Algorithmic Advances in Program Analysis and Their Applications

by

Andreas Pavlogiannis

August 9, 2017

A thesis presented to the
Graduate School
of the
Institute of Science and Technology Austria, Klosterneuburg, Austria
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

The thesis of Andreas Pavlogiannis, titled Algorithmic Advances in Program Analysis and Their Applications, is approved by:

Supervisor: Krishnendu Chatterjee, IST Austria, Klosterneuburg, Austria

Signature: ______________________________

Committee Member: Thomas A. Henzinger, Klosterneuburg, Austria

Signature: ______________________________

Committee Member: Ulrich Schmid, Vienna, Austria

Signature: ______________________________

Committee Member: Martin A. Nowak, Boston, Massachusetts

Signature: ______________________________

Defense Chair: Vladimir Kolmogorov, IST Austria, Klosterneuburg, Austria

Signature: ______________________________

© by Andreas Pavlogiannis, August 9, 2017

All Rights Reserved

I hereby declare that this thesis is my own work and that it does not contain other people's work without this being so stated; this thesis does not contain my previous work without this being stated, and the bibliography contains all the literature that I used in writing the dissertation.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee, and that this thesis has not been submitted for a higher degree to any other university or institution.

I certify that any republication of materials presented in this thesis has been approved by the relevant publishers and co-authors.

Signature: ___________________________

Andreas Pavlogiannis

August 9, 2017

Publications

The research for this dissertation has been partially supported by the Austrian Science Fund (FWF) Grant No P23499- N23, FWF NFN Grant No S11407-N23 (RiSE/SHiNE) and ERC Start grant (279307: Graph Games).

The results appearing in this dissertation have been previously presented in the following conferences.

Chapter 2 is based on joint work with Krishnendu Chatterjee, Amir Kafshdar Goharshady and Rasmus Ibsen-Jensen (appeared in [Chatterjee *et al.*, 2016a]).

Chapter 3 is based on joint work with Krishnendu Chatterjee, Rasmus Ibsen-Jensen and Prateesh Goyal (appeared in [Chatterjee *et al.*, 2015e]), and on joint work with Krishnendu Chatterjee and Rasmus Ibsen-Jensen (appeared in [Chatterjee *et al.*, 2016e]).

Chapter 4 is based on joint work with Krishnendu Chatterjee, Rasmus Ibsen-Jensen and Prateesh Goyal (appeared in [Chatterjee *et al.*, 2015e]).

Chapter 5 is based on joint work with Krishnendu Chatterjee and Bhavya Choudhary (as yet unpublished).

Chapter 6 is based on joint work with Krishnendu Chatterjee and Yaron Venler (appeared in [Chatterjee *et al.*, 2015f]).

Chapter 7 is based on joint work with Krishnendu Chatterjee, Amir Kafshdar Goharshady and Rasmus Ibsen-Jensen (appeared in [Chatterjee *et al.*, 2016a]).

Chapter 8 is based on joint work with Krishnendu Chatterjee and Rasmus Ibsen-Jensen (appeared in [Chatterjee *et al.*, 2015d]).

Chapter 9 is based on joint work with Krishnendu Chatterjee, Nishant Sinha and Kapil Vaidya (as yet unpublished).

Chapter 10 is based on joint work with Krishnendu Chatterjee, Alexander Kößler and Ulrich Schmid (appeared in [Chatterjee *et al.*, 2014c]), as well as on a separate work of these authors (appeared in [Chatterjee *et al.*, 2013]). The credits for the contribution in the modeling and

automated generation of the online algorithms' state spaces in [Chatterjee *et al.*, 2015d] are to Alexander Kößler.

Acknowledgments

I feel obliged to express my gratitude to a number of people who have contributed, directly or not, to the contents of this dissertation.

First, I am thankful to my advisor, Krishnendu Chatterjee, for offering me the opportunity to materialize my scientific curiosity in a remarkably wide range of interesting topics, as well as for his constant availability and continuous support throughout my doctoral studies. I have had the privilege of collaborating with, discussing and getting inspired by all members of my committee: Thomas A. Henzinger, Ulrich Schmid and Martin A. Nowak. The role of the above four people has been very instrumental both to the research carried out for this dissertation, and to the researcher I evolved to in the process.

I have greatly enjoyed my numerous brainstorming sessions with Rasmus Ibsen-Jensen, many of which led to results on low-treewidth graphs presented here. I thank Alex Kößler for our discussions on modeling and analyzing real-time scheduling algorithms, Yaron Velner for our collaboration on the Quantitative Interprocedural Analysis framework, and Nishant Sinha for our initial discussions on partial order reduction techniques in stateless model checking. I also thank Jan Otop, Ben Adlam, Bernhard Kragl and Josef Tkadlec for our fruitful collaborations on topics outside the scope of this dissertation, as well as the interns Prateesh Goyal, Amir Kafshdar Goharshady, Samarth Mishra, Bhavya Choudhary and Marek Chalupa, with whom I have shared my excitement on various research topics. Together with my collaborators, I thank officemates and members of the Chatterjee and Henzinger groups throughout the years, Thorsten Tarrach, Ventsi Chonev, Roopsha Samanta, Przemek Daca, Mirco Giacobbe, Tanja Petrov, Ashutosh Gupta, Arjun Radhakrishna, Petr Novontý, Christian Hilbe, Jakob Ruess, Martin Chmelik, Cezara Drăgoi, Johannes Reiter, Andrey Kupriyanov, Guy Avni, Sasha Rubin, Jessica Davies, Hongfei Fu, Thomas Ferrère, Pavol Cerný, Ali Sezgin, Jan Kretínský, Sergiy Bogomolov, Hui Kong, Benjamin Aminof, Duc-Hiep Chu, and Damien Zufferey. Besides collaborations and

office spaces, with many of the above people I have been fortunate to share numerous whiteboard discussions, as well as memorable long walks and amicable meals accompanied by stimulating conversations. I am highly indebted to Elisabeth Hacker for her continuous assistance in matters that often exceeded her official duties, and who made my integration in Austria a smooth process.

Finally, a special, warm thank you goes to my parents, Theofanis Pavlogiannis and Panagiota Kanellopoulou. This dissertation is illustrative of their successful efforts in shaping a passionate, curiosity-driven personality in me from a young age. Their endless, self-sacrificing, undeniable support has deeply impacted my life and continuous to do so in the present day.

Abstract

This dissertation focuses on algorithmic aspects of program verification, and presents modeling and complexity advances on several problems related to the static analysis of programs, the stateless model checking of concurrent programs, and the competitive analysis of real-time scheduling algorithms. Our contributions can be broadly grouped into five categories.

Our first contribution is a set of new algorithms and data structures for the quantitative and data-flow analysis of programs, based on the graph-theoretic notion of treewidth. It has been observed that the control-flow graphs of typical programs have special structure, and are characterized as graphs of small treewidth. We utilize this structural property to provide faster algorithms for the quantitative and data-flow analysis of recursive and concurrent programs. In most cases we make an algebraic treatment of the considered problem, where several interesting analyses, such as the reachability, shortest path, and certain kind of data-flow analysis problems follow as special cases. We exploit the constant-treewidth property to obtain algorithmic improvements for on-demand versions of the problems, and provide data structures with various tradeoffs between the resources spent in the preprocessing and querying phase. We also improve on the algorithmic complexity of quantitative problems outside the algebraic path framework, namely of the minimum mean-payoff, minimum ratio, and minimum initial credit for energy problems.

Our second contribution is a set of algorithms for Dyck reachability with applications to data-dependence analysis and alias analysis. In particular, we develop an optimal algorithm for Dyck reachability on bidirected graphs, which are ubiquitous in context-insensitive, field-sensitive points-to analysis. Additionally, we develop an efficient algorithm for context-sensitive data-dependence analysis via Dyck reachability, where the task is to obtain analysis summaries of library code in the presence of callbacks. Our algorithm preprocesses libraries in almost linear time, after which the contribution of the library in the complexity of the client analysis is (i) linear in the number of call sites and (ii) only logarithmic in the size of the whole library, as opposed to

linear in the size of the whole library. Finally, we prove that Dyck reachability is Boolean Matrix Multiplication-hard in general, and the hardness also holds for graphs of constant treewidth. This hardness result strongly indicates that there exist no combinatorial algorithms for Dyck reachability with truly subcubic complexity.

Our third contribution is the formalization and algorithmic treatment of the Quantitative Interprocedural Analysis framework. In this framework, the transitions of a recursive program are annotated as good, bad or neutral, and receive a weight which measures the magnitude of their respective effect. The Quantitative Interprocedural Analysis problem asks to determine whether there exists an infinite run of the program where the long-run ratio of the bad weights over the good weights is above a given threshold. We illustrate how several quantitative problems related to static analysis of recursive programs can be instantiated in this framework, and present some case studies to this direction.

Our fourth contribution is a new dynamic partial-order reduction for the stateless model checking of concurrent programs. Traditional approaches rely on the standard Mazurkiewicz equivalence between traces, by means of partitioning the trace space into equivalence classes, and attempting to explore a few representatives from each class. We present a new dynamic partial-order reduction method called the Data-centric Partial Order Reduction (DC-DPOR). Our algorithm is based on a new equivalence between traces, called the observation equivalence. DC-DPOR explores a coarser partitioning of the trace space than any exploration method based on the standard Mazurkiewicz equivalence. Depending on the program, the new partitioning can be even exponentially coarser. Additionally, DC-DPOR spends only polynomial time in each explored class.

Our fifth contribution is the use of automata and game-theoretic verification techniques in the competitive analysis and synthesis of real-time scheduling algorithms for firm-deadline tasks. On the analysis side, we leverage automata on infinite words to compute the competitive ratio of real-time schedulers subject to various environmental constraints. On the synthesis side, we introduce a new instance of two-player mean-payoff partial-information games, and show how the synthesis of an optimal real-time scheduler can be reduced to computing winning strategies in this new type of games.

Table of Contents

Acknowledgments **vii**

Abstract **ix**

List of Tables **xiv**

List of Figures **xviii**

List of Abbreviations **xxiv**

1 Introduction **1**

1.1 Introduction 1

1.2 Motivating Examples 14

1.3 Outline 27

2 Preliminaries **30**

2.1 Introduction 30

2.2 General Notation 31

2.3 Graphs and Tree Decompositions 32

2.4 Recursive State Machines 49

3 Semiring Distance Oracles on Low-treewidth Graphs **54**

3.1 Introduction 54

3.2 Dynamic Algorithms for Preprocess, Update and Query 58

3.3 Optimal Reachability for Low-Treewidth Graphs 66

3.4 Space vs Query Time Tradeoff for Sublinear Space 74

4 Semiring Distances on RSMs of Constant Treewidth **81**

4.1 Introduction 81

4.2 Algorithms for Constant Treewidth RSMs 87
4.3 Experimental Results 98

5 Optimal Dyck Reachability with Applications to Data-dependence and Alias Analysis 102

5.1 Introduction 102
5.2 Preliminaries 108
5.3 Dyck Reachability on Bidirected Graphs 109
5.4 Dyck Reachability on General Graphs 126
5.5 Library/Client Dyck Reachability 128
5.6 Experimental Results 141

6 Quantitative Interprocedural Analysis 149

6.1 Introduction 149
6.2 Definitions 154
6.3 Applications: Theoretical Modeling 157
6.4 An Algorithm for the Quantitative Interprocedural Analysis Problem 165
6.5 Experimental Results: Three Case Studies 180

7 Semiring Distances on Concurrent Systems of Constant-treewidth Components 194

7.1 Introduction 194
7.2 Definitions 201
7.3 Modeling Example 204
7.4 Concurrent Tree Decomposition 206
7.5 Semiring Distances on Concurrent Graphs 211
7.6 Conditional Optimality for Two Graphs 227
7.7 Experimental Results 231

8 Quantitative Verification on Constant-treewidth Graphs 235

8.1 Introduction 235
8.2 Definitions 239
8.3 Minimum Cycle 241
8.4 The Minimum Ratio and Mean Cycle Problems 246
8.5 The Minimum Initial Credit Problem 252

8.6 Experimental Results 267

9 Data-centric Dynamic Partial Order Reduction 273

9.1 Introduction 273

9.2 Preliminaries 277

9.3 Observation Trace Equivalence 285

9.4 Annotations 292

9.5 Data-centric Dynamic Partial Order Reduction 302

9.6 Beyond Acyclic Architectures 308

9.7 Experiments 312

10 Automated Competitive Analysis in Real-time Scheduling with Graphs and Games318

10.1 Introduction 318

10.2 Problem Definition 320

10.3 Modeling Formalisms in Our Framework 323

10.4 Competitive Analysis of On-line Scheduling Algorithms 330

10.5 Competitive Synthesis of On-line Scheduling Algorithms 348

Bibliography 362

List of Tables

3.1 A data structure for handling single-source and pair semiring distance queries on a graph G of n nodes and constant treewidth. 55

3.2 Data structures for pair and single-source reachability queries, on a directed graph G with n nodes, m edges, and a treewidth t. The model of computation is the standard RAM model with wordsize $W = \Theta(\log n)$. We denote by $\alpha(n)$ the inverse of the Ackermann function on input n. Space usage refers to the total space used during the preprocessing and query phase. 56

3.3 Data structures for pair and single-source distance queries, on a weighted directed graph G with n nodes, m edges, and a tree decomposition of width $O(1)$ and height h. The number ϵ can be any fixed number in $[\frac{1}{2}, 1]$ and $\alpha(n)$ is the inverse Ackermann function. Space usage refers to the total space used during the preprocessing and query phase. When measuring space complexity, we do not count the input size. Rows 1-6 are previous results. 57

4.1 Interprocedural same-context semiring distances on RSMs with n nodes, b boxes and constant treewidth, for stack height h. 83

4.2 Interprocedural same-context semiring distances on RSMs with n nodes, b boxes and constant treewidth, where the semiring is over the subset of $|D|$ elements and the plus operator is the meet operator of the IFDS framework. Existing results are taken from [Reps *et al.*, 1995a]. Our results are obtained from Corollary 4.1 and Corollary 4.2 84

4.3 Interprocedural same-context reachability on RSMs with n nodes, b boxes and constant treewidth. Existing results are taken from [Reps *et al.*, 1995a] using the IFDS/IDE framework with $|D| = 1$. Our results are obtained from Corollary 4.3. 84

4.4 Interprocedural same-context distances with non-negative weights for RSMs with n nodes, k CSMs, b boxes and constant treewidth. [1] The preprocessing time is obtained by executing Dijkstra's algorithm b times in each of the k CMSs, followed by executing Dijkstra's algorithm from n source nodes. [2] The single-source and pair query times are obtained by executing Dijkstra's algorithm b times in each of the k CMSs. 85

4.5 Illustration of RSMDistance on the tree decompositions of methods dot_vector and dot_matrix from Fig. 2.6. Table 4.5a shows the local distance maps for each bag and stack height $\ell = 0, 1$. Table 4.5b shows how the distance query $d(1, 6)$ in method dot_vector is handled. 91

4.6 Average statistics gathered from our experiments on the DaCapo benchmark suit. Times are in microseconds. 99

4.7 Average statistics gathered from our experiments on the DaCapo benchmark suit. Times are in microseconds. 100

5.1 Comparison of our results with existing work for Dyck reachability on bidirected graphs with n nodes and m edges. We also prove a matching lower-bound for the worst-case analysis. Thus our algorithm is optimal wrt worst-case complexity. 107

5.2 Library/Client CFL reachability on the library graph of size n_1 and the client graph of size n_2. s is the number of library summary nodes, as defined in [Tang *et al.*, 2015]. k_1 is the number of call sites in the library code, with $k_1 < s$. k_2 is the number of call sites in the client code. 108

5.3 Comparison between our algorithm and the existing from [Zhang *et al.*, 2013]. The first three columns contain the number of fields (Dyck parenthesis), nodes and edges in the SPG of each benchmark. The last two columns contain the running times, in seconds. 143

5.4 Running time of our algorithm vs the TAL and CFL approach for data-dependence analysis with library summarization. Times are in milliseconds. MEM-OUT indicates that the algorithm run out of memory. The number of nodes and treewidth reflects the average case among all methods in each benchmark. 147

5.5 Memory usage of our algorithm vs the TAL and CFL approach for data-dependence analysis with library summarization. Memory usage is in Megabytes. MEM-OUT indicates that the algorithm run out of memory. The number of nodes and treewidth reflects the average case among all methods in each benchmark. . 148

6.1 Experimental results for container usage analysis 184

6.2 Experimental results for frequency of functions 186

6.3 Experimental results for frequency of classes 191

7.1 The algorithmic complexity for computing algebraic path queries wrt a closed, complete semiring on a concurrent graph G which is the product of two constant-treewidth graphs G_1, G_2, with n nodes each. 195

7.2 Percentage of cases for which the transitive closure of the graph G for the given value of λ is at most 5% slower than the time required to obtain the transitive closure of G for the best λ. 232

7.3 Time required for the transitive closure on 2-self concurrent graphs extracted from methods of the java.util.concurrent library. Each constituent graph has $\mathbf{n}$ nodes. $\mathbf{T_o}(\mathrm{s})$ and $\mathbf{T_b}(\mathrm{s})$ correspond to our method and the baseline method respectively. 234

8.1 Time complexity of existing and our solutions for the minimum mean-cycle value and ratio-cycle value problem in constant treewidth weighted graphs with n nodes and largest absolute weight W, when the output is the (irreducible) fraction $\frac{a}{b} \neq 0$. 236

8.2 Complexity of the existing and our solution for the minimum initial credit problem on weighted graphs of n nodes, m edges, and largest absolute weight W.236

8.3 Asymptotic complexity of compared minimum mean cycle algorithms. 268

8.4 The time performance of Fig. 8.4 (in μs). 270

8.5 The space performance of Fig. 8.4 (in KB). 271

8.6 The time performance of Fig. 8.5 in μs. 272

9.1 Notation on the concurrent architecture. 282

9.2 Notation on traces. 284

9.3 Experimental results on two synthetic benchmarks. 315

9.4 Experimental results on four benchmarks from SV-COMP. 316

10.1 The tasksets used to generate Fig. 10.6. 345
10.2 Columns show the mean workload restriction. The check-marks indicate that the corresponding scheduler is optimal for that mean workload restriction, among the six schedulers we examined. We see that the optimal scheduler can vary as the restrictions are tighter, and in a non-monotonic way. LLF, EDF, DSTAR and DOVER were not optimal in any case and hence not mentioned. 346
10.3 Taskset of Fig. 10.7 (left) and Table 10.2 (right). 347
10.4 Competitive ratio of TD1. 347
10.5 Scalability of our approach for tasksets of various sizes N and $D_{\max}$. For each taskset, the size of the state space of the clairvoyant scheduler is shown, along with the mean size of the product LTS, and the mean and maximum time to solve one instance of the corresponding ratio objective. 348

List of Figures

1.1 Example of a program consisting of two methods, their control-flow graphs $G_i = (V_i, E_i')$ where nodes correspond to line numbers, and the corresponding tree decompositions. 15

1.2 Underutilized container analysis. 17

1.3 A system of two processes with two events each. 19

1.4 Task sequences from taskset $\mathcal{T}_1$. Solid lines represent workload. Dashed lines represent deadlines. 22

1.5 Task sequences from taskset $\mathcal{T}_2$. Solid lines represent workload. Dashed lines represent deadlines. 24

2.1 A graph G with treewidth 2 (left) and a corresponding tree-decomposition $\mathrm{Tree}(G)$ (right). 34

2.2 Illustration of Lemma 2.3. If P is the unique simple path $B_1 \rightsquigarrow B_4$ in $\mathrm{Tree}(G)$, then there exist (not necessarily distinct) $x_i \in B_{i-1} \cap B_i$ with $1 < i \leqslant 4$ such that $d(u,v) = d(u,x_2) \otimes d(x_2,x_3) \otimes d(x_3,x_4) \otimes d(x_4,v)$. 36

2.3 Illustration of one recursive step of Rank on a component $\mathcal{C}$ (gray). $\mathcal{C}$ is split into two sub-components $\overline{\mathcal{C}}_1$ and $\overline{\mathcal{C}}_2$ by removing a list of bags $\mathcal{X} = (B_i)_i$. Once every λ recursive calls, $\mathcal{X}$ contains one bag, such that the neighborhood $\mathsf{Nh}(\overline{\mathcal{C}}_i)$ of each $\overline{\mathcal{C}}_i$ is at most half the size of $\mathsf{Nh}(\mathcal{C})$ (i.e., the red area is split in half). In the remaining $\lambda - 1$ recursive calls, $\mathcal{X}$ contains m bags, such that the size of each $\overline{\mathcal{C}}_i$, is at most $\frac{1+\delta}{2}$ fraction the size of $\mathcal{C}$. (i.e., the gray area is split in almost half). 41

2.4 Illustration of Lemma 2.8. Since B_2 belongs to $\mathcal{C}(\mathcal{B})$ and the blue sub-component has not been split yet, the bag B^r is in the neighborhood of the blue sub-component, and thus in the neighborhood of $\mathcal{C}(\mathcal{B})$. 46

2.5 Given the tree-decomposition $\mathrm{Tree}(G)$ on the left, the graph in the middle is the corresponding $\mathsf{R_G}$ and the one on the right is the corresponding tree-decomposition $\widehat{\mathsf{R_G}} = \mathsf{Replace}(\mathsf{R_G})$ after replacing each bag B with $\mathsf{NhV}(B)$. . . 49
2.6 Example of a program consisting of two methods, their control-flow graphs $G_i = (V_i, E_i')$ where nodes correspond to line numbers, and the corresponding tree decompositions, each one achieving treewidth 2. 53

3.1 Illustration of the inductive argument of Preprocess. 61
3.2 Illustration of Query in computing the distance $d(u, v) = \otimes(P)$ as a sequence of U-shaped paths, whose weight has been captured in the local distance map of each bag. When B_z is examined, with $z = \mathsf{argmin}_{x \in P} \mathsf{Lv}(x)$, it will be $\delta_u(z) = d(u, z)$ and $\delta_v(z) = d(z, v)$, and hence by distributivity $d(u, v) = \bigotimes(\delta_u(z), \delta_v(z))$. 65
3.3 a, c: A graph G and a tree-decomposition $\mathrm{Tree}(G)$. b: The sets F_u constructed from step 5 to answer single-source queries. The j-th bit of a set F_u is 1 iff $(u, v) \in E^*$, where v is such that $i_v - i_u = j$. d: The set sequences $(\mathsf{F}_u^i)_i$ and $(\mathsf{T}_u^i)_i$ constructed from step 6 to answer pair queries, for $u = 6$. For every $i \in \{0, 1, 2, 3\}$ and ancestor B_6^i of B_6 at level i, every node $v \in B_u^i$ is assigned a local index $l_v^{B_6^i}$. The j-th bit of set F_6^i (resp. T_6^i) is 1 iff $(6, v) \in E^*$ (resp. $(v, 6) \in E^*$), where v is such that $l_v^{B_6^i} = j$. 69

5.1 A state of BidirectedReach consists of a set of trees, with outgoing edges coming only from the root of each tree. 113
5.2 The intermediate stages of BidirectedReach starting from the stage of Figure 5.1. 115
5.3 A union sequence σ_1 and the corresponding graph G^{σ_1}. 123
5.4 **(5.4a)** A grammar $\mathcal{G}$ for the language $a^n b^n$, **(5.4b)** The gadget graph $G^{\mathcal{G}}$, **(5.4c)** The parse graph $G_s^{\mathcal{G}}$, given a string $s = s_1, \dots d_n$. 127
5.5 Example of a library/client program and the corresponding program-valid data-dependence graph. The library consists of method $g()$ which has a callback function $f(x, y)$. The client implements $f(x, y)$ either as $f_1(x, y)$ or $f_2(x, y)$. The parenthesis-labeled edge model context-sensitive dependencies on parameter passing and return. Note that depending on the implementation of f, there is a data dependence of the variable p on y. 132

5.6 A minimal program and its (bidirected) SPG. Circles and squares represent variable nodes and object nodes, respectively. Only forward edges are shown. . 142

5.7 Running time of our algorithm vs [Zhang *et al.*, 2013] for context-insensitive field-sensitive points-to analysis on SPGs of various benchmarks. The top row shows the total time (in ms) taken for the slowest method to perform the analysis. The total time is taken as the sum of the time spent in analyzing library and client code. The y-axis shows the percentage of time that each method took as compared to the slowest method. 143

5.8 Time comparison of our algorithm with TAL and CFL for performing data dependence analysis via CFL reachability. The top row shows the total time (in ms) taken for the slowest method to perform the analysis. The total time is taken as the sum of the time spent in analyzing library and client code. The y-axis shows the percentage of time that each method took as compared to the slowest method. The benchmark *serial* is missing, as TAL mems-out. 146

5.9 Space comparison of our algorithm with TAL and CFL for performing data dependence analysis via CFL reachability. The top row shows the total memory usage (in MB) of the most memory-demanding method to perform the analysis. The total space is taken as the maximum of the space used in analyzing library and client code. The y-axis shows the percentage of memory that each method used as compared to the most memory-demanding method. The benchmark *serial* is missing, as TAL mems-out. 146

6.1 An RSM that consists of two CSMs which represent the functions main() and foo(). 155

6.2 An example for underutilized container analysis. 159

6.3 Illustration of the Quantitative Interprocedural Analysis problem for capturing the interprocedural WCET. Positive weights indicate good transitions, whereas negative weights indicate bad transitions. 163

6.4 Example of an execution path $\pi = \langle \mathcal{C}_1, \dots, \mathcal{C}_\ell \rangle$, where j indexes the j-th configuration of π, and α_j is the stack of $\mathcal{C}_j$. The configuration $\mathcal{C}_i$ is a local minimum of π. The suffix π_i of π starting at the i-th configuration is a non-decreasing execution path, as the top symbol of α_i is never popped. 170

6.5 Two CSMs f and g and their summary graph. 175

6.6 The ROC curves for the analysis of frequently invoked methods. The left plot shows the results when all methods are analyzed. The right plot shows the results when only the active methods are analyzed. The different dots of each curve correspond to different threshold values λ. 185

6.7 An example from benchmark batik. The method **run** invokes **cleared** in a loop, and in every invocation, one element of elementsById is removed and one element is added. Thus in this loop the total number of elements in elementsById is bounded. 187

6.8 An example from benchmark muffin. The method **findRecords** has a recursive call, and method **addCNAME** adds an element to vector backtrace. A path with recursion depth n adds n elements to backtrace. Hence, backtrace may have unbounded number of elements and it is not underutilized. 188

6.9 An example from benchmark muffin. The method **cgi** allocates the container attrs and potentially adds it many elements. The method **recvReply** performs a **get** operation over attrs in a loop. Since we analyze not only the operations that are nested in the allocation site, we detect that attrs is not overpopulated. 189

6.10 The ROC curves for the analysis of frequently invoked classes. The different dots of each curve correspond to different threshold values λ. 192

6.11 Illustration of the difference between hot methods and hot collections of methods. Here, the collection of the methods of class A is λ-hot for any threshold $\lambda \leqslant 1$. However, each of the methods $f1()$, $f2()$ and $f3()$ of A are at most $\frac{1}{3}$-hot. Hence determining frequently invoked collections of methods requires separate analysis, and cannot rely on simply detecting frequently invoked methods. 193

7.1 Given a concurrent graph G of two constant-treewidth graphs of n nodes each, the figure illustrates the time required by the variants of our algorithms to preprocess G, and then answer i pair queries and j partial pair queries. The different regions correspond to the best variant for handling different number of such queries. In contrast, the current best solution requires $O(n^6 + i + j)$ time. For ease of presentation we omit the $O(\cdot)$ notation. 198

7.2 A concurrent program, its control flow graph, and a tree decomposition of the control-flow graph. 202

7.3 The tree-decomposition $\mathsf{ConcurTree}(G)$ of a concurrent graph G of two constant-treewidth graphs G_1 and G_2. 208

7.4 A graph G (left), and G' that is a 2-self-product of a graph G'' of treewidth 1 (right). The weighted edges of G correspond to weighted red edges on G'. The distance $d(x_i, x_j)$ in G equals the distance $d(\langle x_i, x_i\rangle, \langle x_j, x_j\rangle) = d(\langle \perp, x_i\rangle, \langle \perp, x_j\rangle)$ in G'. 229

7.5 Time required to compute the transitive closure on concurrent graphs of various sizes. Our algorithm is run for $\lambda = 7$. TO denotes that the computation timed out after 30 minutes. 233

8.1 Exponential search followed by a binary search to determine $\lfloor \nu^* \rfloor$ 248

8.2 Solving the value problem using operations on the graph $\mathcal{G}$. Initially we examine $\mathcal{G}^0$, and a non-positive cycle is found (boldface edges) with highest-energy node x. Thus $\mathsf{E}(x) = 0$, and we proceed with $\mathcal{G}^1$, to discover $\mathsf{E}(u) = 0$. In $\mathcal{G}^2$ all cycles are positive, and the energy of each remaining node is minus its distance to z. 261

8.3 Illustration of the $\alpha_1 + \alpha_2$ operation, corresponding to concatenating paths P_1 and P_2. The path P_j^i denotes the i-th prefix of P_j. We have $P = P_1 \circ P_2$, and the corresponding tripplet $\alpha = (a, b, c)$ denotes the weight a of P, its highest-energy node b, and the weight c of a highest-energy prefix. 263

8.4 Average performance of minimum mean cycle algorithms. 269

8.5 Comparison of running times for the minimum initial credit problem. 270

9.1 The control-flow graph CFG_i is a sequential composition of these five atomic graphs. 279

9.2 *(Left)*: A method withdraw executed whenever some amount is to be extracted from the balance of a bank account. *(Right)*: Representation of withdraw in our concurrent model. The root node is x_1. The program structure orders $\mathsf{PS}(e_2, e_4)$. We have $\mathsf{loc}(e_1) = \mathsf{loc}(e_5)$ and $\mathsf{loc}(e_2) = \mathsf{loc}(e_4) = \mathsf{loc}(e_6)$. 280

9.3 Trace exploration on the system of Fig. 9.2 with two processes, where initially balance $\leftarrow 4$ and both withdrawals succeed. 288

9.4 An architecture of two processes with $n + 1$ events each. 290

9.5 An architecture of k processes with two events each. 291

9.6 The reduction of 3SAT over ϕ to ACYCLIC EDGE ADDITION over (G, H). The nodes and solid edges represent the graph G. The dashed edges represent the triplets in H. 298
9.7 A cyclic architecture of three processes. 310
9.8 Converting a cyclic architecture to a star architecture which is acyclic. On the star, solid edges correspond to observation equivalence interleavings, and dashed edges correspond to Mazurkiewicz equivalence interleavings. 313

10.1 EDF for two tasks represented as a deterministic LTS. 325
10.2 Example of a safety LTS $L_{\mathcal{S}}$. 327
10.3 Example of a liveness LTS $L_{\mathcal{L}}$. 328
10.4 Example of a limit-average LTS $L_{\mathcal{W}}$. 329
10.5 An example of a multi-graph G. 333
10.6 The competitive ratio of the examined algorithms in various tasksets under no constraints. Every examined algorithm is optimal in some taskset, among all others. 345
10.7 Restricting the absolute workload generated by the adversary. 346
10.8 Illustration of the construction of a game from a 3-SAT formula. 355

List of Abbreviations

BFS Breadth-First Search

CFG Control-flow Graph

DFS Depth-First Search

fpr false positive rate

EDF Earliest Deadline First

FIFO First In First Out

FPT Fixed Parameter Tractable

JVM Java Virtual Machine

LCA Lowest Common Ancestor

LLF Least Laxity First

LTS Labeled Transition System

MSO Monadic Second Order Logic

POR Partial-order Reduction

DPOR Dynamic Partial-order Reduction

DC-DPOR Data-centric Dynamic Partial-order Reduction

QIA Quantitative Interprocedural Analysis

RAM Random Access Machine

ROC Receiver Operating Characteristic

RSM Recursive State Machine

SAT Satisfiability

SCC Strongly Connected Component

SP Static Priorities

SRT Shortest Remaining Time

tpr True Positive Rate

WCET Worst-case Execution Time

wlog without loss of generality

wrt with respect to

1 Introduction

1.1 Introduction

The central aim of software verification is to guarantee correctness of programs, where correctness is formally specified. For example, a program for sorting names in alphabetical order should not crash, should always produce output, and the output it produces should be indeed a sorted version of the input. Correctness criteria might even specify the outcome when the input is not expected, e.g. if instead of names we accidentally input a list of numbers, we would not expect a sorted list of names as output, but some form of warning. Beyond correctness, there are many other desirable aspects of program behavior we would like to be able to verify, such as responsiveness, efficiency and resource consumption. In all such cases, we would like to verify program behavior systematically, by means of another program, the "verifier", which would take as input the program under consideration, and would deduce whether the desirable aspects are met.

Turing's seminal paper [Turing, 1936] marked the dawn of computer science with a negative result, namely the undecidability of the halting problem. In high level, there is no systematic (algorithmic) way to decide whether a Turing machine halts on some input. Rice's theorem [Rice, 1953] generalizes the halting problem to any non-trivial property (i.e., any property held by some programs but not all) of the language recognized by a Turing machine. The theorem implies that every non-trivial property concerning the behavior of programs written in a Turing-complete language admits no automated solution (see [Hopcroft, 2007; Sipser, 1996] for a general reference).

Traditionally, there have been two general ways to circumvent this universal negative result.

1. Algorithms with relaxed correctness. Such algorithms can operate on Turing-complete models of computation but have weak correctness guarantees, and typically make one-sided or two-sided errors by sacrificing soundness or completeness (or both). For example, given an input program and a specification, the algorithm might generate spurious warnings that the program violates the specification.

2. Algorithms that operate on abstract models of actual programs. Such algorithms come with strong correctness guarantees, but operate on non Turing-complete models of computation, which approximate program behavior.

In many cases these two approaches are combined, for example in order to tackle intractability due to complexity besides undecidability. There also exist approaches which operate on progressively refined abstractions of the program. Typically, each abstraction is complete wrt program behavior but unsound, i.e., some model executions do not correspond to actual program executions. Upon a spurious warning of incorrect program behavior, the abstraction is refined so that the new overapproximation does not include the spurious instances.

In most of the above approaches, once a model of the program and a specification have been fixed, an efficient algorithm is needed to verify the compliance of the model to the specification. The focus of this dissertation is on algorithmic improvements for such verification tasks, which operate on program abstractions represented as finite or recursive graphs, and offer strong worst-case complexity guarantees. In the rest of this introduction we first outline some fundamental concepts in program analysis, quantitative verification, stateless model checking and real-time scheduling. Afterwards, we present some examples that motivate the results presented in this dissertation.

1.1.1 Program Analysis and Verification

Recursive State Machines. Recursive State Machines (RSMs) were first introduced in [Alur *et al.*, 2005] to serve as a model of programs consisting of several functions which invoke each other. Although they are strongly equivalent to Pushdown Systems (PDSs) [Alur *et al.*, 2005], they are a more intuitive and natural model of programs, and the algorithmic complexity of

various relevant problems depends explicitly on natural parameters, such as the number of entries, exits and boxes. A large number of path-related problems have been a subject of extensive study on recursive graphs (either RSMs or PDSs). Reachability has been studied in the works of [Bouajjani *et al.*, 1997; Schwoon, 2002; Chaudhuri, 2008; Alur *et al.*, 2016], and extended to more complex model checking problems, such as linear time logics [Esparza *et al.*, 2000; Schwoon, 2002; Alur *et al.*, 2016]. A notable extension to reachability algorithms is, so called, generalized reachability [Bouajjani *et al.*, 2003a; Reps *et al.*, 2005; Lal *et al.*, 2005; Reps *et al.*, 2007; Lal and Reps, 2008; Chatterjee *et al.*, 2017], where each transition of the RSM is labeled with the weights of a semiring (typically a semi-lattice), and the task is to compute the semiring distance between nodes and configurations of the system. A special type of RSMs, namely single-entry single-exit RSMs have been used on interprocedural analysis, usually under the name "supergraphs". The famous IFDS/IDE framework [Reps *et al.*, 1995a; Reps, 1997; Horwitz *et al.*, 1995; Sagiv *et al.*, 1996; Reps, 1997; Bodden, 2012] reduces a large class of interprocedural analyses to essentially computing semiring distances. In some fundamental cases [Reps *et al.*, 1995a; Reps, 1997], the problem is reduced to the problem of reachability on the "exploded supergraph", which encodes both control-flow and data-flow information. Several variants of recursive graphs have also been used for the verification of concurrent systems as for example in [Harel *et al.*, 1997; Alur *et al.*, 1999; Bouajjani *et al.*, 2003b; Bouajjani *et al.*, 2005; Qadeer and Rehof, 2005; Bozzelli *et al.*, 2006; Kahlon and Gupta, 2007; La Torre *et al.*, 2008; Atig *et al.*, 2008; Suwimonteerabuth *et al.*, 2008; Lal *et al.*, 2008; Lal and Reps, 2009; Kahlon *et al.*, 2013; Farzan *et al.*, 2013] and many practical tools have been developed as well [Qadeer and Rehof, 2005; Lal and Reps, 2009; Suwimonteerabuth *et al.*, 2008; Lal *et al.*, 2012]. RSMs have also been studied in the context of games as e.g. in [Alur *et al.*, 2006; Chatterjee and Velner, 2012].

Static analysis. Static analysis techniques provide ways to obtain information about programs without actually executing the programs on specific inputs. Static analysis explores the program behavior for *all* possible inputs and *all* possible executions. For non-trivial programs, it is impossible to explore all the possibilities, and hence static analysis uses approximations to account for all the possibilities [Cousot and Cousot, 1977b]. The various static analysis methods can be broadly grouped into two categories. In the *intraprocedural setting*, each method of the program is analyzed separately. Such analysis is more lightweight but also less precise, as it ignores the effects of program executions which involve invocations and returns to and from other

procedures. In the *interprocedural setting*, the effect of executions that involve the invocation of program procedures is taken into account. Interprocedural analyses are usually more resource demanding, but also more precise.

Data-flow analysis. The main aim of data-flow analysis is to infer various facts (often called *data facts*) about the values of variables appearing in various program locations (i.e., locations of the underlying control-flow graph). It appears at the heart of numerous applications, ranging from alias analysis, to data dependencies (modification and reference side effect), to constant propagation, to live and use analysis [Reps *et al.*, 1995a; Sagiv *et al.*, 1996; Callahan *et al.*, 1986; Grove and Torczon, 1993; Landi and Ryder, 1991; Knoop *et al.*, 1996; Cousot and Cousot, 1977a; Müller-Olm and Seidl, 2004; Giegerich *et al.*, 1981; Knoop and Steffen, 1992; Reps *et al.*, 2005; Naeem and Lhoták, 2008; Zhang *et al.*, 2014].

A wide range of data-flow problems has an algebraic paths formulation, expressed as a "meet-over-all-paths" analysis [Kildall, 1973]. Perhaps the most well-known case is that of *interprocedural finite distributive subset (IFDS)* flow functions considered in [Reps *et al.*, 1995a]. Given a finite domain D and a universe F of distributive data-flow functions $f : 2^D \to 2^D$, a weight function wt associates each edge of the control-flow graphs with a flow function. A flow function f is distributive if $f(X) = \bigcup_{x \in X} f(\{x\})$ (or $f(X) = \bigcap_{x \in X} f(\{x\})$, depending on the problem under consideration). The weight of a path is then defined as the composition of the flow functions along its edges, and the data-flow distance between two nodes u, v is the meet $\sqcap$ (union or intersection) of the weights of all $u \rightsquigarrow v$ paths. The data-flow analysis then can be reduced to the problem of reachability on the "exploded supergraph", which encodes both control-flow and data-flow information [Reps *et al.*, 1995a; Reps, 1997]. The *interprocedural distributive environment (IDE)* framework [Sagiv *et al.*, 1996] extends the IFDS framework to unbounded domains. In this case, the flow functions (called environment transformers) map elements from the finite domain D to values in an infinite set (e.g., of the form $f : D \to \mathbb{N}$). An environment transformer is denoted as $f[d \to \ell]$, meaning that the element $d \in D$ is mapped to value ℓ, while the mapping of all other elements remains unchanged. Similarly to the case of IFDS, IDE problems are phrased as "meet-over-all-paths" problems between program locations. Data-flow analysis is also a field of intensive study in the context of concurrency (e.g. [Bouajjani *et al.*, 2003a; Grunwald and Srinivasan, 1993; Knoop *et al.*, 1996; Farzan and Madhusudan, 2007; Chugh *et al.*, 2008; Kahlon *et al.*, 2009;

De *et al.*, 2011]), where (part of) the underlying analysis is also based on a "meet-over-all-paths" approach, as in the case of IFDS/IDE.

CFL and Dyck reachability. A very important instance of language reachability in static analysis is *CFL* reachability, where the input language is context-free, which can be used to model, e.g., context-sensitivity or field-sensitivity. The CFL reachability formulation has applications to a very wide range of static analysis problems, such as interprocedural data-flow analysis [Reps *et al.*, 1995b], slicing [Reps *et al.*, 1994], shape analysis [Reps, 1995a], impact analysis [Arnold, 1996], type-based flow analysis [Rehof and Fähndrich, 2001] and alias/points-to analysis [Shang *et al.*, 2012; Sridharan and Bodík, 2006a; Sridharan *et al.*, 2005; Xu *et al.*, 2009a; Yan *et al.*, 2011a; Zheng and Rugina, 2008], etc. In practice, widely-used large-scale analysis tools, such as Wala [Wal, 2003] and Soot [Vallée-Rai *et al.*, 1999; Bodden, 2012], equip CFL reachability techniques to perform such analyses. In most of the above cases, the languages used to define the problem are those of properly-matched parenthesis, which are known as Dyck languages, and form a proper subset of context-free languages. Thus Dyck reachability is at the heart of many problems in static analysis.

Preprocess vs Query. A topic of widespread interest is that of on-demand analysis [Babich and Jazayeri, 1978; Zadeck, 1984; Horwitz *et al.*, 1995; Duesterwald *et al.*, 1995; Reps, 1995b; Sagiv *et al.*, 1996; Reps, 1997; Yuan *et al.*, 1997; Naeem *et al.*, 2010]. The main goal is to avoid analyzing the whole program when the focus is on obtaining solutions for specific locations of the program. Such analysis has several advantages, such as (quoting from [Horwitz *et al.*, 1995; Reps, 1997]) (i) narrowing down the focus to specific points of interest, (ii) narrowing down the focus to specific data-flow facts of interest, (iii) reducing work in preliminary phases, (iv) sidestepping incremental updating problems, and (v) offering demand analysis as a user-level operation. For example, in constant propagation a relevant question is whether some variable remains constant between a pair of control-flow locations. The problem of on-demand analysis allows us to distinguish between a single preprocessing phase (one-time computation), and a subsequent query phase, where queries are answered on demand. The two extremes of the preprocessing and query phase are: (i) *complete preprocessing* (aka *transitive closure* computation) where the result is precomputed for every possible query, and hence queries are answered by simple table lookups; and (ii) *no preprocessing* where every query requires a new computation. In general, there can be a tradeoff between the preprocessing and query

computation. Most of the existing work on on-demand analysis does not make a formal distinction between preprocessing and query phases, as the provided complexities only guarantee the *same worst-case complexity* property, namely that the total time for handling any sequence of queries is no worse than the complete preprocessing. Hence most existing tradeoffs are practical, without any theoretical guarantees.

1.1.2 Quantitative Verification

Boolean vs quantitative verification. The traditional view of verification has been *qualitative (boolean)*, in which traces of a system are classified as “correct" or “incorrect”. In the recent years, motivated by applications to analyze resource-constrained systems (such as embedded systems), there has been a large interest to study *quantitative* properties of systems. Quantitative verification extends the traditional boolean verification in two directions. First, it enjoys a rich expressive power of quantitative properties, such as resource consumption and performance guarantees, which cannot be expressed in a boolean setting. Second, it allows for quantifiable measures of model compliance to specifications, as opposed to the boolean nature of model checking. For example, a boolean property may require that every request is eventually granted, whereas a quantitative property can measure for each trace the average waiting time between requests and corresponding grants. Not only is this property inherently quantitative, quantities can also be used to measure the extent to which a trace deviates from the desired behavior.

Quantitative extensions. Given the importance of quantitative verification, the traditional qualitative view of verification has been extended in several ways. Some notable examples concern quantitative languages and quantitative automata [Droste *et al.*, 2009; Chatterjee *et al.*, 2010a; Chatterjee *et al.*, 2010c; Chatterjee *et al.*, 2014b; Chatterjee *et al.*, 2010b; Velner *et al.*, 2015a; Droste and Meinecke, 2012; Chatterjee *et al.*, 2016c; Chatterjee *et al.*, 2016b; Chatterjee *et al.*, 2016d; Chatterjee *et al.*, 2015c; Boker *et al.*, 2015], as well as quantitative logics for specification languages [Boker *et al.*, 2011; Bouyer *et al.*, 2014; Almagor *et al.*, 2013]. Quantitative objectives have been proposed in several applications such as for worst-case execution time (see [Wilhelm *et al.*, 2008] for survey), power consumption [Tiwari *et al.*, 1994], prediction of cache behavior for timing analysis [Ferdinand *et al.*, 99], and performance measures [Bloem *et al.*, 2009a; Chatterjee *et al.*, 2010a; Filar and Vrieze, 1997; Droste and Meinecke, 2010], to name

a few. Quantitative abstraction-refinement frameworks for finite-state systems with mean-payoff objectives have also been studied in [Cerný *et al.*, 2013]. Quantities are also suitable for expressing robustness of systems, which extends the idea of model checking to model measuring [Černý *et al.*, 2012; Samanta *et al.*, 2013; Henzinger and Otop, 2013; Henzinger and Otop, 2014; Henzinger *et al.*, 2014; Henzinger *et al.*, 2016]. Besides verification, quantities have also been used in quantitative synthesis and repair [Bloem *et al.*, 2009a; Bloem *et al.*, 2009b; Černý *et al.*, 2011; Samanta *et al.*, 2014; Chatterjee *et al.*, 2015b; D'Antoni *et al.*, 2016].

The core algorithmic question in many of the above studies is a graph algorithmic problem that requires to analyze a graph wrt a quantitative property.

The algebraic path problem. The algebraic path problem is phrased wrt a closed semiring $S = (\Sigma, \oplus, \otimes, \overline{\mathbf{0}}, \overline{\mathbf{1}})$ and a weighted graph $G = (V, E, \mathsf{wt})$, where the weight function $\mathsf{wt} : E \to \Sigma$ assigns to each edge a weight of the domain of the semiring. The problem asks to determine the *semiring distance* between pairs of nodes. The algebraic path problem generalizes various related graph path problems, such as the reachability, shortest path, maximum reliability and maximum capacity path problems [Cormen *et al.*, 2009; Aho and Hopcroft, 1974; Mohri, 2002; Zimmermann, 2011; Carré, 1971; Lehmann, 1977; Mahr, 1984; Fink, 1992]. In static program analysis, some data-flow frameworks (consisting of finite, distributive, subset data-flow functions or *IFDS*) reduce to computing semiring distances over an appropriate data-flow semiring [Reps *et al.*, 1995a; Horwitz *et al.*, 1995]. Several existing algorithms for solving specific instances of the problem can be straightforwardly generalized to solve the algebraic path problem, such as Warshall's algorithm [Warshall, 1962] for computing the transitive closure, Floyd's algorithm [Floyd, 1962] for computing all-pairs shortest paths, the Gauss-Jordan algorithm [Cormen *et al.*, 2009] for matrix inversion, Viterbi's algorithm [Viterbi, 1967] used in probabilistic parsing and Kleene's construction [Kleene, 1956] of regular expressions from finite state automata. We refer to [Fink, 1992] for a survey on various algorithms for the algebraic path problem.

IFDS/IDE as algebraic path problems. The IFDS/IDE frameworks make, in essence, an algebraic treatment of the data-flow analysis. This is a consequence of the distributivity of the data-flow functions, which allows the composition operator $\circ$ to distribute over the meet operator $\sqcup$. The underlying semiring forms a semi-lattice $(F, \sqcap, \circ, \varnothing, I)$, where F is the set of all distributive

data-flow functions, and I is the identity function and the identity environment transformer in the case of IFDS and IDE respectively.

Reachability/distance problems. The *pair* reachability and distance problems are special cases of the algebraic path problem phrased on the appropriate semiring (the *boolean semiring* $(\{0,1\}, \vee, \wedge, 0, 1)$ in the case of reachability and the *tropical semiring* $(\mathbb{R}_{\geqslant}0 \cup \{\infty\}, \min, +, \infty, 0)$ in the case of distances). Both problems are two of the most classic graph algorithmic problems which, given a pair of nodes u, v, ask to compute if there is a path from u to v (in the case of reachability) and the weight of the shortest path from u to v (in the case of shortest paths). The *single-source* variant given a node u asks to solve the pair problem u, v for every node v. Finally, the *all pairs* variant asks to solve the pair problem for each pair u, v. While there exist many classic algorithms for the distance problem, such as A^*-algorithm (pair) [Hart *et al.*, 1968], Dijkstra's algorithm (single-source) [Dijkstra, 1959], Bellman-Ford algorithm (single-source) [Bellman, 1958; Ford, 1956; Moore, 1959], Floyd-Warshall algorithm (all pairs) [Floyd, 1962; Warshall, 1962; Roy, 1959], and Johnson's algorithm (all pairs) [Johnson, 1977] and others for various special cases, there exist in essence only two different algorithmic ideas for reachability: Fast matrix multiplication (all pairs) [Fischer and Meyer, 1971] and DFS/BFS (single-source) [Cormen *et al.*, 2009].

Mean payoff, ratio and initial credit for energy problems. Although the mean-payoff, ratio and initial credit for energy properties cannot be phrased as algebraic path problems, they are closely related to the distance problem. The minimum mean-payoff problem on graphs is known as the minimum mean cycle problem of [Karp, 1978]. Informally, the input to the problem is a graph and a weight function that assigns integer weights to the edges of the graph, and the task is to compute an infinite path of the graph with the smallest average weight. The problem has large modeling power, and has been studied extensively [Madani, 2002; Lawler, 1976; Young *et al.*, 1991; Burns, 1991; Orlin and Ahuja, 1992; Dasdan and Gupta, 1998; Hartmann and Orlin, 1993; Karp and Orlin, 1981]. For some special cases there exist faster approximation algorithms [Chatterjee *et al.*, 2014a]. We refer to [Dasdan *et al.*, 1998] for an excellent exposition to various algorithms for the minimum mean-payoff problem. The mean-payoff problem is one of the most well-studied objectives in quantitative games [Liggett and Lippman, 1969; Ehrenfeucht and Mycielski, 1979; Zwick and Paterson, 1996; Bjorklund *et al.*, 2004; Brim *et al.*,

2011], and the graph problem can be viewed as the one-player version. Recently, both the one-player and two-player versions of the mean payoff problem were studied on infinite-state systems and multiple dimensions [Chatterjee and Velner, 2012; Velner, 2012; Chatterjee and Velner, 2013; Velner, 2014; Velner, 2015; Velner *et al.*, 2015b].

The ratio problem generalizes the mean payoff problem in the following way. Instead of a single weight function we are now given two weight functions; the first assigning integer weights and the second assigning positive integer weights to the edges of the graph. The task is to compute an infinite path of the graph for which the ratio of the sum of weights of the first function over the sum of weights of the second function is the smallest possible. The ratio problem has bee studied thoroughly both by the graph-theory and the systems-verification community [Lawler, 1976; Burns, 1991; Gerez *et al.*, 1992; Hartmann and Orlin, 1993; Ito and Parhi, 1995; Mathur *et al.*, 1998; Cochet-terrasson *et al.*, 1998; Chatterjee and Velner, 2012], and one standard way to solve it is by reducing it to the mean-payoff problem. Typically, in practice for both the ratio and mean-payoff problem Howard's policy iteration algorithm [Howard, 1960; Cochet-terrasson *et al.*, 1998] is the fastest and uses $O(m)$ time per iteration. The number of iterations needed for the policy iteration algorithm is not known to be polynomial though.

The minimum initial credit for energy problem takes as input a graph and a weight function which assigns an integer weight to each edge. The task is to determine for each node smallest initial energy value such that there is an infinite path starting from that node and so that the sum of the weights along that path added to the initial energy stays non-negative. The problem has been studied in [Bouyer *et al.*, 2008], in the context of weighted timed automata with additional energy constraints. Energy constraints have also been widely studied in conjunction with mean-payoff objectives, as, for example, in [Brim *et al.*, 2011; Chatterjee *et al.*, 2012; Velner *et al.*, 2015b].

1.1.3 Graphs of Constant Treewidth

The significance of constant treewidth. The notion of treewidth of graphs as an elegant mathematical tool to analyze graphs was introduced in [Robertson and Seymour, 1984]. Informally, the treewidth of a graph measures how close the graph is to a tree (a graph has treewidth 1 precisely if it is a tree). The significance of constant treewidth in graph theory is huge, mainly because several

problems on graphs become complexity-wise easier. Given a tree decomposition of a graph with treewidth t, many NP-complete problems for arbitrary graphs can be solved in time polynomial (even linear) in the size of the graph, but exponential in t [Arnborg and Proskurowski, 1989; Bern *et al.*, 1987; Bodlaender, 1988; Bodlaender, 1993; Bodlaender, 2005]. Even for problems that can be solved in polynomial time, faster algorithms can be obtained for low treewidth graphs, for example, for the distance problem [Chaudhuri and Zaroliagis, 1995]. The constant-treewidth property of graphs has also been used in the context of logic: Monadic Second Order (MSO) logic is a very expressive logic, and a celebrated result of [Courcelle, 1990] showed that for constant-treewidth graphs the decision questions for MSO can be solved in polynomial time; a result extended to deterministic logspace in [Elberfeld *et al.*, 2010]. Dynamic algorithms for the special case of graphs with treewidth 2 have been considered in [Bodlaender, 1994] and extended to various tradeoffs by [Hagerup, 2000]; and [Lacki, 2013] shows how to maintain the strongly connected component decomposition under edge deletions for constant-treewidth graphs. Various other models (such as probabilistic models of Markov decision processes and games played on graphs for synthesis) with the constant-treewidth restriction have also been considered [Chatterjee and Lacki, 2013; Obdrzálek, 2003]. The impact of treewidth in the complexity of various verification problems has been studied in [Ferrara *et al.*, 2005; Madhusudan and Parlato, 2011],

Computing tree decompositions. The problem of computing tree decompositions of constant-treewidth graphs has been studied extensively [Reed, 1992; Robertson and Seymour, 1995; Bodlaender and Hagerup, 1995; Bodlaender, 1996; Feige *et al.*, 2005; Bodlaender *et al.*, 2013]. More importantly, in the context of programming languages, it was shown in [Thorup, 1998] that the control-flow graphs of goto-free programs in many programming languages have constant treewidth. This theoretical result was subsequently extended to more programming languages [Gustedt *et al.*, 2002; Burgstaller *et al.*, 2004]. Even in cases where no small bound on the treewidth of control-flow graphs exists, in practice the treewidth remains small. For example, it was shown in [Gustedt *et al.*, 2002] that though in theory Java programs might not have constant treewidth, in practice Java programs have small treewidth. The small treewidth of control-flow graphs has lead to improvements on the problem of register allocation [Thorup, 1998; Bodlaender *et al.*, 1998; Krause, 2013; Krause, 2014]. For example, the SDCC compilers implements tree-decomposition based algorithms for performance optimizations [Krause, 2013].

We refer to [Bodlaender, 1993; Kloks, 1994; Bodlaender, 1998; Bodlaender, 2005] for detailed expositions to the notion of treewidth and its applications.

1.1.4 Partial-order Reduction Techniques in Stateless Model Checking of Concurrent Programs

Stateless model checking of concurrent programs. The verification of concurrent programs is one of the major challenges in formal methods. Due to the combinatorial explosion on the number of interleavings, errors found by testing are hard to reproduce (often called *Heisenbugs* [Musuvathi *et al.*, 2008]), and the problem needs to be addressed by a systematic exploration of the state space. *Model checking* [Clarke *et al.*, 1999a] addresses this issue, however, since model checkers store a large number of global states, stateless model checking cannot be applied to realistic programs. One solution that is adopted is *stateless model checking* [Godefroid, 1996], which avoids the above problem by exploring the state space without explicitly storing the global states. This is typically achieved by a scheduler, which drives the program execution based on the current interaction between the processes. Well-known tools such as VeriSoft [Godefroid, 1997; Godefroid, 2005] and CHESS [Madan Musuvathi, 2007] have successfully employed stateless model checking.

Partial-Order Reduction (POR). Even though stateless model-checking addresses the global state space issue, it still suffers from the combinatorial explosion of the number of interleavings, which grows exponentially. While there are many approaches to reduce the number of explored interleavings, such as, depth-bounding and context bounding [Lal and Reps, 2009; Musuvathi and Qadeer, 2007], the most well-known method is *partial order reduction (POR)* [Clarke *et al.*, 1999b; Godefroid, 1996; Peled, 1993]. The principle of POR is that two interleavings can be regarded as equivalent if one can be obtained from the other by swapping adjacent, non-conflicting (independent) execution steps. The theoretical foundation of POR is the equivalence class of traces induced by the *Mazurkiewicz trace equivalence* [Mazurkiewicz, 1987], and POR explores at least one trace from each equivalence class. POR provides a full coverage of all behaviors that can occur in any interleaving, even though it explores only a subset of traces. Moreover, POR is sufficient for checking most of the interesting verification properties such as safety properties, race freedom, absence of global deadlocks, and absence of assertion

violations [Godefroid, 1996].

Dynamic Partial-order Reduction (DPOR). Dynamic partial-order reduction (DPOR) [Flanagan and Godefroid, 2005] improves the precision of POR by recording actually occurring conflicts during the exploration and using this information on-the-fly. DPOR guarantees the exploration of at least one trace in each Mazurkiewicz equivalence class when the explored state space is acyclic and finite, which holds for stateless model checking, as usually the length of executions is bounded [Flanagan and Godefroid, 2005; Godefroid, 2005; Musuvathi *et al.*, 2008]. Recently, an optimal method for DPOR was developed [Abdulla *et al.*, 2014].

1.1.5 Real-time Scheduling

Competitive analysis of real-time schedulers. Competitive analysis [Borodin and El-Yaniv, 1998] has been the primary tool for studying the performance of such scheduling algorithms [Baruah *et al.*, 1992]. It allows to compare the performance of an *on-line* algorithm $\mathcal{A}$, which processes a sequence of inputs without knowing the future, with what can be achieved by an optimal *off-line* algorithm $\mathcal{C}$ that does know the future (a *clairvoyant* algorithm): the *competitive factor* gives the worst-case performance ratio of $\mathcal{A}$ vs. $\mathcal{C}$ over all possible scenarios.

In a seminal paper [Baruah *et al.*, 1992], Baruah et al. proved that no on-line scheduling algorithm for single processors can achieve a competitive factor better than $1/4$ over a clairvoyant algorithm in *all* possible job sequences of *all* possible tasksets. The proof is based on constructing a specific job sequence, which takes into account the on-line algorithm's actions and thereby forces any such algorithm to deliver a sub-optimal cumulated utility. For the special case of zero-laxity tasksets of uniform value-density, where utilities equal execution times, they also provided the on-line algorithm TD1 with competitive factor $1/4$, concluding that $1/4$ is a tight bound for this family of tasksets. In [Baruah *et al.*, 1992], the $1/4$ upper bound was also generalized, by showing that there exist tasksets with *importance ratio* k, defined as the ratio of the maximum over the minimum value-density in the taskset, in which no on-line scheduler can have competitive factor larger than $\frac{1}{(1+\sqrt{k})^2}$. In subsequent work [Koren and Shasha, 1995], the on-line scheduler D^{over} was introduced, which provides the performance guarantee of $\frac{1}{(1+\sqrt{k})^2}$ in any taskset with importance ratio k, showing that this bound is also tight.

Firm-deadline tasks Firm-deadline tasks arise in various application domains, e.g., machine scheduling [Gupta and Palis, 2001], multimedia and video streaming [Abeni and Buttazzo, 1998], QoS management in bounded-delay data network switches [Englert and Westermann, 2007] and even networks-on-chip [Lu and Jantsch, 2007], Starting out from [Baruah *et al.*, 1992], classic real-time systems research has studied the competitive factor of both simple and extended real-time scheduling algorithms. The competitive analysis of simple algorithms was been extended in various ways later on: Energy consumption [Aydin *et al.*, 2004; Devadas *et al.*, 2010] (including dynamic voltage scaling), imprecise computation tasks (having both a mandatory and an optional part and associated utilities) [Baruah and Hickey, 1998], lower bounds on slack time [Baruah and Haritsa, 1997], and fairness [Palis, 2004]. Note that dealing with these extensions involved considerable ingenuity and efforts w.r.t. identifying and analyzing appropriate worst case scenarios, which do not necessarily carry over even to minor variants of the problem. Maximizing cumulated utility while satisfying multiple resource constraints is also the purpose of the Q-RAM (QoS-based Resource Allocation Model) [Rajkumar *et al.*, 1997] approach.

Algorithmic game theory in scheduling. Algorithmic game theory [Nisan *et al.*, 2007] has been applied to classic scheduling problems since decades, primarily in economics and operations research, see e.g. [Koutsoupias, 2011] for just one example of some more recent work. It has also been applied for real-time scheduling of *hard* real-time tasks in the past: Besides Altisen et al. [Altisen *et al.*, 2002], who used games for synthesizing controllers dedicated to meeting all deadlines, Bonifaci and Marchetti-Spaccamela [Bonifaci and Marchetti-Spaccamela, 2012] employed graph games for automatic feasibility analysis of sporadic real-time tasks in multiprocessor systems: Given a set of sporadic tasks (where consecutive releases of jobs of the same task are separated at least by some sporadicity interval), the algorithms provided in [Bonifaci and Marchetti-Spaccamela, 2012] allow to decide, in polynomial time, whether some given scheduling algorithm will meet *all* deadlines. A partial-information game variant of their approach also allows to synthesize an optimal scheduling algorithm for a given task set (albeit not in polynomial time).

Competitive analysis in fixed environments. Although the competitive factor characterizes the worst-case performance of an online scheduler in the *worst-case setting*, it is hardly informative of the performance of the algorithm in a *fixed setting*. Since the taskset arising in a particular

application is usually known, a relevant problem is to determine the competitiveness of scheduling algorithms in *fixed* tasksets. The *competitive ratio* of an online scheduler $\mathcal{A}$ in a taskset $\mathcal{T}$ is informally defined as the smallest long-run ratio of the cumulative utility of $\mathcal{A}$ over the cumulative utility received by a clairvoyant scheduler on the same sequence of task releases, where the tasks now come from $\mathcal{T}$. Two relevant problems for the *automated* competitive analysis in fixed tasksets are the following.

(1) The *competitive analysis* question asks to compute the competitive ratio of a given on-line algorithm.

(2) The *competitive synthesis* question asks to construct an on-line algorithm with optimal competitive ratio.

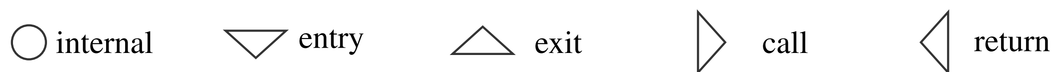
1.2 Motivating Examples

In this section we provide some examples which have motivated the research presented in this dissertation.

1.2.1 Motivating Example: Constant-treewidth in Static Analysis

Consider the program shown in Fig. 1.1 comprising two methods dot_vector and dot_matrix. For simplicity we focus on on-demand local reachability, i.e., given any two nodes of the same control-flow graph, whether there is a path from one to the other. Observe that even on local reachability we have to solve an interprocedural problem, as a path between nodes of the control-flow graph of dot_matrix might go through nodes of the control-flow graph of dot_vector. We are interested in handling multiple reachability queries, which need to be answered fast. For this purpose we are allowed a preprocessing phase, in which we can spend some resources in building a data structure that will allow for fast handling of queries.

Preprocess vs Query. Traditional approaches appear in the following two extremes, where we use n as the input size, and the stated complexities ar upper bounds wrt the worst case:



Method: dot_vector

Input: $x, y \in \mathbb{R}^n$

Output: The dot product $x^\top y$

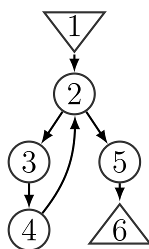
```
1 result ← 0
2 for i ← 1 to n do
3     z ← x[i] · y[i]
4     result ← result + z
5 end
6 return result
```

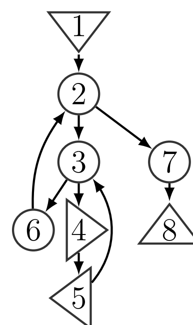
Method: dot_matrix

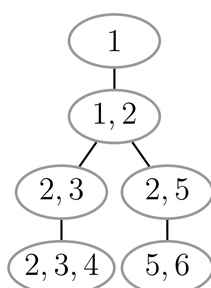
Input: $A \in \mathbb{R}^{n \times k}, B \in \mathbb{R}^{k \times m}$

Output: The dot product $A \times B$

```
1 C ← zero matrix of size n × m
2 for i ← 1 to n do
3     for j ← 1 to m do
4         Call dot_vector(A[i, :], B[:, j])
5         C[i, j] ← the value returned at Line 4
6     end
7 end
8 return C
```







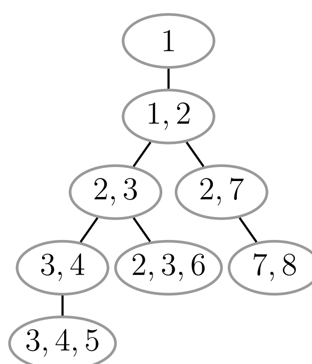


Figure 1.1: Example of a program consisting of two methods, their control-flow graphs $G_i = (V_i, E_i')$ where nodes correspond to line numbers, and the corresponding tree decompositions.

Complete preprocessing, which requires $O(n^2)$ time and space and after which queries are answered in $O(1)$ time, and

No preprocessing, which answers each query in time $O(n)$. Although in this case smarter

approaches exist, such as storing intermediate results of every arriving query, they suffer in two respects. In terms of space, the memory requirements grow as we store each intermediate result. In terms of time, the only given guarantee is that of *same worst-case complexity*, namely that the total time for handling any sequence of queries is no worse than the complete preprocessing.

In this dissertation we provide new data structures for handling algebraic path queries efficiently, by exploiting the constant-treewidth property of control-flow graphs. Our focus is on *same-context* queries, where the two nodes reside in the same function. Note that same-context queries require interprocedural analysis, as although the endpoints of paths are in the same function, the paths are allowed to traverse multiple functions. For example, our new data structures operate on the tree decompositions of the control-flow graphs, which can be preprocessed in $O(n \cdot \log n)$ time and using only $O(n)$ space, after which pair queries are handled in $O(1)$ time.

Preprocess of libraries. Assume that `dot_matrix` is part of some library, and different implementations of `dot_vector` are passed as a callback function to the library each time the library is linked to a main program. More generally, let n_1 and n_2 be the size of the library and the main program, respectively, with $n_2 >> n_1$, and b the number of callback locations. Typically we have $b << n_2$, i.e., the number of callback locations is much smaller than the size of the library. Traditionally, the whole program needs to be analyzed anew with each linking, leading to $O(n_1 \cdot n_2)$ time for the transitive closure each time, by running $O(n_1)$ single-source computations from the locations of the main program. At this point pair reachability queries on the main program can be handled in $O(1)$ time. In contrast, our new data structures can preprocess the library once in $O(n_2)$ time. Each time a new main program links to the library, at the cost of $O(n_1 \cdot \log n_1 + b \cdot \log n_2)$, pair reachability queries can be handled in $O(1)$ time. Hence the full cost of analyzing a huge library is only paid once, and each linking cost has a small dependency on the size of the library.

1.2.2 Motivating Example: Quantitative Interprocedural Analysis

Consider the program shown in Fig. 1.2. Our focus is on the containers that are allocated in Line 9 and in Line 20. We call a container *underutilized* if in every infinite interprocedural path of the corresponding RSM the container holds a bounded number of elements. The intended

```
 1 void qux( Queue q, int x ){
 2     q.push((x, x/2));
 3     if( x > 0 ){
 4         qux( q, x/2 );
 5     }
 6 }
 7
 8 Queue bar( int x ){
 9     return new Queue(x*x);
10 }
```

```
12 void foo( int x ){
13     if( x % 2){
14         Queue q1 = bar(x);
15         qux(q1, x);
16     }
17     else
18     {
19         for( int y = 0 ; y < x ; y++ ){
20             Queue q2 = new Queue(y);
21             q2.push( (y,x) );
22             for( int z = 0 ; z < y ; z++ )
23             {
24                 q2.push((z,y));
25                 ...
26                 q2.pop();
27             }
28         }
29     }
30 }
```

Figure 1.2: Underutilized container analysis.

interpretation is that if the container holds only a few elements in every path, it might not be worthy to pay the large cost in resources for initializing the container, and some other data structure can be used instead.

Our task is to analyze the program for containers that may be underutilized. The problem can be cast in our Quantitative Interprocedural Analysis framework, as follows. We assign a weight of +1 to every transition of the RSM that adds an element to the analyzed container (i.e., the container performs a push() operation). Similarly, we assign a weight of -1 to every transition of the RSM that removes an element from the analyzed container (i.e., the container performs a pop() operation). Additionally, we assign a large finite negative weight to the transition that corresponds to the initialization of the container. This is because a reinitialization of an existing container removes all its elements. Hence, in order to not report the container as possibly underutilized, we need to ignore all elements added to the container up to the reinitialization point, and find a new path from that point on in the RSM that adds an unbounded number of elements. In the end, the algorithmic problem that we need to solve is to detect whether there exists an infinite path of the RSM in which the long run ratio of the positive weights over the negative weights is larger than 0. Note that the proper use of the container might be because of an intraprocedural loop, or because of an interprocedural loop (i.e., due to recursion). Hence the problem we need to solve is an interprocedural mean-payoff problem.

In our example, there exist runs that go through line 14 and properly use the container that is allocated in line 9, since the qux method can add unbounded number of elements to the queue (due to its recursive call). However, the container in line 20 is underutilized, since in every run the number of elements is bounded by 2.

In this dissertation we develop a new framework that allows to capture quantitative properties on recursive programs, called the Quantitative Interprocedural Analysis framework. We provide a generic algorithm for performing Quantitative Interprocedural Analysis, and show how various relevant static program-analysis problems can be expressed as Quantitative Interprocedural Analysis instances.

Process p_1 :	Process p_2 :
1. write x;	1. write x;
2. read x;	2. read x;

Figure 1.3: A system of two processes with two events each.

1.2.3 Motivating Example: Data-centric Partial-order Reduction

Consider a concurrent system that consists of two processes and a single global variable x shown in Fig. 1.3. Denote by w_i and r_i the write and read events to x by process p_i, respectively. The system consists of four events which are all pairwise dependent, except for the pair r_1, r_2. Two traces t and t' are called Mazurkiewicz equivalent, denoted $t \sim_M t'$, if they agree on the order of dependent events. The traditional DPOR based on the Mazurkiewicz equivalence $\sim_M$ will explore at least one representative trace from every class induced on the trace space by the Mazurkiewicz equivalence. There exist $\frac{2^3}{2} = 4$ possible orderings of dependent events, as there are 2^3 possible interleavings, but half of those reorder the independent events r_1, r_2, and thus will not be considered. The traditional Mazurkiewicz-based DPOR will explore the following four traces.

$$t_1 : \; w_1, r_1, w_2, r_2 \qquad\qquad t_2 : \; w_1, w_2, r_1, r_2$$

$$t_3 : \; w_2, w_1, r_1, r_2 \qquad\qquad t_4 : \; w_2, r_2, w_1, r_1$$

Note however that t_1 and t_4 are state-equivalent, in the sense that the local states visited by p_1 and p_2 are identical in the two traces. This is because each read event *observes* the same write event in t_1 and t_4. In contrast, in every pair of traces among t_1, t_2, t_3, there is at least one read event that observes a different write event in that pair. This observation makes it natural to consider two traces equivalent if they contain the same read events, and every read event observes the same write event in both traces. This example illustrates that it is possible to have coarser equivalence on traces than the traditional Mazurkiewicz equivalence. The challenge then is to design an enumerative exploration of the trace space that

1. is optimal wrt the new coarser equivalence, i.e., it examines precisely one (or a constant constant number of) trace(s) per class, and

2. spends only polynomial time per explored trace.

In this dissertation we develop a new equivalence on traces, called the *observation equivalence*. Informally, two traces are observation equivalent if they contain the same events, and every read event observes the same write event in both traces, under sequential consistency memory semantics. We show that the observation equivalence is always coarser than the traditional Mazurkiewicz equivalence, and can be even exponentially coarser. Based on the observation equivalence, we develop a new enumerative exploration of the trace space, called the *data-centric, partial-order reduction (*DC-DPOR*)*.

1. For acyclic architectures, our algorithm is guaranteed to explore *exactly* one representative trace from each observation class, while spending polynomial time per class. Hence, our algorithm is *optimal* wrt the observation equivalence, and in several cases explores exponentially fewer traces than *any* enumerative method based on the Mazurkiewicz equivalence.

2. For cyclic architectures, we consider an equivalence between traces which is finer than the observation equivalence; but coarser than the Mazurkiewicz equivalence, and in cases is exponentially coarser. Our DC-DPOR algorithm remains optimal under this trace equivalence.

1.2.4 Motivating Example: Competitive Analysis of Real-time Schedulers

We illustrate the need for automated, case-specific competitive analysis of real-time schedulers, by focusing on two well-known scheduling policies, First-in First-out (FIFO) and Earliest Deadline First (EDF), and showing that none dominates the other in the firm-deadline setting. In this setting time proceeds in discrete steps, and in each time point some new task instances are released to the system and require to be scheduled for some time. Each task is defined by three positive integer values $\tau_i = (C_i, D_i, V_i)$, where

C_i denotes the total workload that τ_i generates,

D_i denotes the relative deadline of τ_i, and

V_i denotes the utility of τ_i.

When an instance of τ_i is released, the scheduler receives utility V_i if it schedules that instance for C_i time units (not necessarily consecutively) within D_i time units. We additionally restrict every task to release at most once instance per time unit. We will denote by $\mathcal{CR}_{\mathcal{T}}(\mathcal{A})$ the competitive ratio of scheduler $\mathcal{A}$ in taskset $\mathcal{T}$, defined as the smallest long-run ratio of the cumulative utility of $\mathcal{A}$ over the cumulative utility received by a clairvoyant scheduler on the same sequence of task releases. To simplify our analysis we make the following assumptions.

1. We have an infinite sequence of task releases i.e., new tasks are released infinitely often, though we do not restrict the frequency of releases.

2. If at any time the workload of a task is more than its relative deadline, neither policy will schedule it (as it is bound to not complete in time).

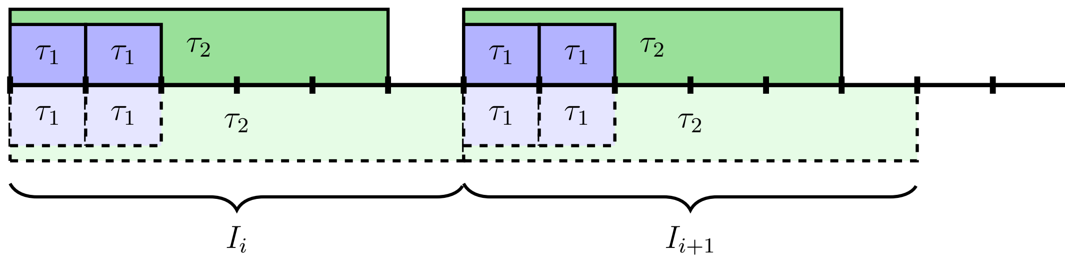
3. If EDF has scheduled at least one unit of workload of some instance of a task, that instance has priority over any other instance (of any task) with the same deadline.

It is easy to see that due to Item 2 and Item 3, any task which is scheduled by either policy for at least one unit of time is in fact scheduled to completion and contributes to the received utility. We will construct two simple tasksets $\mathcal{T}_1, \mathcal{T}_2$, such that FIFO has better competitive ratio than EDF in $\mathcal{T}_1$, and EDF has better competitive ratio than FIFO in $\mathcal{T}_2$. Our analysis is rather crude, but sufficient for the purposes of this example.

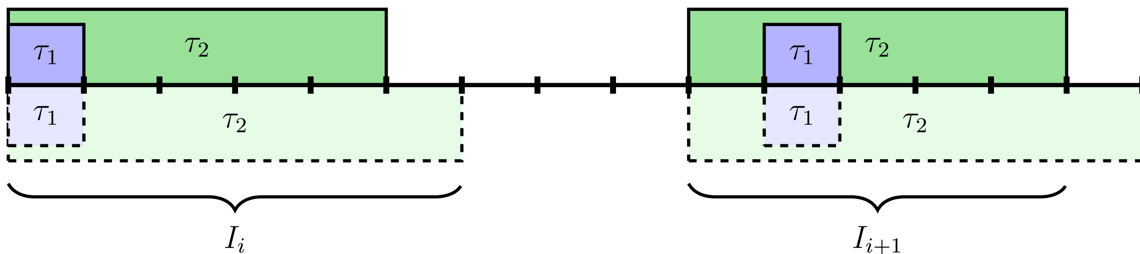
Taskset $\mathcal{T}_1$: FIFO beats EDF. Consider the taskset $\mathcal{T}_1 = \{\tau_1 = (n, n+1, n), \tau_2 = (1, 1, 1)\}$. Here we have a long task τ_1 with laxity 1 and a short, zero-laxity task τ_2.

$\mathcal{CR}_{\mathcal{T}_1}(\mathsf{EDF}) \leqslant \frac{2}{n+1}$**.** Indeed, consider an infinite periodic sequence X which consists of consecutive, non-overlapping intervals $(I_i)_{i\geqslant 1}$ of length $n+1$. The task τ_1 is released once, in the beginning of I_i. The task τ_2 is released twice, one time each in the first two time units of I_i. See Fig. 1.4a for an illustration. Then EDF will schedule to completion the two instances of τ_2, but will miss τ_1, and thus will receive a utility of 2 in I. Instead, a clairvoyant scheduler receives utility of at least $n+1$ in I, by scheduling to completion τ_1 and one instance of τ_2. Hence $\mathcal{CR}_{\mathcal{T}_1}(\mathsf{EDF}) \leqslant \frac{2}{n+1}$.

$\mathcal{CR}_{\mathcal{T}_1}(\mathsf{FIFO}) \geqslant \frac{1}{2}$**.** Let X be any sequence of task releases. We define a sequence of intervals $(I_i)_{i\geqslant 1}$ of length either n or $n+1$ each as follows. I_1 starts the first time the task τ_1 is released in X, and for all $i \geqslant 1$,



(a) The intervals $(I_i)_{i \geqslant 1}$ in the case of EDF. In each I_i EDF misses each instance of τ_2 and thus receives utility at most 2. In contrast a clairvoyant scheduler can receive utility $n+1$ in I_i.



(b) The intervals $(I_i)_{i \geqslant 1}$ in the case of FIFO. In each I_i FIFO will schedule at least the instance of τ_2 to completion regardless of releases of τ_1 instances. Hence FIFO will receive utility at least n by completing tasks released in I_i.

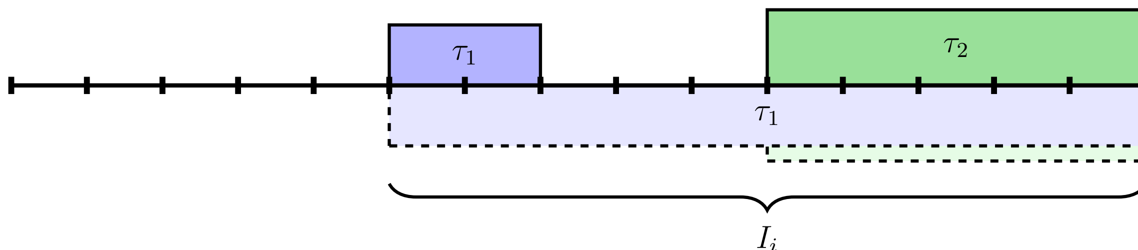
Figure 1.4: Task sequences from taskset $\mathcal{T}_1$. Solid lines represent workload. Dashed lines represent deadlines.

- if τ_2 is released in the beginning of I_i and scheduled by FIFO, I_i lasts $n+1$ time units, otherwise
- I_i lasts n time units.

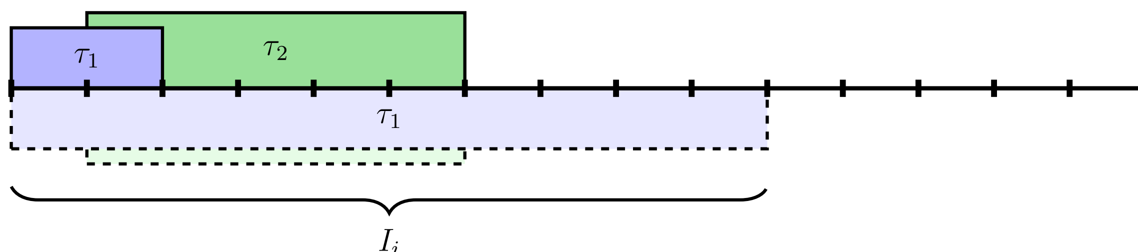
The interval I_{i+1} starts the first time τ_1 is released after the end of I_i. See Fig. 1.4b for an illustration. We first measure the utility that FIFO and any clairvoyant scheduler obtain by completing tasks released in any I_i. Note that at the beginning of each I_i, every released task has been either completed or discarded by FIFO, as FIFO will schedule to completion exactly one instance of τ_1 in I_i (the one that defines the beginning of I_i) and discard all other instances of τ_1. Then FIFO obtains utility at least n, as it schedules τ_1 either right away in I_i, or possibly with one unit time delay, if τ_2 is also released in the beginning of I_i. On the other hand, any clairvoyant scheduler receives utility at most $2 \cdot n$ for tasks released in I_i. Indeed, the clairvoyant scheduler can complete at most two instances of τ_1 released in I_i, in which case it cannot complete any instance of τ_2 released in I_i, and hence it obtains utility $2 \cdot n$. It is not hard to see that in any other case, it obtains utility less than $2 \cdot n$. Hence the ratio of the utility of FIFO over any clairvoyant scheduler in each I_i is at least $\frac{1}{2}$. Finally, we claim that FIFO and any clairvoyant scheduler obtain the same utility by completing tasks released outside the interval $Y = \bigcup_i I_i$. Indeed, by definition τ_1 is not released outside Y, and FIFO does not schedule workload of τ_1 outside Y. Hence, any of the instances of τ_2 released outside Y is scheduled by FIFO. We thus conclude that $\mathcal{CR}_{\mathcal{T}_1}(\mathsf{FIFO}) \geqslant \frac{1}{2}$.

Taskset $\mathcal{T}_2$: EDF beats FIFO. Consider the taskset $\mathcal{T}_2 = \{\tau_1 = (n, n, n), \tau_2 = (2, 2 \cdot n, 2)\}$. Here we have a long, zero laxity task τ_1 and a short task τ_2 with laxity $2 \cdot n - 2$.

$\mathcal{CR}_{\mathcal{T}_2}(\mathsf{EDF}) \geqslant \frac{1}{c}$ **(for fixed c).** Let X be any sequence of task releases. If X contains only finitely many releases of τ_1, then $\mathcal{CR}_{\mathcal{T}_2}(\mathsf{EDF}) = 1$. To see this, first note that we can focus only on the releases of τ_2, since τ_1 contributes only finite utility. Then, observe that any set of instances of τ_2 scheduled to completion by any clairvoyant algorithm can be scheduled non preemptively and in an earliest-deadline first policy. Finally, let x be the first instance of τ_2 scheduled by the clairvoyant scheduler and not by EDF. Then there must be an instance y with deadline earlier than x, such that EDF schedules x but the clairvoyant scheduler does not. Observe that swapping x and y in the schedule of the clairvoyant



(a) The intervals $(I_i)_{i \geqslant 1}$. In each I_i, EDF will either schedule some instance of τ_2 or not. If not, when the last instance of τ_2 is released, EDF must schedule an instance of τ_1 with earlier deadline. That instance was released at least n rounds ago, and since it was not completed, EDF was busy with scheduling some other workload during the past n rounds. In all cases EDF will receive utility at least n by completing tasks with deadline within I_i.



(b) The intervals $(I_i)_{i \geqslant 1}$ in the case of FIFO. In each I_i FIFO misses each instance of τ_2 and thus receives utility at most 2. In contrast a clairvoyant scheduler can receive utility $n + 2$ in I_i.

Figure 1.5: Task sequences from taskset $\mathcal{T}_2$. Solid lines represent workload. Dashed lines represent deadlines.

scheduler, i.e., dropping y and scheduling x does not modify the utility obtained by the clairvoyant scheduler. Hence, up to the release of instance x, EDF and the clairvoyant scheduler have identical behavior and thus obtain the same utility. The claim then follows by an easy induction of such utility-preserving task swaps.

We now consider the case where there are infinitely many releases of τ_1. We define a sequence of intervals $(I_i)_{i\geqslant 1}$ of length $2 \cdot n$ each (except possibly for I_1 which might have smaller length) as follows. I_1 ends at the deadline of the the first released instance of τ_1, and interval I_{i+1} ends at the deadline of the first instance of τ_1 which is released at least n time units after the end of I_i. See Fig. 1.5a for an illustration. We first measure the utility that EDF and any clairvoyant scheduler obtain by completing tasks whose deadline ends in any of I_i. Let x_i be the instance of τ_1 that defines I_i. EDF obtains utility at least n, as it either schedules x_i to completion, in which case it obtains utility at least n, or it does not. In the second case there is an instance of a task with deadline before the end of I_i, which prevents EDF from scheduling x_i. If there is such an instance of τ_1, it is scheduled to completion and EDF obtains utility at least n in I_i. Otherwise there is an instance y of τ_2 released at least n times before x_i, and y_i has not been completed when x_i is released. This means that EDF is scheduling some instance in the whole interval between the release of y_i and the release of x_i, and each such instance has a deadline within I_i (otherwise EDF would have completed y_i before x_i is released). Since EDF schedules every task to completion and the tasks have uniform value density (equal to 1), it will obtain utility at least n. On the other hand, any clairvoyant scheduler can obtain utility at most $4 \cdot n$, by completing every task instance that has deadline in I_i, and there are at most $4 \cdot n$ units of workload belonging to such tasks. Hence the ratio of the utility of EDF over any clairvoyant scheduler in each I_i is at least $\frac{1}{4}$.

Let $Y = \bigcup_i I_i$, and we turn our attention to the utility of EDF and any clairvoyant scheduler obtained by completing tasks with deadlines outside Y. Focus on any contiguous interval Z_i between I_i and I_{i+1}. Observe that by definition, there are no more than n releases of τ_1 in Z_i. Hence, after time point $2 \cdot n$ in Z_i there exist only instances of τ_2 in the system, and the argument which supports the infinite releases of τ_1 can be used to show that EDF will complete at least as many instances of τ_2 as the clairvoyant scheduler does after that point. It follows that in the worst case, the number of instances with deadline in Z_i completed

by the clairvoyant scheduler is bounded by a constant multiple of n, so that the utility received is bounded by $c' \cdot n$, for some c'. Assuming that (in the worst case) EDF obtains no utility in Z_i, the competitive ratio is $\mathcal{C}_{\tau_2}(\mathsf{EDF}) \geqslant \frac{1}{c}$, where $c = c' + 4$.

$\mathcal{CR}_{\mathcal{T}_2}(\mathsf{FIFO}) \leqslant \frac{2}{n+2}$. Indeed, consider an infinite periodic sequence X which consists of consecutive, non-overlapping intervals $(I_i)_{i\geqslant 1}$ of length $2 \cdot n$. In the beginning of I_i we have a release of τ_2, and in the next time unit a release of τ_1. See Fig. 1.5b for an illustration. FIFO will schedule to completion τ_2 but will miss τ_1, and hence will obtain utility 2. Instead, a clairvoyant scheduler can obtain utility $n + 2$ by first scheduling τ_2, then preempting τ_2 to schedule τ_1 to completion, and then completing with the second workload of τ_2. Hence $\mathcal{CR}_{\mathcal{T}_2}(\mathsf{FIFO}) \leqslant \frac{2}{n+2}$.

Conclusion. Note that as $n \to \infty$ we have

$$\lim_{n\to\infty} \frac{\mathcal{C}_{\mathcal{T}_1}(\mathsf{FIFO})}{\mathcal{C}_{\mathcal{T}_1}(\mathsf{EDF})} = \infty \qquad \text{and} \qquad \lim_{n\to\infty} \frac{\mathcal{C}_{\mathcal{T}_2}(\mathsf{EDF})}{\mathcal{C}_{\mathcal{T}_2}(\mathsf{FIFO})} = \infty$$

and thus FIFO is "arbitrarily more competitive" than EDF in $\mathcal{T}_1$, and EDF is "arbitrarily more competitive" than FIFO in $\mathcal{T}_2$. Observe that this is insensitive to how the two scheduling policies break ties. Additionally, all tasks have the same value density (i.e., the fraction $\frac{V_i}{C_i}$ which captures the average utility per unit workload is the same in all tasks τ_i) and thus our conclusion also holds in settings where the utility of each task equals the workload of the task.

The above exposition serves as a simple illustration of the need for automated, case-specific competitive analysis to determine the optimal scheduler in a given setting. Going one step further, one might look for a synthesis approach that automatically constructs the optimal scheduler given a taskset.

In this dissertation we develop a framework for the automated *competitive analysis* and *competitive synthesis* of real-time scheduling algorithms.

1. In the case of competitive analysis, the framework takes as input a fixed taskset, a finite-state real-time scheduling algorithm and optional constraints on the released tasks, expressed using finite-state automata. The framework automatically computes the competitive ratio of the scheduling algorithm in the given setting.

2. In the case of competitive synthesis, the framework takes as input a fixed taskset. The output is a real-time scheduling algorithm that is optimal for the given taskset, i.e., it

achieves the highest competitive ratio among all real-time scheduling algorithms, in the given taskset.

1.3 Outline

The rest of this document is organized as follows.

In Chapter 2 we introduce the main formalism, which is that of finite graphs and recursive (infinite) graphs (or RSMs), whose transitions are annotated with weights coming from the domain of a closed semiring. We also introduce the notion of tree decompositions and treewidth, and present several existing and new results regarding computing tree decompositions, which prove helpful in later chapters. Our central result is a strong notion of balanced tree decompositions, together with an efficient algorithm for constructing such tree decompositions of graphs of constant treewidth. Informally, a strongly balanced tree-decomposition is a binary tree-decomposition in which the number of descendants of each bag is typically approximately half of that of its parent. This is a stronger notion of balancing that the usual one which requires that the tree decomposition simply has height logarithmic on its size.

In Chapter 3 we deal with the algebraic path problem on finite graphs of constant treewidth. To this end, we first develop a dynamic data structure for handling general semiring distances on constant-treewidth graphs. The data structure has a preprocessing phase, after which it can handle efficiently (i) queries on the semiring distance between nodes, and (ii) dynamic updates of edge weights. We also provide data structures for the special case of reachability and distance queries, and obtain important improvements over the general case of algebraic path queries wrt arbitrary semirings.

In Chapter 4 we consider the algebraic path problem on RSMs. We utilize the data structures from Chapter 3 both to provide an efficient solution to the general algebraic path problem and to obtain improvements for special cases of IFDS/IDE, reachability and distance queries.

In Chapter 5 we focus on the problem of Dyck reachability, and its applications to data-

dependence and alias analysis. We develop an algorithm for Dyck reachability on bidirected graphs, and prove that it is optimal. Additionally, we present an approach for context-sensitive data-dependence analysis of libraries, and the problem of creating summaries to speed-up subsequent client analysis. Finally, we prove that Dyck reachability is, in general, Boolean Matrix Multiplication-hard, thereby proving the (conditional) optimality of the existing, cubic-time algorithms for the problem.

In Chapter 6 we present the Quantitative Interprocedural Analysis (QIA) framework. In this framework, the transitions of a recursive program are annotated as good, bad or neutral, and receive a weight which measures the magnitude of their respective effect. The Quantitative Interprocedural Analysis problem asks to determine whether there exists an infinite run of the program where the long-run ratio of the bad weights over the good weights is above a given threshold. We illustrate how several quantitative problems related to static analysis of recursive programs can be instantiated in this framework, and present some case studies to this direction.

In Chapter 7 we consider the algebraic path problem on concurrent systems of constant-treewidth components. Here the weight function assigns weights to global transitions of the concurrent system (as opposed to local weights for each component). We make use of our balancing notion of tree decompositions from Chapter 2 to provide several algorithmic tradeoffs between preprocessing and query times. Our results come with several optimality guarantees, which imply that further complexity improvements over our algorithms are either unlikely, or impossible.

In Chapter 8 we study three classic quantitative verification properties, namely, the minimum mean-payoff, minimum ratio, and minimum initial credit for energy properties. Although these properties cannot be expressed in the algebraic path framework, they are closely related to the shortest path problem. We provide several algorithms for exact and approximate solutions to the minimum mean-payoff and minimum ratio problems on graphs of constant treewidth. We also study the minimum initial credit for energy problem on arbitrary graphs, and obtain a significant improvement over the existing approach. Finally, we present a significant algorithmic improvement for the problem restricted on constant-treewidth graphs.

In Chapter 9 we present a new dynamic partial-order reduction method for stateless model checking of concurrent programs, called the Data-centric Partial Order Reduction (DC-DPOR). Our algorithm is based on a new equivalence between traces, called the observation equivalence. DC-DPOR explores a coarser partitioning of the trace space than any exploration method based on the standard Mazurkiewicz equivalence. Depending on the program, the new partitioning can be even exponentially coarser. Additionally, DC-DPOR spends only polynomial time in each explored class.

In Chapter 10 we study the competitive analysis and competitive synthesis problems on real-time schedulers for firm-deadline tasks. Given a fixed taskset, we first provide algorithms for computing the competitive ratio of any real-time scheduler represented as a labeled transition system, and then establish the computational complexity of synthesizing an optimal real-time scheduler for that taskset (i.e., one that achieves the largest possible competitive ratio).

2 Preliminaries

2.1 Introduction

In this chapter we introduce some notation that will be helpful for the exposition of the ideas in this dissertation. The main formalism is that of finite graphs and recursive (infinite) graphs (or RSMs), whose transitions are annotated with weights coming from the domain of a closed semiring. We also introduce the notion of tree decompositions and treewidth, and present several existing and new results regarding computing tree decompositions. One crucial result in this section concerns the construction of *strongly balanced* tree decompositions. Informally, a strongly balanced tree-decomposition is a binary tree-decomposition in which the number of descendants of each bag is typically approximately half of that of its parent. This is a stronger notion of balancing that the one found usually in the literature, which requires that the tree decomposition simply has height logarithmic on its size. Strongly balanced tree decompositions are of importance in Chapter 7, where we deal with concurrent systems of constant-treewidth components.

Organization. The rest of this chapter is organized as follows.

1. In Section 2.2 we introduce some general notation.
2. In Section 2.3 we introduce notation on graphs and tree decompositions, and present several facts regarding tree decompositions. Additionally, we state the algebraic path problem on graphs.

3. In Section 2.4 we introduce notation on RSMs, and state the algebraic path problem on RSMs.

2.2 General Notation

Notation on sets and sequences. Given a number $r \in \mathbb{N}$, we denote by $[r] = \{1, 2, \dots, r\}$ the natural numbers from 1 to r. Given a set X and a $k \in \mathbb{N}$, we denote by $X^k = \prod_{i=1}^{k} X$, the k times Cartesian product of X. A finite sequence $x_1, \dots x_k$ is denoted for short by $(x_i)_{1 \leqslant i \leqslant k}$ or $(x^i)_{1 \leqslant i \leqslant k}$, and write $(x_i)_i$ or $(x^i)_i$ respectively when k is implied from the context. Similarly, we write $\{x_i\}_{1 \leqslant i \leqslant k}$, or simply $\{x_i\}_i$ then k is implied from the context, to denote the set $\{x_1, \dots x_k\}$. Given a sequence Y and integers $i, j \geqslant 1$ with $i < j$, we denote by $Y[i, j]$ the subsequence of Y from position i to position j (inclusive). Given a sequence Y, we denote by $y \in Y$ the fact that y appears in Y, and given additionally a set X, denote by $Y \cap X$ the set of elements that appear both in Y and X.

Semirings. We fix a *closed semiring* $S = (\Sigma, \oplus, \otimes, \overline{\mathbf{0}}, \overline{\mathbf{1}})$ where Σ is a countable set, $\oplus$ and $\otimes$ are binary operators on Σ, and $\overline{\mathbf{0}}, \overline{\mathbf{1}} \in \Sigma$, and the following properties hold:

1. $\oplus$ is infinitely associative, infinitely commutative, and $\overline{\mathbf{0}}$ is the neutral element,
2. $\otimes$ is associative, and $\overline{\mathbf{1}}$ is the neutral element,
3. $\otimes$ infinitely distributes over $\oplus$,
4. $\overline{\mathbf{0}}$ absorbs in multiplication, i.e., $\forall a \in \Sigma : a \otimes \overline{\mathbf{0}} = \overline{\mathbf{0}}$.

Additionally, we consider that

1. S idempotent, i.e., for every $s \in \Sigma$ we have that $s \oplus s = s$, and
2. S is equipped with a *closure* operator *, such that $\forall s \in \Sigma : \; s^* = \overline{\mathbf{1}} \oplus (s \otimes s^*) = \overline{\mathbf{1}} \oplus (s^* \otimes s)$ (i.e., the semiring is *closed*).

Conventionally, we let $\oplus(\varnothing) = \overline{\mathbf{0}}$ and $\otimes(\varnothing) = \overline{\mathbf{1}}$. The idempotence property defines a partial order $\preceq \subseteq \Sigma \times \Sigma$, such that $\forall s_1, s_2 \in \Sigma$, we have that $s_1 \preceq s_2$ iff $s_1 \oplus s_2 = s_1$.

Computational model. Our model of computation is the standard RAM model with word size $W = \Theta(\log n)$. Several of the results presented here rely on, so called, "word tricks", which perform operations on $\Theta(\log n)$ bits in constant time, by grouping the bits to $O(1)$ machine words. We also use constant-time lowest common ancestor queries, which also make use of word tricks.

2.3 Graphs and Tree Decompositions

2.3.1 Graphs

In the current section we fix a semiring $S = (\Sigma, \oplus, \otimes, \overline{\mathbf{0}}, \overline{\mathbf{1}})$.

Graphs, weighted paths and semiring distances. We denote by $G = (V, E, \mathsf{wt})$ a weighted finite directed graph (henceforth called simply a graph) where V is a set of n nodes and $E \subseteq V \times V$ is an edge relation of m edges, along with a weight function $\mathsf{wt} : E \to \Sigma$ that assigns to each edge of G an element from Σ. In several cases where the weight function is of no interest, we will simply let $G = (V, E)$ be a directed graph. The *undirected variant* of G is an unweighted undirected graph $G' = (V, E')$ such that $(u, v) \in E'$ iff $(u, v) \in E$ or $(v, u) \in E$, i.e., G' is obtained from G by ignoring the direction on the edges. Given a set of nodes $X \subseteq V$, we denote by $G[X] = (X, E \cap (X \times X))$ the subgraph of G induced by X. A path $P : u \rightsquigarrow v$ is a sequence of nodes $(x_1, \dots, x_k)$ such that $x_1 = u$, $x_k = v$, and for all $1 \leqslant i < k$ we have $(x_i, x_{i+1}) \in E$. The length of P is $|P| = k - 1$, and a single node is itself a 0-length path. A path P is *simple* if no node repeats in the path (i.e., the path does not contain a cycle). Given two paths $P_1 = (x_1, \dots x_k)$ and $P_2 = (y_1, y_\ell)$ with $x_k = y_1$, we denote by $P_1 \circ P_2$ the *concatenation* of P_2 on P_1. We use the notation $x \in P$ to denote that a node x appears in P, and $e \in P$ to denote that an edge e appears in P. Given a set $B \subseteq V$, we denote by $P \cap B$ the set of nodes of B that appear in P. Given a path $P = (x_1, \dots, x_k)$, the weight of P is $\otimes(P) = \bigotimes(\mathsf{wt}(x_i, x_{i+1}))_i$ if $|P| \geqslant 1$ else $\otimes(P) = \overline{\mathbf{1}}$. Given nodes $u, v \in V$, the *semiring distance* (or simply, *distance*) $d(u, v)$ is defined as $d(u, v) = \bigoplus_{P : u \rightsquigarrow v} \otimes(P)$, and $d(u, v) = \overline{\mathbf{0}}$ if no such P exists.

Trees. A (rooted) tree $T = (V_T, E_T)$ is an undirected graph with a distinguished node r which is the root such that there is a unique simple path $P_u^v : u \rightsquigarrow v$ for each pair of nodes u, v. We

denote by $|T| = |V_T|$ the number of nodes in T. Given a tree T with root r, the *level* $\mathsf{Lv}(u)$ of a node u is the length of the simple path P_u^r from u to the root r, and every node in P_u^r is an *ancestor* of u. If v is an ancestor of u, then u is a *descendant* of v. Note that a node u is both an ancestor and descendant of itself. For a pair of nodes $u, v \in V_T$, the *lowest common ancestor (LCA)* of u and v is the common ancestor of u and v with the largest level. The *parent* u of v is the unique ancestor of v in level $\mathsf{Lv}(v) - 1$, and v is a *child* of u. A *leaf* of T is a node with no children. For a node $u \in V_T$, we denote by $T(u)$ the subtree of T rooted in u (i.e., the tree consisting of all descendants of u). A tree is called *k-ary* if every node has at most k children (e.g., in a binary tree every node has at most two children). A *full k-ary tree* is a k-ary tree in which every non-leaf node has exactly k children. The *height* of T is $\max_u \mathsf{Lv}(u)$ (i.e., it is the length of the longest path P_u^r). Given a tree T, a *connected component* $\mathcal{C} \subseteq V_T$ of T is a set of nodes of T such that for every pair of nodes $u, v \in \mathcal{C}$, the unique simple path P_u^v in T visits only nodes in $\mathcal{C}$. We call the graph $T[\mathcal{C}]$ a *contiguous subtree* of T.

Balanced trees. A tree $T = (V_T, E_T)$ is called *balanced* if its height is $O(\log |V_T|)$. Given two constants $0 < \beta < 1$ and $\gamma \in \mathbb{N}^+$, we say that a node u of T is *(β, γ)-balanced* if for every descendant v of u with $\mathsf{Lv}(v) - \mathsf{Lv}(u) \geqslant \gamma$, we have that $|T(v)| \leqslant \beta \cdot |T(u)|$. Intuitively, every subtree rooted far below u must contain only a constant fraction of the nodes of $T(u)$. A tree T is called *(β, γ)-balanced* if every node in V_T is (β, γ)-balanced.

The algebraic path problem. Given a closed semiring $S = (\Sigma, \oplus, \otimes, \overline{\mathbf{0}}, \overline{\mathbf{1}})$ and a weighted graph $G = (V, E, \mathsf{wt})$, the *algebraic path problem* for a pair of nodes (u, v) asks for the *semiring distance* from u to v, defined as

$$d(u, v) = \bigoplus_{P: u \rightsquigarrow v} \mathsf{wt}(P)$$

In this dissertation we will deal with various small variations of this problem, such as, for example, the *single-source* version, the *all-pairs* version, as well as *dynamic* versions of the problem where some edge weights might change over time.

Hereinafter, we will use the term *semiring distance* or *distance wrt a semiring* to refer to the general definition of $d(u, v)$ introduced here. To stay consistent with the literature, we will refer to the semiring distance wrt the tropical semiring simply as *distance*.

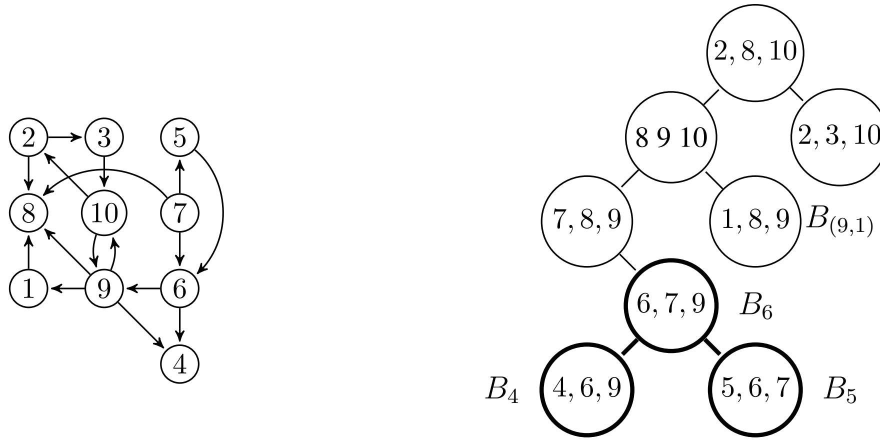


Figure 2.1: A graph G with treewidth 2 (left) and a corresponding tree-decomposition $\mathrm{Tree}(G)$ (right).

2.3.2 Tree Decompositions

Tree decompositions and treewidth [Robertson and Seymour, 1984]. Given a graph G, a tree-decomposition $\mathrm{Tree}(G) = (V_T, E_T)$ is a tree with the following properties.

C1: $V_T = \{B_1, \dots, B_b :$ for all $1 \leqslant i \leqslant b.\ B_i \subseteq V\}$ and $\bigcup_{B_i \in V_T} B_i = V$. That is, each node of $\mathrm{Tree}(G)$ is a subset of nodes of G, and each node of G appears in some node of $\mathrm{Tree}(G)$.

C2: For all $(u, v) \in E$ there exists $B_i \in V_T$ such that $u, v \in B_i$. That is, the endpoints of each edge of G appear together in some node of $\mathrm{Tree}(G)$.

C3: For all B_i, B_j and any bag B_k that appears in the simple path $B_i \rightsquigarrow B_j$ in $\mathrm{Tree}(G)$, we have $B_i \cap B_j \subseteq B_k$. That is, every node of G is contained in a contiguous subtree of $\mathrm{Tree}(G)$.

To distinguish between the nodes of G and the nodes of $\mathrm{Tree}(G)$, the sets B_i are called *bags*. The *width* of a tree-decomposition $\mathrm{Tree}(G)$ is the size of the largest bag minus 1 and the *treewidth* of G is the width of a minimum-width tree decomposition of G. It follows from the definition that if G has constant treewidth, then $m = O(n)$. A graph has treewidth 1 precisely if its undirected variant is a tree.

Notation on tree decompositions. Let G be a graph and $T = \mathrm{Tree}(G)$ a tree decomposition of G. We denote by $\mathsf{Lv}(B_i)$ the level of B_i in $\mathrm{Tree}(G)$. For $u \in V$, we say that a bag B is

the *root bag* of u if B is the bag with the smallest level among all bags that contain u, i.e., $B_u = \arg\min_{B \in V_T:\, u \in B} \mathsf{Lv}\,(B)$. By definition, there is exactly one root bag for each node u. We often write B_u for the root bag of node u, and denote by $\mathsf{Lv}(u) = \mathsf{Lv}\,(B_u)$. Additionally, we denote by $B_{(u,v)}$ the bag of the largest level that is the root bag of one of u, v. Finally, given a bag B, we denote by

- $T(B)$ the subtree of T rooted at B, by
- $V_T(B)$ the nodes of G that appear in the bags of $T(B)$, and by
- $\mathcal{V}_T(B)$ the nodes of G that appear in B and its ancestors in T.

Example 2.1 (Graph and tree decomposition). The treewidth of a graph G is an intuitive measure which represents the proximity of G to a tree, though G itself not a tree. The treewidth of G is 1 precisely if G is itself a tree [Robertson and Seymour, 1984]. Consider an example graph and its tree decomposition shown in Fig. 2.1. It is straightforward to verify that all the three conditions of tree decomposition are met. Each node in the tree is a bag, and labeled by the set of nodes it contains. Since each bag contains at most three nodes, the tree decomposition has width 2. The bag $\{2, 8, 10\}$ is the root of $\mathrm{Tree}(G)$, the root bag of node 6 is $B_6 = \{6, 7, 9\}$, the level of node 9 is $\mathsf{Lv}(9) = \mathsf{Lv}(\{8, 9, 10\}) = 1$, and the bag of the edge $(9, 1)$ is $B_{(9,1)} = \{1, 8, 9\}$. We have $V_T(B_6) = \{6, 7, 9, 4, 5\}$ and $\mathcal{V}_T(B_6) = \{6, 7, 9, 8, 10, 2\}$. The subtree $T(B_6)$ is shown in bold.

Separator property. A key property of a tree-decomposition $\mathrm{Tree}(G)$ is that the nodes of each bag B form a *separator* of G. Removing B splits $\mathrm{Tree}(G)$ into a number of connected components. The separator property states that every path between nodes that appear in bags of different components has to go through some node in B. This is formally stated in the following lemma. This separator property serves as a basis for some useful lemmas regarding node distances on a graph given a tree-decomposition (Lemmas 2.3 and 2.4).

Lemma 2.1 ([Bodlaender, 1998, Lemma 3])**.** *Consider a graph $G = (V, E)$, a tree-decomposition $T = \mathrm{Tree}(G)$, and a bag B of T. Let $(\mathcal{C}_i)_i$ be the components of T created by removing B from T, and let V_i be the set of nodes that appear in bags of component $\mathcal{C}_i$. For every $i \neq j$, nodes $u \in V_i$, $v \in V_j$ and path $P : u \rightsquigarrow v$, we have that $P \cap B \neq \emptyset$ (i.e., all paths between u and v go through some node in B).*

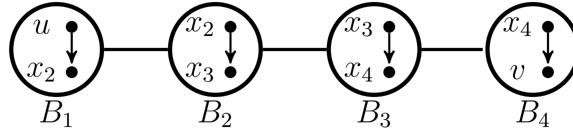


Figure 2.2: Illustration of Lemma 2.3. If P is the unique simple path $B_1 \rightsquigarrow B_4$ in $\mathrm{Tree}(G)$, then there exist (not necessarily distinct) $x_i \in B_{i-1} \cap B_i$ with $1 < i \leqslant 4$ such that $d(u,v) = d(u,x_2) \otimes d(x_2,x_3) \otimes d(x_3,x_4) \otimes d(x_4,v)$.

Using Lemma 2.1, we prove the following stronger version of the separator property, which will be useful throughout the paper.

Lemma 2.2. *Consider a graph $G = (V,E)$ and a tree-decomposition $\mathrm{Tree}(G)$. Let $u, v \in V$, and consider two distinct bags B_1 and B_j such that $u \in B_1$ and $v \in B_j$. Let $P' : B_1, B_2, \ldots, B_j$ be the unique simple path in T from B_1 to B_j. For each $i \in \{2, \ldots, j\}$ and for each path $P : u \rightsquigarrow v$, there exists a node $x_i \in (B_{i-1} \cap B_i \cap P)$.*

Proof. Let $T = \mathrm{Tree}(G)$. Fix a number $i \in \{2, \ldots, j\}$. We argue that for each path $P : u \rightsquigarrow v$, there exists a node $x_i \in (B_{i-1} \cap B_i \cap P)$. We construct a tree T', which is similar to T except that instead of having an edge between bag B_{i-1} and bag B_i, there is a new bag B, that contains the nodes in $B_{i-1} \cap B_i$, and there is an edge between B_{i-1} and B and one between B and B_i. It is easy to see that T' satisfies the properties C1-C3 of a tree-decomposition of G. By Lemma 2.1, each bag B' in the unique path $P'' : B_1, \ldots, B_{i-1}, B, B_i, \ldots, B_j$ in T' separates u from v in G. Hence, each path $u \rightsquigarrow v$ must go through some node in B, and the result follows. $\square$

The following lemma states that for nodes that appear in bags B, B' of the tree-decomposition $T = \mathrm{Tree}(G)$, their distance can be written as a sum of distances $d(x_i, x_{i+1})$ between pairs of nodes (x_i, x_{i+1}) that appear in bags B_i that constitute the unique $B \rightsquigarrow B'$ path in T. See Fig. 2.2 for an illustration.

Lemma 2.3. *Consider a weighted graph $G = (V, E, \mathsf{wt})$ and a tree-decomposition $\mathrm{Tree}(G)$. Let $u, v \in V$, and $P' : B_1, B_2, \ldots, B_j$ be a simple path in T such that $u \in B_1$ and $v \in B_j$. Let $A = \{u\} \times \left(\prod_{1<i\leqslant j} (B_{i-1} \cap B_i)\right) \times \{v\}$. Then $d(u,v) = \bigoplus_{(x_1,\ldots,x_{j+1}) \in A} \bigoplus_{i=1}^{j} d(x_i, x_{i+1})$.*

Proof. Consider a witness path $P : u \rightsquigarrow v$ such that $\mathsf{wt}(P) = d(u,v)$. By Lemma 2.2, there exists some node $x_i \in (B_{i-1} \cap B_i \cap P)$, for each $i \in \{1, \ldots, j\}$. It easily follows that $d(u,v) = \bigotimes_{i=1}^{j} d(x_i, x_{i+1})$ with $x_1, \ldots x_{j+1} \in A$. $\square$

Lemma 2.3 reduces the problem of computing the distance $d(u,v)$ between any pair (u,v), to computing the *local distance* between every pair nodes that appear together in some bag. The following crucial lemma states that given a tree decomposition of constant width, such local distances can be computed in time linear in the size of the tree-decomposition.

Lemma 2.4 ([Chaudhuri and Zaroliagis, 1995, Lemma 7])**.** *Given a weighted graph $G = (V, E, \mathsf{wt})$ of treewidth t and a tree-decomposition $T = (V_T, E_T)$ of G of width $O(t)$, we can compute for all bags $B \in V_T$ a* local distance map $\mathsf{LD}_B : B \times B \to \Sigma$ *with* $\mathsf{LD}_B(u,v) = d(u,v)$ *in total time $O(|V_T| \cdot t^3)$ and space $O(|V_T| \cdot t^2)$.*

We note that the above lemma is found as [Chaudhuri and Zaroliagis, 1995, Lemma 7], where the complexity bounds are stated at a factor t larger. However, that lemma is concerned with also computing a shortest-path witness of each local distance. It follows easily from the proofs of [Chaudhuri and Zaroliagis, 1995, Lemma 7] that if we are only interested in the local distances, the obtained upperbounds are those stated in Lemma 2.4.

Various types of tree decompositions. There exist various types of tree decompositions, mostly based on the number of elements a bag B differs from its neighbors B_i [Bodlaender, 1998]. Here we introduce a few more notions, which will be useful throughout. As is often the case, transforming a tree decomposition from one type to another is typically an efficient operation. Let G be a graph of treewidth t, and $\mathrm{Tree}(G) = (V_T, E_T)$ a tree decomposition of G. Then, $\mathrm{Tree}(G)$ is called

- α-*approximate*, for some fixed $\alpha \in \mathbb{N}$, if it has width at most $\alpha \cdot (t+1) - 1$;
- (α, β, γ)-*balanced* if it is α-approximate and (β, γ)-balanced;
- *nicely rooted*, if every bag is the root bag of at most one node of G;
- *small*, if $|V_T| = O(\frac{n}{t})$;
- *smooth*, if (i) for every bag $B \in V_T$ we have $|B| = t + 1$, and (ii) for every pair of bags $(B_1, B_2) \in E_T$ we have $|B_1 \cap B_2| = t$.

Note that the various notions on trees carry over to tree decompositions, e.g., $\mathrm{Tree}(G)$ might be k-ary, balanced e.t.c.

Constructing and manipulating tree decompositions. Given a graph G, the problem of determining the treewidth t of G and constructing a tree-decomposition $\mathrm{Tree}(G)$ of width t is known to be NP-complete [Arnborg *et al.*, 1987]. However, the problem is fixed-parameter tractable *(FPT)* on the treewidth, and for constant-treewidth graphs there exist various linear-time, or almost-linear-time algorithms for constructing a tree decomposition $\mathrm{Tree}(G)$ whose width either achieves, or approximates the treewidth of G [Bodlaender, 1996; Bodlaender *et al.*, 2013]. Here we will make use of the following theorem, which states that balanced tree decompositions of constant-treewidth graphs can be constructed efficiently in both time and space.

Theorem 2.1. *For every graph G with n nodes and bounded treewidth $t = O(1)$, a balanced binary tree decomposition $\mathrm{Tree}(G)$ of $O(n)$ bags and*

1. *width $3 \cdot t + 2$ can be constructed in $O(n)$ time and space [Bodlaender and Hagerup, 1995],*

2. *width $4 \cdot t + 3$ can be constructed in* DLOGSPACE*(and hence polynomial time) [Elberfeld* et al.*, 2010].*

In several cases, we find it useful to work with binary tree decompositions. The following lemma states that any tree decomposition can be made binary efficiently, while preserving the width and increasing the height by at most a logarithmic term.

Lemma 2.5 (Binary Tree Decompositions). *Let G be a graph, and $T_1 = (V_{T_1}, E_{T_1})$ be a tree decomposition G with width t and height η. A binary tree decomposition $T_2 = (V_{T_2}, E_{T_2})$ of G with width t, size $|V_{T_2}| = O(V_{T_1})$ and height $O(\eta + \log |V_{T_1}|)$ can be constructed in $O(n \cdot t)$ time.*

Proof. We turn T_1 to a binary tree-decomposition $T_2 = (V_2, E_2)$, as follows. We traverse T_1 bottom-up, and we replace every bag B of T_1 with $k > 2$ children with a binary tree T_B of height $\lfloor \log k \rfloor$. The leaves of T_B are the children of B, whereas every internal node of T_B consists of a copy of B. We call these copies the *new bags* of T_2. Note that T_B has size $O(k)$, and thus T_2 has size $O(n)$. Finally, note that a bag in T_2 has a single child iff it is not a new bag in T_2, and it has a single child in T_1 as well. Hence every path from the root to a leaf of T_2 can traverse at most $O(\log |V_{T_1}|)$ new bags, and thus the height of T_2 increases by at most a $O(\log |V_{T_1}|)$ term compared to T_1. Finally, it is easy to see that T_2 is a tree decomposition of G, and has the same width as T_1. The time bound follows trivially. $\square$

The following lemma states that a tree decomposition can be made nicely rooted efficiently.

Lemma 2.6 (Nicely-rooted Tree Decompositions). *Given a tree decomposition* $\mathrm{Tree}(G)$ *of* G *of width* $O(t)$ *and* $O(n)$ *bags, a nicely rooted, binary tree decomposition* $\mathrm{Tree}'(G)$ *of width* $O(t)$ *can be constructed in* $O(n \cdot t)$ *time. If* $t = O(1)$ *and* $\mathrm{Tree}(G)$ *is balanced, then so is* $\mathrm{Tree}'(G)$*.*

Proof. First, $\mathrm{Tree}(G)$ can be turned into a binary tree decomposition T_1 by Lemma 2.5, and T_1 remains balanced. Then, we can make T_1 nicely rooted simply by replacing each bag B which is the root of $k > 1$ nodes $x_1, \dots x_k$ with a chain of bags $B_1, \dots, B_k = B$, where each B_i is the parent of B_{i+1}, and $B_{i+1} = B_i \cup \{x_{i+1}\}$. Note that this keeps the tree binary and increases its height by at most a factor t, hence if $t = O(1)$ and $\mathrm{Tree}(G)$ is balanced, then the resulting tree is also balanced. □

Hence, combined with Theorem 2.1, a nicely rooted, balanced binary tree decomposition of a constant-treewidth graph can be constructed in $O(n)$ time. The following lemma states that a tree decomposition can be made small efficiently.

Lemma 2.7 (Small Tree Decompositions). *Given a tree decomposition* $\mathrm{Tree}(G)$ *of* G *of width* $O(t)$ *and* $O(n)$ *bags, a small, binary tree decomposition* $\mathrm{Tree}'(G)$ *of width* $O(t)$ *can be constructed in* $O(n \cdot t)$ *time. Moreover, if* $\mathrm{Tree}(G)$ *is balanced, then so is* $\mathrm{Tree}'(G)$*.*

Proof. Let $k = O(t)$ be the width of $\mathrm{Tree}(G)$. The construction is achieved using the following steps.

1. Following the steps of [Bodlaender, 1996, Lemma 2.4], we turn $\mathrm{Tree}(G)$ to a *smooth* tree-decomposition $T_1 = (V_1, E_1)$. The process of [Bodlaender, 1996, Lemma 2.4] can be performed $O(n \cdot t)$ time and increases the height by at most a factor 2, hence if $\mathrm{Tree}(G)$ is balanced, T_1 is also balanced, and by [Bodlaender, 1996, Lemma 2.5], we have $|V_1| = O(n)$.

2. We turn T_1 to a binary decomposition T_2 in $O(n \cdot t)$ time using Lemma 2.5. Note that if T_1 is balanced, then so is T_2.

3. We construct a tree-decomposition $T_3 = (V_3, E_3)$ by partitioning T_2 to disjoint connected components of size between $\frac{k}{2}$ and k each (the last component might have size less than $\frac{k}{2}$) and contracting each such component to a single bag in T_3. Since T_2 is smooth, the

number of nodes in the union of the bags of each component is at most $2 \cdot k$. Hence the width of T_3 is $O(k)$. The partitioning is done as follows. We traverse T_2 bottom-up and group bags into components in a greedy way. In particular, given that the traversal is on a current bag B, we keep track of the number of bags i_B below B (not including B) that have not been grouped to a component yet. The first time we find $i_B \geqslant t$, let B' be the child of B with the largest number $i_{B'}$ among the children of B. We group B' and its ungrouped descendants into a new component $\mathcal{C}$, and continue with the traversal. Observe that the size of $\mathcal{C}$ is $\frac{k}{2} \leqslant |\mathcal{C}| < k$.

4. Finally, we construct $\mathrm{Tree}'(G)$ by turning T_3 to a binary tree-decomposition as in Step 2.

Note that all steps above require $O(n \cdot t)$ time. The desired result follows. □

Hence, combined with Theorem 2.1, a small, balanced binary tree decomposition of a constant-treewidth graph can be constructed in $O(n)$ time. The following important theorem states that (α, β, γ)-balanced tree decompositions can be constructed efficiently. Recall that the factor α determines how close to optimal the width of the tree decomposition is, while the parameters (β, γ) determine how strongly-balanced it is. Theorem 2.2 provides a tradeoff between the two properties, by constructing tree decompositions that have tunable width and balance, based on two parameters δ and λ. A tree decomposition can be made more strongly balanced, if one allows its width to increase. As the proof is technical, we provide only a sketch here, and devote the next subsection in the formal proof.

Theorem 2.2 (**(α, β, γ)-tree decompositions).** *For every graph G with n nodes and constant treewidth, for any fixed $\delta > 0$ and $\lambda \in \mathbb{N}$ with $\lambda \geqslant 2$, let $\alpha = \frac{4\cdot\lambda}{\delta}$, $\beta = \left(\frac{1+\delta}{2}\right)^{\lambda-1}$, and $\gamma = \lambda$. A binary (α, β, γ) tree-decomposition $\mathrm{Tree}(G)$ with $O(n)$ bags can be constructed in $O(n \cdot \log n)$ time and $O(n)$ space.*

(Sketch). Here we only outline the construction. The formal proof can be found in Section 2.3.3. The construction considers that a tree-decomposition $\mathrm{Tree}'(G)$ of width t and $O(n)$ bags is given (which can be obtained using e.g. [Bodlaender, 1996] in $O(n)$ time). Given the parameters $\delta > 0$ and $\lambda \in \mathbb{N}$ with $\lambda \geqslant 2$, $\mathrm{Tree}'(G)$ is turned to an (α, β, γ) tree-decomposition, for $\alpha = \frac{4\cdot\lambda}{\delta}$, $\beta = \left(\frac{1+\delta}{2}\right)^{\lambda-1}$, and $\gamma = \lambda$, in two conceptual steps.

1. A tree of bags $\mathsf{R_G}$ is constructed, which is (β, γ)-balanced.

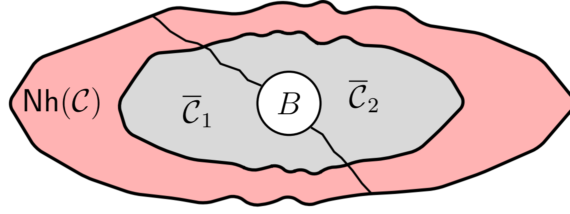


Figure 2.3: Illustration of one recursive step of Rank on a component $\mathcal{C}$ (gray). $\mathcal{C}$ is split into two sub-components $\overline{\mathcal{C}}_1$ and $\overline{\mathcal{C}}_2$ by removing a list of bags $\mathcal{X} = (B_i)_i$. Once every λ recursive calls, $\mathcal{X}$ contains one bag, such that the neighborhood $\mathsf{Nh}(\overline{\mathcal{C}}_i)$ of each $\overline{\mathcal{C}}_i$ is at most half the size of $\mathsf{Nh}(\mathcal{C})$ (i.e., the red area is split in half). In the remaining $\lambda - 1$ recursive calls, $\mathcal{X}$ contains m bags, such that the size of each $\overline{\mathcal{C}}_i$, is at most $\frac{1+\delta}{2}$ fraction the size of $\mathcal{C}$. (i.e., the gray area is split in almost half).

2. $\mathsf{R_G}$ is turned to an α-approximate tree decomposition of G.

The first construction is obtained by a recursive algorithm Rank, which operates on inputs $(\mathcal{C}, \ell)$, where $\mathcal{C}$ is a component of $\mathrm{Tree}'(G)$, and $\ell \in [\lambda]$ specifies the type of operation the algorithm performs on $\mathcal{C}$. Given such a component $\mathcal{C}$, we denote by $\mathsf{Nh}(\mathcal{C})$ the *neighborhood* of $\mathcal{C}$, defined as the set of bags of $\mathrm{Tree}'(G)$ that are incident to $\mathcal{C}$. Informally, on input $(\mathcal{C}, \ell)$, the algorithm partitions $\mathcal{C}$ into two sub-components $\overline{\mathcal{C}}_1$ and $\overline{\mathcal{C}}_2$ such that either (i) the size of each $\overline{\mathcal{C}}_i$ is approximately half the size of $\mathcal{C}$, or (ii) the size of the neighborhood of each $\overline{\mathcal{C}}_i$ is approximately half the size of the neighborhood of $\mathcal{C}$. In more detail,

1. If $\ell > 0$, then $\mathcal{C}$ is partitioned into components $\mathcal{Y} = (\mathcal{C}_1, \ldots, \mathcal{C}_r)$, by removing a list of bags $\mathcal{X} = (B_1, \ldots B_m)$, such that $|\mathcal{C}_i| \leqslant \frac{\delta}{2} \cdot |\mathcal{C}|$. The union of $\mathcal{X}$ yields a new bag $\mathcal{B}$ in $\mathsf{R_G}$. Then $\mathcal{Y}$ is merged into two components $\overline{\mathcal{C}}_1, \overline{\mathcal{C}}_2$ with $|\overline{\mathcal{C}}_1| \leqslant |\overline{\mathcal{C}}_2| \leqslant \frac{1+\delta}{2} \cdot |\mathcal{C}|$. Finally, each $\overline{\mathcal{C}}_i$ is passed on to the next recursive step with $\ell = (\ell + 1) \mod \lambda$.

2. If $\ell = 0$, then $\mathcal{C}$ is partitioned into two components $\overline{\mathcal{C}}_1, \overline{\mathcal{C}}_2$ such that $|\mathsf{Nh}(\overline{\mathcal{C}}_i) \cap \mathsf{Nh}(\mathcal{C})| \leqslant \frac{|\mathsf{Nh}(\mathcal{C})|}{2}$ by removing a single bag B. This bag becomes a new bag $\mathcal{B}$ in $\mathsf{R_G}$, and each $\overline{\mathcal{C}}_i$ is passed on to the next recursive step with $\ell = (\ell + 1) \mod \lambda$.

Fig. 2.3 provides an illustration. The second construction is obtained simply by inserting in each bag $\mathcal{B}$ of $\mathsf{R_G}$ the nodes contained in the neighborhood $\mathsf{Nh}(\mathcal{C})$ of the component $\mathcal{C}$ from which $\mathcal{B}$ was constructed. Finally, the tree decomposition is made binary using Lemma 2.5, which keeps the size and height of the tree asymptotically the same. $\square$

Remark 2.1. The notion of balanced tree decompositions exists in the literature [Elberfeld *et al.*, 2010; Baruah and Hickey, 1998], but balancing only requires that the height of the tree is logarithmic in its size. In Theorem 2.2 we develop a stronger notion of balancing, which will become crucial in obtaining the results of latter sections.

2.3.3 Proof of Theorem 2.2

We now present in detail the construction of an (α, β, γ)-balanced tree decomposition. Given constants $0 < \delta \leqslant 1$ and $\lambda \geqslant 2$, throughout this section we fix

$$\alpha = \frac{4 \cdot \lambda}{\delta}; \quad \beta = \left(\frac{1+\delta}{2}\right)^{\lambda - 1}; \quad \gamma = \lambda$$

We show how given a graph G of treewidth t and a tree-decomposition $\mathrm{Tree}'(G)$ of b bags and width t, we can construct in $O(b \cdot \log b)$ time and $O(b)$ space a (α, β, γ) tree-decomposition with b bags. That is, the resulting tree-decomposition has width at most $\alpha \cdot (t+1)$, and for every bag B and descendant B' of B that appears γ levels below, we have that $|T(B')| \leqslant \beta \cdot |T(B)|$ (i.e., the number of bags in $T(B')$ is at most β times as large as that in $T(B)$). The result is established in two steps.

Tree components and operations Split **and** Merge**.** Given a tree-decomposition $T = (V_T, E_T)$, and a connected component $\mathcal{C}$ of T, the *neighborhood* $\mathsf{Nh}(\mathcal{C})$ of $\mathcal{C}$ is the set of bags in $V_T \setminus \mathcal{C}$ that have a neighbor in $\mathcal{C}$, i.e.

$$\mathsf{Nh}(\mathcal{C}) = \{B \in V_T \setminus \mathcal{C} : \ (\{B\} \times \mathcal{C}) \cap E_T \neq \varnothing\}$$

Given a component $\mathcal{C}$, we define the operation Split as $\mathsf{Split}(\mathcal{C}) = (\mathcal{X}, \mathcal{Y})$, where $\mathcal{X} \subseteq \mathcal{C}$ is a list of bags $(B_1, \dots B_{\frac{2}{\delta}})$ and $\mathcal{Y}$ is a list of sub-components $(\mathcal{C}_1, \dots \mathcal{C}_r)$ such that removing each bag B_i from $\mathcal{C}$ splits $\mathcal{C}$ into the subcomponents $\mathcal{Y}$, and for every i we have $|\mathcal{C}_i| \leqslant \frac{\delta}{2} \cdot |\mathcal{C}|$. Note that since $\mathcal{C}$ is a component of a tree, we can find a single separator bag that splits $\mathcal{C}$ into sub-components of size at most $\frac{|\mathcal{C}|}{2}$. Applying this step recursively for $\log \frac{2}{\delta}$ levels yields the desired separator set $\mathcal{X}$. For technical convenience, if this process yields less than $\frac{2}{\delta}$ bags, we repeat some of these bags until we have $\frac{2}{\delta}$ many.

Consider a list of components $\mathcal{Y} = (\mathcal{C}_1, \ldots \mathcal{C}_r)$, and let $z = \sum_i |\mathcal{C}_i|$. Let j be the largest integer such that $\sum_{i=1}^{j} |\mathcal{C}_i| \leqslant \frac{z}{2}$. We define the operation $\mathsf{Merge}(\mathcal{Y}) = (\overline{\mathcal{C}}_1, \overline{\mathcal{C}}_2)$, where $\overline{\mathcal{C}}_1 = \bigcup_{i=1}^{j} \mathcal{C}_i$ and $\overline{\mathcal{C}}_2 = \bigcup_{i=j+1}^{r} \mathcal{C}_i$. The following claim is trivially obtained.

Claim 2.1. *If $|\mathcal{C}_i| < \frac{\delta}{2} \cdot z$ for all i, then $|\overline{\mathcal{C}}_1| \leqslant |\overline{\mathcal{C}}_2| \leqslant \frac{1+\delta}{2} \cdot z$.*

Proof. By construction, $\frac{1-\delta}{2} \cdot z < |\overline{\mathcal{C}}_1| \leqslant \frac{1}{2} \cdot z$, and since $\overline{\mathcal{C}}_1$ and $\overline{\mathcal{C}}_2$ partition $\mathcal{Y}$, we have $|\overline{\mathcal{C}}_1| + |\overline{\mathcal{C}}_2| = z$. The result follows. □

Construction of a (β, γ)-balanced rank tree. In the following, we consider that $T_G = \mathrm{Tree}'(G) = (V_T, E_T)$ is a tree-decomposition of G and has $|V_T| = b$ bags. Given the parameters $\lambda \in \mathbb{N}$ with $\lambda \geqslant 2$ and $0 < \delta < 1$, we use the following algorithm Rank to construct a tree of bags $\mathsf{R_G}$. Rank operates recursively on inputs $(\mathcal{C}, \ell)$ where $\mathcal{C}$ is a component of T_G and $\ell \in \{0\} \cup [\lambda - 1]$, as follows.

1. If $|\mathcal{C}| \cdot \frac{\delta}{2} \leqslant 1$, construct a bag $\mathcal{B} = \bigcup_{B \in \mathcal{C}} B$, and return $\mathcal{B}$.
2. Else, if $\ell > 0$, let $(\mathcal{X}, \mathcal{Y}) = \mathsf{Split}(\mathcal{C})$. Construct a bag $\mathcal{B} = \bigcup_{B_i \in \mathcal{X}} B_i$, and let $(\overline{\mathcal{C}}_1, \overline{\mathcal{C}}_2) = \mathsf{Merge}(\mathcal{Y})$. Call Rank recursively on input $(\overline{\mathcal{C}}_1, (\ell + 1) \mod \lambda)$ and $(\overline{\mathcal{C}}_2, (\ell + 1) \mod \lambda)$, and let $\mathcal{B}_1$, $\mathcal{B}_2$ be the returned bags. Make $\mathcal{B}_1$ and $\mathcal{B}_2$ the left and right child of $\mathcal{B}$, and return the resulting tree.
3. Else, if $\ell = 0$, if $|\mathsf{Nh}(\mathcal{C})| > 1$, find a bag B whose removal splits $\mathcal{C}$ into connected components $\overline{\mathcal{C}}_1, \overline{\mathcal{C}}_2$ with $|\mathsf{Nh}(\overline{\mathcal{C}}_i) \cap \mathsf{Nh}(\mathcal{C})| \leqslant \frac{|\mathsf{Nh}(\mathcal{C})|}{2}$. Call Rank recursively on input $(\overline{\mathcal{C}}_1, (\ell+1) \mod \lambda)$ and $(\overline{\mathcal{C}}_2, (\ell+1) \mod \lambda)$, and let $\mathcal{B}_1$, $\mathcal{B}_2$ be the returned bags. Make $\mathcal{B}_1$ and $\mathcal{B}_2$ the left and right child of $\mathcal{B}$, and return the resulting tree. Finally, if $|\mathsf{Nh}(\mathcal{C})| \leqslant 1$, call Rank recursively on input $(\mathcal{C}, (\ell - 1) \mod \lambda)$, and return the tree obtained by this recursive call.

Algorithm 1 provides the formal description of the algorithm.

In the following we use the symbols B and $\mathcal{B}$ to refer to bags of T_G and $\mathsf{R_G}$ respectively. Given a bag $\mathcal{B}$, we denote by $\mathcal{C}(\mathcal{B})$ the input component of Rank when $\mathcal{B}$ was constructed, and define the *neighborhood* of $\mathcal{B}$ as $\mathsf{Nh}(\mathcal{B}) = \mathsf{Nh}(\mathcal{C}(\mathcal{B}))$. Additionally, we denote by $\mathsf{Bh}(\mathcal{B})$ the set of separator bags $B_1, \ldots B_r$ of $\mathcal{C}$ that were used to construct $\mathcal{B}$. It is straightforward that $\mathsf{Bh}(\mathcal{B}_1) \cap \mathsf{Bh}(\mathcal{B}_2) = \varnothing$ for every distinct $\mathcal{B}_1$ and $\mathcal{B}_2$.

Algorithm 1: Rank

Input: A component $\mathcal{C}$ of T_G, a natural number $\ell \in [\lambda]$

Output: A rank tree $\mathsf{R_G}$

```
1  Assign 𝒯 ← an empty tree
2  if |𝒞| · δ/2 ⩽ 1 then
3  |  Assign ℬ ← ⋃_{B∈𝒞} B and make ℬ the root of 𝒯
4  else if ℓ > 0 then
5  |  Assign (𝒳, 𝒴) ← Split(𝒞)
6  |  Assign ℬ ← ⋃_{B_i∈𝒳} B_i
7  |  Assign (𝒞̄_1, 𝒞̄_2) ← Merge(𝒴)
8  |  Assign 𝒯_1 ← Rank(𝒞̄_1, (ℓ+1) mod λ)
9  |  Assign 𝒯_2 ← Rank(𝒞̄_2, (ℓ+1) mod λ)
10 |  Make ℬ the root of 𝒯 and 𝒯_1 and 𝒯_2 its left and right subtree
11 else
12 |  if |Nh(𝒞)| > 1 then
13 |  |  Let B ← a bag of 𝒞 whose removal splits 𝒞 to 𝒞̄_1, 𝒞̄_2 with |Nh(𝒞̄_i) ∩ Nh(𝒞)| ⩽ |Nh(𝒞)|/2
14 |  |  Assign ℬ ← B
15 |  |  Assign 𝒯_1 ← Rank(𝒞̄_1, (ℓ+1) mod λ)
16 |  |  Assign 𝒯_2 ← Rank(𝒞̄_2, (ℓ+1) mod λ)
17 |  |  Make ℬ the root of 𝒯 and 𝒯_1 and 𝒯_2 its left and right subtree
18 |  else
19 |  |  Assign 𝒯 ← Rank(𝒞, (ℓ−1) mod λ)
20 |  end
21 end
22 return 𝒯
```

Claim 2.2. *Let $\mathcal{B}$ and $\mathcal{B}'$ be respectively a bag and its parent in* $\mathsf{R_G}$*. Then* $\mathsf{Nh}(\mathcal{B}) \subseteq \mathsf{Nh}(\mathcal{B}') \cup \mathsf{Bh}(\mathcal{B}')$*, and thus* $|\mathsf{Nh}(\mathcal{B})| \leqslant |\mathsf{Nh}(\mathcal{B}')| + \frac{2}{\delta}$.

Proof. Every bag in $\mathsf{Nh}(\mathcal{C}(\mathcal{B}))$ is either a bag in $\mathsf{Nh}(\mathcal{C}(\mathcal{B}'))$, or a separator bag of $\mathcal{C}(\mathcal{B}')$, and thus a bag of $\mathsf{Bh}(B')$. □

Note that every bag B of T_G belongs in $\mathsf{Bh}(\mathcal{B})$ of some bag $\mathcal{B}$ of $\mathsf{R_G}$, and thus the bags of $\mathsf{R_G}$ already cover all nodes and edges of G (i.e., properties C1 and C2 of a tree decomposition). In the following we show how $\mathsf{R_G}$ can be modified to also satisfy condition C3, i.e., that every node u appears in a contiguous subtree of $\mathsf{R_G}$. Given a bag $\mathcal{B}$, we denote by $\mathsf{NhV}(\mathcal{B}) = \mathcal{B} \cup \bigcup_{B \in \mathsf{Nh}(\mathcal{B})} B$, i.e., $\mathsf{NhV}(\mathcal{B})$ is the set of nodes of G that appear in $\mathcal{B}$ and its neighborhood. In the sequel, to distinguish between paths in different trees, given a tree of bags T (e.g. T_G or $\mathsf{R_G}$) and bags B_1, B_2 of T, we write $B_1 \rightsquigarrow_T B_2$ to denote the unique simple path from B_1 to B_2 in T.

We say that a pair of bags $(\mathcal{B}_1, \mathcal{B}_2)$ *form a gap* of some node u in a tree of bags T (e.g., $\mathsf{R_G}$) if $u \in \mathcal{B}_1 \cap \mathcal{B}_2$ and for the unique simple path $P : \mathcal{B}_1 \rightsquigarrow_T \mathcal{B}_2$ we have that $|P| \geqslant 2$ (i.e., there is at least one intermediate bag in P) and for all intermediate bags $\mathcal{B}$ in P we have $u \not\in \mathcal{B}$. The following crucial lemma shows that if $\mathcal{B}_1$ and $\mathcal{B}_2$ form a gap of u in $\widehat{\mathsf{R_G}}$, then for every intermediate bag $\mathcal{B}$ in the path $P : \mathcal{B}_1 \rightsquigarrow_{\mathsf{R_G}} \mathcal{B}_2$, u must appear in some bag of $\mathsf{Nh}(\mathcal{B})$.

Lemma 2.8. *For every node u, and pair of bags $(\mathcal{B}_1, \mathcal{B}_2)$ that form a gap of u in* $\mathsf{R_G}$*, such that $\mathcal{B}_1$ is an ancestor of $\mathcal{B}_2$, for every intermediate bag $\mathcal{B}$ in $P : \mathcal{B}_1 \rightsquigarrow_{\mathsf{R_G}} \mathcal{B}_2$ in* $\mathsf{R_G}$*, we have that* $u \in \mathsf{NhV}(\mathcal{B})$.

Proof. Fix any such a bag $\mathcal{B}$, and since $\mathcal{B}_1$ and $\mathcal{B}_2$ form a gap of u, there exist bags $B_1 \in \mathsf{Bh}(\mathcal{B}_1)$ and $B_2 \in \mathsf{Bh}(\mathcal{B}_2)$ with $u \in B_1 \cap B_2$. Consider the time point j that bag $\mathcal{B}$ was constructed. Let B^r be the rightmost bag of the path $P_1 : B_1 \rightsquigarrow_{T_G} B_2$ that had been chosen as a separator in some previous step $j' < j$ of the algorithm. Note that B_1 has been chosen as such a separator, therefore B^r is well defined. We argue that $B^r \in \mathsf{Nh}(\mathcal{B})$, which implies that $u \in \mathsf{NhV}(\mathcal{B})$. This is done in two steps.

1. Since $\mathcal{B}_2$ is a descendant of $\mathcal{B}$, we have that $B_2 \in \mathcal{C}(\mathcal{B})$, i.e., B_2 is a bag of the component when $\mathcal{B}$ was constructed.

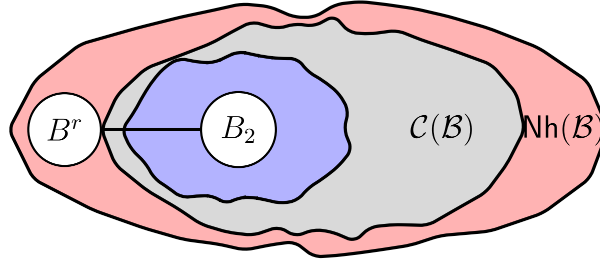


Figure 2.4: Illustration of Lemma 2.8. Since B_2 belongs to $\mathcal{C}(\mathcal{B})$ and the blue sub-component has not been split yet, the bag B^r is in the neighborhood of the blue sub-component, and thus in the neighborhood of $\mathcal{C}(\mathcal{B})$.

2. By the choice of B^r, for every intermediate bag B^i in the path $B^r \rightsquigarrow_{T_G} B_2$ we have that at the time $\mathcal{B}$ was constructed, each B^i belongs to the same component as B_2, and hence B^r is incident to that component.

These two points imply that $B^r \in \mathsf{Nh}(\mathcal{B})$. From the properties of tree decomposition we know that $u \in B^r$. It follows that $u \in \mathsf{NhV}(\mathcal{B})$, as desired. Figure 2.4 provides an illustration of the argument. □

Turning the rank tree to a tree decomposition. Lemma 2.8 suggests a way to turn the rank tree $\mathsf{R_G}$ to a tree-decomposition. Let $\widehat{\mathsf{R_G}} = \mathsf{Replace}(\mathsf{R_G})$ be the tree obtained by replacing each bag B of $\mathsf{R_G}$ with $\mathsf{NhV}(B)$. For a bag $\mathcal{B}$ in $\mathsf{R_G}$ let $\widehat{\mathcal{B}}$ be the corresponding bag in $\widehat{\mathsf{R_G}}$ and vice versa.

Claim 2.3. *If there is a pair of bags $\widehat{\mathcal{B}}_1$, $\widehat{\mathcal{B}}_2$ that form a gap of some node u in $\widehat{\mathsf{R_G}}$, then there is a pair of bags $\widehat{\mathcal{B}}'_1$, $\widehat{\mathcal{B}}'_2$ that also form a gap of u, and $\widehat{\mathcal{B}}'_1$ is ancestor of $\widehat{\mathcal{B}}'_2$.*

Proof. First, note that neither parent of the bags $\widehat{\mathcal{B}}_1$ and $\widehat{\mathcal{B}}_2$ in $\widehat{\mathsf{R_G}}$ contains u. Assume that neither of $\widehat{\mathcal{B}}_1$, $\widehat{\mathcal{B}}_2$ is ancestor of the other.

1. If for some $i \in \{1, 2\}$ there is no bag $B_i \in \mathsf{Bh}(\mathcal{B}_i)$ such that $u \in B_i$, then $u \in \mathsf{NhV}(\mathcal{B}_i) \setminus \mathcal{B}_i$ and hence there is an ancestor $\mathcal{B}'_i$ of $\mathcal{B}_i$ such that $u \in \mathsf{NhV}(\mathcal{B}'_i)$. Thus $\widehat{\mathcal{B}}'_i$ and $\widehat{\mathcal{B}}_i$ form a gap of u in $\widehat{\mathsf{R_G}}$.

2. Else, there exists a $B_1 \in \mathsf{Bh}(\mathcal{B}_1)$ and $B_2 \in \mathsf{Bh}(\mathcal{B}_2)$ such that $u \in B_1 \cap B_2$. Let B be first bag in the path $B_1 \rightsquigarrow_{T_G} B_2$ that was chosen as a separator. We have $B \in \mathsf{Bh}(\mathcal{B})$ for some

ancestor $\mathcal{B}$ of $\mathcal{B}_1$ and $\mathcal{B}_2$, therefore $u \in \mathsf{NhV}(\mathcal{B})$, and thus $\widehat{\mathcal{B}}$ forms a gap of u with both $\widehat{\mathcal{B}}_1$ and $\widehat{\mathcal{B}}_2$ in $\widehat{\mathsf{R_G}}$.

It follows that in both cases there exists an ancestor $\widehat{\mathcal{B}}_i'$ of some $\widehat{\mathcal{B}}_i$ so that the two form a gap of u in $\widehat{\mathsf{R_G}}$. □

The following lemma states that $\widehat{\mathsf{R_G}}$ is a tree decomposition of G.

Lemma 2.9. $\widehat{\mathsf{R_G}} = \mathsf{Replace}(\mathsf{R_G})$ *is a tree-decomposition of* G.

Proof. It is straightforward to see that the bags of $\widehat{\mathsf{R_G}}$ cover all nodes and edges of G (properties C1 and C2 of the definition of tree-decomposition), because for each bag $\mathcal{B}$, we have that $\mathcal{B} \subseteq \widehat{\mathcal{B}}$. It remains to show that every node u appears in a contiguous subtree of $\widehat{\mathsf{R_G}}$ (i.e., that property C3 is satisfied).

Assume towards contradiction otherwise, and by Claim 2.3 it follows that there exist bags $\widehat{\mathcal{B}}_1$ and $\widehat{\mathcal{B}}_2$ in $\widehat{\mathsf{R_G}}$ that form a gap of some node u such that $\widehat{\mathcal{B}}_1$ is an ancestor of $\widehat{\mathcal{B}}_2$. Let $\widehat{P} : \widehat{\mathcal{B}}_1 \rightsquigarrow_{\widehat{\mathsf{R_G}}} \widehat{\mathcal{B}}_2$ be the path between them, and $P : \mathcal{B}_1 \rightsquigarrow_{\mathsf{R_G}} \mathcal{B}_2$ the corresponding path in $\mathsf{R_G}$. By Lemma 2.8 we have $u \not\in \mathcal{B}_1 \cap \mathcal{B}_2$, otherwise for every intermediate bag $\mathcal{B} \in \widehat{P}$ we would have $u \in \mathsf{NhV}(\mathcal{B})$ and thus $u \in \widehat{\mathcal{B}}$. Additionally, we have $u \in \mathcal{B}_2$, otherwise by Claim 2.2, we would have $u \in \mathsf{NhV}(\mathcal{B}_2')$, where $\mathcal{B}_2'$ is the parent of $\mathcal{B}_2$, and thus $u \in \widehat{\mathcal{B}}_2'$, contradicting the assumption that $\widehat{\mathcal{B}}_1$ and $\widehat{\mathcal{B}}_2$ form a gap of u. Hence $u \not\in \mathcal{B}_1$. A similar argument as that of Claim 2.3 shows that for the parent $\mathcal{B}_1'$ of $\mathcal{B}_1$, we have that $u \in \mathcal{B}_1'$, and wlog, take $\mathcal{B}_1'$ to be the lowest ancestor of $\mathcal{B}_1$ with this property. Then $\mathcal{B}_1'$ is also an ancestor of $\mathcal{B}_2$, and $\mathcal{B}_1'$ and $\mathcal{B}_2$ form a gap of u in $\mathsf{R_G}$. Then by Lemma 2.8, for every intermediate bag $\mathcal{B}$ in the path $\mathcal{B}_1' \rightsquigarrow_{\mathsf{R_G}} \mathcal{B}_2$ we have that $u \in \mathsf{NhV}(\mathcal{B})$, thus $u \in \widehat{\mathcal{B}}$. Since the path $\widehat{\mathcal{B}}_1 \rightsquigarrow \widehat{\mathsf{R_G}}\widehat{\mathcal{B}}_2$ is a suffix of $\widehat{\mathcal{B}}_1' \rightsquigarrow_{\widehat{\mathsf{R_G}}} \widehat{\mathcal{B}}_2$, we have that $\widehat{\mathcal{B}}_1$ and $\widehat{\mathcal{B}}_2$ cannot form a gap of u. We have thus arrived at a contradiction, and the desired result follows. □

Properties of the tree-decomposition $\widehat{\mathsf{R_G}}$. Lemma 2.9 states that $\widehat{\mathsf{R_G}}$ obtained by replacing each bag of $\mathsf{R_G}$ with $\mathsf{NhV}(B)$ is a tree-decomposition of G. The remaining of the section focuses on showing that $\widehat{\mathsf{R_G}}$ is a (α, β, γ) tree-decomposition of G, and that it can be constructed in $O(b \cdot \log b)$ time and $O(b)$ space. Recall the definition of the parameters

$$\alpha = \frac{4 \cdot \lambda}{\delta}; \quad \beta = \left(\frac{1+\delta}{2}\right)^{\lambda - 1}; \quad \gamma = \lambda$$

Lemma 2.10. *The following assertions hold:*

1. *Every bag $\widehat{\mathcal{B}}$ of $\widehat{\mathsf{R_G}}$ is (β, γ)-balanced.*

2. *For every bag $\widehat{\mathcal{B}}$ of $\widehat{\mathsf{R_G}}$, we have $|\widehat{\mathcal{B}}| \leqslant \alpha \cdot (t+1)$.*

Proof. We prove each item separately.

1. For every bag $\mathcal{B}$ constructed by Rank, in at least $\gamma - 1$ out of every γ levels, Item 2 of the algorithm applies, and by Claim 2.1, the recursion proceeds on components $\overline{\mathcal{C}}_1$ and $\overline{\mathcal{C}}_2$ that are at most $\frac{1+\delta}{2}$ times as large as the input component $\mathcal{C}$ in that recursion step. Thus $\mathcal{B}$ is (β, γ)-balanced in $\mathsf{R_G}$, and hence $\widehat{\mathcal{B}}$ is (β, γ)-balanced in $\widehat{\mathsf{R_G}}$.

2. It suffices to show that for every bag $\mathcal{B}$, we have $|\mathsf{Nh}(\mathcal{B})| \leqslant \alpha - 1 = 2 \cdot (\frac{2}{\delta}) \cdot \lambda - 1$. Assume towards contradiction otherwise. Let $\mathcal{B}$ be the first bag that Rank constructed such that $|\mathsf{Nh}(\mathcal{B})| \geqslant 2 \cdot (\frac{2}{\delta}) \cdot \lambda$. Let $\mathcal{B}'$ be the lowest ancestor of $\mathcal{B}$ in $\mathsf{R_G}$ that was constructed by Rank on some input $(\mathcal{C}, \ell)$ with $\ell = 1$, and let $\mathcal{B}''$ be the parent of $\mathcal{B}'$ in $\mathsf{R_G}$ (note that $\mathcal{B}'$ can be $\mathcal{B}$ itself). By Item 3 of Rank, it follows that $|\mathsf{Nh}(\mathcal{B}')| \leqslant \lfloor \frac{|\mathsf{Nh}(\mathcal{B}'')|}{2} \rfloor + 1$. Note that $\mathcal{B}'$ is at most $\lambda - 1$ levels above $\mathcal{B}$ (as we allow $\mathcal{B}'$ to be $\mathcal{B}$). By Claim 2.2, the neighborhood of a bag can increase by at most $(\frac{2}{\delta})$ from the neighborhood of its parents, hence $\mathsf{Nh}(\mathcal{B}') \geqslant (\frac{2}{\delta}) \cdot (\lambda + 1)$. The last two inequalities lead to $|\mathsf{Nh}(\mathcal{B}'')| \geqslant 2 \cdot (\frac{2}{\delta}) \cdot \lambda$, which contradicts our choice of B.

The desired result follows. ☐

Example 2.2 (Balancing a tree decomposition). Fig. 2.5 illustrates an example of $\widehat{\mathsf{R_G}}$ constructed out of a tree-decomposition $\mathrm{Tree}'(G)$. First, $\mathrm{Tree}'(G)$ is turned into a binary and balanced tree $\mathsf{R_G}$ and then into a binary and balanced tree $\widehat{\mathsf{R_G}}$. If the numbers are pointers to bags, such that $\mathrm{Tree}'(G)$ is a tree-decomposition for G, then $\widehat{\mathsf{R_G}}$ is a binary and balanced tree-decomposition of G. The values of λ and δ are immaterial for this example, as $\widehat{\mathsf{R_G}}$ becomes perfectly balanced (i.e., $(\frac{1}{2}, 1)$-balanced).

We conclude this subsection with the proof of Theorem 2.2.

Proof of Theorem 2.2. By Theorem 2.1, an initial tree-decomposition $\mathrm{Tree}'(G)$ of G with width t and $b = O(n)$ bags can be constructed in $O(n)$ time. Lemma 2.9 and Lemma 2.10 prove that the constructed $\widehat{\mathsf{R_G}}$ is a (α, β, γ) tree-decomposition of G. The time and space complexity come from the construction of $\mathsf{R_G}$ by the recursion of Rank. It can be easily seen that every level of

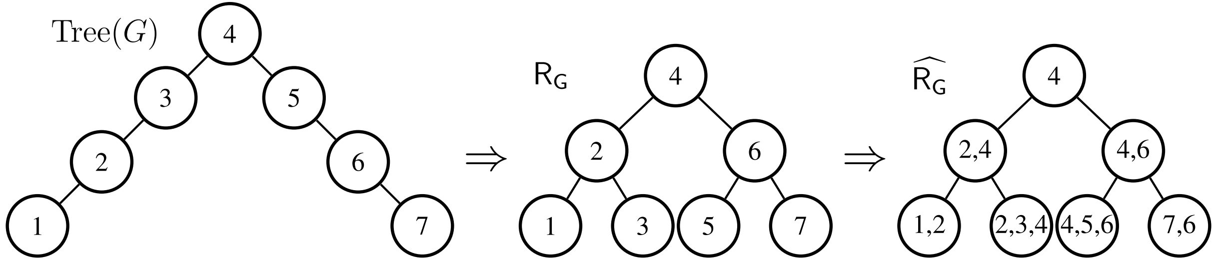


Figure 2.5: Given the tree-decomposition $\mathrm{Tree}(G)$ on the left, the graph in the middle is the corresponding $\mathsf{R_G}$ and the one on the right is the corresponding tree-decomposition $\widehat{\mathsf{R_G}} = \mathsf{Replace}(\mathsf{R_G})$ after replacing each bag B with $\mathsf{NhV}(B)$.

the recursion processes disjoint components $\mathcal{C}_i$ of $\mathrm{Tree}'(G)$ in $O(|\mathcal{C}_i|)$ time, thus one level of the recursion requires $O(b)$ time in total. There are $O(\log b)$ such levels, since every λ levels, the size of each component has been reduced to at most a factor $\left(\frac{1+\delta}{2}\right)^{\lambda-1}$. Hence the time complexity is $O(b \cdot \log b) = O(n \cdot \log n)$. The space complexity is that of processing a single level of the recursion, hence $O(b) = O(n)$. □

2.4 Recursive State Machines

Recursive State Machines (RSMs). A *single-entry single-exit recursive state machine* (RSM from now on) over an alphabet Σ, as defined in [Alur *et al.*, 2005], is a set $\mathsf{RSM} = \{A_1, A_2, \dots, A_k\}$, such that for each $1 \leqslant i \leqslant k$, the *component state machine* (CSM) $A_i = (B_i, Y_i, V_i, E_i, \mathsf{wt}_i)$, where $V_i = N_i \cup \{\mathsf{en}_i\} \cup \{\mathsf{ex}_i\} \cup \mathsf{Calls}_i \cup \mathsf{Returns}_i$, consists of:

- A set B_i of b_i *boxes*.
- A map Y_i, mapping each box in B_i to an index in $\{1, 2, \dots, k\}$. We say that a box $b \in B_i$ *corresponds* to the CSM with index $Y_i(b)$.
- A set V_i of *nodes*, consisting of the union of the sets N_i, $\{\mathsf{en}_i\}$, $\{\mathsf{ex}_i\}$, Calls_i and $\mathsf{Returns}_i$. The number n_i is the size of V_i. Each of these sets, besides V_i, are w.l.o.g. assumed to be pairwise disjoint.
  - The set N_i is the set of *internal nodes*.
  - The node en_i is the *entry node*.

   - The node ex_i is the *exit node*.
   - The set Calls_i is the set of *call nodes*. Each call node is a pair (x, b), where b is a box in B_i and x is the unique entry node $\mathsf{en}_{Y_i(b)}$ of the corresponding CSM with index $Y_i(b)$.
   - The set $\mathsf{Returns}_i$ is the set of *return nodes*. Each return node is a pair (y, b), where b is a box in B_i and y is the unique exit node $\mathsf{ex}_{Y_i(b)}$ of the corresponding CSM with index $Y_i(b)$.

- A set E_i of *internal edges*. Each edge is a pair in $(N_i \cup \{\mathsf{en}_i\} \cup \mathsf{Returns}_i) \times (N_i \cup \{\mathsf{ex}_i\} \cup \mathsf{Calls}_i)$.
- A map $\mathsf{wt}_i E_i \to \Sigma$, mapping each edge in E_i to a label in the domain Σ of the semiring S.

We let $N = \bigcup_i N_i$, $E = \bigcup_i E_i$ $B = \bigcup_i B_i$, $V = \bigcup_i V_i$, $\mathsf{En} = \{\mathsf{en}_i\}_i$, $\mathsf{Ex} = \{\mathsf{ex}_i\}_i$ $\mathsf{Calls} = \bigcup_i \mathsf{Calls}_i$ and $\mathsf{Returns} = \bigcup_i \mathsf{Returns}_i$. Additionally, we let $n_i = |N_i|$, $n = |N|$, $m = |E_i|$, $m = |E|$, and denote by $\mathsf{wt} : E \to \Sigma$ the union of all weight functions $\{\mathsf{wt}_i\}_i$.

Control-flow graphs of CSMs and the treewidth of RSMs. Given an RSM $\mathsf{RSM} = \{A_1, A_2, \ldots, A_k\}$, the *control-flow graph* $G_i = (V_i, E_i')$ of CSM A_i consists of V_i as the set of nodes and E_i' as the set of edges, where E_i' consists of the edges E_i of A_i, and for each box b, the call node (v, b) of that box (i.e. for $v = \mathsf{en}_{Y_i(b)}$) has an edge to the return node (v', b) of that box (i.e. for $v' = \mathsf{ex}_{Y_i(b)}$). We say that the RSM has *treewidth* t, if t is the smallest integer such that for each index $1 \leqslant i \leqslant k$, the graph $G_i = (V_i, E_i')$ has treewidth at most t.

It is known that the control-flow graphs of structured programs from most programming languages have constant treewidth. This was first proved in [Thorup, 1998], and later followed by other works which extend the result to other programming languages. (e.g. [Burgstaller *et al.*, 2004]).

Theorem 2.3 ([Thorup, 1998; Burgstaller *et al.*, 2004])**.** *The following bounds hold for the treewidth of control-flow graphs of programs from various programming languages.*

- *Goto-free Algol and Pascal programs have control-flow graphs of treewidth* $\leqslant 3$.
- *All Modula-2 programs have control-flow graphs of treewidth* $\leqslant 5$.

- *Goto-free C programs have control-flow graphs of treewidth* $\leqslant 6$.

- *Goto-free without labeled loops have control-flow graphs of treewidth* $\leqslant 6$.

Example 2.3 (RSM and tree decompositions). Fig. 2.6 shows an example of a program for matrix multiplication consisting of two methods (one for vector multiplication invoked by the one for matrix multiplication). The corresponding control-flow graphs, and their tree decompositions that achieve treewidth 2 are also shown in the figure.

Configurations and transitions. A *configuration* of an RSM is a pair $\mathcal{C} = (v, L)$, where v is a node in $(N \cup \{\mathsf{en}\} \cup \mathsf{Returns})$ and L is a sequence of boxes. The *stack height* $\mathsf{SH}(\mathcal{C})$ of a configuration $\mathcal{C} = (v, L)$ is the number of boxes in the sequence L. The set of *transitions* E are edges between configurations. The *global weight function* wt maps each transition in E to a label in the domain Σ of the semiring S. We have that there is a transition from configuration $\mathcal{C}_1 = (v_1, L_1)$, where $v_1 \in V_i$ for some $1 \leqslant i \leqslant k$, to configuration $\mathcal{C}_2 = (v_2, L_2)$ with label $\sigma = \mathsf{wt}(\mathcal{C}_1, \mathcal{C}_2)$ if and only if one of the following holds:

- **Internal transition:** We have that $v_2 \in N_i$ (i.e., v_2 is an internal node of A_i) and the following hold: (i) $L_1 = L_2$; and (ii) $(v_1, v_2) \in E_i$; and (iii) $\sigma = \mathsf{wt}_i((v_1, v_2))$.

- **Entry transition:** We have that $v_2 = \mathsf{en}_{Y_i(b)}$ (i.e., v_2 is the entry node of $A_{Y_i(b)}$), for some box b, and the following hold: (i) $L_1 \circ b = L_2$; and (ii) $(v_1, (v_2, b)) \in E_i$; and (iii) $\sigma = \mathsf{wt}_i((v_1, (v_2, b)))$.

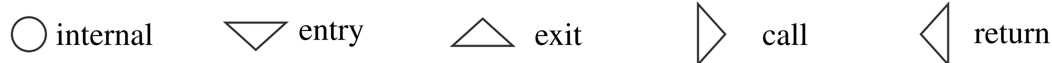
- **Return transition:** We have that $v_2 = (v, b) \in \mathsf{Returns}_j$ is a *return* node, for some exit node $v = \mathsf{ex}_i$ and some box b with $Y_j(b) = i$, and the following hold: (i) $L_1 = L_2 \circ b$; and (ii) $(v_1, v) \in E_i$; and (iii) $\sigma = \mathsf{wt}_i((v_1, v))$.

Note that in a configuration (v, L), the node v cannot be ex_i or in Calls_i. In essence, the corresponding configuration is at the corresponding return node, instead of at the exit node, or corresponding entry node, instead of at the call node, respectively.

Execution paths and stack heights. An *execution path* is a sequence of configurations and labels $\pi = \langle \mathcal{C}_1, \sigma_1, \mathcal{C}_2, \sigma_2 \ldots, \sigma_{\ell-1}, \mathcal{C}_\ell \rangle$, such that for each integer i where $1 \leqslant i \leqslant \ell - 1$, we have that $(\mathcal{C}_i, \mathcal{C}_{i+1}) \in E$ and $\sigma_i = \mathsf{wt}(\mathcal{C}_i, \mathcal{C}_{i+1})$. Occasionally our interest is only on the configurations of an execution path, in which case we simply write $\pi = \langle \mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_\ell \rangle$. The

length of π is ℓ. We say that the *stack height* $\mathsf{SH}(\pi)$ of an execution path $\pi = \langle \mathcal{C}_1, \mathcal{C}_2, \dots, \mathcal{C}_\ell \rangle$ is $\mathsf{SH}(\pi) = \max_i \mathsf{SH}(\mathcal{C}_i)$, i.e., it is the maximum stack height of a configuration in the execution path. The *additional stack height* of π is the additional height of the stack in the segment of the path, i.e., $\mathsf{ASH}(\pi) = \mathsf{SH}(\pi) - \max(\mathsf{SH}(\mathcal{C}_1), \mathsf{SH}(\mathcal{C}_\ell)$. For a pair of configurations $\mathcal{C}, \mathcal{C}'$, the set $\{\mathcal{C} \rightsquigarrow \mathcal{C}'\}$ is the set of execution paths $\langle \mathcal{C}_1, \sigma_1, \mathcal{C}_2, \sigma_2 \dots, \sigma_{\ell-1}, \mathcal{C}_\ell \rangle$, for any ℓ, where $\mathcal{C} = \mathcal{C}_1$ and $\mathcal{C}' = \mathcal{C}_\ell$. Given a set X of execution paths and some $h \in \mathbb{N}$, the set $R(X, h) \subseteq X$ is the subset of execution paths with stack height at most h. Given a complete semiring $S = (\Sigma, \oplus, \otimes, \overline{\mathbf{0}}, \overline{\mathbf{1}})$, the *weight* of a execution path $\pi = \langle \mathcal{C}_1, \sigma_1, \mathcal{C}_2, \sigma_2 \dots, \sigma_{\ell-1}, \mathcal{C}_\ell \rangle$ is $\otimes(\pi) = \bigotimes(\sigma_1, \dots, \sigma_{\ell-1})$.

The algebraic path problem for RSMs. Given configurations c, c', the *configuration semiring distance* $d(c, c')$ is defined as $d(c, c') = \bigoplus_{\pi : c \rightsquigarrow c'} \otimes(\pi)$. Given configurations c, c' and a stack height $h \in \mathbb{N}$, the *bounded-height configuration semiring distance* $d(c, c', h)$ is defined as $d(c, c', h) = \bigoplus_{\pi \in R(\{c \rightsquigarrow c'\}, h)} \otimes(\pi)$. Given a CSM A_i and two nodes $u, v \in \{\mathsf{en}\} \cup N_i \cup \mathsf{Returns}_i$, the *same-context semiring distance* $d(u, v)$ from u to v is defined as $d(u, v) = d((u, \varnothing), (v, \varnothing))$. Similarly, given a stack height $h \in \mathbb{N}$, the *bounded-height same-context distance* from u to v is defined as $d(u, v) = d((u, \varnothing), (v, \varnothing), h)$. Note that the above definition of semiring distances only allows for so called *valid* paths [Reps *et al.*, 1995a; Sagiv *et al.*, 1996], i.e., paths that fully respect the calling contexts of an execution.



Method: dot_vector

Input: $x, y \in \mathbb{R}^n$

Output: The dot product $x^\top y$

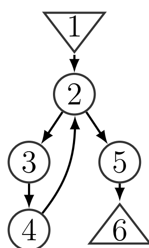
```
1 result ← 0
2 for i ← 1 to n do
3     z ← x[i] · y[i]
4     result ← result + z
5 end
6 return result
```

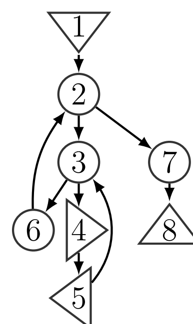
Method: dot_matrix

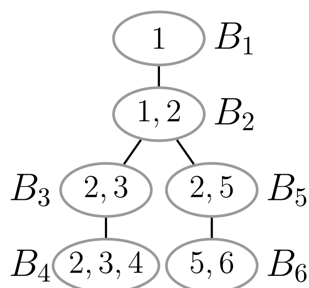
Input: $A \in \mathbb{R}^{n \times k}, B \in \mathbb{R}^{k \times m}$

Output: The dot product $A \times B$

```
1 C ← zero matrix of size n × m
2 for i ← 1 to n do
3     for j ← 1 to m do
4         Call dot_vector(A[i, :], B[:, j])
5         C[i, j] ← the value returned at Line 4
6     end
7 end
8 return C
```







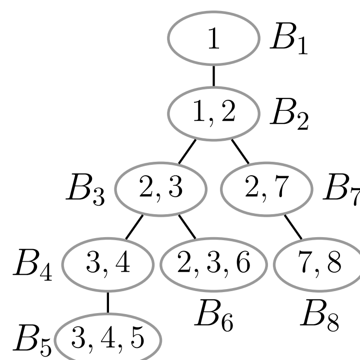


Figure 2.6: Example of a program consisting of two methods, their control-flow graphs $G_i = (V_i, E_i')$ where nodes correspond to line numbers, and the corresponding tree decompositions, each one achieving treewidth 2.

3 Semiring Distance Oracles on Low-treewidth Graphs

3.1 Introduction

In this chapter we focus on the algebraic path problem on graphs of constant treewidth. We first consider queries wrt an arbitrary complete, closed semiring S, and then focus on the important special cases of reachability (phrased on the boolean semiring) and shortest path (phrased on the tropical semiring). In all cases the input is a graph G with n nodes, and a tree-decomposition $\mathrm{Tree}(G)$ of G with $b = O(n)$ bags and width t. The computational model is the standard RAM with wordsize $W = \Theta(\log n)$.

Previous results. Semiring distances on constant-treewidth graphs have been previously considered in [Hagerup, 2000]. The algorithmic question of the distance (pair, single-source, all pairs) problem wrt the tropical semiring for low-treewidth graphs has been considered extensively in the literature, and many data structures have been presented [Akiba *et al.*, 2012; Chaudhuri and Zaroliagis, 1995; Planken *et al.*; Akiba *et al.*, 2013; Bauer *et al.*, 2013; Columbus, 2012]. The previous results are incomparable, in the sense that the best data structure depends on the treewidth and the number of queries. The pair reachability query for low-treewidth graphs has been considered in [Yano *et al.*, 2013]. Despite many results for constant-(or low-) treewidth graphs, none of them improves the complexity for the basic single-source reachability problem, i.e., the bound for DFS/BFS has not been improved in any of the previous works.

Our results. Our algorithms take as input a graph G with n nodes and treewidth t, and a tree-decomposition $\mathrm{Tree}(G)$ of $O(n)$ bags and width $O(t)$. Our main contributions are as follows

Preprocessing time	**Update time**	**Query time**
$O(n)$	$O(\log n)$	$O(\log n)$

Table 3.1: A data structure for handling single-source and pair semiring distance queries on a graph G of n nodes and constant treewidth.

(summarized in Tables 3.1 to 3.3):

1. Our first contribution is a data structure for handling semiring distances in G. It supports preprocessing G in $O(n)$ time, after which it can handle weight updates in $O(\log n)$ time and pair semiring distance queries in $O(\log n)$ time each.

2. Our second contribution is a data structure that supports reachability queries in G. The computational complexity we achieve is as follows: (i) $O(n \cdot t^2)$ preprocessing (construction) time; (ii) $O(n \cdot t)$ space; (iii) $O(\lceil t/\log n \rceil)$ pair-query time; and (iv) $O(n \cdot t/\log n)$ time for single-source queries. Note that for constant-treewidth graphs, the data structure is *optimal* in the sense that it only uses linear preprocessing time, and supports answering queries in the size of the output (the output for single-source queries requires one bit per node, and thus has size $\Theta(n/W) = \Theta(n/\log n)$). Moreover, also for constant-treewidth graphs, the data structure answers single-source queries faster than DFS/BFS, after linear preprocessing time (which is asymptotically the same as for DFS/BFS). Thus there exists a constant c_0 such that the total of the preprocessing and querying time of the data structure is smaller than that of DFS/BFS for answering at least c_0 single-source queries.

3. Our third contribution is a space-time tradeoff data structure that supports distance pair queries in G and given a number $\epsilon \in [\frac{1}{2}, 1]$. The weights of G come from the set of integers $\mathbb{Z}$, but we do not allow negative cycles. For constant-treewidth graphs, our data structure requires (i) polynomial preprocessing time; (ii) $O(n^{\epsilon})$ working space; and (iii) $O(n^{1-\epsilon} \cdot \alpha(n))$ time for pair queries.

Technical contributions. Our results rely on three key technical contributions:

1. Our data structure for general semiring distances relies on the newly introduced notion of U-shaped paths. Informally, given a bag B of a tree-decomposition T, a path is U-shaped in B if all its intermediate nodes are contained in the subtree of T rooted at

	Preprocessing time	Space	Query		Reference
			Single-source	Pair	
	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(\log n)$	[Yano *et al.*, 2013]
	$O(n)$	$O(n)$	$O(n)$	$O(\alpha(n))$	[Chaudhuri and Zaroliagis, 1995]
DFS/BFS	–	$O(n)$	$O(m)$	$O(m)$	[Cormen *et al.*, 2009]
Our Result	$O(n)$	$O(n)$	$O\left(\frac{n}{\log n}\right)$	$O(1)$	Theorem 3.2

Table 3.2: Data structures for pair and single-source reachability queries, on a directed graph G with n nodes, m edges, and a treewidth t. The model of computation is the standard RAM model with wordsize $W = \Theta(\log n)$. We denote by $\alpha(n)$ the inverse of the Ackermann function on input n. Space usage refers to the total space used during the preprocessing and query phase.

B. In the preprocessing and update phases, the data structure maintains the semiring distance between every pair of nodes (u, v), restricted to U-shaped paths (i.e., the weight of the smallest-weight U-shaped path from u to v). In the query phase, the data structure combines $O(\log n)$ such "U-shaped distances" to obtain the semiring distance between nodes.

2. For pair reachability queries, the key idea is to store reachability information from each node to $O(\log n)$ other nodes. For single-source queries, for some nodes this reachability information might be of size $\Theta(n)$, but on average remains $O(\log n)$. Our data structure computes reachability information in such a way that allows for compact representation and fast retrieval using word tricks, which, for constant-treewidth graphs leads to asymptotically optimal preprocessing and query (both pair and single-source) bounds. The idea of storing $O(\log n)$ information per node has appeared before ([Yano *et al.*, 2013; Chaudhuri and Zaroliagis, 1995]) however those algorithms follow different approaches, where word tricks do not seem to be applicable (at least not without significantly modifying the algorithms).

3. For distance queries, we devise a procedure for shrinking a tree-decomposition of size $O(n)$ to one of size $O(n^{1-\epsilon})$, by partitioning the tree-decomposition to components of sufficient size. A key property of this partitioning is that each component has only a constant number of neighbor components. We show how this shrank tree-decomposition

Row	Preprocessing time	Space usage	Pair query time	Single-source query time	From
1	$O(n^2)$	$O(n^2)$	$O(1)$	$O(n)$	[Planken *et al.*] [a]
2	$O(n)$	$O(n)$	$O(\alpha(n))$	$O(n)$	[Chaudhuri and Zaroliagis, 1995]
3	$O(n \cdot \log h)$	$O(n)$	$O(\log\log n)$	$O(n \cdot \log\log n)$ [b]	[Akiba *et al.*, 2012]
4	$O(n \cdot \log^2 n)$	$O(n \cdot \log n)$	$O(\log n)$	$O(n \cdot \log n)$[b]	[Akiba *et al.*, 2013]
5	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(\log^2 n)$	$O(n \cdot \log^2 n)$ [b]	[Bauer *et al.*, 2013; Columbus, 2012]
6	Not given	$O(n^{\epsilon} \cdot \log^2 n)$ [c]	$O(n^{1-\epsilon} \cdot \log n)$	– [d]	[Akiba *et al.*, 2012] [e]
Our Result	Polynomial	$O(n^{\epsilon})$	$O(n^{1-\epsilon} \cdot \alpha(n))$	– [d]	Theorem 3.3

Table 3.3: Data structures for pair and single-source distance queries, on a weighted directed graph G with n nodes, m edges, and a tree decomposition of width $O(1)$ and height h. The number ϵ can be any fixed number in $[\frac{1}{2}, 1]$ and $\alpha(n)$ is the inverse Ackermann function. Space usage refers to the total space used during the preprocessing and query phase. When measuring space complexity, we do not count the input size. Rows 1-6 are previous results.

[a] This data structure solves the all pairs problem in the given time and space bounds.

[b] Obtained by multiplying the time for a pair query by n.

[c] This is the space usage after preprocessing.

[d] Not given/supported since the size of the output is larger than the data structure.

[e] Note that [Akiba *et al.*, 2012] does not explicitly state the tradeoff given (they only state linear space), but it follows from their technique by picking other values for their variable k. Also, note that [Akiba *et al.*, 2012] requires a tree-decomposition to be part of the input, whereas our data structure only requires that the graph G is part of the input.

can be preprocessed for answering pair distance queries in the stated bounds.

Organization. The rest of this chapter is organized as follows.

1. In Section 3.2 we present our dynamic data structure for handling general semiring distance queries on graphs of low treewidth, with also supporting weight updates.

2. In Section 3.3 we present our data structure for handling reachability queries on graphs of low treewidth.

3. In Section 3.4 we present our data structure for handling distance queries (wrt the tropical semiring) on graphs of low treewidth, while providing a space-time tradeoff between the

space used in preprocessing and the query time.

3.2 Dynamic Algorithms for Preprocess, Update and Query

In the current section we present a data structure that takes as input a weighted graph $G = (V, E, \mathsf{wt})$ of n nodes, treewidth $t = O(1)$ and weight function $\mathsf{wt} : E \to \Sigma$ over a complete, closed semiring S, and a nicely rooted, balanced, binary tree decomposition $\mathrm{Tree}(G)$ of width $O(t)$, and achieves the following tasks:

1. Preprocessing the tree-decomposition $\mathrm{Tree}(G)$ to semiring distance queries fast.
2. Updating the preprocessed $\mathrm{Tree}(G)$ upon change of the weight $\mathsf{wt}(u, v)$ of an edge (u, v).
3. Querying the preprocessed $\mathrm{Tree}(G)$ to retrieve the distance $d(u, v)$ of any pair of nodes (u, v).

3.2.1 Algorithms Preprocess, Update and Query

Intuition and U-shaped paths. A central concept in our algorithms is that of U-shaped paths. Given a bag B and nodes $u, v \in B$ we say that a path $P : u \rightsquigarrow v$ is U-shaped in B, if one of the following conditions hold:

1. Either $|P| > 1$ and for all intermediate nodes $w \in P$, we have that B is an ancestor of B_w,
2. or $|P| \leqslant 1$ and B is B_u or B_v (i.e., B is the root bag of u or v).

Informally, given a bag B, a U-shaped path in B is a path that traverses intermediate nodes whose root bag is either B or some descendant bag of B in $\mathrm{Tree}(G)$. In the following we present three algorithms for (i) preprocessing a tree decomposition, (ii) updating the data structures of the preprocessing upon a weight change $\mathsf{wt}(u, v)$ of an edge (u, v), and (iii) querying for the distance $d(u, v)$ for any pair of nodes u, v. The intuition behind the overall approach is that for every path $P : u \rightsquigarrow v$ and $z = \mathsf{argmin}_{x \in P} \mathsf{Lv}(x)$, the path P can be decomposed to paths $P_1 : u \rightsquigarrow z$ and $P_2 : z \rightsquigarrow v$. By Lemma 2.1, if we consider the path $P' : B_u \rightsquigarrow B_z$ and any bag $B_i \in P'$, we can find nodes $x, y \in B_i \cap P_1$ (not necessarily distinct). Then P_1 is decomposed to a sequence of

U-shaped paths P_1^i, one for each such B_i, and the weight of P_1 can be written as the $\otimes$-product of the weights of P_1^i, i.e., $\otimes(P_1) = \bigotimes(\otimes(P_1^i))$. A similar observation holds for P_2. Hence, the task of preprocessing and updating is to summarize in each B_i the weights of all such U-shaped paths between all pairs of nodes appearing in B_i. To answer the query, the algorithm traverses upwards the tree $\mathrm{Tree}(G)$ from B_u and B_v, and combines the summarized paths to obtain the weights of all such paths P_1 and P_2, and eventually P, such that $\otimes(P) = d(u, v)$.

Informal description of preprocessing. Algorithm Preprocess (Algorithm 2) associates with each bag B a *local* U-*shaped distance* map $\mathsf{LUD}_B : B \times B \to \Sigma$. Upon a weight change, algorithm Update (Algorithm 3) updates the local U-shaped distance map of some bags. It will hold that after the preprocessing and each subsequent update, $\mathsf{LUD}_B(u, v) = \bigoplus_{P:u \rightsquigarrow v}\{\otimes(P)\}$, where all P are U-shaped paths in B. Given this guarantee, we later present algorithm Query (Algorithm 5) for answering (u, v) queries with $d(u, v)$, the distance from u to v.

Algorithm Preprocess is a dynamic programming algorithm. It traverses $\mathrm{Tree}(G)$ bottom-up, and for a currently examined bag B that is the root bag of a node x, it calls Merge to compute the local U-shaped distance map LUD_B. In turn, Merge computes LUD_B depending only on the local U-shaped distance maps LUD_{B_i} of the children $\{B_i\}$ of B, and uses the closure operator $*$ to capture possibly unbounded traversals of cycles whose smallest-level node is x. See Algorithms 1 and 2 for a formal description.

Algorithm 1: Merge

Input: A bag B_x with children $\{B_i\}_i$

Output: A local U-shaped distance map LUD_{B_x}

1 Assign $\mathsf{wt}'(x, x) \leftarrow \left(\bigotimes\{\mathsf{LUD}_{B_1}(x, x)^*, \dots, \mathsf{LUD}_{B_j}(x, x)^*\}\right)^*$
2 **foreach** $u \in B_x$ *with* $u \neq x$ **do**
3 | Assign $\mathsf{wt}'(x, u) \leftarrow \bigoplus\{\mathsf{wt}(x, u), \mathsf{LUD}_{B_1}(x, u), \dots, \mathsf{LUD}_{B_j}(x, u)\}$
4 | Assign $\mathsf{wt}'(u, x) \leftarrow \bigoplus\{\mathsf{wt}(u, x), \mathsf{LUD}_{B_1}(u, x), \dots, \mathsf{LUD}_{B_j}(u, x)\}$
5 **end**
6 **foreach** $u, v \in B_x$ **do**
7 | Assign $\delta \leftarrow \bigotimes(\mathsf{wt}'(u, x), \mathsf{wt}'(x, x), \mathsf{wt}'(x, v))$
8 | Assign $\mathsf{LUD}_{B_x}(u, v) \leftarrow \bigoplus\{\delta, \mathsf{LUD}_{B_1}(u, v), \dots, \mathsf{LUD}_{B_j}(u, v)\}$
9 **end**

Algorithm 2: Preprocess

Input: A tree-decomposition $\mathrm{Tree}(G) = (V_T, E_T)$

Output: A local U-shaped distance map LUD_B for each bag $B \in V_T$

1 Traverse $\mathrm{Tree}(G)$ bottom up and examine each bag B with children $\{B_i\}_i$

2 **if** *B is the root bag of some node x* **then**

3 | Assign $\mathsf{LUD}_B \leftarrow$ Merge on B

4 **else**

5 | **foreach** $u, v \in B$ **do**

6 | | Assign $\mathsf{LUD}_B(u,v) \leftarrow \bigoplus\{\mathsf{LUD}_{B_1}(u,v), \dots, \mathsf{LUD}_{B_j}(u,v)\}$

7 | **end**

8 **end**

Lemma 3.1. *At the end of* Preprocess*, for every bag B and nodes $u, v \in B$, we have* $\mathsf{LUD}_B(u,v) = \bigoplus_{P:u\rightsquigarrow v}\{\otimes(P)\}$*, where all P are* U*-shaped paths in B.*

Proof. The proof is by induction on the parents. Initially, B is a leaf, and root of some node x, thus each such path P can only go through x, and hence will be captured by Preprocess. Now assume that the algorithm examines a bag B, and by the induction hypothesis the statement is true for all $\{B_i\}$ children of B_x. The correctness follows easily if B is not the root bag of any node, since every such P is a U-shaped path in some child B_i of B. Now consider that B is the root bag some node x, and any U-shaped path $P' : u \rightsquigarrow v$ that additionally visits x, and decompose it to paths $P_1 : u \rightsquigarrow x$, $P_2 : x \rightsquigarrow x$ and $P_3 : x \rightsquigarrow v$, such that x is not an intermediate node in either P_1 or P_3, and we have by distributivity:

$$\begin{aligned}\bigoplus_{P'} \otimes(P') &= \bigoplus_{P_1,P_2,P_3} \bigotimes(\otimes(P_1), \otimes(P_2), \otimes(P_3)) \\ &= \bigotimes\left(\bigoplus_{P_1}\otimes(P_1), \bigoplus_{P_2}\otimes(P_2), \bigoplus_{P_3}\otimes(P_3)\right)\end{aligned}$$

Note that P_1 and P_3 are also U-shaped in one of the children bags B_i of B_x, hence by the induction hypothesis in Line 3 and Line 2 of Merge we have $\mathsf{wt}'(u,x) = \bigoplus_{P_1}\otimes(P_1)$ and $\mathsf{wt}'(x,v) = \bigoplus_{P_3}\otimes(P_3)$. Also, by decomposing P_2 into a (possibly unbounded) sequence of paths $P_2^i : x \rightsquigarrow x$ such that x is not intermediate node in any P_2^i, we get that each such P_2^i is a

U-shaped path in some child B_{l_i} of B, and we have by distributivity and the induction hypothesis

$$\begin{aligned}\bigoplus_{P_2} \otimes(P_2) &= \bigoplus_{P_2^1, P_2^2, \dots} \bigotimes \left(\otimes(P_2^1), \otimes(P_2^2), \dots\right) \\ &= \bigoplus_{B_{l_1}, B_{l_2}, \dots} \bigotimes \left(\bigoplus_{P_2^1} \otimes(P_2^1), \bigoplus_{P_2^2} \otimes(P_2^2), \dots\right) \\ &= \bigoplus_{B_{l_1}, B_{l_2}, \dots} \bigotimes \left(\mathsf{LUD}_{B_{l_1}}(x,x), \mathsf{LUD}_{B_{l_2}}(x,x), \dots\right)\end{aligned}$$

and the last expression equals $\mathsf{wt}'(x,x)$ from Algorithm 1 of Merge. The above conclude that in Line 6 of Merge we have $\delta = \bigoplus_{P'} \otimes(P')$.

Finally, each U-shaped path $P : u \rightsquigarrow v$ in B either visits x, or is U-shaped in one of the children B_i. Hence after Line 8 of Merge has run on B, for all $u, v \in B$ we have that $\mathsf{LUD}_B(u,v) = \bigoplus_{P:u\rightsquigarrow v} \otimes(P)$ where all paths P are U-shaped in B. The desired results follows. □

Lemma 3.2. Preprocess *requires* $O(n)$ *semiring operations.*

Proof. Merge requires $O(t^2) = O(1)$ operations, and Preprocess calls Merge at most once for each bag, hence requiring $O(n)$ operations. □

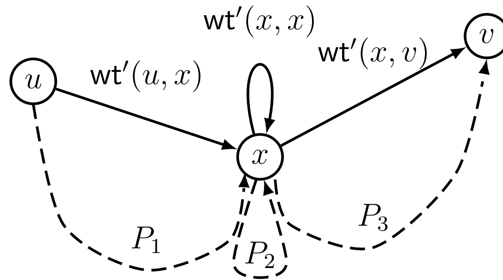


Figure 3.1: Illustration of the inductive argument of Preprocess.

Informal description of updating. Algorithm Update is called whenever the weight $\mathsf{wt}(x,y)$ of an edge of G has changed. Given the guarantee of Lemma 3.1, after Update has run on an edge update $\mathsf{wt}(x,y)$, it restores the property that for each bag B we have $\mathsf{LUD}_B(u,v) = \bigoplus_{P:u\rightsquigarrow v}\{\otimes(P)\}$, where all P are U-shaped paths in B. See Algorithm 3 for a formal description.

Lemma 3.3. *At the end of each run of* Update*, for every bag B and nodes $u, v \in B$, we have* $\mathsf{LUD}_B(u,v) = \bigoplus_{P:u\rightsquigarrow v}\{\otimes(P)\}$*, where all P are* U*-shaped paths in B.*

Algorithm 3: Update

Input: An edge (x, y) with new weight $\mathsf{wt}(x, y)$

Output: A local U-shaped distance map LUD_B for each bag $B \in V_T$

1 Assign $B \leftarrow B_{(x,y)}$, the highest bag containing the edge (x, y)
2 **repeat**
3   Call Merge on B
4   Assign $B \leftarrow B'$ where B' is the parent of B
5 **until** $\mathsf{Lv}(B) = 0$

Proof. First, by the definition of a U-shaped path P in B it follows that the statement holds for all bags not processed by Update, since for any such bag B and U-shaped path P in B, the path P cannot traverse (u, v). For the remaining bags, the proof follows an induction on the parents updated by Update, similar to that of Lemma 3.1. □

Lemma 3.4. Update *requires* $O(\log n)$ *operations per update.*

Proof. Merge requires $O(t^2) = O(1)$ operations, and Update calls Merge once for each bag in the path from $B_{(u,v)}$ to the root. Recall that the height of $\mathrm{Tree}(G)$ is $O(\log n)$, and the result follows. □

Informal description of querying. Algorithm Query answers a (u, v) query with the distance $d(u, v)$ from u to v. Because of Lemma 2.2, every path $P : u \rightsquigarrow v$ is guaranteed to go through the least common ancestor (LCA) B_L of B_u and B_v, and possibly some of the ancestors B of B_L. Given this fact, algorithm Query uses the procedure Climb to climb up the tree from B_u and B_v until it reaches B_L and then the root of $\mathrm{Tree}(G)$. For each encountered bag B along the way, it computes maps $\delta_u(w) = \bigoplus_{P_1}\{\otimes(P_1)\}$, and $\delta_v(w) = \bigoplus_{P_2}\{\otimes(P_2)\}$ where all $P_1 : u \rightsquigarrow w$ and $P_2 : w \rightsquigarrow v$ are such that the root bag of each intermediate node y is a descendant of B. This guarantees that for path P such that $d(u, v) = \otimes(P)$, when Query examines the bag B_z that is the root bag of $z = \mathsf{argmin}_{x \in P} \mathsf{Lv}(x)$, it will be $d(u, v) = \bigotimes(\delta_u(z), \delta_v(z))$. Hence, for Query it suffices to maintain a current best solution δ, and update it with $\delta \leftarrow \bigoplus\{\delta, \bigotimes(\delta_u(x), \delta_v(x))\}$ every time it examines a bag B that is the root bag of some node x. Fig. 3.2 presents a pictorial illustration of Query and its correctness. Algorithm 4 presents the Climb procedure which, given a current distance map of a node δ, a current bag B and a flag Up, updates δ with the distance to

(if $\mathsf{Up} = \mathsf{True}$), or from (if $\mathsf{Up} = \mathsf{False}$) each node in B. See Algorithm 4 and Algorithm 5 for a formal description.

Algorithm 4: Climb

Input: A bag B, a map δ, a flag Up

Output: A new map δ

```
1 Remove from δ all w ∉ B
2 Assign δ(w) ← 0̄ for all w ∈ B and not in δ
3 if B is the root bag of some node x then
4 |  if Up then                                                  /* Climbing up */
5 |  |  Update δ with δ(w) ← ⊕{δ(w), ⊗(δ(x), LUD_B(x, w))}
6 |  else                                                        /* Climbing down */
7 |  |  Update δ with δ(w) ← ⊕{δ(w), ⊗(δ(x), LUD_B(w, x))}
8 |  end
9 return δ
```

Lemma 3.5. Query *returns* $\delta = d(u, v)$.

Proof. Let $P : u \rightsquigarrow v$ be any path from u to v, and $z = \mathsf{argmin}_{x \in P} \mathsf{Lv}(x)$ the lowest level node in P. Decompose P to $P_1 : u \rightsquigarrow z$, $P_2 : z \rightsquigarrow v$, and it follows that $\otimes(P) = \bigotimes(\otimes(P_1), \otimes(P_2))$. We argue that when Query examines B_z, it will be $\delta_u(z) = \bigoplus_{P_1} \otimes(P_1)$ and $\bigoplus_{P_2} \delta_v(z) = \otimes(P_2)$. We only focus on the $\delta_u(z)$ case here, as the $\delta_v(z)$ is similar. We argue inductively that when algorithm Query examines a bag B_x, for all $w \in B_x$ we have $\delta_u(w) = \bigoplus_{P'} \{\otimes(P')\}$, where all P' are such that for each intermediate node y we have $\mathsf{Lv}(y) \geqslant \mathsf{Lv}(x)$. Initially (Line 1), it is $x = u$, $B_x = B_u$, and every such P' is U-shaped in B_u, hence $\mathsf{LUD}_{B_x}(x, w) = \bigoplus_{P'} \{\otimes(P')\}$ and $\delta_u(w) = \bigoplus_{P'} \{\otimes(P')\}$. Now consider that Query examines a bag B_x (Lines 7 and 18) and the claim holds for $B_{x'}$ a descendant of B_x previously examined by Query. If x does not occur in P', it is a consequence of Lemma 2.2 that $w \in B_{x'}$, hence by the induction hypothesis, P' has been considered by Query. Otherwise, x occurs in P' and decompose P' to P'_1, P'_2, such that P'_1 ends with the first occurrence of x in P', and it is $\otimes(P) = \bigotimes(\otimes(P'_1), \otimes(P'_2))$. Note that P'_2 is a U-shaped path in B_x, hence $\mathsf{LUD}_{B_x}(x, w) = \bigoplus_{P'_2} \{\otimes(P'_2)\}$. Finally, as a consequence of Lemma 2.2, we have that $x \in B_{x'}$, and by the induction hypothesis, $\delta_u(x) = \bigoplus_{P'_1} \{\otimes(P'_1)\}$. It follows that after Query processes B_x, it will be $\delta_u(w) = \bigoplus_{P'} \{\otimes(P')\}$. By the choice of z, when Query examines the bag B_z, it will be $\delta_u(z) = \bigoplus_{P_1} \{\otimes(P_1)\}$. A similar argument shows that

Algorithm 5: Query

Input: A pair (u, v)

Output: The distance $d(u, v)$ from u to v

```
1  Initialize map δ_u with δ_u(w) ← LUD_{B_u}(u, w)
2  Initialize map δ_v with δ_v(w) ← LUD_{B_v}(w, v)
3  Assign B_L ← the LCA of B_u, B_v in Tree(G)
4  Assign B ← B_u
5  repeat
6  |   Assign B ← B' where B' is the parent of B
7  |   Call Climb on B and δ_u with flag Up set to True
8  until B = B_L
9  Assign B ← B_v
10 repeat
11 |   Assign B ← B' where B' is the parent of B
12 |   Call Climb on B and δ_v with flag Up set to False
13 until B = B_L
14 Assign B ← B_L
15 Assign δ ← ⊕_{x∈B_L} ⊗(δ_u(x), δ_v(x))
16 repeat
17 |   Assign B ← B' where B' is the parent of B
18 |   Call Climb on B and δ_u with flag Up set to True
19 |   Call Climb on B and δ_v with flag Up set to False
20 |   if B is the root bag of some node x then
21 |   |   Assign δ ← ⊕{δ, ⊗(δ_u(x), δ_v(x))}
22 until Lv(B) = 0
23 return δ
```

at that point it will also be $\delta_v(z) = \bigoplus_{P_2}\{\otimes(P_2)\}$, hence at that point $\delta = \bigotimes(\otimes(P_1), \otimes(P_2)) = d(u,v)$. □

Lemma 3.6. Query *requires* $O(\log n)$ *semiring operations.*

Proof. Climb requires $O(t^2) = O(1)$ operations and Query calls Climb once for every bag in the paths from B_u and B_v to the root. Recall that the height of $\mathrm{Tree}(G)$ is $O(\log n)$, and the result follows. □

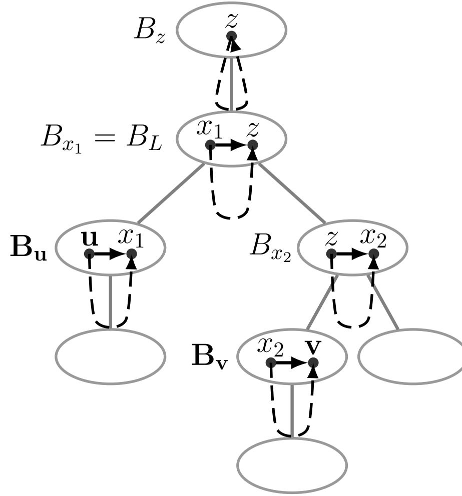


Figure 3.2: Illustration of Query in computing the distance $d(u,v) = \otimes(P)$ as a sequence of U-shaped paths, whose weight has been captured in the local distance map of each bag. When B_z is examined, with $z = \mathsf{argmin}_{x\in P}\mathsf{Lv}(x)$, it will be $\delta_u(z) = d(u,z)$ and $\delta_v(z) = d(z,v)$, and hence by distributivity $d(u,v) = \bigotimes(\delta_u(z), \delta_v(z))$.

We summarize the results of this section in the following theorem.

Theorem 3.1. *Consider a graph* $G = (V,E)$ *of* n *nodes and constant treewidth, and a nicely rooted, balanced, binary tree decomposition* $\mathrm{Tree}(G)$ *of constant width. The following assertions hold:*

1. Preprocess *requires* $O(n)$ *semiring operations;*

2. Update *requires* $O(\log n)$ *semiring operations per edge-weight update; and*

3. Query *correctly answers distance queries using* $O(\log n)$ *semiring operations.*

Witness paths. Our algorithms so far have only been concerned with returning the distance $d(u,v)$ of the pair query u, v. In several cases of semirings (e.g. Boolean, Tropical), the distance

$d(u, v)$ is realized by a single acyclic path. Then, it is straightforward to also obtain a witness path, i.e., a path $P : u \rightsquigarrow v$ such that $\otimes(P) = d(u, v)$, with some minor additional preprocessing. Here we outline how.

Whenever Merge updates the local U-shaped distance $\mathsf{LUD}_B(u, v)$ between two nodes in a bag B, it does so by considering the distances to and from an intermediate node x. It suffices to remember that intermediate node for every such local U-shaped distance. Then, the witness path to a local U-shaped distance in B can be obtained straightforwardly by a top-down computation on $\mathrm{Tree}(G)$ starting from B. Recall that in essence, Query answers a distance query u, v by combining several local U-shaped distances along the paths $B_u \rightsquigarrow B_z$ and $B_z \rightsquigarrow B_v$, where z is the node with the minimum level in a path $P : u \rightsquigarrow v$ such that $\otimes(P) = d(u, v)$. Since from every such local U-shaped distance a witness sub-path P_i can be obtained, P is reconstructed by juxtaposition of all such P_i. Finally, this process costs $O(|P|)$ time.

3.3 Optimal Reachability for Low-Treewidth Graphs

In this section we present algorithms for building and querying a data structure $\mathsf{Reachability}$, which handles single-source and pair reachability queries over an input a weighted graph G of n nodes and treewidth t.

Intuition. Informally, the preprocessing consists of first obtaining a small, balanced and binary tree-decomposition T of G, and computing the local reachability information in each bag B (i.e., the pairs $(u, v) \in E^*$ with $u, v \in B$) using Lemma 2.4. Then, the whole of preprocessing is done on T, by constructing two types of sets, which are represented as bit sequences and packed into words of length $W = \Theta(\log n)$. Initially, every node u receives an *index* i_u, such that for every bag B, the indices of nodes whose root bag is in $T(B)$ form a contiguous interval. Additionally, for every appearance of node u in a bag B, the node u receives a *local index* l_u^B in B. For brevity, a sequence $(A^0, A^1, \dots A^k)$ will be denoted by $(A^i)_{0 \leqslant i \leqslant k}$. When k is implied, we simply write $(A^i)_i$. The following two types of sets are constructed.

1. Sets that store information about subtrees. Specifically, for every node u, the set F_u stores the relative indices of nodes v that can be reached from u, and whose root bag is in $T(B_u)$. These sets are used to answer single-source queries.

2. Sets that store information about ancestors. Specifically, for every node u, two sequences of sets are stored $(\mathsf{F}_u^i)_{0 \leqslant i \leqslant \mathsf{Lv}(u)}$, $(\mathsf{T}_u^i)_{0 \leqslant i \leqslant \mathsf{Lv}(u)}$, such that F_u^i (resp., T_u^i) contains the local indices of nodes v in the ancestor bag B_u^i of B_u at level i, such that $(u, v) \in E^*$ (resp., $(v, u) \in E^*$). These sets are used to answer pair queries.

The sets of the first type are constructed by a bottom-up pass, whereas the sets of the second type are constructed by a top-down pass. Both passes are based on the separator property of tree decompositions (recall Lemmas 2.1 and 2.2), which informally states that reachability properties between nodes in distant bags will be captured transitively, through nodes in intermediate bags.

Reachability **Preprocessing.** We now give a formal description of the preprocessing of Reachability that takes as input a graph G of n nodes and treewidth t, and a balanced tree-decomposition $T = \mathrm{Tree}(G)$ of width $O(t)$. After the preprocessing, Reachability supports single-source and pair reachability queries. We say that we "insert" set A to set A' meaning that we replace A' with $A \cup A'$. Sets are represented as bit sequences where 1 denotes membership in the set, and the operation of inserting a set A "at the i-th position" of a set A' is performed by taking the bit-wise logical OR between A and the segment $[i, i + |A|]$ of A'. The preprocessing consists of the following steps.

1. Turn T to a small, balanced binary tree-decomposition of G of width $O(t)$, using Lemma 2.7.

2. Preprocess T to answer LCA queries in $O(1)$ time [Harel and Tarjan, 1984].

3. Use Lemma 2.4 to compute the local distance map $\mathsf{LD}_B : B \times B \to \{0, 1\}$ for every bag B w.r.t reachability, i.e., for any bag B and nodes $u, v \in B$, we have $\mathsf{LD}_B(u, v) = 1$ iff $(u, v) \in E^*$.

4. Apply a preorder traversal on T, and assign an incremental index i_u to each node u at the time the root bag B of u is visited. If there are multiple nodes u for which B is the root bag, assign the indices to those nodes in some arbitrary order. Additionally, store the number s_u of nodes whose root bag is in $T(B)$ and have index at least i_u. Finally, for each bag B and $u \in B$, assign a unique local index l_u^B to u, and store in B the number of nodes (with multiplicities) a_B contained in all ancestors of B, and the number b_B of nodes in B.

5. For every node u, initialize a bit set F_u of length s_u, pack it into words, and set the first bit

to 1.

6. Traverse T bottom-up, and for every bag B execute the following step. For every pair of nodes $u, v \in B$ such that B is the root bag of v and $i_u < i_v$ and $\mathsf{LD}_B(u, v) = 1$, insert F_v to the segment $[i_v - i_u, i_v - i_u + s_v]$ of F_u (the nodes reachable from v now become reachable from u, through v).

7. For every node u initialize two sequences of bit sets $(\mathsf{T}^i_u)_{0\leqslant i\leqslant \mathsf{Lv}(u)}$, $(\mathsf{F}^i_u)_{0\leqslant i\leqslant \mathsf{Lv}(u)}$, and pack them into consecutive words. Each set T^i_u and F^i_u has size $b_{B^i_u}$, where B^i_u is the ancestor of B_u at level i.

8. Traverse T top-down, and for B the bag currently visited, for every node $x \in B$, maintain two sequences of bit sets $(\overline{\mathsf{T}}^i_x)_{0\leqslant i\leqslant \mathsf{Lv}(B)}$ and $(\overline{\mathsf{F}}^i_x)_{0\leqslant i\leqslant \mathsf{Lv}(B)}$. Each set $\overline{\mathsf{T}}^i_x$ and $\overline{\mathsf{F}}^i_x$ has size b_{B^i}, where B^i is the ancestor of B at level i. Initially, B is the root of T (hence $\mathsf{Lv}(B) = 0$), and set the position l^B_w of $\overline{\mathsf{F}}^0_x$ (resp., $\overline{\mathsf{T}}^0_x$) to 1 for every node w such that $\mathsf{LD}_B(x, w) = 1$ (resp., $\mathsf{LD}_B(w, x) = 1$). For each other bag B encountered in the traversal, do as follows. Let $S = B \cap B'$, where B' is the parent of B in T, and let x range over S.

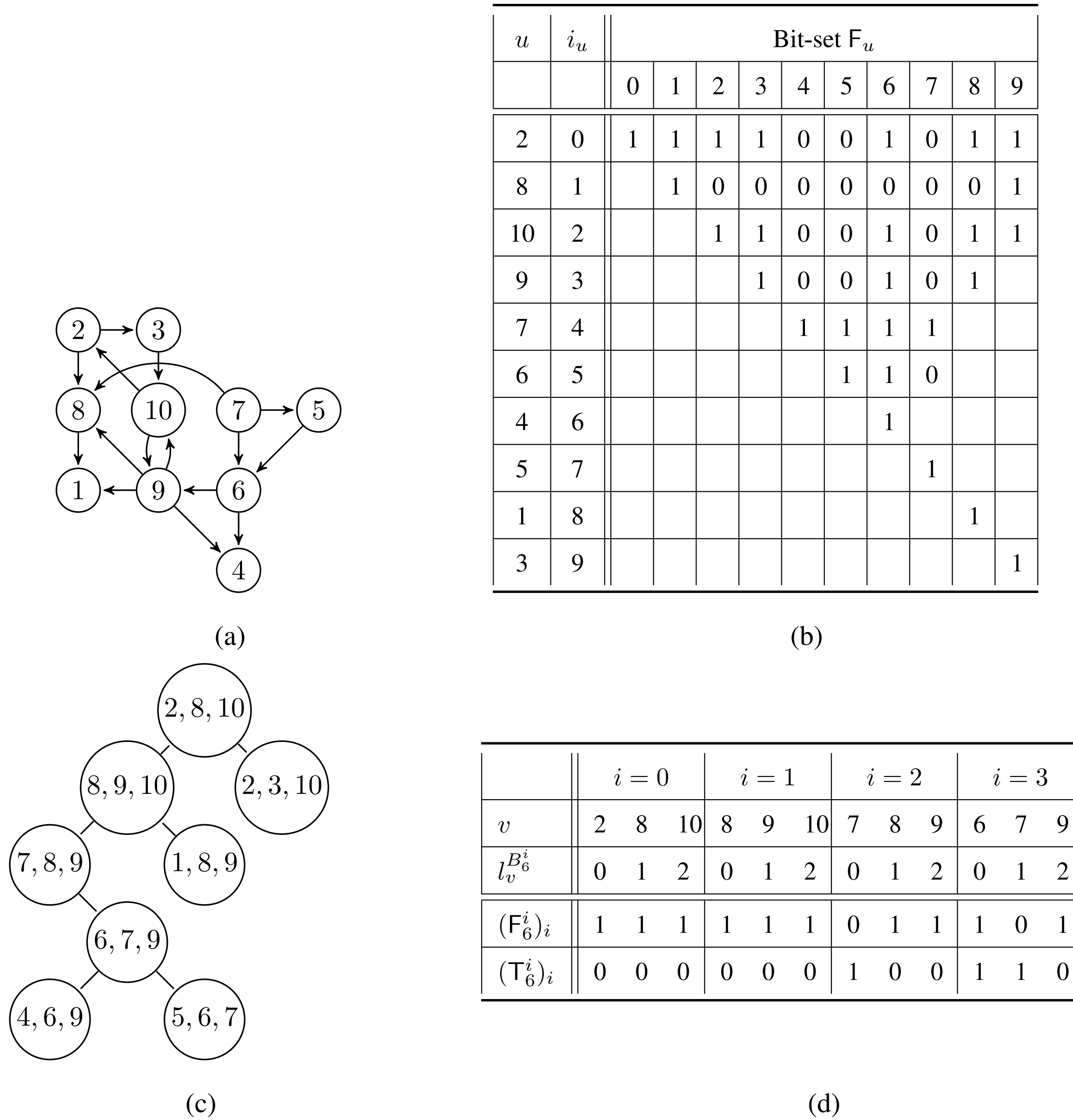
   (a) For each node x, create a set $\overline{\mathsf{T}}_x$ (resp., $\overline{\mathsf{F}}_x$) of 0s of length b_B, and for every $w \in B$ such that $\mathsf{LD}_B(x, w) = 1$ (resp., $\mathsf{LD}_B(w, x) = 1$), set the l^B_w-th bit of $\overline{\mathsf{F}}_x$ (resp., $\overline{\mathsf{T}}_x$) to 1. Append the set $\overline{\mathsf{T}}_x$ (resp., $\overline{\mathsf{F}}_x$) to $(\overline{\mathsf{T}}^i_x)_i$ (resp., $(\overline{\mathsf{F}}^i_x)_i$). Now each set sequence $(\overline{\mathsf{T}}^i_x)_i$ and $(\overline{\mathsf{F}}^i_x)_i$ has size $a_B + b_B$.

   (b) For each $u \in B$ whose root bag is B, initialize set sequences $(\overline{\mathsf{F}}^i_u)_i$ and $(\overline{\mathsf{T}}^i_u)_i$ with 0s of length $a_B + b_B$ each, and set the bit at position l^B_u of $\overline{\mathsf{F}}^{\mathsf{Lv}(B)}_u$ and $\overline{\mathsf{T}}^{\mathsf{Lv}(B)}_u$ to 1. For every $w \in B$ with $\mathsf{LD}_B(u, w) = 1$ (resp., $\mathsf{LD}_B(w, u) = 1$), insert $(\overline{\mathsf{F}}^i_w)_i$ to $(\overline{\mathsf{F}}^i_u)_i$ (resp., $(\overline{\mathsf{T}}^i_w)_i$ to $(\overline{\mathsf{T}}^i_u)_i$). Finally, set $(\mathsf{F}^i_u)_i$ equal to $(\overline{\mathsf{F}}^i_u)_i$ (resp., $(\mathsf{T}^i_u)_i$ equal to $(\overline{\mathsf{T}}^i_u)_i$).

Fig. 3.3 illustrates the constructed sets on a small example.

It is fairly straightforward that at the end of the preprocessing, the i-th position of each set F_u is 1 only if $(u, v) \in E^*$, where v is such that $i_v - i_u = i$. The following lemma states the opposite direction, namely that each such i-th position will be 1, as long as the path $P : u \rightsquigarrow v$ only visits nodes with certain indices.

Lemma 3.7. *At the end of preprocessing, for every pair of nodes u and v with $i_u \leqslant i_v \leqslant i_u + s_u$, if there exists a path $P : u \rightsquigarrow v$ such that for every $w \in P$, we have $i_u \leqslant i_w \leqslant i_u + s_u$, then the*



u	i_u	Bit-set F_u									
		0	1	2	3	4	5	6	7	8	9
2	0	1	1	1	1	0	0	1	0	1	1
8	1		1	0	0	0	0	0	0	0	1
10	2			1	1	0	0	1	0	1	1
9	3				1	0	0	1	0	1	
7	4					1	1	1	1		
6	5						1	1	0		
4	6							1			
5	7								1		
1	8									1	
3	9										1

	$i=0$			$i=1$			$i=2$			$i=3$		
v	2	8	10	8	9	10	7	8	9	6	7	9
$l_v^{B_6^i}$	0	1	2	0	1	2	0	1	2	0	1	2
$(\mathsf{F}_6^i)_i$	1	1	1	1	1	1	0	1	1	1	0	1
$(\mathsf{T}_6^i)_i$	0	0	0	0	0	0	1	0	0	1	1	0

Figure 3.3: a, c: A graph G and a tree-decomposition $\mathrm{Tree}(G)$. b: The sets F_u constructed from step 5 to answer single-source queries. The j-th bit of a set F_u is 1 iff $(u,v) \in E^*$, where v is such that $i_v - i_u = j$. d: The set sequences $(\mathsf{F}_u^i)_i$ and $(\mathsf{T}_u^i)_i$ constructed from step 6 to answer pair queries, for $u = 6$. For every $i \in \{0,1,2,3\}$ and ancestor B_6^i of B_6 at level i, every node $v \in B_u^i$ is assigned a local index $l_v^{B_6^i}$. The j-th bit of set F_6^i (resp. T_6^i) is 1 iff $(6,v) \in E^*$ (resp. $(v,6) \in E^*$), where v is such that $l_v^{B_6^i} = j$.

$(i_v - i_u)$-th bit of F_u *is* 1.

Proof. We prove inductively the following claim. For every ancestor B of B_v, if there exists $w \in B$ and a path $P_1 : w \rightsquigarrow v$, then exists $x \in B \cap P_1$ such that $i_x \leqslant i_v \leqslant i_x + s_x$ and the $i_v - i_x$-th bit of F_x is 1. The proof is by induction on the length of the simple path $P_2 : B \rightsquigarrow B_v$.

1. If $|P_2| = 0$, the statement is true by taking $x = v$, since the 0-th bit of F_v is 1.

2. If $|P_2| > 0$, examine the child B' of B in P_2. By Lemma 2.2, there exists $x \in B \cap B' \cap P$, and let $P_3 : x \rightsquigarrow v$. By the induction hypothesis there exists some $y \in B' \cap P_3$ with $i_y \leqslant i_v \leqslant i_y + s_y$ and the $i_v - i_y$-th bit of F_y is 1. If $y \in B$, we take $x = y$. Otherwise, B' is the root bag of y, and by the local distance computation of Lemma 2.4, it is $\mathsf{LD}_{B'}(x, y) = 1$. By the choice of x, y we have that B_x is an ancestor of B_y. Thus, by construction we have $i_x < i_y$ and $s_x \geqslant s_y + i_y - i_x$, and hence $i_x \leqslant i_v \leqslant i_x + s_x$. Then in step 5, F_y is inserted in position $i_y - i_x$ of F_x, thus the bit at position $i_y - i_x + i_v - i_y = i_v - i_x$ of F_x will be 1, and we are done.

When B_u is examined, by the above claim there exists $x \in P$ such that $i_x \leqslant i_v$ and the $i_v - i_x$-th bit of F_x is 1. If $x = u$ we are done. Otherwise, by the choice of P, we have $i_u < i_x$, which can only happen if B_u is also the root bag of x. Then in step 5, F_x is inserted in position $i_x - i_u$ of F_u, and hence the bit at position $i_x - i_u + i_v - i_x = i_v - i_u$ of F_x will be 1, as desired. □

Similarly, given a node u and an ancestor bag B_u^i of B_u at level i, the j-th position of the set F_u^i (resp., T_u^i) is 1 only if $(u, v) \in E^*$ (resp., $(v, u) \in E^*$), where $v \in B_u^i$ is such that $l_v^{B_u^i} = j$. The following lemma states that the inverse is also true.

Lemma 3.8. *At the end of preprocessing, for every node u, for every $v \in B_u^i$ where B_u^i is the ancestor of B_u at level i, we have that if $(u, v) \in E^*$ (resp., $(v, u) \in E^*$), then the $l_v^{B_u^i}$-th bit of F_u^i (resp., T_u^i) is 1 .*

Lemma 3.9. *Given a graph G with n nodes and treewidth t, let $\mathcal{T}(G)$ be the time and $\mathcal{S}(G)$ be the space required for constructing a balanced tree-decomposition of G with $O(n)$ bags and width $O(t)$. The preprocessing phase of* Reachability *on G requires $O(\mathcal{T}(G) + n \cdot t^2)$ time and $O(\mathcal{S}(G) + n \cdot t)$ space.*

Proof. First, we construct a balanced tree-decomposition $T = \mathrm{Tree}(G)$ of G in $\mathcal{T}(G)$ time and $\mathcal{S}(G)$ space. We establish the complexity of each preprocessing step separately.

1. Using Lemma 2.7, this step requires $O(n \cdot t)$ time. From this point on, T consists of $b = O(\frac{n}{t})$ bags, has height $h = O(\log n)$, and width $t' = O(t)$.

2. By a standard construction for balanced trees, preprocessing T to answer LCA queries in $O(1)$ time requires $O(b) = O(\frac{n}{t})$ time.

3. By Lemma 2.4, this step requires $O(b \cdot t'^3) = O(\frac{n}{t} \cdot t^3) = O(n \cdot t^2)$ time and $O(b \cdot t'^2) = O(\frac{n}{t} \cdot t^2) = O(n \cdot t)$ space.

4. Every bag B is visited once, and each operation on B takes constant time. We make $O(t')$ such operations in B, hence this step requires $O(b \cdot t') = O(n)$ time in total.

5-6. The space required in this step is the space for storing all the sets F_u of size s_u each, packed into words of length W:

$$\sum_{u \in V} \left\lceil \frac{s_u}{W} \right\rceil = \sum_{i=0}^{h} \sum_{u:\mathsf{Lv}(u)=i} \left\lceil \frac{s_u}{W} \right\rceil \leqslant \sum_{i=0}^{h} \sum_{u:\mathsf{Lv}(u)=i} \left(\frac{s_u}{W} + 1 \right)$$
$$= \frac{1}{W} \cdot \sum_{i=0}^{h} \sum_{u:\mathsf{Lv}(u)=i} s_u + \sum_{i=0}^{h} \sum_{u:\mathsf{Lv}(u)=i} 1 \leqslant \frac{1}{W} \cdot \sum_{i=0}^{h} n \cdot (t'+1) + n = O(n \cdot t)$$

since $h = O(\log n)$, $t' = O(t)$ and $W = \Theta(\log n)$. Note that we have $\sum_{u:\mathsf{Lv}(u)=i} s_u \leqslant n \cdot (t'+1)$ because $|\bigcup_u \mathsf{F}_u| \leqslant n$ (as there are n nodes) and every element of $\bigcup_u \mathsf{F}_u$ belongs to at most $t'+1$ such sets F_u (i.e., for those u that share the same root bag at level i). The time required in this step is $O(n \cdot t)$ in total for iterating over all pairs of nodes (u, v) in each bag B such that B is the root bag of either u or v, and $O(n \cdot t^2)$ for the set operations, by amortizing $O(t)$ operations per word used.

6. The time and space required for storing each sequence of the sets $(\mathsf{F}_u^i)_{0 \leqslant i \leqslant \mathsf{Lv}(u)}$ and $(\mathsf{T}_u^i)_{0 \leqslant i \leqslant \mathsf{Lv}(u)}$ is:

$$\sum_{u \in V} 2 \cdot \left\lceil \frac{a_{B_u} + b_{B_u}}{W} \right\rceil \leqslant 2 \cdot n \cdot \left\lceil \frac{(t'+1) \cdot h}{W} \right\rceil = O(n \cdot t)$$

since $a_{B_u} + b_{B_u} \leqslant (t'+1) \cdot h$, $h = O(\log n)$ and $W = \Theta(\log n)$.

7. The space required is the space for storing the set sequences $(\overline{\mathsf{T}}_v^i)_i$ and $(\overline{\mathsf{F}}_v^i)_i$, which is $O(t^2)$ by a similar argument as in the previous item. The time required is $O(t)$ for initializing every new set sequence $(\overline{\mathsf{T}}_u^i)_i$ and $(\overline{\mathsf{F}}_u^i)_i$ and this will happen once for each node u at its root bag B_u, hence the total time is $O(n \cdot t)$.

□

$\mathsf{Reachability}$ **Querying.** We now turn our attention to the querying phase.

- *Pair query.* Given a pair query (u, v), find the LCA B of bags B_u and B_v. Obtain the sets $\mathsf{F}_u^{\mathsf{Lv}(B)}$ and $\mathsf{T}_v^{\mathsf{Lv}(B)}$ of size b_B. Each set starts in bit position a_B of the corresponding sequence $(\mathsf{F}_u^i)_i$ and $(\mathsf{T}_v^i)_i$. Return True iff the logical-AND of $\mathsf{F}_u^{\mathsf{Lv}(B)}$ and $\mathsf{T}_v^{\mathsf{Lv}(B)}$ contains an entry which is 1.

- *Single-source query.* Given a single-source query u, create a bit set A of size n, initially all 0s. For every node $x \in B_u$ with $i_x \leqslant i_u$, if the $l_x^{B_u}$-th bit of $\mathsf{F}_u^{\mathsf{Lv}(u)}$ is 1, insert F_x to the segment $[i_x, i_x + s_x]$ of A. Then traverse the path from B_u to the root of T, and let B_u^i be the ancestor of B_u at level $i < \mathsf{Lv}(B_u)$. For every node $x \in B_u^i$, if the $l_x^{B_u^i}$-th bit of F_u^i is 1, set the i_x-th bit of A to 1. Additionally, if B_u^i has two children, let B be the child of B_u^i that is not ancestor of B_u, and $j_{\min}$ and $j_{\max}$ the smallest and largest indices, respectively, of nodes whose root bag is in $T(B)$. Insert the segment $[j_{\min} - i_x, j_{\max} - i_x]$ of F_x to the segment $[j_{\min}, j_{\max}]$ of A. Report that the nodes v reached from u are those v for which the i_v-th bit of A is 1.

The following lemma establishes the correctness and complexity of the query phase.

Lemma 3.10. *After the preprocessing phase of* $\mathsf{Reachability}$*, pair and single-source reachability queries are answered correctly in* $O\left(\left\lceil \frac{t}{\log n} \right\rceil\right)$ *and* $O\left(\frac{n \cdot t}{\log n}\right)$ *time respectively.*

Proof. Let $t' = O(t)$ be the width of the small tree-decomposition constructed in Step 1. The correctness of the pair query comes immediately from Lemmas 2.1 and 3.8 , which implies that every path $u \rightsquigarrow v$ must go through the LCA of B_u and B_v. The time complexity follows from the $O\left(\left\lceil \frac{t}{W} \right\rceil\right)$ word operations on the sets $\mathsf{F}_u^{\mathsf{Lv}(B)}$ and $\mathsf{T}_v^{\mathsf{Lv}(B)}$ of size $O(t)$ each.

Now consider the single-source query from a node u and let v be any node such that there is a path $P : u \rightsquigarrow v$. Let B be the LCA of B_u, B_v, and by Lemma 2.1, there is a node $y \in B \cap P$. Let x be the last such node in P, and let $P' : x \rightsquigarrow v$ be the suffix of P from x. It follows that P' is a path such that for every $w \in P'$ we have $i_x \leqslant i_w \leqslant i_x + s_x$.

1. If B_v is an ancestor of B_u, then necessarily $x = v$, and by Lemma 3.8, the l_v^B-th bit of $\mathsf{F}_u^{\mathsf{Lv}(B)}$ is 1. Then the algorithm sets the i_v-th bit of A to 1.

2. Else, B_x is an ancestor of B_v (recall that a bag is an ancestor of itself), and by Lemma 3.7,

the $(i_v - i_x)$-th bit of F_x is 1.

(a) If B is B_u, the algorithm will insert F_x to the segment $[i_x, i_x + s_x]$ of A, thus the $i_x + i_v - i_x = i_v$-th bit of A is set to 1.

(b) If B is not B_u, it can be seen that $j_{\min} \leqslant i_v \leqslant j_{\max}$, where $j_{\min}$ and $j_{\max}$ are the smallest and largest indices of nodes whose root bag is in $T(B')$, with B' the child of B that is not ancestor of B_u. Since the $(i_v - i_x)$-th bit of F_x is 1, the $(i_v - j_{\min})$-th bit of the $[j_{\min}, j_{\max}]$ segment of F_x is 1, thus the $j_{\min} + i_v - j_{\min} = i_v$-th bit of A is set to 1.

Regarding the time complexity, the algorithm performs $O(h \cdot t') = O(h \cdot t)$ set insertions to A. For every position j of A, the number of such set insertions that overlap on j is at most $t' + 1$ (once for every node in the LCA of B_u and B_v, where v is such that $i_v = j$). Hence if H_i is the size of the i-th insertion in A, we have $\sum_i H_i \leqslant n \cdot (t' + 1)$. Since the insertions are word operations, the total time spent for the single source query is

$$\sum_{i=0}^{h} \left\lceil \frac{H_i}{W} \right\rceil \leqslant h + \sum_{i=0}^{h} \frac{H_i}{W} \leqslant h + \frac{n \cdot (t' + 1)}{W} = O\left(\frac{n \cdot t}{\log n}\right)$$

since $h = O(\log n)$, $t' = O(t)$ and $W = \Theta(\log n)$. $\square$

We summarize the results of this section in the following theorem.

Theorem 3.2. *Given a graph G of n nodes and treewidth t, let $\mathcal{T}(G)$ be the time and $\mathcal{S}(G)$ be the space required for constructing a balanced tree-decomposition* $\mathrm{Tree}(G)$ *of $O(n)$ bags and width $O(t)$ on the standard RAM with wordsize $W = \Theta(\log n)$. The data structure* $\mathsf{Reachability}$ *correctly answers reachability queries and requires*

1. *$O(\mathcal{T}(G) + n \cdot t^2)$ preprocessing time;*
2. *$O(\mathcal{S}(G) + n \cdot t)$ preprocessing space;*
3. *$O\left(\left\lceil \frac{t}{\log n} \right\rceil\right)$ pair query time; and*
4. *$O\left(\frac{n \cdot t}{\log n}\right)$ single-source query time.*

For constant-treewidth graphs we have that $\mathcal{T}(G) = O(n)$ and $\mathcal{S}(G) = O(n)$ (Theorem 2.1) and thus from Theorem 3.2 we obtain the following corollary.

Corollary 3.1. *Given a graph G of n nodes and constant treewidth, the data structure* Reachability *requires $O(n)$ preprocessing time and space, and correctly answers (i) pair reachability queries in $O(1)$ time, and (ii) single-source reachability queries in $O\left(\frac{n}{\log n}\right)$ time.*

3.4 Space vs Query Time Tradeoff for Sublinear Space

In this section we present the data structure LowSpDis, for low-space distance queries. Our results make use of the following lemma, where $\alpha(n)$ is the inverse of the Ackermann function on input (n, n).

Lemma 3.11 ([Chaudhuri and Zaroliagis, 1995])**.** *Consider a weighted graph $G = (V, E, \mathsf{wt})$ of n nodes and constant-treewidth, and a tree-decomposition T of G of $O(n)$ nodes and constant width. There exists a data structure* DistanceLP *that answers distance queries on G and requires*

1. *$O(n)$ preprocessing time and space; and*

2. *$O(\alpha(n))$ pair query time.*

The main idea is to partition the initial tree-decomposition T to sufficiently large components, and discard all bags that don't appear in the boundary of their component. We use Lemma 3.11 to preprocess $\overline{T}$ and the induced graph. Answering a pair query (u, v) is performed similarly as in Lemma 3.11, but requires additional time for processing the components in which u and v appear (since they have not been preprocessed). The challenge comes in performing these computations within the targeted space and time bounds.

Informal description. Here we outline the key steps required for LowSpDis to achieve the bounds stated in Theorem 3.3. Throughout this section we fix a constant $\epsilon \in [\frac{1}{2}, 1]$. The preprocessing consists of the following conceptual steps.

1. A binary tree-decomposition $T = \mathrm{Tree}(G)$ of $O(n)$ bags is constructed in polynomial time and logarithmic space, using Theorem 2.1. Hence, LowSpDis does not store T explicitly, but uses the logspace construction of Theorem 2.1 to traverse T and access its bags.

2. A tree-partitioning algorithm LowSpTreePart is used to partition T into $O(n^{1-\epsilon})$ components $\mathcal{C}$ of size $O(n^{\epsilon})$ each. A key point in this construction is that every such component

$\mathcal{C}$ contains a constant number of bags on its boundary.

3. Given a list of components $\mathcal{C} = (\mathcal{C}_1, \dots, \mathcal{C}_\ell)$ constructed in the previous step, a tree of bags called *summary tree* $\overline{T}$ is constructed. The summary tree occurs by contracting every component $\mathcal{C}_i$ of T to a single bag $\mathcal{B}_i$. Moreover, $\mathcal{B}_i$ contains precisely the nodes that appear in the bags of the boundary of $\mathcal{C}_i$. Since there are $O(1)$ such bags for every component, each $\mathcal{B}_i$ has constant size. The key point in this step is that $\overline{T}$ is a tree-decomposition of G restricted on the nodes that appear in bags of $\overline{T}$. Moreover, $\overline{T}$ has size $O(n^{1-\epsilon})$ instead of $O(n)$, which is the size of the initial tree-decomposition T.

4. Since $\overline{T}$ is a tree-decomposition, Lemma 3.11 applies to preprocess $\overline{T}$ in the stated bounds.

5. An algorithm LowSpLD is used to compute the distance $d(u, v)$ between any pair of nodes u, v that appear together in some boundary bag of a component $\mathcal{C}_i$. This is achieved by traversing T in a particular way, and applying a standard, linear-space computation on each component $\mathcal{C}_i$ separately. Since $|\mathcal{C}_i| = O(n^\epsilon)$, this requires $O(n^\epsilon)$ space. Since the boundary bags of $\mathcal{C}_i$ are constantly many, the algorithm only needs to store constant-size information per component, and thus $O(n^{1-\epsilon}) = O(n^\epsilon)$ information in total.

6. Finally, given a node u, it is crucial to obtain the set V_u of nodes that u can reach going through nodes v that appear in bags of $\overline{T}$. Moreover, this set needs to be obtained in linear time in the size of the component, i.e., $O(n^{1-\epsilon})$. This is achieved by a graph traversal on G starting from u, in combination with perfect hashing for testing in $O(1)$ time whether a node v appears in bags of $\overline{T}$.

A query u, v is answered by LowSpDis using the following conceptual steps.

1. First, the algorithm retrieves the sets V_u and V_v. If $v \in V_u$, then the distance $d(u, v)$ is retrieved by constructing a tree-decomposition T_u of $G[V_u]$, and using standard methods for solving the problem in T_u, in $O(n^\epsilon)$ time. Similarly if $u \in V_v$.

2. If $v \notin V_u$ and $u \notin V_v$, then the algorithm again constructs the tree-decompositions T_u and T_v of $G[V_u]$ and $G[V_v]$ respectively. The algorithm retrieves two bags $\mathcal{B}_u$ and $\mathcal{B}_v$ of $\overline{T}$ with $\mathcal{B}_u \subseteq V_u$ and $\mathcal{B}_v \subseteq V_v$, and uses the standard methods of the previous item to obtain the distances $d(u, x)$ and $d(y, v)$, for every node $x \in \mathcal{B}_u$ and $\mathcal{B}_v$. Additionally, the algorithm uses Lemma 3.11 to obtain the distance $d(x, y)$ between every such pair x, y. Finally, the

algorithm returns the value $\min_{x\in\mathcal{B}_u, y\in\mathcal{B}_v}(d(u,x)+d(x,y)+d(y,v))$.

In the remaining of this section we describe in detail the above phases of LowSpDis.

Tree partitioning: The algorithm LowSpTreePart. We first describe algorithm LowSpTreePart, which operates on a binary tree-decomposition $T=(V_T,E_T)$ of $O(n)$ bags. Given a constant ϵ, LowSpTreePart splits T to $O(n^{1-\epsilon})$ connected components $\mathcal{C}\subseteq V_T$ of size $|\mathcal{C}|=O(n^{\epsilon})$. Each component $\mathcal{C}$ is implicitly represented as a list of bags $\mathcal{C}(B_1,\dots,B_k)$, which mark the boundaries of $\mathcal{C}$ in T. The *root* of $\mathcal{C}(B_1,\dots,B_k)$ is $B=\arg\min_{B_i}\mathsf{Lv}(B_i)$, i.e., the smallest-level bag among all B_i. We will consider w.l.o.g. that B_1 is always the root bag of component $\mathcal{C}(B_1,\dots,B_k)$. A bag B' belongs to $\mathcal{C}$ iff the $\mathsf{Lv}(B')\geqslant\mathsf{Lv}(B_1)$ and the unique simple path $B\rightsquigarrow B_1$ in T does not contain any of the B_i as intermediate bags.

The algorithm traverses T in post-order, and maintains a two variables $x,y\in\mathbb{N}$, that represent the size of the current component $\mathcal{C}$ and the number of components that appear directly below $\mathcal{C}$. As the algorithm backtracks to a bag B, it updates $x=x_1+x_2+1$ and $y=y_1+y_2$, where x_i,y_i is the pair corresponding to the child B'_i of B (recall that T is binary), or sets $x=x_1+1$ and $y=y_1$ if B has only one child B'_1. If $x\geqslant n^{\epsilon}$ or $y\geqslant 3$, the algorithm creates a new component $\mathcal{C}(B_1,\dots,B_k)$, where B_1 is the current bag B, and $B_2,\dots,B_k$ are parents of roots of components that have been constructed already (or leaves of T). Finally, the algorithm sets $x=0$ and $y=1$, and proceeds to the parent of B.

Lemma 3.12. LowSpTreePart *constructs $O(n^{1-\epsilon})$ components. For every constructed component $\mathcal{C}(B_1,\dots,B_k)$ we have $|\mathcal{C}|\leqslant 2\cdot n^{\epsilon}-1$ and $k\leqslant 5$.*

Proof. If $|\mathcal{C}|>2\cdot n^{\epsilon}-1$, then, before backtracking to B_1, the algorithm examined a child B of B_1 with value $x\geqslant j$, and thus would have grouped B and B_1 in different components. It is easy to see that every root of a component appears in the same component with its children, a contradiction. A similar argument holds for showing that $k\leqslant 5$. We now argue that LowSpTreePart constructs $O(n^{1-\epsilon})$ components. We say that the algorithm "performs a type A cut" and "performs a type B cut" when it constructs a component based on the criterion $x\geqslant j$ and $y\geqslant 3$ respectively. Let X and Y be the number of type A and type B cuts. Every type A cut constructs a component of size at least j, hence $X=O(n^{1-\epsilon})$. Additionally, we have $Y\leqslant X$, hence $X+Y=O(n^{1-\epsilon})$, as desired. To see that $Y\leqslant X$, let Z be a counter that counts the sum of the y values that LowSpTreePart maintains at any point in the traversal. Observe that a type A cut increases Z by

at most one, and a type B cut decreases Z by at least one. Since Z is always non-negative, we have that there is at least one type A cut for each type B cut, thus $Y \leqslant X$. The desired result follows. $\square$

We denote by $\mathsf{Root}(\mathcal{C})$ the root bag of a component $\mathcal{C}$. Given two components $\mathcal{C}_1, \mathcal{C}_2$ constructed by $\mathsf{LowSpTreePart}$, we say that $\mathcal{C}_1$ is the *parent* of $\mathcal{C}_2$ if $\mathsf{Root}(\mathcal{C}_1)$ is the lowest ancestor of $\mathsf{Root}(\mathcal{C}_2)$ among all bags that appear as roots in some component. In such case, $\mathcal{C}_2$ is a *child* of $\mathcal{C}_1$. Given a component $\mathcal{C}$ that is the parent of components $\mathcal{C}_1, \dots, \mathcal{C}_i$, we let $\mathsf{Merge}(\mathcal{C}) = \mathcal{C} \cup \bigcup_j \mathcal{C}_j$.

The summary-tree construction $\mathsf{SummaryTree}$**.** Let $\mathcal{C} = (\mathcal{C}_1, \dots, \mathcal{C}_\ell) = \mathsf{LowSpTreePart}(T)$ be the list of components that $\mathsf{LowSpTreePart}$ returns, where each component is implicitly represented by the bags of its boundary, i.e., $\mathcal{C}_i = \mathcal{C}_i(B^i_1, \dots, B^i_{k_i})$. We construct a *summary tree* of bags $\overline{T} = \mathsf{SummaryTree}(\mathcal{C}) = (\overline{V}, \overline{E})$ as follows.

1. $\overline{V}$ consists of bags $\mathcal{B}_i$ for $1 \leqslant i \leqslant \ell$, where $\mathcal{B}_i = B^i_1 \cup \dots \cup B^i_{k_i}$, i.e., $\mathcal{B}_i$ is the union of all bags in the boundary of $\mathcal{C}_i$.

2. We have $(\mathcal{B}_i, \mathcal{B}_j) \in \overline{E}$ if $\mathcal{C}_i$ is a parent of $\mathcal{C}_j$.

The following lemma follows easily from Lemma 3.12 and the above construction.

Lemma 3.13. *Let $V_S = \bigcup_{\mathcal{B}_i \in \overline{V}} \mathcal{B}_i$ be the set of nodes of G that appear in bags of the summary tree $\overline{T}$. Then $\overline{T}$ is a tree-decomposition of the graph $G[V_S]$ induced by V_S. $\overline{T}$ has $O(n^{1-\epsilon})$ bags and constant width.*

Local distance computation in low space $\mathsf{LowSpLD}$**.** Let $\mathcal{C} = (\mathcal{C}_1, \dots, \mathcal{C}_\ell) = \mathsf{LowSpTreePart}(T)$ be the list of components constructed by $\mathsf{LowSpTreePart}$. We describe algorithm $\mathsf{LowSpLD}$, which computes the distance $d(u, v)$ between any pair of nodes u, v that appear in the root bag $\mathsf{Root}(\mathcal{C}_i)$ of some component $\mathcal{C}_i$. Let $T_i = \mathrm{Tree}(G)[\mathsf{Merge}(\mathcal{C}_i)]$ be the subtree of $\mathrm{Tree}(G)$ restricted in the bags of component $\mathcal{C}_i$ and its children components, and $V_i = \bigcup_{B \in \mathsf{Merge}(\mathcal{C}_i)} B$ the set of nodes that appear in bags of $\mathsf{Merge}(\mathcal{C}_i)$. It is easy to verify that T_i is a subtree of T, and thus a tree decomposition of the graph $G[V_i] = (V_i, E_i)$ induced by V_i. The algorithm $\mathsf{LowSpLD}$ operates as follows. For every component $\mathcal{C}$, it maintains a local distance map $\mathsf{LD}_{\mathsf{Root}(\mathcal{C})} : \mathsf{Root}(\mathcal{C}) \times \mathsf{Root}(\mathcal{C}) \to \mathbb{R}$. Initially, $\mathsf{LD}_{\mathsf{Root}(\mathcal{C})}(u, v) = \mathsf{wt}(u, v)$ for every component $\mathcal{C}$ and pair of nodes $u, v \in \mathsf{Root}(\mathcal{C})$. Then, $\mathsf{LowSpLD}$ performs the following two passes.

1. Traverse $\overline{T}$ bottom-up, and for every encountered bag $\mathcal{B}$ that corresponds to component $\mathcal{C}$, let $\mathcal{C}_1, \dots, \mathcal{C}_k$ be the children components of $\mathcal{C}$. Obtain the tree-decomposition T_i, and construct a weight function $\mathsf{wt}_i : E_i \to \mathbb{R}$ defined as follows:

$$\mathsf{wt}_i(u,v) = \begin{cases} \mathsf{LD}_{\mathsf{Root}(\mathcal{C})}(u,v) & \text{if } u,v \in \mathsf{Root}(\mathcal{C}) \\ \mathsf{LD}_{\mathsf{Root}(\mathcal{C}_i)}(u,v) & \text{if } u,v \in \mathsf{Root}(\mathcal{C}_i) \text{ for some } 1 \leqslant i \leqslant k \\ \mathsf{wt}(u,v) & \text{otherwise} \end{cases}$$

   and execute the local distance computation of Lemma 2.4 Afterwards, update $\mathsf{LD}_{\mathsf{Root}(\mathcal{C})}$ and $\mathsf{LD}_{\mathsf{Root}(\mathcal{C}_i)}$ for all $1 \leqslant i \leqslant k$ with the newly discovered distances.

2. Traverse $\overline{T}$ top-down, and for every encountered bag $\mathcal{B}$ execute the steps of Step 1.

Lemma 3.14. *At the end of* LowSpLD*, for every component* $\mathcal{C}$ *and nodes* $u, v \in \mathsf{Root}(\mathcal{C})$ *we have* $\mathsf{LD}_{\mathsf{Root}(\mathcal{C})}(u,v) = d(u,v)$. *Moreover,* LowSpLD *operates in* $O(n^{\epsilon})$ *space and polynomial time.*

Proof. The correctness of LowSpLD follows straightforwardly from Lemmas 2.3 and 2.4. Since T has constant width, the size of each local distance map $\mathsf{LD}_{\mathsf{Root}(\mathcal{C})}$ has constant size. Hence the space used by the algorithm is asymptotically the space required for storing $\overline{T}$, plus the space for constructing each tree-decomposition T_i. By Lemma 3.13 the former requires $O(n^{1-\epsilon})$ space, while by Lemma 3.12 the latter $O(n^{\epsilon})$ space. Since $\epsilon \geqslant \frac{1}{2}$, we conclude that the space usage is $O(n^{\epsilon})$. The polynomial time bound follows from the space bound. □

Fast component retrieval GetCompNodes**.** Given a node u of G, we are interested in retrieving the set V_u of nodes that u can reach in G without going through nodes v that appear in bags of $\overline{T}$. The desired set V_u can be obtained in $O(n^{\epsilon})$ time by a performing any standard graph traversal on G starting from u, and making sure that the traversal never expands a node v that appears in the bags of $\overline{T}$. This can be done if testing whether v appears in any of the bags of $\overline{T}$ can be performed in constant time. Let $V_S = \bigcup_{\mathcal{B}_i \in \overline{V}} \mathcal{B}_i$ be the set of all such nodes, and $k = |V_S| = O(n^{1-\epsilon})$. We cannot store V_S as a standard bit-set which allows $O(1)$ membership testing, as this would require linear space (i.e., beyond our space bound $O(n^{\epsilon})$). The problem can be solved using standard techniques from perfect hashing to store the set V_S. In the query phase, given a node u, GetCompNodes detects that $u \in V_S$ by testing whether u equals its entry in the hash table.

$\mathsf{LowSpDis}$ **Preprocessing.** We now describe the preprocessing phase of $\mathsf{LowSpDis}$. The input is a weighted graph $G = (V, E, \mathsf{wt})$ of constant treewidth, and a constant $\epsilon \in [\frac{1}{2}, 1]$.

1. Construct a binary tree-decomposition $T = \mathrm{Tree}(G)$ in logspace Theorem 2.1.

2. Use $\mathsf{LowSpTreePart}$ to construct a list of components $\mathcal{C} = (\mathcal{C}_1, \dots, \mathcal{C}_\ell) = \mathsf{LowSpTreePart}(T)$, with $\ell = n^{1-\epsilon}$ (i.e., $\mathsf{LowSpTreePart}$ is executed with $j = n^\epsilon$).

3. Construct the local distance maps $\mathsf{LD}_{\mathsf{Root}(\mathcal{C})}$ using $\mathsf{LowSpLD}$.

4. Construct the summary tree $\overline{T} = \mathsf{SummaryTree}(\mathcal{C}) = (\overline{V}, \overline{E})$. For every component $\mathcal{C}_i$ that corresponds to $\mathcal{B}_i$ in $\overline{T}$, find a node $z \notin \mathcal{B}_i$ that appears in bags of $\mathcal{C}_i$, and associate z with $\mathcal{B}_i$.

5. Use Lemma 3.11 to build a data structure $\mathsf{DistanceLP}$ on $G[V_S]$ and $\overline{T}$.

6. Let $V_S = \bigcup_{\mathcal{B}_i \in \overline{V}} \mathcal{B}_i$ be the set of nodes of G that appear in bags of the summary tree $\overline{T}$. Construct the data structure $\mathsf{GetCompNodes}$ on V_S.

$\mathsf{LowSpDis}$ **Querying.** We now turn our attention to the query phase of $\mathsf{LowSpDis}$.

1. Use the data structure $\mathsf{GetCompNodes}$ to construct the sets V_u and V_v.

2. Construct the tree-decompositions T_u and T_v of the graphs $G[V_u]$ and $G[V_v]$ induced by V_u and V_v. This is done using some standard linear-time algorithm, e.g. [Bodlaender and Hagerup, 1995, Lemma 2]. If $u \in V_v$, insert u to every bag of T_v, and use Lemma 2.4 to obtain the distance $d(u, v)$. Similarly if $v \in V_u$.

3. If $u \notin V_v$ and $v \notin V_u$ let $\mathcal{B}_u$ be the unique bag of $\overline{T}$ with that is associated with a node $z_u \in V_u$, and $\mathcal{B}_v$ the unique bag of $\overline{T}$ that is associated with a node $z_v \in V_v$. Insert every node of $\mathcal{B}_u$ in every bag of T_u, and every node of $\mathcal{B}_v$ in every bag of T_u, and use Lemma 2.4 to obtain the distances $d(u, x)$ and $d(y, v)$ for every node $x \in \mathcal{B}_u$ and $y \in \mathcal{B}_v$. Return the value $\min_{x \in \mathcal{B}_u, y \in \mathcal{B}_v}(d(u, x) + d(x, y) + d(y, v))$ where for every pair x, y the distance $d(x, y)$ is obtained by querying $\mathsf{DistanceLP}$.

We arrive at the following theorem.

Theorem 3.3. *Let (1) a constant* $\epsilon \in [\frac{1}{2}, 1]$*; and (2) a weighted graph* $G = (V, E, \mathsf{wt})$ *with* n *nodes and of constant treewidth, be given. The data structure* LowSpDis *correctly answers pair distance queries on* G *and requires*

1. *Polynomial in* n *preprocessing time;*

2. $O(n^{\epsilon})$ *working space; and*

3. $O(n^{1-\epsilon} \cdot \alpha(n))$ *pair query time.*

Proof. It is clear from Lemmas 3.11 to 3.14 that the preprocessing of LowSpDis requires polynomial time and $O(n^{\epsilon})$ space, where $\epsilon \geqslant \frac{1}{2}$. In the query phase, LowSpDis uses $O(n^{\epsilon})$ time and space for extracting the sets V_u and V_v, since each has size $O(n^{\epsilon})$. Using a linear time and space algorithm for constructing the tree-decompositions T_u and T_v, this step also requires $O(n^{1-\epsilon})$ time and space. If $u \in V_v$ or $v \in V_u$, applying Lemma 2.4 on T_u and T_v is also done in $O(n^{1-\epsilon})$ time and space.

If $u \notin V_v$ and $v \notin V_u$, note that by Lemma 3.13 $\mathcal{B}_u$ and $\mathcal{B}_v$ have constant size, hence after inserting every node of $\mathcal{B}_u$ to every bag of T_u and every node of $\mathcal{B}_v$ to every bag of T_v, T_u and T_v still have constant width. Hence all distances $d(u, x)$ and $d(v, y)$ can be obtained using Lemma 2.4 in $O(n^{1-\epsilon})$ time and space. Finally, DistanceLP will be queried for the distances $d(x, y)$ of a constant number of pairs x, y, and by Lemma 3.11, all such queries can be served in $O(n^{1-\epsilon} \cdot \alpha(n))$ time. □

4 Semiring Distances on RSMs of Constant Treewidth

4.1 Introduction

In this chapter we focus on the algebraic path problem for RSMs, which models various interprocedural analysis problems. In turn, these problems have numerous applications, ranging from alias analysis, to data dependencies (modification and reference side effect), to constant propagation, to live and use analysis [Reps *et al.*, 1995a; Sagiv *et al.*, 1996; Callahan *et al.*, 1986; Grove and Torczon, 1993; Landi and Ryder, 1991; Knoop *et al.*, 1996; Cousot and Cousot, 1977a; Giegerich *et al.*, 1981; Knoop and Steffen, 1992; Naeem and Lhoták, 2008; Zhang *et al.*, 2014]. We obtain algorithmic improvements for the general algebraic path problem, by exploiting the fact that the control-flow graphs of typical programs have small treewidth.

The algebraic path problem for RSMs. To specify properties of traces of a RSM we consider a very general framework, where edges of the RSM are labeled from a closed semiring (which subsumes bounded and finite distributive semirings), and we refer to the labels of the edges as weights. For a given path, the weight of the path is the semiring product of the weights on the edges of the path, and to choose among different paths we use the semiring plus operator. For example, (i) with Boolean semiring (with semiring product as AND, and semiring plus as OR) we can express the reachability property; (ii) with tropical semiring (with real-edge weights, semiring product as standard sum, and semiring plus as minimum) we can express the shortest path property; and (iii) with Viterbi semiring (with probability value on edges, semiring product as standard multiplication and semiring plus as maximum) we can express the most probable path property. The algebraic path problem expressed in our framework subsumes the IFDS/IDE

frameworks [Reps *et al.*, 1995a; Sagiv *et al.*, 1996] which consider finite semirings and meet over all paths as the semiring plus operator. Since IFDS/IDE are subsumed in our framework, the large and important class of data-flow analysis problems that can be expressed in IFDS/IDE frameworks can also be expressed in our framework.

Two important aspects. In the traditional algorithms for interprocedural analysis, the starting point is typically *fixed* as the entry point of a specific method. In graph theoretic parlance, graph algorithms can consider two types of queries: (i) a *pair query* that given nodes u and v (called (u, v)-pair query) asks for the semiring distance from u to v; and (ii) a *single-source* query that given a node u asks for the answer of (u, v)-pair queries for all nodes v. Thus the traditional algorithms for interprocedural analysis have focused on the answer for *one* single-source query. Moreover, the existing algorithms also consider that the input control-flow graph is arbitrary, and do not exploit the fact that most control-flow graphs satisfy some elegant structural properties. Here we consider two new aspects, namely, (i) multiple pair and single-source queries, and (ii) exploit the fact that typically the control-flow graphs of programs satisfy an important structural property, namely they are graphs of small treewidth. We describe in details the two aspects.

- *Multiple queries.* We first describe the relevance of pair and multiple pair queries, and then the significance of even multiple single-source queries. For example, in constant propagation, given a function call, a relevant question is whether some variable remains constant within the entry and exit of the function (in general it can be between a pair of nodes of the program). This shows that the pair query problem, and the multiple pair queries are relevant in many applications. Finally, consider a run-time optimization scenario, where the goal is to decide whether a variable remains constant from *now on*. This corresponds to a single-source query, where the starting point is the current execution point of the program. Thus, multiple pair queries and multiple single-source queries are relevant for several important static analysis problems.

- *Constant treewidth.* A very well-known concept in graph theory is the notion of *treewidth* of a graph, which is a measure of how similar a graph is to a tree (a graph has treewidth 1 precisely if it is a tree) [Robertson and Seymour, 1984]. The treewidth of a graph is defined based on a *tree decomposition* of the graph [Halin, 1976] (see Section 2.3 for a formal definition). Besides the mathematical elegance of the treewidth property for graphs,

	Preprocessing time	Space	Query		Reference
			Single-source	Pair	
Our	$O(n \cdot \log n + h \cdot b \cdot \log n)$	$O(n \cdot \log n)$	$O(n)$	$O(1)$	Theorem 4.1
Result	$O(n + h \cdot b \cdot \log n)$	$O(n)$	$O(n)$	$O(\log n)$	Theorem 4.1

Table 4.1: Interprocedural same-context semiring distances on RSMs with n nodes, b boxes and constant treewidth, for stack height h.

there are many classes of graphs which arise in practice and have constant treewidth. The most important example is that the control-flow graph for goto-free programs for many programming languages are of constant treewidth [Thorup, 1998], and it was also shown in [Gustedt *et al.*, 2002] that typically all Java programs have constant treewidth.

Our contributions. We consider RSMs where every CSM (Component State Machine) has constant treewidth, and the algorithmic question of answering multiple single-source and multiple pair semiring distance queries, where each query is a *same-context* query (a same-context query starts and ends with an empty stack, see [Chaudhuri, 2008] for the significance of same-context queries). In the analysis of multiple queries, there is a very important algorithmic distinction between *one-time* preprocessing (denoted as the preprocessing time), and the work done for each individual query (denoted as the query time). There are two end-points in the spectrum of tradeoff between the preprocessing and query resources that can be obtained by using the classic algorithms for one single-source query, namely, (i) the *complete preprocessing*, and (ii) the *no preprocessing*. In complete preprocessing, the single-source answer is precomputed with every node as the starting point (for example, in graph reachability this corresponds to computing the all-pairs reachability problem with the classic BFS/DFS algorithm [Cormen *et al.*, 2009], or with fast matrix multiplication [Fischer and Meyer, 1971]). In no preprocessing, there is no preprocessing done, and the algorithm for one single-source query is used on demand for each individual query. We consider the computation on a standard RAM with wordsize $W = \Theta(\log n)$, and focus on various possible tradeoffs in preprocessing vs query time. Our main contributions are as follows:

- *(General result).* Since we consider arbitrary semirings (i.e., not restricted to finite semirings) we consider the stack height bounded problem, where the distance between

	Preprocessing time	Space	Query		Reference
			Single-source	Pair	
IDE/IFDS (complete preprocessing)	$O(n^2 \cdot \lvert D\rvert^3)$	$O(n^2 \cdot \lvert D\rvert^2)$	$O(n \cdot \lvert D\rvert)$	$O(\lvert D\rvert)$	[Reps *et al.*, 1995a]
IDE/IFDS (no preprocessing)	-	$O(n \cdot \lvert D\rvert^2)$	$O(n \cdot \lvert D\rvert^3)$	$O(n \cdot \lvert D\rvert^3)$	[Reps *et al.*, 1995a]
Our Results	$O(n \cdot \log n \cdot \lvert D\rvert^3)$	$O(n \cdot \log n \cdot \lvert D\rvert^2)$	$O(n \cdot \lvert D\rvert^2)$	$O(\lvert D\rvert^2)$	Corollary 4.1
	$O((n + b \cdot \log n) \cdot \lvert D\rvert^3)$	$O(n \cdot \lvert D\rvert^2)$	$O(n \cdot \lvert D\rvert^2)$	$O(\log n \cdot \lvert D\rvert^2)$	Corollary 4.1
$\lvert D\rvert = \Omega(\log n)$	$O(n \cdot \lvert D\rvert^3)$	$O(n \cdot \lvert D\rvert^2)$	$O(n \cdot \lvert D\rvert^2/\log n)$	$O(\lvert D\rvert^2/\log n)$	Corollary 4.2

Table 4.2: Interprocedural same-context semiring distances on RSMs with n nodes, b boxes and constant treewidth, where the semiring is over the subset of $|D|$ elements and the plus operator is the meet operator of the IFDS framework. Existing results are taken from [Reps *et al.*, 1995a]. Our results are obtained from Corollary 4.1 and Corollary 4.2

	Preprocessing time	Space	Query		Reference
			Single-source	Pair	
Complete preprocessing	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	[Reps *et al.*, 1995a]
No preprocessing	-	$O(n)$	$O(n)$	$O(n)$	[Reps *et al.*, 1995a]
Our Result	$O(n + b \cdot \log n)$	$O(n)$	$O\left(\frac{n}{\log n}\right)$	$O(1)$	Corollary 4.3

Table 4.3: Interprocedural same-context reachability on RSMs with n nodes, b boxes and constant treewidth. Existing results are taken from [Reps *et al.*, 1995a] using the IFDS/IDE framework with $|D| = 1$. Our results are obtained from Corollary 4.3.

	Preprocessing time	Space	Query		Reference
			Single-source	Pair	
Complete preprocessing [1]	$O(n^2 \cdot \log n)$	$O(n^2)$	$O(n)$	$O(1)$	[Schwoon, 2002]
No preprocessing [2]	-	$O(n)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$	[Schwoon, 2002]
Our	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n)$	$O(1)$	Corollary 4.4
Result	$O(n + b \cdot \log n)$	$O(n)$	$O(n)$	$O(\log n)$	Corollary 4.4

Table 4.4: Interprocedural same-context distances with non-negative weights for RSMs with n nodes, k CSMs, b boxes and constant treewidth.

[1] The preprocessing time is obtained by executing Dijkstra's algorithm b times in each of the k CMSs, followed by executing Dijkstra's algorithm from n source nodes.

[2] The single-source and pair query times are obtained by executing Dijkstra's algorithm b times in each of the k CMSs.

nodes is witnessed by paths whose stack height is bounded by a parameter h. While in general for arbitrary semirings there does not exist a bound on the stack height, if the semiring contains subsets of a finite universe D, and the semiring plus operator is intersection or union, then solving the problem with sufficiently large bound on the stack height is equivalent to solving the problem without any restriction on stack height. Our main result is an algorithm where the one-time preprocessing phase requires $O(n \cdot \log n + h \cdot b \cdot \log n)$ semiring operations, and then each subsequent bounded stack height pair query can be answered in constant number of semiring operations, where n is the number of nodes of the RSM and b the number of boxes (see Table 4.1 and Theorem 4.1). If we specialize our result to the IFDS/IDE setting with finite semirings from a finite universe of distributive functions $2^D \rightarrow 2^D$, and meet over all paths as the semiring plus operator, then we obtain the results shown in Table 4.2 (Corollary 4.1). For example, our approach with a factor of $O(\log n)$ overhead for one-time preprocessing, as compared to no preprocessing, can answer subsequent pair queries by a factor of $\Omega(n \cdot |D|)$ faster. Additionally, when $|D| = \Omega(\log n)$, our algorithm requires only $O(n \cdot |D|^3)$ preprocessing after which pair queries are answered in $O(|D|^2 / \log n)$ time. Note that the complexity of the standard IFDS/IDE algorithm is $O(n \cdot |D|^3)$ for answering one single-source query, whereas in the

same preprocessing time, our algorithm handles every pair query efficiently. An important feature of our algorithms is that they are simple and implementable.

- *(Reachability and distance).* We now discuss the significance of our result for the important special cases of reachability and distances with non-negative weights.

  - *(Reachability).* Observe that reachability follows as a special case of our general result, by setting $|D| = 1$. We improve further this case, by combining our general result with Theorem 3.2 from Section 3.3. We obtain a data structure that handles same-context queries for interprocedural reachability and uses (i) $O(n \cdot \log n)$ preprocessing time; (ii) $O(n)$ space; (iii) $O(1)$ pair query time; and (iv) $O(n/\log n)$ time for single-source queries. For example, if we consider $\Theta(n)$ pair queries, then both full preprocessing and no preprocessing take quadratic time in total, whereas our approach requires $O(n \cdot \log n + n) = O(n \cdot \log n)$ time. See Table 4.3.

  - *(Distances with non-negative weights).* We now consider the problem of distances with non-negative weights, where the current best-known algorithm for RSMs with unique entries and exists comes from [Schwoon, 2002]. Each single-source query requires $O(n \cdot \log n)$ based on a variant of Dijkstra's shortest-path algorithm phrased on RSMs. The complete preprocessing requires $O(n^2 \cdot \log n)$ time for computing the transitive closure using n single-source queries. The complete preprocessing additionally requires $\Theta(n^2)$ space, at the cost of which single-source and pair distance queries are handled in $O(n)$ and $O(1)$ time respectively. In contrast, we show that (i) with $O(n + b \cdot \log n)$ preprocessing time and $O(n)$ space, we can answer single-source (resp. pair) queries in $O(n)$ (resp. $O(\log n)$) time; and (i) with $O(n \cdot \log n)$ time and space, we can answer single-source (resp. pair) queries in $O(n)$ (resp. $O(1)$) time. Thus our approach provides a significant theoretical improvement over the existing approaches. See Table 4.4.

- *(Experimental results).* Besides the theoretical improvements, we demonstrate the effectiveness of our approach on several well-known benchmarks from programming languages. We have used the tool for computing tree decompositions from [van Dijk *et al.*, 2006a], and all benchmarks of our experimental results have small treewidth. We have implemented our algorithms for reachability (both intraprocedural and interprocedural) and distance

with non-negative weights (only intraprocedural), and compare their performance against complete and no preprocessing approaches for same-context queries. Our experimental results show that our approach obtains a significant improvement over the existing approaches (of complete and no preprocessing).

A byproduct of our results: on demand analysis. Several previous works such as [Horwitz *et al.*, 1995] have stated the importance and asked for the development of data structures and analysis techniques to support dynamic updates. Though our main results are for the problem where the RSM is given and fixed, our main technical contribution is a dynamic algorithm that can also be used in other applications to support dynamic updates, and is thus also of independent interest. For example, consider a large library which is linked to different main programs upon compilation. Traditionally, for every different main program, the resulting program linked with the library needs to be analyzed anew. In contrast, our data structures can be used to preprocess the library once, so that each new analysis spends little time on the library.

Organization. The rest of this chapter is organized as follows.

1. In Section 4.2 we present our data structures for handling bounded-height same-context semiring distances on RSMs of constant treewidth.

2. In Section 4.3 we present an experimental evaluation of our algorithms on RSMs of benchmark programs.

4.2 Algorithms for Constant Treewidth RSMs

We consider the bounded-height same-context semiring-distance problem on RSMs of constant treewidth. The input is (i) a constant-treewidth RSM $\mathsf{RSM} = \{A_1, A_2, \dots, A_k\}$, where A_i consists of n_i nodes and b_i boxes; (ii) a complete semiring $(\Sigma, \oplus, \otimes, \overline{\mathbf{0}}, \overline{\mathbf{1}})$; and (iii) a maximum stack height h. Our task is to create a data structure that after some preprocessing can answer queries of the following form:

1. Given a pair $((u, \varnothing), (v, \varnothing))$ of configurations, compute the semiring distance $d((u, \varnothing), (v, \varnothing), h)$.

2. Given a source configuration $(u, \varnothing)$, compute the semiring distance $d((u, \varnothing), (v, \varnothing), h)$ for every node v in the same CSM as u.

For this purpose, we present the algorithm RSMDistance, which performs such preprocessing using a data structure $\mathcal{D}$ consisting of the algorithms Preprocess, Update and Query of Section 3.2. At the end of RSMDistance it will hold that pair semiring-distance queries in a CSM A_i can be answered in $O(\log n_i)$ semiring operations. We later present some additional preprocessing which suffers a factor of $O(\log n_i)$ in the preprocessing space, but reduces the pair query time to constant.

Our algorithm RSMDistance can be viewed as a Bellman-Ford computation on the call graph of the RSM (i.e., a graph where every node corresponds to a CSM, and an edge connects two CSMs if one appears as a box in the other). Informally, RSMDistance consists of the following steps.

1. In a preprocessing phase, it uses Theorem 2.1 and Lemma 2.6 to compute a nicely rooted, balanced, binary tree decomposition $\mathrm{Tree}(G_i)$ of the CFG G_i each CSM A_i.

2. It preprocesses the control-flow graphs $G_i = (V_i, E_i')$ of the CSMs A_i using Preprocess of Section 3.2, where the weight function wt_i for each G_i is extended such that $\mathsf{wt}_i((en, b), (ex, b)) = \overline{\mathbf{0}}$ for all pairs of call and return nodes to the same box b. This allows the computation of $d(u, v, 0)$ for all pairs of nodes (u, v), since no call can be made while still having zero stack height.

3. Then, iteratively for each ℓ, where $1 \leqslant \ell \leqslant h$, given that we have a dynamic data structure $\mathcal{D}$ (concretely, an instance of the dynamic algorithms Update and Query from Section 3.2) for computing $d(u, v, \ell - 1)$, the algorithm does as follows: First, for each G_i whose entry to exit distance $d(\mathsf{en}_i, \mathsf{ex}_i, \ell - 1)$ has changed from the last iteration and for each G_j that contains a box pointing to G_i, it updates the call to return distance of the corresponding nodes, using Query.

4. Then, it obtains the entry to exit distance $d(\mathsf{en}_j, \mathsf{ex}_j, \ell)$ to see if it was modified, and continues with the next iteration of $\ell + 1$.

See Algorithm 6 for the formal description.

Correctness and logarithmic pair query time. The algorithm RSMDistance is described so

Algorithm 6: RSMDistance

Input: A set of control-flow graphs $\mathcal{G} = \{G_i\}_{1 \leqslant i \leqslant k}$, stack height h

```
1  foreach G_i ∈ 𝒢 do
2  |  Construct a nicely rooted, balanced, binary tree-decomposition Tree(G_i)
3  |  Call Preprocess on Tree(G_i)
4  end
5  distances ← [Call Query on (en_i, ex_i) of G_i]_{1⩽i⩽k}
6  modified ← {1, ..., k}
7  for ℓ ← 1 to h do
8  |  modified′ ← ∅
9  |  foreach i ∈ modified do
10 |  |  foreach G_j that contains boxes b_{j_1}, ..., b_{j_l} s.t. Y_j(b_{j_x}) = i do
11 |  |  |  Call Update on G_j for the weight change wt((en_i, b_{j_l}), (ex_i, b_{j_x})) ← distances[i]
12 |  |  |  δ ← Query(en_j, ex_j)
13 |  |  |  if δ ≠ distances[j] then
14 |  |  |  |  modified′ ← modified′ ∪ {j}
15 |  |  |  |  distances[j] ← δ
16 |  |  end
17 |  end
18 |  modified ← modified′
19 end
```

that a proof by induction is straightforward for correctness. Initially, running the algorithm Preprocess from Section 3.2 on each of the graphs G_i allows queries for the distances $d(u, v, 0)$ for all pairs of nodes (u, v), since no method call can be made. Also, the induction follows directly since for every CSM A_i, updating the distance from call nodes (en, b) to the corresponding return nodes (ex, b) of every box b that corresponds to a CSM A_j whose distance $d(\mathsf{en}_j, \mathsf{ex}_j)$ was changed in the last iteration ℓ, ensures that the distance $d(u, v, \ell + 1)$ of every pair of nodes u, v in A_i is computed correctly. This is also true for the special pair of nodes en_i, ex_i, which feeds the next iteration of RSMDistance. Finally, RSMDistance requires $O(\sum_{i=1}^{k}(n_i))$ time to construct a nicely rooted, balanced, binary tree decomposition (Theorem 2.1 and Lemma 2.6), $O(n)$ time to preprocess all G_i initially, and $O(\sum_{i=1}^{k}(b_i \cdot \log n_i))$ to update all G_i for one iteration of the loop of Line 4 (from Theorem 3.1). Hence, RSMDistance uses $O(\sum_{i=1}^{k}(n_i + h \cdot b_i \cdot \log n_i))$ preprocessing semiring operations. Finally, it is easy to verify that all preprocessing is done in $O(\sum_i n_i) = O(n)$ space.

After the last iteration of algorithm RSMDistance, we have a data structure $\mathcal{D}$ that occupies $O(n)$ space and answers distance queries $d(u, v, h)$ in $O(\log n_i)$ time, with $u, v \in V_i$, by calling Query from Section 3.2 for the distance $d(u, v)$ in G_i.

Example 4.1 (RSMDistance **on the RSM of Fig. 2.6).** We now present a small example of how RSMDistance is executed on the RSM of Fig. 2.6 for the case of reachability. In this case, for any pair of nodes (u, v), we have $d(u, v) = \mathsf{True}$ iff u reaches v. Table 4.5(a) illustrates how the local distance maps LUD_{B_x} look for each bag B_x of each of the CSMs of the two methods dot_vector and dot_matrix. Each column represents the local distance map of the corresponding bag B_x, and an entry (u, v) means that $\mathsf{LUD}_{B_x}(u, v) = \mathsf{True}$ (i.e., u reaches v). For brevity, in the table we hide self loops (i.e., entries of the form (u, u)) although they are stored by the algorithms. Initially, the stack height $\ell = 0$, and Preprocess is called for each graph (Line 3). The new reachability relations discovered by Merge are shown in bold. Note that at this point we have $\mathsf{wt}(4, 5) = \mathsf{False}$ in method dot_matrix, as we do not know whether the call to method dot_vector actually returns. Afterwards, Query is called to discover the distance $d(1, 6)$ in method dot_vector (Line 5). Table 4.5 (b) shows the sequence in which Query examines the bags of the tree decomposition, and the distances δ_1, δ_6 and δ it maintains. When B_2 is examined, $\delta = \mathsf{True}$ and hence at the end Query returns $\delta = \mathsf{True}$. Finally, since Query returns $\delta = \mathsf{True}$, the weight $\mathsf{wt}(4, 5)$ between the call-return pair of nodes $(4, 5)$ in method dot_matrix is set to True. An execution of Update (Line 11) with this update on the corresponding tree

	dot_vector						dot_matrix							
ℓ/LUD_{B_x}	B_1	B_2	B_3	B_4	B_5	B_6	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
$\ell = 0$ (Preprocess)	$-$	$(1,2)$	$(2,3)$	$(2,3)$ $(3,4)$ $(4,2)$ $(\mathbf{2},\mathbf{4})$	$(2,5)$	$(5,6)$	$-$	$(1,2)$	$(2,3)$	$(3,4)$	$(3,4)$ $(5,3)$	$(2,6)$ $(3,6)$ $(6,2)$ $(\mathbf{3},\mathbf{2})$	$(2,7)$	$(7,8)$
$\ell = 1$ (Update)	$-$	$(1,2)$	$(2,3)$	$(2,3)$ $(3,4)$ $(4,2)$ $(2,4)$	$(2,5)$	$(5,6)$	$-$	$(1,2)$	$(2,3)$	$(3,4)$	$(3,4)$ $(5,3)$ $(\mathbf{4},\mathbf{5})$ $(\mathbf{4},\mathbf{3})$	$(2,6)$ $(3,6)$ $(6,2)$ $(3,2)$	$(2,7)$	$(7,8)$

(a)

	dot_vector			
	B_6	B_5	B_2	B_1
Query	$\delta_6 = \{5,6\}$	$\delta_6 = \{2,5\}$	$\delta_6 = \{1,2\}$	$\delta_6 = \{1\}$
$d(1,6)$	$-$	$-$	$\delta_1 = \{1,2\}$	$\delta_1 = \{1\}$
	$-$	$-$	$\delta = \mathsf{True}$	$\delta = \mathsf{True}$

(b)

Table 4.5: Illustration of $\mathsf{RSMDistance}$ on the tree decompositions of methods dot_vector and dot_matrix from Fig. 2.6. Table 4.5a shows the local distance maps for each bag and stack height $\ell = 0, 1$. Table 4.5b shows how the distance query $d(1,6)$ in method dot_vector is handled.

decomposition (Table 4.5(a) for $\ell = 1$) updates the entries $(4,5)$ and $(4,3)$ in LUD_{B_5} of method dot_matrix (shown in bold). From this point, any same-context distance query can be answered in logarithmic time in the size of its CSM by further calls to Query.

Linear single-source query time. In order to handle single-source queries, some additional preprocessing is required. The basic idea is to use $\mathsf{RSMDistance}$ to process the graphs G_i, and then use additional preprocessing on each G_i by applying existing algorithms for graphs with constant treewidth. This is achieved using Lemma 2.4, which states that for each bag B of each tree-decomposition $\mathrm{Tree}(G_i)$, a *local distance map* $\mathsf{LD}_B : B \times B \to \Sigma$ with $\mathsf{LD}_B(u,v) = d(u,v)$ can be computed in time and space $O(n_i)$. After all such maps LD_B have been computed for each B, it is straightforward to answer single-source queries from some node u in linear time.

The algorithm simply maintains a map $A : V_i \to \Sigma$, and initially $A(v) = d(u,v)$ for all $v \in B_u$, and $A(v) = \overline{\mathbf{0}}$ otherwise. Then, it traverses $\mathrm{Tree}(G_i)$ in a BFS manner starting at B_u, and for every encountered bag B and $v \in B$, if $A(v) = \overline{\mathbf{0}}$, it sets $A(v) = \bigoplus_{z\in B} \bigotimes(A(z), d(z,v))$. The correctness follows directly from Lemma 2.3. For constant treewidth, this results in a constant number of semiring operations per bag, and hence $O(n_i)$ time in total.

Constant pair query time. After RSMDistance has returned, it is possible to further preprocess the graphs G_i to reduce the pair query time to constant, while increasing the space by a factor of $\log n_i$. For constant treewidth, this can be obtained by adapting [Chaudhuri and Zaroliagis, 1995, Theorem 10] to our setting, which in turn is based on a rather complicated algorithmic technique of [Alon and Schieber, 1987]. We present a more intuitive, simpler and implementable approach that has a dynamic programming nature. In Section 4.3 we present some experimental results obtained by this approach.

Recall that the extra preprocessing for answering single-source queries in linear time consists in computing the local distance maps LD_B for every bag B. To handle pair queries in constant time, we further traverse each $\mathrm{Tree}(G_i)$ one last time, bottom-up, and for each node u we store maps $\mathsf{F}_u, \mathsf{T}_u : V_i^{B_u} \to \Sigma$, where $V_i^{B_u}$ is the subset of V_i of nodes that appear in B_u and its descendants in $\mathrm{Tree}(G_i)$. The maps are such that $\mathsf{F}_u(v) = d(u,v)$ and $\mathsf{T}_u = d(v,u)$. Hence, F_u stores the distances from u to nodes in $V_i^{B_u}$, and T_u stores the distances from nodes in $V_i^{B_u}$ to u. The maps are computed in a dynamic programming fashion, as follows:

1. Initially, the maps F_u and T_u are constructed for all u that appear in a bag B which is a leaf of $\mathrm{Tree}(G_i)$. The information required has already been computed as part of the preprocessing for answering single-source queries. Then, $\mathrm{Tree}(G_i)$ is traversed up, level by level.

2. When examining a bag B such that the computation has been performed for all its children, for every node $u \in B$ and $v \in V_i^B$, we set $\mathsf{F}_u(v) = \bigoplus_{z\in B} \bigotimes\{d(u,z), \mathsf{F}_z(v)\}$, and similarly for $\mathsf{T}_u = \bigoplus_{z\in B} \bigotimes\{d(z,u), \mathsf{T}_z(v)\}$.

An application of Lemma 2.2 inductively on the levels processed by the algorithm can be used to show that when a bag B is processed, for every node $u \in B$ and $v \in V_i^B$, we have $\mathsf{T}_u(v) = \bigoplus_{P:v\rightsquigarrow u} \otimes(P)$ and $\mathsf{F}_u(v) = \bigoplus_{P:u\rightsquigarrow v} \otimes(P)$. Finally, there are $O(n_i)$ semiring operations done at each level of $\mathrm{Tree}(G_i)$, and since there are $O(\log n_i)$ levels, $O(n_i \cdot \log n_i)$ operations are required

in total. Hence, the space used is also $O(n_i \cdot \log n_i)$. We furthermore preprocess $\mathrm{Tree}(G_i)$ in linear time and space to answer LCA queries in constant time (note that since $\mathrm{Tree}(G_i)$ is balanced, this is standard). To answer a pair query u, v, it suffices to first obtain the LCA B of B_u and B_v, and it follows from Lemma 2.3 that $d(u,v) = \bigoplus_{z \in B} \bigotimes \{T_z(u), F_z(v)\}$, which requires a constant number of semiring operations.

Theorem 4.1. *Fix the following input: (i) a constant treewidth RSM $A = \{A_1, A_2, \dots, A_k\}$, where A_i consists of n_i nodes and b_i boxes; (ii) a complete semiring $(\Sigma, \oplus, \otimes, \overline{\mathbf{0}}, \overline{\mathbf{1}})$; and (iii) a maximum stack height h. Let $n = \sum_i n_i$.* RSMDistance *operates in the following complexity bounds.*

1. *Using $O(n + h \cdot \sum_{i=1}^{k} b_i \cdot \log n_i)$ semiring operations and $O(n)$ space, it correctly answers same-context semiring distance pair queries in $O(\log n_i)$, and same-context semiring distance single-source queries in $O(n_i)$ semiring operations.*

2. *Using $O(\sum_{i=1}^{k}(n_i \cdot \log n_i + h \cdot b_i \cdot \log n_i))$ semiring operations $O(\sum_{i=1}^{k}(n_i \cdot \log n_i))$ space, it correctly answers same-context semiring distance pair queries in $O(1)$ semiring operations, and same-context semiring distance single-source queries in $O(n_i)$ semiring operations.*

Remark 4.1. We note that the pair query time of Item 1 in Theorem 4.1 can be improved further to $O(\alpha(n_i))$ time, where $\alpha(n_i)$ is the inverse of the Ackermann function on input (n_i, n_i). This is achieved using [Chaudhuri and Zaroliagis, 1995, Theorem 10, Item (ii)], instead of the process described above. This result is only of theoretical interest, as (i) the hidden constants are large, and (ii) the data structure for achieving such bounds is hard to be implemented in practice.

IFDS/IDE framework. In the special case where the algebraic path problem belongs to the IFDS/IDE framework, we have a meet-composition semiring $(F, \sqcap, \circ, \varnothing, I)$, where F is a set of distributive flow functions $2^D \to 2^D$, D is a set of data facts, $\sqcap$ is the meet operator (either union or intersection), $\circ$ is the flow function composition operator, and I is the identity flow function. For a fair comparison, the $\circ$ semiring operation does not induce a unit time cost, but instead a cost of $O(|D|)$ per data fact (as functions are represented as bipartite graphs [Reps *et al.*, 1995a]). Because the set D is finite, and the meet operator is either union or intersection, it follows that the image of every data fact will be updated at most $|D|$ times. Hence, in the preprocessing phase where $\mathsf{Preprocess}(G_i)$ is called for each graph G_i, the total time spent for

each G_i is $O(n_i \cdot |D|^3)$. Additionally, Line 7 of RSMDistance needs to change so that instead of h iterations, the body of the loop is carried up to a fixpoint. The amortized cost for all edge updates per G_i is then $O(b_i \cdot \log n_i \cdot |D|^3)$ (as there are $|D|$ data facts), and we have the following corollary (also see Table 4.2).

In the query phase we fix a source node u (in the case of single-source queries) or a source/destination pair (u, v) (in the case of pair queries), as well as the set of data facts X that hold in the source node u (of either query). Since we deal with sets of data facts and not flow functions, each application of the composition operator yields a new set of data facts, and the meet operator corresponds to the union or intersection of two data-fact sets. Each such operation incurs a cost $O(|D|^2)$. We thus arrive at the following corollary.

Corollary 4.1 (IFDS/IDE). *Fix the following input a (i) constant treewidth RSM $A = \{A_1, A_2, \dots, A_k\}$, where A_i consists of n_i nodes and b_i boxes; and (ii) a meet-composition semiring $(F, \sqcap, \circ, \varnothing, I)$ where F is a set of distributive flow functions $D \to D$, $\circ$ is the flow function composition operator and $\sqcap$ is the meet operator. Let $n = \sum_i n_i$.* RSMDistance *operates in the following complexity bounds.*

1. *Using $O(|D|^3 \cdot (n + \sum_{i=1}^k b_i \cdot \log n_i)$ preprocessing time and $O(n \cdot |D|^2)$ space, it correctly answers same-context algebraic pair queries in $O(\log n_i \cdot |D|^2)$ time, and same-context algebraic single-source queries in $O(n_i \cdot |D|^2)$ time.*

2. *Using $O(|D|^3 \cdot \sum_{i=1}^k (n_i \cdot \log n_i))$ preprocessing time and $O(|D|^2 \cdot \sum_{i=1}^k (n_i \cdot \log n_i))$ space, it correctly answers same-context algebraic pair queries in $O(|D|^2)$ time, and same-context algebraic single-source queries in $O(n_i \cdot |D|^2)$ time.*

A speedup for large data-fact domains. Here we outline a speedup for the algebraic path problem wrt the IFDS framework when the domain of data facts D is such that $|D| = \Omega(\log n)$. In this case, sets of data facts can be represented as bit sets, where the i-th bit of a bit set X is one iff it contains the i-th data fact. When $|D| = \Omega(\log n)$, such bit sets can be stored compactly in machine words. Since in the standard RAM model each machine word has size $\Theta(\log n)$, such a set X can be stored using $O(|D|/\log n)$ words. The meet $\sqcap$ (union/intersection) of two data-fact sets can be performed in $O(|D|/\log n)$ time, by computing the bit-wise OR/AND operation on the corresponding machine words. Similarly, a distributive flow function $f : 2^D \to 2^D$ can be represented using $O(|D|^2/\log n)$ words, by storing a bit set X_i for every data fact d_i for

which $f(d_i) = X_i$. Using bit sets, every update of a flow function with a new data flow pair $d_i \to d_j$ incurs a $O(|D|/\log n)$ time cost, simply by performing the bit-wise OR/AND operation (depending on whether the meet operator is union/intersection) between the data-fact sets $f(d_i)$ and $f(d_j)$. Since there can be at most $|D|^2$ updates of data flow pairs $d_i \to d_j$ per graph edge, the total preprocessing cost for each graph G_i is $n_i \cdot |D|^3/\log n$. Similarly, in the update phase (Line 7) the amortized cost per G_i is $b_i \cdot \log n_i \cdot |D|^3/\log n$. Finally, in the query phase, where we track data facts rather than data-flow functions, data-fact operations require $O(|D|/\log n)$ word operations per data fact. We thus arrive at the following corollary.

Corollary 4.2 (IFDS/IDE, large domain)**.** *Fix the following input a (i) constant treewidth RSM $A = \{A_1, A_2, \dots, A_k\}$, where A_i consists of n_i nodes and b_i boxes; and (ii) a meet-composition semiring $(F, \sqcap, \circ, \varnothing, I)$ where F is a set of distributive flow functions $D \to D$ with $|D| = \Omega(\log n)$, $\circ$ is the flow function composition operator and $\sqcap$ is the meet operator. Let $n = \sum_i n_i$.* RSMDistance *operates in the following complexity bounds.*

1. *Using $O(n/\log n) \cdot |D|^3 \cdot \sum_{i=1}^{k} b_i \cdot \log n_i)$ preprocessing time and $O((n/\log n) \cdot |D|^2)$ space, it correctly answers same-context algebraic pair queries in $O(|D|^2)$ time, and same-context algebraic single-source queries in $O(n_i \cdot |D|^2/\log n)$ time.*

2. *Using $O(n \cdot |D|^3)$ preprocessing time and $O(n \cdot |D|^2)$ space, it correctly answers same-context algebraic pair queries in $O(|D|^2/\log n)$ time, and same-context algebraic single-source queries in $O(n_i \cdot |D|^2/\log n)$ time.*

Reachability. The special case of reachability is obtained from Corollary 4.1 by setting $|D| = 1$. In conjunction with Theorem 4.1, we obtain an improvement. This is achieved by executing the algorithm RSMDistance as before, in order to infer in every CSM A_i the reachability information between every pair of call and return nodes (c, r). Afterwards, every corresponding graph G_i can be viewed independently, so that we can use Reachability of Section 3.3 to further preprocess it in order to handle same-context reachability queries. This approach yields the following corollary.

Corollary 4.3 (Interprocedural Reachability)**.** *Fix the following input a (i) constant treewidth RSM $A = \{A_1, A_2, \dots, A_k\}$, where A_i consists of n_i nodes and b_i boxes. Let $n = \sum_i n_i$.* RSMDistance *uses $O(n + \sum_{i=1}^{k} b_i \cdot \log n_i)$ preprocessing time and $O(n)$ space, and correctly answers same-context pair reachability queries in $O(1)$ time, and same-context single-source reachability queries in $O(\frac{n_i}{\log n})$ time.*

Distances with non-negative weights. The distance (or shortest path) problem can be formulated on the tropical semiring $(\mathbb{R}_{\geqslant 0} \cup \{\infty\}, \min, +, \infty, 0)$. We consider that both semiring operators cost unit time (i.e., the weights occurring in the computation fit in a constant number of machine words). Since we consider non-negative weights, the distance between any pair of nodes is realized by an interprocedural path of stack height at most b, as no boxes need to appear more than once at any time in the stack of the path. Hence, we can solve the distance problem by setting $h = b$ in Theorem 4.1. However, our restriction to non-negative weights allows for a significant speedup, achieved by algorithm RSMDistanceTrop (see Algorithm 7). RSMDistanceTrop is obtained from RSMDistance by using a priority queue to store the distances from entries to exits. In each iteration of the while loop in Line 7 we extract the element of the queue with the smallest entry-to-exit distance, and update the entry-to-exit distances of all remaining elements in the queue that correspond to CSMs which invoke the CSM that corresponds to the extracted element. The algorithm has similar flavor to the classic Dijkstra's algorithm for distances on finite graphs with non-negative edge weights [Dijkstra, 1959; Cormen *et al.*, 2009]. The time complexity is $O(n)$ time for executing Preprocess, plus the time required for each execution of Update and Query. Note that Update is executed at least as many times as Query, and since both require time logarithmic in the size of the respective G_i, it suffices to count the total time spent on Update. Since Line 10 is executed at most once per box, the total time spent on Update is $O(\sum_{i=1}^{k} b_i \cdot \log n_i)$. We thus obtain the following corollary for distances with non-negative weights.

Corollary 4.4 (Interprocedural Shortest paths)**.** *Fix the following input a (i) constant treewidth RSM $A = \{A_1, A_2, \ldots, A_k\}$, where A_i consists of n_i nodes and b_i boxes; (ii) a tropical semiring* $(\mathbb{R}_{\geqslant 0} \cup \{\infty\}, \min, +, \infty, 0)$. *Let $n = \sum_i n_i$.* RSMDistanceTrop *operates in the following complexity bounds.*

1. *Using $O(n + \sum_{i=1}^{k} b_i \cdot \log n_i))$ preprocessing time and $O(n)$ space, it correctly answers same-context shortest path pair queries in $O(\log n_i)$, and same-context single-source distance queries in $O(n_i)$ time.*

2. *Using $O(\sum_{i=1}^{k} n_i \cdot \log n_i)$ preprocessing time and $O(\sum_{i=1}^{k}(n_i \cdot \log n_i))$ space, it correctly answers same-context pair distance queries in $O(1)$ time.*

Interprocedural witness paths. As in the case of simple graphs from Section 3.2, we can

Algorithm 7: RSMDistanceTrop

Input: A set of control-flow graphs $\mathcal{G} = \{G_i\}_{1 \leqslant i \leqslant k}$

```
1  foreach G_i ∈ 𝒢 do
2      Construct a nicely rooted, balanced, binary tree-decomposition Tree(G_i)
3      Call Preprocess on Tree(G_i)
4  end
5  distances ← [Call Query on (en_i, ex_i) of G_i]_{1⩽i⩽k}
6  PriorityQueueQ ← [(i, Call Query on (en_i, ex_i) of G_i)]_{1⩽i⩽k}
7  while Q is not empty do
8      (i, δ) ← Q.pop()
9      foreach G_j that contains boxes b_{j_1}, ..., b_{j_l} s.t. Y_j(b_{j_x}) = i do
10         Call Update on G_j for the weight change wt((en_i, b_{j_l}), (ex_i, b_{j_x})) ← δ
11         δ' ← Query(en_j, ex_j)
12         if δ' < distances[j] then
13             distances[j] ← δ'
14             Decrease the key of j in Q to δ'
15     end
16 end
```

retrieve a witness path for any distance $d(u, v, h)$ that is realized by acyclic interprocedural paths $P : (u, \varnothing) \rightsquigarrow (v, \varnothing)$, without affecting the stated complexities. The process is straightforward. Let A_i contain the pair of nodes u, v on which the query is asked. Initially, we obtain the witness intraprocedural path $P' : u \rightsquigarrow v$, as described in Section 3.2. Then, we proceed recursively to obtain a witness path P_j between the entry en_j and exit ex_j nodes of every CSM A_j such that P' contains an edge between a call node (en, b) and a return node (ex, b) with $Y_i(B) = j$. That is, we reconstruct a witness path for every call to a CSM whose weight has been summarized locally in A_i. This process constructs an interprocedural witness path $P : u \rightsquigarrow v$ such that $\otimes(P) = d(u, v)$ in $O(|P|)$ time.

4.3 Experimental Results

Setup. We have implemented our algorithms for linear-time single-source and constant-time pair queries presented in Section 4.2 and have tested them on graphs obtained from the DaCapo benchmark suit [Blackburn, 2006] that contains several, real-world Java applications. Every benchmark is represented as a RSM that consists of several CSMs, and each CSM corresponds to the control-flow graph of a method of the benchmark. We have used the Soot framework [Vallée-Rai *et al.*, 1999] for obtaining the control-flow graphs, where every node of the graph corresponds to one Jimple statement of Soot, and the tool of [van Dijk *et al.*, 2006a] to obtain their tree decompositions. Our experiments were run on a standard desktop computer with a 3.4GHz CPU, on a single thread.

Interprocedural reachability and intraprocedural distances. In our experiments, we focus on the important special case of reachability and distances with non-negative weights We have considered CSMs of moderate to large size (all CSMs with at least five hundred nodes), as for small CSMs the running times are negligible. The first step was to execute an interprocedural reachability algorithm from the program entry to discover all actual call to return edges $((\mathsf{en}, b), (\mathsf{ex}, b))$ of every CSM A_i (i.e., all invocations that actually return), and then consider the control-flow graphs G_i independently.

- *(Reachability).* For every G_i, the complete preprocessing in the case of reachability was done by executing n_i DFSs, one from each source node. The single-source query from u

Benchmarks			Interprocedural Reachability					
			Preprocessing		Query			
					Single-source		Pair	
	n	t	Ours	Complete	Ours	No Prepr.	Ours	No Prepr.
antlr	698	1.0	76316	136145	15.3	166.3	0.15	14.34
bloat	696	2.3	27597	54335	3.9	72.5	0.10	14.34
chart	1159	1.5	22191	90709	2.3	80.9	0.13	22.32
eclipse	656	1.6	37010	138905	6.7	239.1	0.19	15.76
fop	1209	1.7	30189	91795	2.9	60.6	0.12	43.0
hsqldb	698	1.0	55668	180333	13.0	219.0	0.14	13.89
jython	748	1.5	43609	68687	7.2	85.7	0.11	12.84
luindex	885	1.3	36015	142005	5.6	202.7	0.16	26.44
lusearch	885	1.3	51375	189251	12.8	211.4	0.13	26.01
pmd	644	1.4	31483	52527	2.5	83.9	0.13	12.5
xalan	698	1.0	57734	138420	8.0	235.0	0.19	14.28
Jflex	1091	1.6	51431	91742	3.1	50.8	0.11	20.46
muffin	1022	1.7	29905	66708	2.6	52.7	0.10	18.57
javac	711	1.8	32981	59793	4.8	75.2	0.11	11.86
polyglot	698	1.0	68643	150799	12.2	184.5	0.14	14.14

Table 4.6: Average statistics gathered from our experiments on the DaCapo benchmark suit. Times are in microseconds.

is answered by executing one DFS from u, and the pair query u, v is handled similarly, but we stop as soon as v is reached. This methodology correctly answers interprocedural same-context reachability queries on CMSs reachable from the program entry.

- *(Distances).* In the case of distances we performed intraprocedural analysis on each G_i. We assign both positive and negative weights to each edge of G_i uniformly at random from the range $[-10, 10]$. For general semiring path properties, the Bellman-Ford algorithm [Cormen *et al.*, 2009] is a very natural one, which in the case of distances weights can handle positive and negative weights, as long as there is no negative cycle. To have a meaningful comparison with Bellman-Ford (as a representative of a general

Benchmarks			Intraprocedural Shortest path					
			Preprocessing		Query			
					Single-source		Pair	
	n	t	Ours	Complete	Ours	No Prepr.	Ours	No Prepr.
antlr	698	1.0	221578	$1.13 \cdot 10^7$	251	24576	0.36	24576
bloat	696	2.3	87950	$1.15 \cdot 10^7$	257	25239	0.37	25239
chart	1159	1.5	125468	$1.24 \cdot 10^8$	398	88856	0.39	88856
eclipse	656	1.6	152293	$1.07 \cdot 10^7$	533	23639	0.46	23639
fop	1209	1.7	153728	$3.94 \cdot 10^8$	1926	113689	2.71	113689
hsqldb	698	1.0	215063	$1.23 \cdot 10^7$	236	24322	0.36	24322
jython	748	1.5	159085	$1.42 \cdot 10^7$	386	29958	0.32	29958
luindex	885	1.3	163108	$2.97 \cdot 10^7$	258	51192	0.37	51192
lusearch	885	1.3	219015	$2.90 \cdot 10^7$	254	50719	0.34	50719
pmd	644	1.4	140974	$9.14 \cdot 10^6$	327	22572	0.37	22572
xalan	698	1.0	186695	$1.10 \cdot 10^7$	380	24141	0.43	24141
Jflex	1091	1.6	154818	$1.24 \cdot 10^8$	231	83093	0.36	83093
muffin	1022	1.7	125938	$1.02 \cdot 10^8$	265	80878	0.38	80878
javac	711	1.8	117390	$1.31 \cdot 10^7$	370	26180	0.34	26180
polyglot	698	1.0	228758	$1.15 \cdot 10^7$	244	24400	0.35	24400

Table 4.7: Average statistics gathered from our experiments on the DaCapo benchmark suit. Times are in microseconds.

semiring framework), we considered both positive and negative weights, but do not allow negative cycles. We note that although our algorithm Algorithm 7 assumes non-negative weights, the case of intraprocedural distances, handled by algorithms Preprocess and Query of Section 3.2 as well as the algorithms presented in this section for constant time pair-query time can handle negative weights. For complete preprocessing we run the classic Floyd-Warshall algorithm. Under no preprocessing, for every single-source and pair query we run the Bellman-Ford algorithm.

Results. Our experimental results are shown in Tables 4.6 and 4.7. In each table, the second (resp. third) column shows the average number of nodes (resp. treewidth) of CSMs of each benchmark.

The running times of preprocessing are gathered by averaging over all CSMs in each benchmark. The running times of querying are gathered by averaging over all possible single-source and pair queries in each CSM, and then averaging over all CSMs in each benchmark. Below we make some observations on the experimental results.

1. The average treewidth of control-flow graphs is confirmed to be very small, and does not scale with the size of the graph. In fact, even the largest treewidth is four.

2. The preprocessing time of our algorithm is significantly less than the complete preprocessing, by factor of 1.5 to 4 times in the case of reachability, and by orders of magnitude in the case of distances.

3. In both reachability and distances, all queries are handled significantly faster after our preprocessing, than without preprocessing. We also note that for distance queries, Bellman-Ford answers single-source and pair queries in the same time, which is significantly slower than both our single-source and pair queries. Finally, we note that after our preprocessing, our data structure handles reachability queries faster than the DFS. Hence, the theoretical sublinear times of Corollary 3.1 are also obtained in practice.

Since our work focuses on same-context queries and the IFDS/IDE framework does not have this restriction, a direct comparison with the IFDS/IDE framework would be biased in our favor. In the experimental results for interprocedural reachability with same-context queries we show that we are faster than even DFS (which is faster than IFDS/IDE).

5 Optimal Dyck Reachability with Applications to Data-dependence and Alias Analysis

5.1 Introduction

In this chapter we present improved upper bounds, lower bounds, and experimental results for algorithmic problems related to Dyck reachability, which is a fundamental problem in static analysis. We present the problem description, its main applications, the existing results, and our contributions.

Alias analysis. Alias analysis has been one of the major types of static analysis and a subject of extensive study [Sridharan *et al.*, 2013; Choi *et al.*, 1993; Landi and Ryder, 1992; Hind, 2001]. The task is to decide whether two pointer variables may point to the same object during program execution. As the problem is computationally expensive [Horwitz, 1997; Ramalingam, 1994], practically relevant results are obtained via approximations. One popular way to perform alias analysis is via points-to analysis, where two variables may alias if their points-to sets intersect. Points-to analysis is typically phrased as a Dyck reachability problem on Symbolic Points-to Graphs (SPGs), which contain information about variables, heap objects and parameter passing due to method calls [Xu *et al.*, 2009a; Yan *et al.*, 2011a]. In alias analysis there is an important distinction between context and field sensitivity, which we describe below.

- *Context vs field sensitivity.* Typically, the Dyck parenthesis are used in SPGs to specify two types of constraints. *Context sensitivity* refers to the requirement that reachability

paths must respect the calling context due to method calls and returns. *Field sensitivity* refers to the requirement that reachability paths must respect field accesses of composite types in Java [Sridharan and Bodík, 2006a; Sridharan *et al.*, 2005; Xu *et al.*, 2009a; Yan *et al.*, 2011a], or references and dereferences of pointers [Zheng and Rugina, 2008] in C. Considering both types of sensitivity makes the problem undecidable [Reps, 2000]. Although one recent workaround is approximation algorithms [Zhang and Su, 2017], the standard approach has been to consider only one type of sensitivity. Field sensitivity has been reported to produce better results, and being more scalable [Lhoták and Hendren, 2006]. We focus on context-insensitive, but field-sensitive points-to analysis.

Data-dependence analysis. Data-dependence analysis aims to identify the def-use chains in a program. It has many applications, including slicing [Reps *et al.*, 1994], impact analysis [Arnold, 1996] and bloat detection [Xu *et al.*, 2010a]. It is also used in compiler optimizations, where data dependencies are used to infer whether it is safe to reorder or parallelize program statements [Kuck *et al.*, 1981]. Here we focus on the distinction between *library vs client analysis* and the challenge of *callbacks*.

- *Library vs Client.* Modern-day software is developed in multiple stages and is interrelated. The vast majority of software development relies on existing libraries and third-party components which are typically huge and complex. At the same time, the analysis of client code is ineffective if not performed in conjunction with the library code. These dynamics give rise to the potential of analyzing library code once, in an offline stage, and creating suitable analysis summaries that are relevant to client behavior only. The benefit of such a process is two-fold. First, library code need only be analyzed once, regardless of the number of clients that link to it. Second, it offers fast client-code analysis, since the expensive cost of analyzing the huge libraries has been spent offline, in an earlier stage. Data-dependence analysis admits a nice separation between library and client code, and has been studied in [Tang *et al.*, 2015; Palepu *et al.*, 2017].

- *The challenge of callbacks.* As pointed out recently in [Tang *et al.*, 2015], one major obstacle to effective library summarization is the presence of callbacks. Callback functions are declared and used by the library, but are implemented by the client. Since these functions are missing when the library code is analyzed, library summarization is ineffective and the whole library needs to be reanalyzed on the client side, when callback functions become

available.

Algorithmic formulations and existing results. We describe below the key algorithmic problems in the applications mentioned above and the existing results. We focus on data-dependence and alias analysis via Dyck reachability, which is the most standard way for performing such analysis. Recall that the problem of Dyck reachability takes as input a (directed) graph, where some edges are marked with opening and closing parenthesis, and the task is to compute for every pair of nodes whether there exists a path between them such that the parenthesis along its edges are matched.

1. *Points-to analysis.* Context-insensitive, field-sensitive points-to analysis via Dyck reachability is phrased on an SPG G with n nodes and m edges. Additionally, the graph is *bidirected*, meaning that if G has an edge (u, v) labeled with an opening parenthesis, then it must also have the edge (v, u) labeled with the corresponding closing parenthesis. Bidirected graphs are found in all existing works on alias analysis via Dyck reachability, and their importance has been remarked in various works [Yuan and Eugster, 2009; Zhang *et al.*, 2013].

   The best existing algorithms for the problem appear in the recent work of [Zhang *et al.*, 2013], where two algorithms are proposed. The first has $O(n^2)$ *worst-case* time complexity; and the second has $O(m \cdot \log n)$ *average-case* time complexity and $O(m \cdot n \cdot \log n)$ *worst-case* complexity. Note that for dense graphs $m = \Theta(n^2)$, and the first algorithm has better average-case complexity too. We elaborate further on the difference between average-case and worst-case complexities in Remark 5.3 of Section 5.3.2.

2. *Library/Client data-dependence analysis.* The standard algorithmic formulation of context-sensitive data-dependence analysis is via Dyck reachability, where the parenthesis are used to properly match method calls and returns in a context-sensitive way [Reps, 2000; Tang *et al.*, 2015]. The algorithmic approach to Library/Client Dyck reachability consists of considering two graphs G_1 and G_2, for the library and client code respectively. The computation is split into two phases. In the *preprocessing phase*, the Dyck reachability problem is solved on G_1 (using a CFL/Dyck reachability algorithm), and some summary information is maintained, which is typically in the form of some subgraph G_1' of G_1. In the *query phase*, the Dyck reachability problem is solved on the combination of the two

graphs G_1' and G_2. Let n_1, n_2 and n_1' be the sizes of G_1, G_2 and G_1' respectively. The algorithm spends $O(n_1^3)$ time in the preprocessing phase, and $O((n_1' + n_2)^3)$ time in the query phase. Hence we have an improvement if $n_1' >> n_1$.

In the presence of callbacks, library summarization via CFL reachability is ineffective, as n_1' can be as large as n_1. To face this challenge, the recent work of [Tang *et al.*, 2015] introduced TAL reachability. This approach spends $O(n_1^6)$ time on the client code (hence more than the CFL reachability algorithm), and is able to produce a summary of size $s < n_1$ even in the presence of callbacks. Afterwards, the client analysis is performed in $O((s + n_2)^6)$ time, and hence the cost due to the library only appears in terms of its summary.

3. *Dyck reachability on general graphs.* As we have already mentioned, Dyck reachability is a fundamental algorithmic formulation of many types of static analysis. For general graphs (not necessarily bidirected), the existing algorithms require $O(n^3)$ time, and they essentially solve the more general CFL reachability problem [Yannakakis, 1990]. The current best algorithm is due to [Chaudhuri, 2008], which utilizes the well-knwon Four Russians' Trick to exhibit complexity $O(n^3/\log n)$. The combinatorial cubic barrier for CFL parsing [Lee, 2002] implies the same barrier for CFL reachability [Reps, 1997]. On the other hand, Dyck parsing is linear-time solvable, which leaves a lot of room for truly sub-cubic algorithms for Dyck reachability.

Our contributions. Our main contributions can be characterized in three parts: (a) improved upper bounds; (b) lower bounds with optimality guarantees; and (c) experimental results. We present the details of each of them below.

Improved upper bounds. Our improved upper bounds are as follows:

1. For Dyck reachability on bidirected graphs with n nodes and m edges, we present an algorithm with the following bounds: (a) The worst-case complexity bound is $O(m + n \cdot \alpha(n))$ time and $O(m)$ space, where $\alpha(n)$ is the inverse Ackermann function, improving the previously known $O(n^2)$ time bound. Note that $\alpha(n)$ is an extremely slowly growing function, and for all practical purposes, $\alpha(n) \leqslant 4$, and hence practically the worst-case bound of our algorithm is linear. (b) The average-case complexity is $O(m)$ improving the previously known $O(m \cdot \log n)$ bound. See Table 5.1 for a summary.

2. For *Library/Client Dyck reachability* we exploit the fact that the data-dependence graphs that arise in practice have special structure, namely they contain components of small treewidth. We denote by n_1 and n_2 the size of the library graph and client graph, and by k_1 and k_2 the number of call sites in the library graph and client graph, respectively. We present an algorithm that analyzes the library graph in $O(n_1 + k_1 \cdot \log n_1)$ time and $O(n_1)$ space. Afterwards, the library and client graphs are analyzed together only in $O(n_2 + k_1 \cdot \log n_1 + k_2 \cdot \log n_2)$ time and $O(n_1 + n_2)$ space. Hence, since typically $n_1 >> n_2$ and $n_i >> k_i$, the cost of analyzing the large library occurs only in the preprocessing phase. When the client code needs to be analyzed, the cost incurred due to the library code is small. See Table 5.2 for a summary.

Lower bounds and optimality guarantees. Along with improved upper bounds we present lower bound and conditional lower bound results that imply optimality guarantees. We note that optimal guarantees for graph algorithms are rare.

1. For Dyck reachability on bidirected graphs we present a matching lower bound of $\Omega(m + n \cdot \alpha(n))$ for the worst-case time complexity. Thus we obtain matching lower and upper bounds for the worst-case complexity, and thus our algorithm is optimal wrt to worst-case complexity. Since the average-case complexity of our algorithm is linear, the algorithm is also optimal wrt the average-case complexity.

2. For *Library/Client Dyck reachability* note that $k_1 \leqslant n_1$ and $k_2 \leqslant n_2$. Hence our algorithm for analyzing library and client code is almost linear time, and hence optimal wrt polynomial improvements.

3. For Dyck reachability on general graphs we present a conditional lower bound. In algorithmic study, a standard problem for showing conditional cubic lower bounds is Boolean Matrix Multiplication (BMM) [Lee, 2002; Henzinger *et al.*, 2015; Vassilevska Williams and Williams, 2010; Abboud and Vassilevska Williams, 2014]. While fast matrix multiplication algorithms exist (such as Strassen's algorithm [Strassen, 1969]), these algorithms are not "combinatorial"[1]. The standard conjecture (called the BMM conjecture) is that

[1]Not combinatorial means algebraic methods [Le Gall, 2014], which are algorithms with large constants. In contrast, combinatorial algorithms are discrete and non-algebraic; for detailed discussion see [Henzinger *et al.*, 2015]

	Worst-case Time	Average-case Time	Space	Reference
Existing	$O\left(n^2\right)$	$O\left(\min\left(n^2, m \cdot \log n\right)\right)$	$O(m)$	[Zhang *et al.*, 2013]
Our Result	$O(m + n \cdot \alpha(n))$	$O(m)$	$O(m)$	Theorem 5.1 , Corollary 5.1

Table 5.1: Comparison of our results with existing work for Dyck reachability on bidirected graphs with n nodes and m edges. We also prove a matching lower-bound for the worst-case analysis. Thus our algorithm is optimal wrt worst-case complexity.

there is no truly sub-cubic[2] combinatorial algorithm for BMM, which has been widely used in algorithmic studies for obtaining various types of hardness results [Lee, 2002; Henzinger *et al.*, 2015; Vassilevska Williams and Williams, 2010; Abboud and Vassilevska Williams, 2014]. We show that Dyck reachability on general graphs is BMM hard. More precisely, we show that for any $\delta > 0$, any algorithm that solves Dyck reachability on general graphs in $O(n^{3-\delta})$ time implies an algorithm that solves BMM in $O(n^{3-\delta/3})$ time. Since all algorithms for Dyck reachability are combinatorial, it establishes a conditional hardness result (under the BMM conjecture) for general Dyck reachability. Our hardness shows that the existing cubic algorithms are optimal (modulo logarithmic factor improvements), under the BMM conjecture.

Experimental results. A key feature of our algorithms are that they are simple to implement. We present experimental results both on alias analysis (see Section 5.6.1) as well as library/client data-dependence analysis (see Section 5.6.2) and show that our algorithms outperform previous approaches for the problems on real-world benchmarks.

Organization The rest of this chapter is organized as follows.

1. In Section 5.2 we present some definitions regarding Dyck languages and Dyck reachability.

2. In Section 5.3 we present a new algorithm for performing Dyck reachability on bidirected graphs, and prove its optimality wrt worst-case and average-case complexity.

[2]Truly sub-cubic means polynomial improvement, in contrast to improvement by logarithmic factors such as $O(n^3/\log n)$

Approach	**Time**		**Space**		Reference
	Library	Client	Library	Client	
CFL	$O\left(n_1^3\right)$	$O\left((n_1+n_2)^3\right)$	$O\left(n_1^2\right)$	$O\left((n_1+n_2)^2\right)$	[Tang *et al.*, 2015]
TAL	$O\left(n_1^6\right)$	$O\left((s+n_2)^6\right)$	$O\left(n_1^4\right)$	$O\left((s+n_2)^4\right)$	[Tang *et al.*, 2015]
Our Result	$O\left(n_1 + k_1 \cdot \log n_1\right)$	$O\left(n_2 + \sum_{i=1}^{2} k_i \cdot \log n_i\right)$	$O(n_1)$	$O(n_1+n_2)$	Theorem 5.5

Table 5.2: Library/Client CFL reachability on the library graph of size n_1 and the client graph of size n_2.
s is the number of library summary nodes, as defined in [Tang *et al.*, 2015].
k_1 is the number of call sites in the library code, with $k_1 < s$.
k_2 is the number of call sites in the client code.

3. In Section 5.4 we show that Dyck reachability on general graphs is Boolean Matrix Multiplication-hard. The result also holds for graphs of low treewidth.

4. In Section 5.5 we present our approach to library summarization wrt Dyck reachability for context-sensitive data-dependence analysis.

5. In Section 5.6 we present an experimental evaluation of the algorithms described in Section 5.3 and Section 5.5, and compare them to the current state-of-the-art algorithms found in the literature.

5.2 Preliminaries

Dyck Languages. Given a nonnegative integer $k \in \mathbb{N}$, we denote by $\Sigma_k = \{\epsilon\} \cup \{\alpha_i, \overline{\alpha}_i\}_{i=1}^{k}$ a finite *alphabet* of k parenthesis types, together with a null element ϵ. We denote by $\mathcal{L}_k$ the Dyck language over Σ_k, defined as the language of strings generated by the following context-free grammar $\mathcal{G}_k$:

$$\mathcal{S} \to \mathcal{S}\,\mathcal{S} \mid \mathcal{A}_1\,\overline{\mathcal{A}}_1 \mid \ \dots \ \mid \mathcal{A}_k\,\overline{\mathcal{A}}_k \mid \epsilon\ ; \qquad \mathcal{A}_i \to \alpha_i\,\mathcal{S}\ ; \qquad \overline{\mathcal{A}}_i \to \mathcal{S}\,\overline{\alpha}_i$$

Given a string s and a non-terminal symbol X of the above grammar, we write $X \vdash s$ to denote that X produces s according to the rules of the grammar. In the rest of the chapter we consider an alphabet Σ_k and the corresponding Dyck language $\mathcal{L}_k$. We also let $\Sigma_k^O = \{\alpha_i\}_{i=1}^k$ and $\Sigma_k^C = \{\overline{\alpha}_i\}_{i=1}^k$ be the subsets of Σ_k of only opening and closing parenthesis, respectively.

Labeled graphs, Dyck reachability, and Dyck SCCs (DSCCs). We denote by $G = (V, E)$ a Σ_k-labeled directed graph where V is the set of nodes and $E \subseteq V \times V \times \Sigma_k$ is the set of edges labeled with symbols from Σ_k. Hence, an edge e is of the form $e = (u, v, \lambda)$ where $u, v \in V$ and $\lambda \in \Sigma_k$. We require that for every $u, v \in V$, there is a unique label λ such that $(u, v, \lambda) \in E$. Often we will be interested only on the endpoints of an edge e, in which case we represent $e = (u, v)$, and will denote by $\lambda(e)$ the label of e. Given a path P, we define the *label* of P as $\lambda(P) = \lambda(e_1) \dots \lambda(e_r)$. Given two nodes u, v, we say that v is Dyck-reachable from u if there exists a path $P : u \rightsquigarrow v$ such that $\lambda(P) \in \mathcal{L}_k$. In that case, P is called a *witness path* of the reachability. A set of nodes $X \subseteq V$ is called a *Dyck SCC* (or *DSCC*) if for every pair of nodes $u, v \in X$, we have that u reaches v and v reaches u. Note that there might exist a DSCC X and a pair of nodes $u, v \in X$ such that every witness path $P : u \rightsquigarrow v$ might be such that $P \not\subseteq X$, i.e., the witness path contains nodes outside the DSCC. Note that is in contrast to the usual SCCs.

5.3 Dyck Reachability on Bidirected Graphs

In this section we present an optimal algorithm for solving the Dyck reachability problem on Σ_k-labeled *bidirected* graphs G. First, in Section 5.3.1, we formally define the problem. Second, in Section 5.3.2, we describe an algorithm BidirectedReach that solves the problem in time $O(m + n \cdot \alpha(n))$, where n is the number of nodes of G, m is the number of edges of G, and $\alpha(n)$ is the inverse Ackermann function. Finally, in Section 5.3.3, we present an $\Omega(m + n \cdot \alpha(n))$ lower bound.

5.3.1 Problem definition

We start with the problem definition of Dyck reachability on bidirected graphs. For the modeling power of bidirected graphs we refer to [Yuan and Eugster, 2009; Zhang *et al.*, 2013] and our

results Section 5.6.

Bidirected Graphs. A Σ_k labeled graph $G = (V, E)$ is called *bidirected* if for every pair of nodes $u, v \in V$, the following conditions hold. (1) $(u, v, \epsilon) \in E \quad \text{iff} \quad (v, u, \epsilon) \in E$; and (2) for all $1 \leqslant i \leqslant k$ we have that $(u, v, \alpha_i) \in E \quad \text{iff} \quad (v, u, \overline{\alpha}_i) \in E$. Informally, the edge relation is symmetric, and the labels of symmetric edges are complimentary wrt to opening and closing parenthesis.

Remark 5.1 ([Zhang *et al.*, 2013]). For bidirected graphs the Dyck reachability relation forms an equivalence, i.e., for all bidirected graphs G, for every pair of nodes u, v, we have that v is Dyck-reachable from u iff u is Dyck-reachable from v.

Remark 5.2. We consider without loss of generality that a bidirected graph G has no edge (u, v) such that $\lambda(u, v) = \epsilon$, i.e., there are no ϵ-labeled edges. This is because in such a case, u, v form a DSCC, and can be merged into a single node. Merging all nodes that share an ϵ-labeled edge requires only linear time, and hence can be applied as a preprocessing step at (asymptotically) no extra cost.

Dyck reachability on bidirected graphs. We are given a Σ_k-labeled bidirected graph $G = (V, E)$, and our task is to compute for every pair of nodes u, v whether v is Dyck-reachable from u. As customary, we consider that $k = O(1)$, i.e., k is fixed wrt to the input graph [Chaudhuri, 2008]. In view of Remark 5.1, it suffices that the output is a list of DSCCs. Note that this way the output has size $\Theta(n)$ instead of $\Theta(n^2)$ that would be required for storing one bit of information per u, v pair. Additionally, the pair query time is $O(1)$, simply by testing whether the two nodes belong to the same DSCC.

5.3.2 An almost linear-time algorithm

We present our algorithm BidirectedReach, for Dyck reachability on bidirected graphs, with almost linear-time complexity.

Informal description of BidirectedReach. We start by providing a high-level description of BidirectedReach. The main idea is that for any two distinct nodes u, v to belong to some DSCC X, there must exist two (not necessarily distinct) nodes x, y that belong to some DSCC Y (possibly $X = Y$) and a closing parenthesis $\overline{\alpha}_i \in \Sigma_k^C$ such that $(x, u, \overline{\alpha}_i), (y, v, \overline{\alpha}_i) \in E$. The

algorithm uses a Disjoint Sets data structure to maintain DSCCs discovered so far. Each DSCC is represented as a tree T rooted on some node $x \in V$, and x is the only node of T that has outgoing edges. However, any node of T can have incoming edges. See Fig. 5.1 for an illustration. Upon discovering that a root node x of some tree T has two or more outgoing edges $(x, u_1, \overline{\alpha}_i), (x, u_2, \overline{\alpha}_i), \dots (x, u_r, \overline{\alpha}_i)$, for some $\overline{\alpha}_i \in \Sigma_k^C$, the algorithm uses r Find operations of the Disjoint Sets data structure to determine the trees T_i that the nodes u_i belong to. Afterwards, a Union operation is performed between all T_i to form a new tree T, and all the outgoing edges of the root of each T_i are merged to the outgoing edges of the root of T.

Complexity overview. The cost of every Find and Union operation is bounded by the inverse Ackermann function $\alpha(n)$ (see [Tarjan, 1975]), which, for all practical purposes, can be considered constant. Additionally, every edge-merge operation requires constant time, using a linked list for storing the outgoing edges. Although list merging in constant time creates the possibility of duplicate edges, such duplicates come at no additional complexity cost. Since every Union of k trees reduces the number of existing edges by $k-1$, the overall complexity of BidirectedReach is $O(m \cdot \alpha(n))$. We later show how to obtain the $O(m + n \cdot \alpha(n))$ complexity.

We are now ready to give the formal description of BidirectedReach. We start with introducing the Union-Find problem, and its solution given by a disjoint sets data structure.

The Union-Find problem. The Union-Find problem is a well-studied problem in the area of algorithms and data structures [Galil and Italiano, 1991; Cormen *et al.*, 2009]. The problem is defined over a *universe* X of n elements, and the task is to maintain partitions of X under set union operations. Initially, every element $x \in X$ belongs to a singleton set $\{x\}$. A *union-find* sequence σ is a sequence of m (typically $m \geqslant n$) operations of the following two types.

1. Union(x, y), for $x, y \in X$, performs a union of the sets that x and y belong to.

2. Find(x), for $x \in X$, returns the name of the unique set containing x.

The sequence σ is presented online, i.e., an operation needs to be completed before the next one is revealed. Additionally, a Union(x, y) operation is allowed in the i-th position of σ only if the prefix of σ up to position $i-1$ places x and y on different sets. The output of the problem consists of the answers to Find operations of σ. It is known that the problem can be solved in $O(m \cdot \alpha(n))$ time, by an appropriate Disjoint Sets data structure [Tarjan, 1975], and that this

complexity is optimal [Aspvall *et al.*, 1979; Banachowski, 1980].

The DisjointSets **data structure.** We consider at our disposal a Disjoint Sets data structure DisjointSets which maintains a set of subsets of V under a sequence of set union operations. At all times, the name of each set X is a node $x \in X$ which is considered to be the representative of x. DisjointSets provides the following operations.

1. For a node u, $\mathsf{MakeSet}(u)$ constructs the singleton set $\{u\}$.

2. For a node u, $\mathsf{Find}(u)$ returns the representative of the set that u belongs to.

3. For a set of nodes $S \subseteq V$ which are pairwise in different sets, and a distinguished node $x \in S$, $\mathsf{Union}(S, x)$ performs the union of the sets that the nodes in S belong to, and makes x the representative of the new set.

The DisjointSets data structure can be straightforwardly obtained from the corresponding Disjoint Sets data structures used to solve the Union-Find problem [Tarjan, 1975], and has $O(\alpha(n))$ amortized complexity per operation. Typically each set is stored as a rooted tree, and the root node is the representative of the set.

Formal description of BidirectedReach**.** We are now ready to present formally BidirectedReach in Algorithm 8 (for the initialization phase) and Algorithm 9 (for the computation phase). Recall that, in view of Remark 5.2, we consider that the input graph has no ϵ-labeled edges. In the initialization phase (Algorithm 8), the algorithm constructs a map $\mathsf{Edges} : V \times \Sigma_k^C \to V^*$. For each node $u \in V$ and closing parenthesis $\overline{\alpha}_i \in \Sigma_k^C$, $\mathsf{Edges}[u][\overline{\alpha}_i]$ will store the nodes that are found to be reachable from u via a path P such that $\overline{\mathcal{A}}_i \vdash \lambda(P)$ (i.e., the label of P has matching parenthesis except for the last parenthesis $\overline{\alpha}_i$). Observe that all such nodes must belong to the same DSCC.

The main computation happens in Algorithm 9. Upon extracting an element $(u, \overline{\alpha}_i)$ from the queue, the algorithm obtains the representatives v of the sets of the nodes in $\mathsf{Edges}[u][\overline{\alpha}_i]$. Since all such nodes belong to the same DSCC, the algorithm chooses an element x to be the new representative, and performs a Union operation of the underlying sets. The new representative x gathers the outgoing edges of all other nodes $v \in \mathsf{Edges}[u][\overline{\alpha}_i]$, and afterwards $\mathsf{Edges}[u][\overline{\alpha}_i]$ points only to x. We note that our requirement for specifying the representative of a set union

operation (i.e., the x in $\mathsf{Union}(S, x)$) is only so that x and u are distinct nodes in the the main loop of Algorithm 9, which makes the algorithm easier to present.

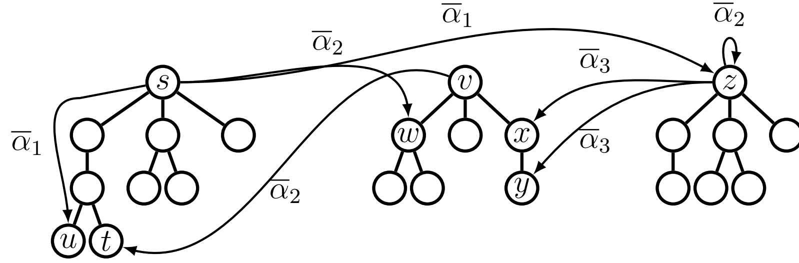


Figure 5.1: A state of BidirectedReach consists of a set of trees, with outgoing edges coming only from the root of each tree.

Algorithm 8: BidirectedReach Initialization

Input: A Σ_k-labeled bidirected graph $G = (V, E)$

Output: A DisjointSets map of DSCCs

```
   // Initialization
1  Q ← an empty queue
2  Edges ← a map V × Σ_k^C → V* implemented as a linked list
3  DisjointSets ← a disjoint-sets data structure over V
4  foreach u ∈ V do
5  |  DisjointSets.MakeSet(u)
6  |  for i ← 1 to k do
7  |  |  Edges[u][ᾱ_i] ← (v : (u, v, ᾱ_i) ∈ E)
8  |  |  if |Edges[u][ᾱ_i]| ⩾ 2 then
9  |  |  |  Insert (u, ᾱ_i) in Q
10 |  end
11 end
```

Example. We illustrate our algorithm on an example. Consider the state of the algorithm given by Figure 5.1. There are currently 3 DSCCS, with representatives s, v and z. Observe that nodes s and z have at two outgoing edges each that have the same type of parenthesis, hence they must have been inserted in the queue $\mathcal{Q}$ at some point. Assume that $\mathcal{Q} = [(s, \overline{\alpha}_1), (z, \overline{\alpha}_3)]$. The algorithm will exhibit the following sequence of steps.

Algorithm 9: BidirectedReach Computation

Input: A Σ_k-labeled bidirected graph $G = (V, E)$

Output: A DisjointSets map of DSCCs

```
   // Computation
1  while Q is not empty do
2  |  Extract (u, ᾱ_i) from Q
3  |  if u = DisjointSets.Find(u) then
4  |  |  Let S ← {DisjointSets.Find(w) : w ∈ Edges[u][ᾱ_i]}
5  |  |  if |S| ⩾ 2 then
6  |  |  |  Let x ← some arbitrary element of S \ {u}
7  |  |  |  Make DisjointSets.Union(S, x)
8  |  |  |  for j ← 1 to k do
9  |  |  |  |  foreach v ∈ S \ {x} do
10 |  |  |  |  |  if u ≠ v or i ≠ j then
11 |  |  |  |  |  |  Move Edges[v][ᾱ_j] to Edges[x][ᾱ_j]
12 |  |  |  |  |  else
13 |  |  |  |  |  |  Append (x) to Edges[x][ᾱ_j]
14 |  |  |  |  |  end
15 |  |  |  |  end
16 |  |  |  |  if |Edges[x][ᾱ_j]| ⩾ 2 then
17 |  |  |  |  |  Insert (x, ᾱ_j) in Q
18 |  |  |  end
19 |  |  else
20 |  |  |  Let x ← the single node in S
21 |  |  end
22 |  |  if u ∉ S or |S| = 1 then
23 |  |  |  Edges[u][ᾱ_i] ← (x)
24 end
25 return DisjointSets
```

1. The element $(z, \overline{\alpha}_3)$ is extracted from $\mathcal{Q}$. We have $\mathsf{Edges}[z][\overline{\alpha}_3] = (x, y)$. Observe that x and y belong to the same DSCC rooted at v, hence in Line 4 the algorithm will construct $S = \{v\}$. Since $|S| = 1$, the algorithm will simply set $\mathsf{Edges}[z][\overline{\alpha}_3] = (v)$ in Line 22, and no new DSCC has been formed.

2. The element $(s, \overline{\alpha}_1)$ is extracted from $\mathcal{Q}$. We have $\mathsf{Edges}[s][\overline{\alpha}_1] = (u, z)$. Since u and z belong to different DSCCs, the algorithm will construct $S = \{s, z\}$, and perform a $\mathsf{DisjointSets.Union}(S, x)$ operation, where $x = z$. Note that union-by-rank will make the tree of z a subtree of the tree of s, i.e., z will become a child of s. Afterwards, the algorithm swaps the names of z and s, as required by the choice of x in Line 6. Finally, in Line 11, the algorithm will move $\mathsf{Edges}[s][\overline{\alpha}_i]$ to $\mathsf{Edges}[z][\overline{\alpha}_i]$ for $i = 1, 2$. Since now $|\mathsf{Edges}[z][\overline{\alpha}_2]| \geqslant 2$, the algorithm inserts $(z, \overline{\alpha}_2)$ in $\mathcal{Q}$. See Figure 5.2a.

3. The element $(z, \overline{\alpha}_2)$ is extracted from $\mathcal{Q}$. We have $\mathsf{Edges}[z][\overline{\alpha}_2] = (v, z)$. Since v and z belong to different DSCCs, the algorithm will construct $S = \{v, z\}$, and perform a $\mathsf{DisjointSets.Union}(S, x)$ operation, where $x = v$. Note that union-by-rank will make the tree of v a subtree of the tree of z, i.e., v will become a child of z. Afterwards, the algorithm swaps the names of v and z, as required by the choice of x in Line 6. Finally, in Line 11, the algorithm will move $\mathsf{Edges}[z][\overline{\alpha}_2]$ to $\mathsf{Edges}[v][\overline{\alpha}_2]$. Since now $|\mathsf{Edges}[v][\overline{\alpha}_2]| \geqslant 2$, the algorithm inserts $(v, \overline{\alpha}_2)$ in $\mathcal{Q}$. See Figure 5.2b.

4. The element $(v, \overline{\alpha}_2)$ is extracted from $\mathcal{Q}$. We have $\mathsf{Edges}[v][\overline{\alpha}_2] = (v, t)$. Observe that v and t belong to the same DSCC rooted at v, hence in Line 4 the algorithm will construct $S = \{v\}$. Since $|S| = 1$, the algorithm will simply set $\mathsf{Edges}[v][\overline{\alpha}_2] = (v)$ in Line 22, and will terminate.

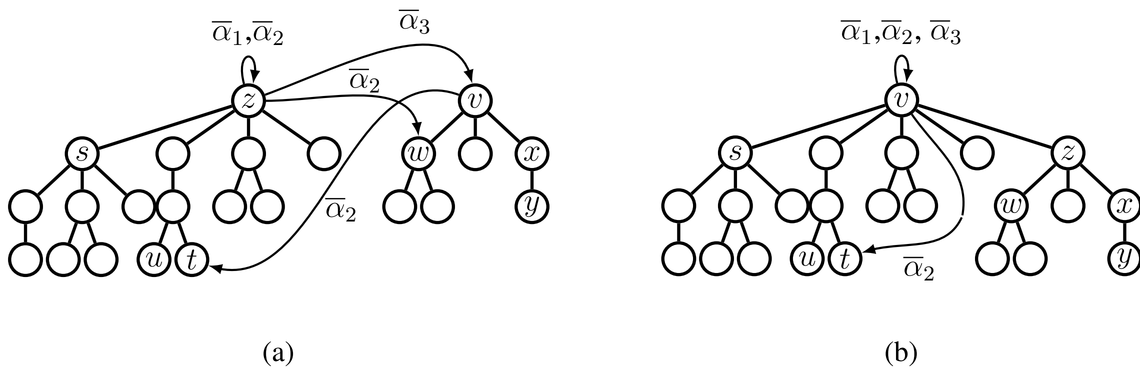


Figure 5.2: The intermediate stages of $\mathsf{BidirectedReach}$ starting from the stage of Figure 5.1.

Correctness. We start with the correctness statement of BidirectedReach, which is established in two parts, namely the soundness and completeness, which are shown in the following two lemmas.

Lemma 5.1 (Soundness). *At the end of* BidirectedReach, *for every pair of nodes* $u, v \in V$, *if* DisjointSets.Find(u) = DisjointSets.$Find(v)$ *then* u *and* v *belong to the same DSCC.*

Proof. The proof is by showing by induction on the number of times the main loop of Algorithm 9 is executed, that

1. every set of DisjointSets forms a DSCC of G, and
2. for every pair $u, v \in V$ and $\overline{\alpha}_i \in \Sigma^C$, if v is in the same set of DisjointSets as some node $w \in$ Edges$[u][\overline{\alpha}_i]$ then there are paths $P_v^u : v \rightsquigarrow u$ and $P_u^v : u \rightsquigarrow v$ such that $\mathcal{A}_i \vdash \lambda(P_v^u)$ and $\overline{\mathcal{A}}_i \vdash \lambda(P_u^v)$.

The claim follows easily after the initialization phase, since every set of DisjointSets is a single node, and the Edges map is initialized with the edges of G. Now, assume that the claim holds before an execution of the main loop of Algorithm 9. Let u be the node as defined in Line 2, and v_1, v_2 be two nodes of the set S constructed in Line 4. By the induction hypothesis, there exist paths $P_{v_1}^u : v_1 \rightsquigarrow u$ and $P_u^{v_2} : u \rightsquigarrow v_2$ such that $\mathcal{A}_i \vdash \lambda(P_{v_1}^u)$ and $\overline{\mathcal{A}}_i \vdash \lambda(P_u^{v_2})$, and thus $\mathcal{S} \vdash \lambda(P_{v_1}^u \circ P_u^{v_2})$ and $\lambda(P_{v_1}^u \circ P_u^{v_2}) \in \mathcal{L}_k$. Hence v_2 is Dyck reachable from v_1 and Item 1 holds.

We now consider Item 2 of the claim, and let u be the node defined in the previous paragraph. Let x be the node of S defined in Line 6, and v any other node of S. By the induction hypothesis, for every j in Line 8, every $y \in$ Edges$[v][\overline{\alpha}_j]$ and every z in the same set as y, we have that there exist paths $P_v^z : v \rightsquigarrow z$ and $P_z^v : z \rightsquigarrow v$ such that $\mathcal{A}_j \vdash \lambda(P_v^z)$ and $\overline{\mathcal{A}}_j \vdash \lambda(P_z^v)$. Additionally, there exist paths $P_v^u : v \rightsquigarrow u$, $P_u^x : u \rightsquigarrow x$, $P_x^u : x \rightsquigarrow u$ and $P_u^v : u \rightsquigarrow v$ such that $\mathcal{S} \vdash \lambda(P_v^u \circ P_u^x), \lambda(P_x^u \circ P_u^v)$. Let $P_z^x = P_z^v \circ P_v^u \circ P_u^x$ and $P_x^z = P_x^u \circ P_u^v \circ P_v^z$. Observe that $P_v^x : v \rightsquigarrow x$ and $P_x^z = x \rightsquigarrow z$, and additionally $\mathcal{A}_j \vdash P_z^x$ and $\overline{\mathcal{A}}_j \vdash P_x^z$. Hence Item 2 of the claim holds after Line 11 is executed.

The desired result follows.

□

Lemma 5.2 (Completeness). *At the end of* BidirectedReach, *for every pair of nodes* $u, v \in V$ *in the same DSCC,* u *and* v *belong to the same set of* DisjointSets.

Proof. The proof is by showing inductively that for every even r, for every pair of nodes $u, v \in V$ such that v is Dyck-reachable from u via a path $P_u^v : u \rightsquigarrow v$ with $|P| \leqslant r$, BidirectedReach will reach a state where u and v belong to the same set of DisjointSets.

After the initialization phase (Algorithm 8), the claim holds for $r = 0$. Now assume that the claim holds for some r, and we will show that it holds for $r + 2$. Let v be Dyck-reachable from u via a path P_u^v of length $r + 2$. We consider two cases.

1. There exists a node $x \neq u, v$ such that x is Dyck-reachable from u via a path $P_u^x : u \rightsquigarrow x$ and v is Dyck-reachable from x via a path $P_x^v : x \rightsquigarrow v$ such that $|P_u^x|, |P_x^v| \leqslant r$. Then, by the induction hypothesis, BidirectedReach will reach a state where x is placed in the same set as u, and a state where x is placed in the same set as v. After BidirectedReach has reached both states, u and v are in the same set, as required.

2. If there exists no such node x, then there exists a parenthesis pair $\alpha_i, \overline{\alpha}_i \in \Sigma_k$ such that $\alpha_i \ \mathcal{S} \ \overline{\alpha}_i \vdash \lambda(P_u^v)$. Thus there exist two nodes y, z and a path $P_u^v = u \rightarrow y \rightsquigarrow z \rightarrow v$, where $u \rightarrow y$ and $z \rightarrow v$ are single edges, and $P_y^z : y \rightsquigarrow z$ is a path from y to z of length r, and additionally (i) $\lambda(u, y) = \alpha_i$, (ii) $\lambda(z, v) = \overline{\alpha}_i$, and (iii) $\mathcal{S} \vdash \lambda(P_z^y)$. Note that possibly $y = z$, or even $u = y = z$. By the induction hypothesis, BidirectedReach will reach a state where y and z are placed in the same set of DisjointSets. Let x be the representative of that set at that point, and by Line 11 (or Line 13, if $x = y$ or $x = z$) we obtain that $u, v \in \mathsf{Edges}[\overline{\alpha}_i][x]$ at that point. Since $|\mathsf{Edges}[\overline{\alpha}_i][x]| \geqslant 2$, the element $(u, \overline{\alpha}_i)$ was inserted in $\mathcal{Q}$ at that point. It is easy to verity that at some later point, an element $(w, \overline{\alpha}_i)$ is extracted from $\mathcal{Q}$ (possibly $w = x$) such that w is the representative of the set of x and $u, v \in \mathsf{Edges}[\overline{\alpha}_i][w]$. Since w is a representative of the set it belongs to, we have $w = \mathsf{DisjointSets.Find}(w)$, and the condition in Line 3 holds true. If, at that point, u and v are still in different sets of DisjointSets, then for the set S constructed in Line 4 we have $|S| \geqslant 2$. Hence, the condition of Line 5 evaluates to true, and after $\mathsf{DisjointSets.Union}(S, x)$ has been executed in Line 7, u and v will be placed in the same set of DisjointSets.

The desired result follows. □

Complexity. We now establish the complexity of BidirectedReach, in a sequence of lemmas.
Lemma 5.3. *The main loop of Line 1 in Algorithm 9 will be executed $O(n)$ times.*

Proof. Initially $\mathcal{Q}$ is populated by Line 9 of Algorithm 8, which inserts $O(n)$ elements, as $k = O(1)$. Afterwards, for every $\ell \leqslant k = O(1)$ elements $(u, \overline{\alpha}_j)$ inserted in $\mathcal{Q}$ via Line 17, there is at least one node $v \in S$ which stops being a representative of its own set in DisjointSets, and thus will not be in S in further iterations. Hence $\mathcal{Q}$ will contain $O(n)$ elements in total, and the result follows. □

The sets S_j and S'_j. Consider an element $(u, \overline{\alpha}_i)$ extracted from $\mathcal{Q}$ in the j-th iteration of the algorithm in Line 1. We denote by S'_j the set $\mathsf{Edges}[u][\overline{\alpha}_i]$, and by S_j the set S constructed in Line 4. If S was not constructed in that iteration (i.e., the condition in Line 3 does not hold), then we let $S_j = \varnothing$. It is easy to see that $|S_j| \leqslant |S'_j|$ for all j. The following crucial lemma bounds the total sizes of the sets S'_j constructed throughout the execution of BidirectedReach.
Lemma 5.4. *Let r be the number of iterations of the main loop in Line 1 of Algorithm 9. We have $\sum_{j=1}^{r} |S'_j| = O(m)$.*

Proof. By Lemma 5.3 we have $r = O(n)$. Let $J = \{j : |S'_j| \geqslant 2\}$, and it suffices to prove that $\sum_{j \in J} S'_j = O(m)$.

We first argue that after a pair $(u, \overline{\alpha}_i)$ has been extracted from $\mathcal{Q}$ in some iteration $j \in J$, the number of edges in Edges decreases by at least $|S'_j| - 1$. We consider the following complementary cases depending on the condition of Line 22.

1. If the condition holds, then we have $|\mathsf{Edges}[u][\overline{\alpha}_i]| = 1$ after Line 23 has been executed.

2. Otherwise, we must have $u \in S$ and $|S| \geqslant 2$, hence there exists some $x \in S \setminus \{u\}$ chosen in Line 6, and all edges in $\mathsf{Edges}[u]$ will be moved to $\mathsf{Edges}[x]$ for some $v = u$ in Line 9. Hence $|\mathsf{Edges}[u][\overline{\alpha}_i]| = 0$.

Note that because of Line 10, the edges in $\mathsf{Edges}[u][\overline{\alpha}_i]$ are not moved to $\mathsf{Edges}[x][\overline{\alpha}_i]$, hence all $\mathsf{Edges}[u][\overline{\alpha}_i]$ (except possibly one) will no longer be present at the end of the iteration. Since

$S'_j = \mathsf{Edges}[u][\overline{\alpha}_i]$ at the beginning of the iteration, we obtain that the number of edges in Edges decreases by at least $|S'_j| - 1$.

We define a potential function $\Phi : \mathbb{N} \to \mathbb{N}$, such that $\Phi(j)$ equals the number of elements in the data structure Edges at the beginning of the j-th iteration of the main loop in Line 1 of Algorithm 9. Note that (i) initially $\Phi(1) = m$, (ii) $\Phi(j) \geqslant 0$ for all j, and (iii) $\Phi(j+1) \leqslant \Phi(j)$ for all j, as new edges are never added to Edges. Let $(u, \overline{\alpha}_i)$ be an element extracted from $\mathcal{Q}$ at the beginning of the j-the iteration, for some $j \in J$. As shown above, at the end of the iteration we have removed at least $|S'_j| - 1$ edges from Edges, and since $|S'_j| \geqslant 2$, we obtain $\Phi(j+1) \leqslant \Phi(j) - |S'_j|/2$. Summing over all $j \in J$, we obtain

$$
\begin{aligned}
\sum_{j \in J} |S'_j| &\leqslant 2 \cdot \sum_{j \in J} (\Phi(j) - \Phi(j+1)) && \left[\text{as } \Phi(j+1) \leqslant \Phi(j) - |S'_j|/2\right] \\
&= 2 \cdot \sum_{\ell=1}^{|J|} (\Phi(j_\ell) - \Phi(j_\ell + 1)) && [\text{for } j_\ell < j_{\ell+1}] \\
&\leqslant 2 \cdot \Phi(j_1) && [\text{as } \Phi \text{ is decreasing and thus } \Phi(j_{\ell+1}) \leqslant \Phi(j_\ell + 1)] \\
&\leqslant 2 \cdot m && [\text{as } \Phi(j_1) \leqslant \Phi(1) = m]
\end{aligned}
$$

The desired result follows. □

Finally, we are ready to establish the complexity of BidirectedReach.

Lemma 5.5 (Complexity). BidirectedReach *requires $O(m \cdot \alpha(n))$ time and $O(m)$ space.*

Proof. The $O(m)$ space bound follows easily by the data-structures, hence our focus will be on the time complexity. It is straightforward that the initialization phase (Algorithm 8) requires $O(m)$ time, and our focus will be on the main computation (Algorithm 9).

First we bound the amount of time spent in the operations of the data structure DisjointSets. Since a $\mathsf{DisjointSets.Find}(w)$ operation has amortized time $O(\alpha(n))$ [Tarjan, 1975], using Lemma 5.4, the total time for constructing the sets S_j in Line 4 is bounded by

$$
\alpha(n) \cdot \sum_{j=1}^{r} |S'_j| = O(m \cdot \alpha(n))
$$

Similarly, the total time required for making the $\mathsf{DisjointSets.Union}(S_j, x)$ operations in Line 7 is

$$\alpha(n) \cdot \sum_{j=1}^{r} |S_j| \leqslant \alpha(n) \cdot \sum_{j=1}^{r} |S'_j| = O(m \cdot \alpha(n))$$

Second, we bound the amount of time spent within BidirectedReach, i.e., without counting the time in DisjointSets. All lines except for the loop in Line 9 are executed a number of times proportional to the iterations of the main loop of Algorithm 9, hence by Lemma 5.3, the total time spent outside the inner loop of Line 9 is $O(n)$. Finally, observe that every line of the inner loop is executed $O(|S'_j|)$ times. Hence, by Lemma 5.4, we obtain that the total time spent in the inner loop is $O(m)$. Note that each move operation in Line 11 and append operation in Line 13 takes constant time, by using a linked-list implementation of each set $\mathsf{Edges}[u][\overline{\alpha}_i]$.

The desired result follows. □

A speedup for non-sparse graphs. Observe that in the case of sparse graphs $m = O(n)$, and Lemma 5.5 yields the complexity $O(n \cdot \alpha(n))$. Here we describe a modification of BidirectedReach that reduces the complexity from $O(m \cdot \alpha(n))$ to $O(m + n \cdot \alpha(n))$, and thus is faster for graphs where the edges are more than a factor $\alpha(n)$ as many as the nodes (i.e., $m = \omega(n \cdot \alpha(n))$). The key idea is that if a node u has more than k outgoing edges initially, then it has two distinct outgoing edges labeled with the same closing parenthesis $\overline{\alpha}_i \in \Sigma_k^C$, and hence the corresponding neighbors can be merged to a single DSCC in a preprocessing step. Once such a merging has taken place, u only needs to keep a single outgoing edge labeled with $\overline{\alpha}_i$ to that DSCC. This preprocessing phase requires $O(m)$ time for all nodes, after which there are only $O(n)$ edges present, by amortizing at most k edges per node of the original graph (recall that $k = O(1)$). After this preprocessing step has taken place, BidirectedReach is executed with $O(n)$ edges in its input, and by Lemma 5.5 the complexity is $O(n \cdot \alpha(n))$. We conclude the results of this section with the following theorem.

Theorem 5.1 (Worst-case complexity). *Let $G = (V, E)$ be a Σ_k-labeled bidirected graph of n nodes and $m = \Omega(n)$ edges.* BidirectedReach *correctly computes the DSCCs of G and requires $O(m + n \cdot \alpha(n))$ time and $O(m)$ space.*

Linear-time considerations. Note that $\alpha(n)$ is an extremely slowly growing function, and for all practical purposes $\alpha(n) \leqslant 4$. Indeed, the smallest n for which $\alpha(n) = 5$ far exceeds the estimated

number of atoms in the observable universe. Additionally, since it is known that a Disjoint Sets data structure operates in amortized constant expected time per operation [Doyle and Rivest, 1976; Yao, 1985], we obtain the following corollary regarding the expected time complexity of our algorithm.

Corollary 5.1 (Average-case complexity). *For bidirected graphs, the algorithm* BidirectedReach *requires $O(m)$ expected time for computing DSCCs.*

Remark 5.3 (**Comparison with existing work.**). We briefly compare our work with the previous best-known results of [Zhang *et al.*, 2013].

- *Algorithm 5.* Algorithm 5 of [Zhang *et al.*, 2013] solves Dyck reachability on bidirected graphs, and the complexity is $O(m \cdot \log n)$ (see [Zhang *et al.*, 2013, Theorem 4]). Although not explicitly mentioned in the theorem, the complexity bound is for the *average-case complexity*, and not the *worst-case complexity*. The average case comes from their Fast-Doubly-Linked-List (FDLL) data structure, the query and deletion time of which are taken to be $O(1)$ in the average case. However, the worst-case time complexity of each of these is $O(n)$. Using the worst-case time for each operation in FDLL yields the upper bound of $O(n \cdot m \cdot logn)$ time, since (i) as argued in [Zhang *et al.*, 2013, Theorem 4] the main loop of the algorithm is executed $O(m \cdot \log n)$ times, and (ii) every FDLL query and delete operation inside that loop takes $O(n)$ time instead of $O(1)$.

- *Algorithm 2.* The same work presents Algorithm 2 which has worst-case complexity $O(n^2)$ (shown in [Zhang *et al.*, 2013, Theorem 2]), and thus dominates $O(n \cdot m \cdot \log n)$ on graphs with no isolated nodes.

Hence, until now, the best worst-case complexity for the problem has been $O(n^2)$, and the best average-case complexity has been $O(\min\{n^2, m \cdot \log n\})$. The new bounds we establish are $O(m + n \cdot \alpha(n))$ and $O(m)$, respectively.

5.3.3 An $\Omega(m + n \cdot \alpha(n))$ lower bound

Theorem 5.1 implies that Dyck reachability on bidirected graphs can be solved in almost linear-time. A theoretically interesting question is whether the problem can be solved in linear time in the worst case. We answer this question in the negative by proving that every algorithm for the

problem requires $\Omega(m + n \cdot \alpha(n))$ time, and thereby proving that our algorithm $\mathsf{BidirectedReach}$ is indeed optimal wrt worst-case complexity.

The Separated Union-Find problem. A sequence σ of Union-Find operations is called *separated* if all Find operations occur at the end of σ. Hence $\sigma = \sigma_1 \circ \sigma_2$, where σ_1 contains all Union operations of σ. We call σ_1 a *union sequence* and σ_2 a *find sequence*. The *Separated Union-Find* problem is the regular Union-Find problem over separated union-find sequences. Note that this version of the problem has an *offline* flavor, as, at the time when the algorithm is needed to produce output (i.e. when the suffix of Find operations starts) the input has been fixed (i.e., all Union operations are known). We note that the Separated Union-Find problem is different from the Static Tree Set Union problem [Gabow and Tarjan, 1985], which restricts the type of allowed Union operations, and for which a linear time algorithm exists on the RAM model. The following lemma states a lower bound on the worst-case complexity of the problem.

Lemma 5.6. *The Separated Union-Find problem over a universe of size n and sequences of length n has worst-case complexity $\Omega(n \cdot \alpha(n))$.*

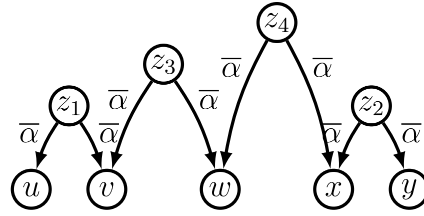
Proof. The proof is essentially the proof of [Aspvall *et al.*, 1979, Theorem 4.4], by observing that the sequences constructed there to prove the lower bound are actually separated union-find sequences. □

The union graph G^{σ_1}. Let σ_1 be a union sequence over some universe X. The *union graph* of σ_1 is a Σ_1-labeled bidirected graph $G^{\sigma_1} = (V^{\sigma_1}, E^{\sigma_1})$, defined as follows.

1. The node set is $V^{\sigma_1} = X \cup \{z_i\}_{1 \leqslant i \leqslant |\sigma_1|}$ where the nodes z_i do not appear in X.
2. The edge set is $E^{\sigma_1} = \{(z_i, x_i, \overline{\alpha}), (z_i, y_i, \overline{\alpha})\}_{1 \leqslant i \leqslant |\sigma_1|}$, where $x_i, y_i \in X$ are the elements such that the i-th operation of σ_1 is $\mathsf{Union}(x_i, y_i)$.

See Fig. 5.3 for an illustration.

A lower bound for Dyck reachability on bidirected graphs. We are now ready to prove our lower bound. The proof consists in showing that there exists no algorithm that solves the problem in $o(n \cdot \alpha(n))$ time. Assume towards contradiction otherwise, and let A' be an algorithm that solves the problem in time $o(n \cdot \alpha(n))$. We construct an algorithm A that solves the Separated Union-Find problem in the same time.



$\sigma_1 = \mathsf{Union}(u, v),\ \mathsf{Union}(x, y),\ \mathsf{Union}(w, v),\ \mathsf{Union}(w, x)$

Figure 5.3: A union sequence σ_1 and the corresponding graph G^{σ_1}.

Let $\sigma = \sigma_1 \circ \sigma_2$ be a separated union-find sequence, where σ_1 is a union sequence and σ_2 is a find sequence. The algorithm A operates as follows. It performs no operations until the whole of σ_1 has been revealed. Then, A' constructs the union graph G^{σ_1}, and uses A' to solve the Dyck reachability problem on G^{σ_1}. Finally, every $\mathsf{Find}(x)$ operation encountered in σ_2 is handled by A by using the answer of A' on G^{σ_1}.

It is easy to see that A handles the input sequence σ correctly. Indeed, for any sequence of union operations $\mathsf{Union}(x_i, y_i), \dots, \mathsf{Union}(x_j, y_j)$ that bring two elements x and y to the same set, the edges $(z_i, x_i, \overline{\alpha}), (z_i, y_i, \overline{\alpha}), \dots, (z_j, x_j, \overline{\alpha}), (z_j, y_j, \overline{\alpha})$ must bring x and y to the same DSCC of G^{Σ_1}. Finally, the algorithm A requires $O(n)$ time for constructing G and answering all queries, plus $o(n \cdot \alpha(n))$ time for running A' on G^{Σ_1}. Hence A operates in $o(n \cdot \alpha(n))$ time, which contradicts Lemma 5.6.

The above establishes that any Dyck reachability algorithm for bidirected graphs of n nodes and m edges requires $\Omega(n \cdot \alpha(n))$ time. Additionally, any such algorithm requires $\Omega(m)$ time, since the size of the input is $\Omega(m)$. Hence we establish the following theorem.

Theorem 5.2 (Lower-bound). *Any Dyck reachability algorithm for bidirected graphs with n nodes and $m = \Omega(n)$ edges requires $\Omega(m + n \cdot \alpha(n))$ time in the worst case.*

Theorem 5.2 together with Theorem 5.1 yield the following corollary.

Corollary 5.2 (Optimality)**.** *The Dyck reachability algorithm* $\mathsf{BidirectedReach}$ *for bidirected graphs is optimal wrt to worst-case complexity.*

5.3.4 An $\Omega(m + n \cdot \alpha(n))$ lower bound for Dyck reachability on constant-treewidth bidirected graphs

In this section we consider Dyck reachability on constant-treewidth bidirected graphs. We start with a remark about the question and our result.

Remark 5.4. In Theorem 5.2 we establish a lower bound for Dyck reachability on general bidirected graphs. For Dyck reachability on bidirected trees, a linear-time ($O(n)$-time) algorithm is known [Zhang *et al.*, 2013]. For a large variety of problems, the complexity on graphs with constant treewidth coincides with the complexity on trees, e.g., for shortest paths [Chaudhuri and Zaroliagis, 1995], various combinatorial optimization problems [Bertele and Brioschi, 1972], and even NP-complete problems [Arnborg and Proskurowski, 1989; Bern *et al.*, 1987; Bodlaender, 1988]. Thus a natural question is whether linear-time algorithm for Dyck reachability can be obtained for constant-treewdith bidirected graphs. Quite unexpectedly, we present a superlinear lower bound of $\Omega(n \cdot \alpha(n))$ time for Dyck reachability on constant-treewdith bidirected graphs. We show that for Dyck reachability on bidirected graphs, graphs of treewidth 3 are as hard to solve as arbitrary graphs.

3-access sequences. A union-find sequence is called *3-access* if every element appears in at most 3 Union operations. As the following lemma shows, the union-find problem over 3-access sequences is as hard as over general sequences.

Lemma 5.7. *Any algorithm for the Union-Find problem over 3-access sequences requires* $\Omega(m \cdot \alpha(n))$ *time, where* n *is the length of the input sequence.*

Proof. Consider any algorithm A' that solves the Union-Find problem over 3-access sequences. We will describe an algorithm A that can handle arbitrary sequences using A as an oracle. Given a sequence σ of length n, the algorithm A will be constructing a new, 3-access sequence σ', and running A' on σ'.

The algorithm simply uses a fresh surrogate symbol $x_i \notin X$ to replace the i-th appearance of the symbol $x \in X$ in σ in a union operation. Consider an operation $\mathsf{Union}(x, y)$, where x, y appear for the i-th and j-th time in σ, respectively. The algorithm A introduces two new surrogate symbols x_i, y_j, and extends σ' by the following operations: $\mathsf{Union}(x_{i-1}, x_i)$, $\mathsf{Union}(y_{j-1}, y_j)$, $\mathsf{Union}(x_i, y_j)$.

It is easy to see that σ' is a 3-access sequence of length at most $3 \cdot m$. Hence the running time of A is asymptotically equal to that of A', i.e. $o(m \cdot \alpha(n))$. This contradicts the lower bound of Lemma 5.6.

The desired result follows. $\square$

In the next lemma we show that the union graph G^{σ} of a 3-access sequence σ has constant treewidth.

Lemma 5.8. *Given a 3-access union-find sequence σ, the graph G^{σ} has treewidth at most* 3.

Proof. Let X be the universe of σ. We construct a tree decomposition $\mathrm{Tree}(G^{\sigma})$ as follows.

1. The node set of $\mathrm{Tree}(G^{\sigma})$ contains one bag B_x per node $x \in X$. The contents the bag are $B_x = \{x, z_{i_1}, z_{i_2}, z_{i_3}\}$, where z_{i_j}, for $1 \leqslant j \leqslant 3$, are the nodes of G^{σ} that have an outgoing edge $(z_{i_j}, x, \overline{\alpha}) \in E^{\sigma}$.
2. Given two nodes $x, y \in X$, there exists an edge (B_x, B_y) in $\mathrm{Tree}(G^{\sigma})$ iff σ contains an operation $\mathsf{Union}(x, y)$.

First, observe that $\mathrm{Tree}(G^{\sigma})$ is indeed a tree, as if there exists a cycle C, the edge of C that corresponds to the last Union operation of σ represents some $\mathsf{Union}(x, y)$ such that x and y were already in the same set at that point. By the definition of the Union-Find problem, $\mathsf{Union}(x, y)$ was not allowed at that point, and σ is an invalid sequence.

Second, we argue that $\mathrm{Tree}(G^{\sigma})$ is a tree decomposition. It is easy to see that every node and edge of G^{σ} is covered by some bag of $\mathrm{Tree}(G^{\sigma})$. To argue that every node appears in a contiguous subtree of $\mathrm{Tree}(G^{\sigma})$, note that (i) every node $x \in X$ appears in a single bag B_x, and (ii) every node $z_{i_j} \in V^{\sigma} \setminus X$ appears only in two bags B_x, B_y, such that the i_j-th operation of σ is $\mathsf{Union}(x, y)$, and these two bags are connected by an edge.

Finally, it follows easily that $\mathrm{Tree}(G^{\sigma})$ has width at most 3, since, by construction, every bag contains at most 4 nodes. We conclude that G^{σ} has treewidth at most 3. The desired result follows.

$\square$

Lemma 5.7 and Lemma 5.8 together with the reduction of Theorem 5.2 lead to the following theorem.

Theorem 5.3 (**Lower-bound for low-treewidth graphs**)**.** *Any Dyck reachability algorithm requires* $\Omega(n \cdot (\alpha(n)))$ *time for the class of constant-treewidth bidirected graphs of* n *nodes.*

5.4 Dyck Reachability on General Graphs

We present a hardness result regarding the Dyck reachability problem on general graphs. Recall Dyck languages are a subset of Context-free languages. The problem of language parsing of a string of length n is a special case of language reachability on a graph of $n+1$ nodes arranged in a line.

Theoretical question. In parsing, there is a big difference between Dyck and general CFL languages. CFL parsing is known to Boolean Matrix Multiplication hard [Lee, 2002], whereas Dyck parsing can be easily solved in $O(n)$ time.

Given the linear-time algorithm for Dyck parsing, an important theoretical question is whether Dyck reachability for general graphs can be solved in truly sub-cubic time, since none of the existing algorithms is truly sub-cubic. Note that since Dyck reachability is a combinatorial graph problem, techniques such as fast-matrix multiplication (e.g. Strassen's algorithm [Strassen, 1969]) are unlikely to be applicable. Hence we consider combinatorial (i.e., discrete, graph-theoretic) algorithms. The standard BMM-conjecture [Lee, 2002; Henzinger *et al.*, 2015; Vassilevska Williams and Williams, 2010; Abboud and Vassilevska Williams, 2014] states that there is no truly sub-cubic ($O(n^{3-\delta})$, for $\delta > 0$) combinatorial algorithm for Boolean Matrix Multiplication. We resolve the question for Dyck reachability on general graphs in negative, under the BMM-conjecture, that is, we show that Dyck reachability on general graphs is BMM-hard. We establish this by showing Dyck reachability on general graphs is hard as CFL parsing, which we present below.

The gadget graph $G^{\mathcal{G}}$**.** Given a Context-free grammar $\mathcal{G}$ in Chomsky normal form, we construct the *gadget graph* $G^{\mathcal{G}} = (V^{\mathcal{G}}, E^{\mathcal{G}})$ as follows.

1. The node set $V^{\mathcal{G}}$ contains two distinguished nodes x, y, together with a node x_i for the i-th

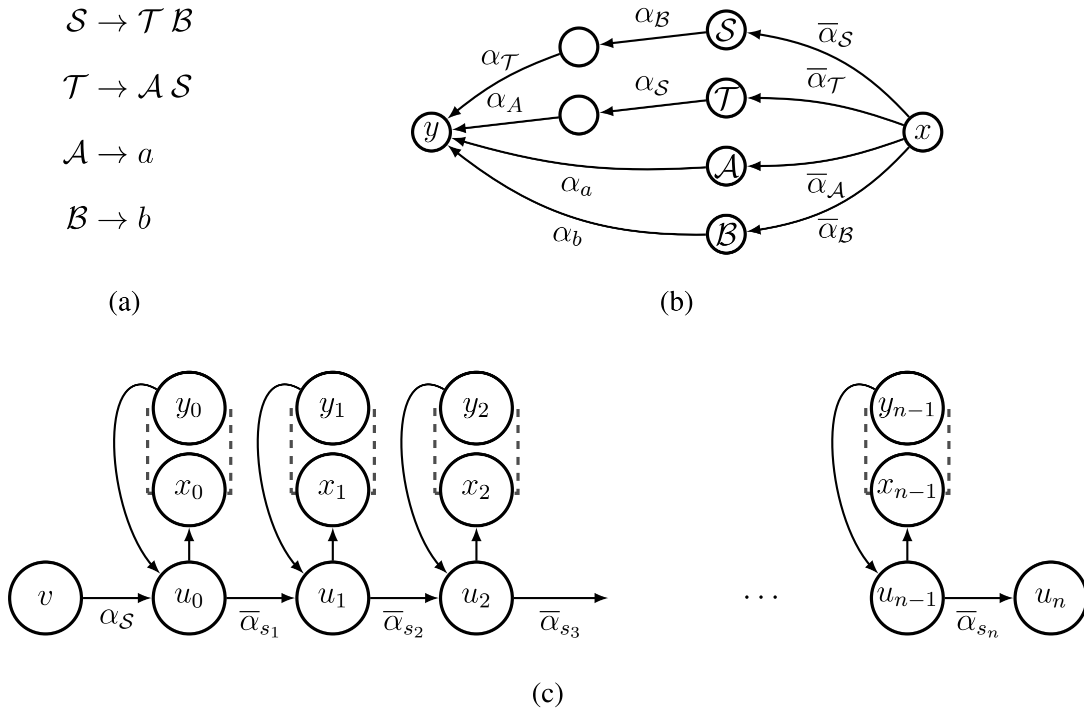


Figure 5.4: **(5.4a)** A grammar $\mathcal{G}$ for the language $a^n b^n$, **(5.4b)** The gadget graph $G^{\mathcal{G}}$, **(5.4c)** The parse graph $G^{\mathcal{G}}_s$, given a string $s = s_1, \ldots d_n$.

production rule p_i. Additionally, if p_i is of the form $\mathcal{A} \to \mathcal{B}\,\mathcal{C}$, then $V^{\mathcal{G}}$ contains a node y_i.

2. The edge set $E^{\mathcal{G}}$ contains an edge $(x, x_i, \overline{\alpha}_{\mathcal{A}})$, where $\mathcal{A}$ is the left hand side symbol of the i-th production rule p_i of $\mathcal{G}$. Additionally,

   (a) if p_i is of the form $\mathcal{A} \to a$, then $E^{\mathcal{G}}$ contains an edge (x_i, y, α_a), else

   (b) if p_i is of the form $\mathcal{A} \to \mathcal{B}\,\mathcal{C}$, then $E^{\mathcal{G}}$ contains the edges $(x_i, y_i, \alpha_{\mathcal{C}})$ and $(y_i, y, \alpha_{\mathcal{B}})$.

See Fig. 5.4a, Fig. 5.4b for an illustration.

The parse graph $G^{\mathcal{G}}_s$. Given a grammar $\mathcal{G}$ and an input string $s = s_1, \ldots s_n$, we construct the *parse graph* $G^{\mathcal{G}}_s = (V^{\mathcal{G}}_s, E^{\mathcal{G}}_s)$ as follows. The graph consists of two parts. The first part is a line graph that contains nodes $v, u_0, u_1, \ldots u_n$, with the edges $(v, u_0, \alpha_{\mathcal{S}})$ and $(u_{i-1}, u_i, \overline{\alpha}_{s_i})$ for all $1 \leqslant i \leqslant n$. The second part consists of a n copies of the gadget graph $G^{\mathcal{G}}$, counting from 0 to $n-1$. Finally, we have a pair of edges (u_i, x_i, ϵ), (y_i, u_i, ϵ) for every $0 \leqslant i < n$, where x_i (resp. y_i) is the distinguished x node (resp. y node) of the i-th gadget graph. See Fig. 5.4c for an illustration.

Lemma 5.9. *The node u_n is Dyck-reachable from node v iff s is generated by $\mathcal{G}$.*

Proof. Given a path P, we denote by $\overline{\lambda}(P)$ the substring of $\lambda(P)$ that consists of all the closing-parenthesis symbols of $\lambda(P)$. The proof follows directly from the following observation: the parse graph $G_s^{\mathcal{G}}$ contains a path $P : v \rightsquigarrow u_n$ with $\lambda(P) \in \mathcal{L}$ if and only if $\overline{\lambda}(P)$ corresponds to a pre-order traversal of a derivation tree of the string s wrt the grammar $\mathcal{G}$. □

Theorem 5.4 (CFL-parsing hardness). *If there exists a combinatorial algorithm that solves the Dyck reachability problem in time $\mathcal{T}(n)$, where n is the number of nodes of the input graph, then there exists a combinatorial algorithm that solves the CFL parsing problem in time $O(n+\mathcal{T}(n))$.*

Since CFL-parsing is BMM-hard, combining Theorem 5.4 with [Lee, 2002, Theorem 2] we obtain the following corollary.

Corollary 5.3 (BMM hardness: Conditional cubic lower bound). *For any fixed $\delta > 0$, if there is a combinatorial algorithm that solves the Dyck reachability problem in $O(n^{3-\delta})$ time, then there is a combinatorial algorithm that solves Boolean Matrix Multiplication in $O(n^{3-\delta/3})$ time.*

Remark 5.5 (**BMM hardness for low-treewidth graphs**). Note that since the size of the grammar $\mathcal{G}$ is constant, the parse graph $G_s^{\mathcal{G}}$ has constant treewidth. Hence the BMM hardness of Corollary 5.3 also holds if we restrict our attention to Dyck reachability on graphs of constant treewidth.

5.5 Library/Client Dyck Reachability

In this section we present some new results for library/client Dyck reachability with applications to context-sensitive data-dependence analysis. One crucial step to our improvements is the fact that we consider that the underlying graphs are not arbitrary, but have special structure. We start with Section 5.5.1 which defines formally the graph models we deal with, and their structural properties. Afterwards, in Section 5.5.2 we present our algorithms.

5.5.1 Problem definition

Here we present a formal definition of the input graphs that we will be considering for library/-client Dyck reachability with application to context-sensitive data-dependence analysis. Each

input graph G is not an arbitrary Σ_k-labeled graph, but has two important structural properties.

1. G can be naturally partitioned to subgraphs $G_1, \dots G_\ell$, such that every G_i has only ϵ-labeled edges. Each such $G_i = (V_i, E_i)$ corresponds to a method of the input program. There are only a few nodes of V_i that have *incoming* edges that are non-ϵ-labeled. Similarly, there are only a few nodes of V_i that have *outgoing* edges that are non-ϵ-labeled. These are nodes that correspond to the input parameters and return statements of the i-th method of the program, which are almost always only a few.

2. Each G_i is a graph of *low treewidth*. This is an important graph-theoretic property which, informally, means that G_i is similar to a tree (although G_i is not a tree).

In the following definitions, we make the above structural properties formal and precise. We start with the first structural property, we captures the fact that the input graph G consists of many local graphs G_i, one for each method of the input program, and the parenthesis-labeled edges model context sensitivity.

Program-valid partitionings. Let $G = (V, E)$ be a Σ_k-labeled graph. Given some $1 \leqslant i \leqslant k$, we define the following sets.

$$V_c(\alpha_i) = \{u : \exists(u, v, \alpha_i) \in E\} \qquad V_e(\alpha_i) = \{v : \exists(u, v, \alpha_i) \in E\}$$

$$V_x(\overline{\alpha}_i) = \{u : \exists(u, v, \overline{\alpha}_i) \in E\} \qquad V_r(\overline{\alpha}_i) = \{v : \exists(u, v, \alpha_i) \in E\}$$

In words, (i) $V_c(\alpha_i)$ contains the nodes that have a α_i-labeled outgoing edge, (ii) $V_e(\alpha_i)$ contains the nodes that have a α_i-labeled incoming edge, (iii) $V_x(\overline{\alpha}_i)$ contains the nodes that have a $\overline{\alpha}_i$-labeled outgoing edge, and (iv) $V_r(\overline{\alpha}_i)$ contains the nodes that have a $\overline{\alpha}_i$-labeled incoming edge. Additionally, we define the following sets.

$$V_c = \bigcup_i V_c(\alpha_i) \quad V_e = \bigcup_i V_e(\alpha_i) \quad V_x = \bigcup_i V_x(\overline{\alpha}_i) \quad V_r = \bigcup_i V_r(\overline{\alpha}_i)$$

Consider a partitioning $\mathcal{V} = \{V_1, \dots, V_\ell\}$ of the node set V, i.e., $\bigcup_i V_i = V$ and $V_i \cap V_j = \varnothing$ for all $1 \leqslant i, j \leqslant \ell$. We say that $\mathcal{V}$ is *program-valid* if the following conditions hold: for every $1 \leqslant i \leqslant k$, there exist some $1 \leqslant j_1, j_2 \leqslant \ell$ such that (i) $V_c(\alpha_i), V_r(\overline{\alpha}_i) \subseteq V_{j_1}$, and (ii) $V_e(\alpha_i), V_x(\overline{\alpha}_i) \subseteq V_{j_2}$. Intuitively, the parenthesis-labeled edges of G correspond to method

calls and returns, and thus model context sensitivity. Each parenthesis type models the calling context, and each $G[V_i]$ corresponds to a single method of the program. Since the calling context is tied to two methods (the caller and the callee), conditions (i) and (ii) must hold for the partitioning.

A program-valid partitioning $\mathcal{V} = \{V_1, \dots V_\ell\}$ is called b-bounded if there exists some $b \in \mathbb{N}$ such that for all $1 \leqslant j \leqslant \ell$ we have that $|V_e \cap V_j|, |V_x \cap V_j| \leqslant b$. Note that since $\mathcal{V}$ is program-valid, this condition also yields that for all $1 \leqslant i \leqslant k$ we have that $|V_c(\alpha_i)|, |V_r(\overline{\alpha}_i)| \leqslant b$. In this chapter we consider that $b = O(1)$, i.e., b is constant wrt the size of the input graph.

This is true since the sets $V_e \cap V_j$ and $V_x \cap V_j$ represent the input parameters and the return statements of the j-th method in the program. Similarly, the sets $V_c(\alpha_i)$, $V_r(\overline{\alpha}_i)$ represent the variables that are passed as input and the variables that capture the return, respectively, of the method that the i-th call site refers to. In all practical cases, each of the above sets has constant size (or even size 1, in the case of return variables).

Program-valid graphs. The graph G is called *program-valid* if there exists a constant $b \in \mathbb{N}$ such that G has b-bounded program valid partitioning. Given a such a partitioning $\mathcal{V} = \{V_1, \dots, V_\ell\}$, we call each graph $G_i = (V_i, E_i) = G[V_i]$ a *local graph*. Given a partitioning of V to the library partition V^1 and client partition V^2, $\mathcal{V}$ induces a program-valid partitioning on each of the library subgraph $G^1 = G[V^1]$ and $G^2 = G[V^2]$. See Fig. 5.5 for an example.

We now present the second structural property of input graphs that we exploit for data-dependence analysis. Namely, for a program-valid input graph G with a program-valid partitioning $\mathcal{V} = \{V_1, \dots, V_\ell\}$ the local graphs $G_i = G[V_i]$ are graphs of *low treewidth*. It is known that the control-flow graphs (CFGs) of goto-free programs have small treewidth [Thorup, 1998]. The local graphs G_i are not CFGs, but rather graphs defined by def-use chains. As we show in our experiments Section 5.6.2, the local def-use graphs of real-world benchmarks also have small treewidth in each method. Below, we make the above notions precise.

Program-valid treewidth. Let $G = (V, E)$ be a Σ_k-labeled program-valid graph, and $\mathcal{V} = \{V_1, \dots, V_\ell\}$ a program-valid partitioning of G. For each $1 \leqslant i \leqslant \ell$, let $G_i = (V_i, E_i) = G[V_i]$. We define the graph $G'_i = (V_i, E'_i)$ such that

$$E'_i = E_i \bigcup_{1 \leqslant j \leqslant k} (V_c(\alpha_j) \cap V_i) \times (V_r(\overline{\alpha}_j) \cap V_i)$$

and call G'_i the *maximal* graph of G_i. In words, the graph G'_i is identical to G_i, with the exception that G'_i contains an extra edge for every pair of nodes $u, v \in V_i$ such that u has opening-parenthesis-labeled outgoing edges, and v has closing-parenthesis-labeled incoming edges. We define the treewidth of $\mathcal{V}$ to be the smallest integer t such that the treewidth of each G'_i is at most t. We define the width of the pair $(G, \mathcal{V})$ as the treewidth of $\mathcal{V}$, and the *program-valid treewidth* of G to be the smallest treewidth among its program-valid partitionings.

The Library/Client Dyck reachability problem on program-valid graphs. Here we define the algorithmic problem that we solve in this section. Let $G = (V, E)$ be a Σ_k-labeled, program-valid graph and $\mathcal{V}$ a program-valid partitioning of G that has constant treewidth (k need not be constant). The set $\mathcal{V}$ is further partitioned into two sets, $\mathcal{V}^1$ and $\mathcal{V}^2$ that correspond to the *library* and *client* partitions, respectively. We let $V^1 = \bigcup_{V_i \in \mathcal{V}^1} V_i$ and $V^2 = \bigcup_{V_i \in \mathcal{V}^2} V_i$, and define the *library graph* $G^1 = (V^1, E^1) = G[V^1]$ and the *client graph* $G^2 = (V^2, E^2) = G[V^1]$.

The task is to answer Dyck reachability queries on G, where the queries are either (i) single source queries from some node $u \in V^2$, or (ii) pair queries for some pair $u, v \in V^2$. The computation takes place in two phases. In the *preprocessing phase*, only the library graph G^1 is revealed, and we are allowed to do some preprocessing to compute reachability summaries. In the *query phase*, the whole graph G is revealed, and our task is to handle queries fast, by utilizing the preprocessing done on G^1.

5.5.2 Library/Client Dyck reachability on Program-valid Graphs

We are now ready to present our method for computing library summaries on program-valid graphs in order to speed up the client-side Dyck reachability. The approach is very similar to the RSMDistance algorithm in Chapter 4 for handling interprocedural semiring distances on RSMs.

Outline of our approach. Our approach consists of the following conceptual steps. We let the input graph $G = (V, E)$ be any program-valid graph of constant treewidth, with a partitioning of V into the library component V^1 and the client component V^2. Since G is program-valid, it has a constant-treewidth, program-valid partitioning $\mathcal{V}$, and we consider $\mathcal{V}^1$ to be the restriction of $\mathcal{V}$ to the set V^1. Hence we have $\mathcal{V}^1 = \{V_1, \dots V_\ell\}$ be a program-valid partitioning of $G[V^1]$, which also has constant treewidth. Our approach consists of the following steps.

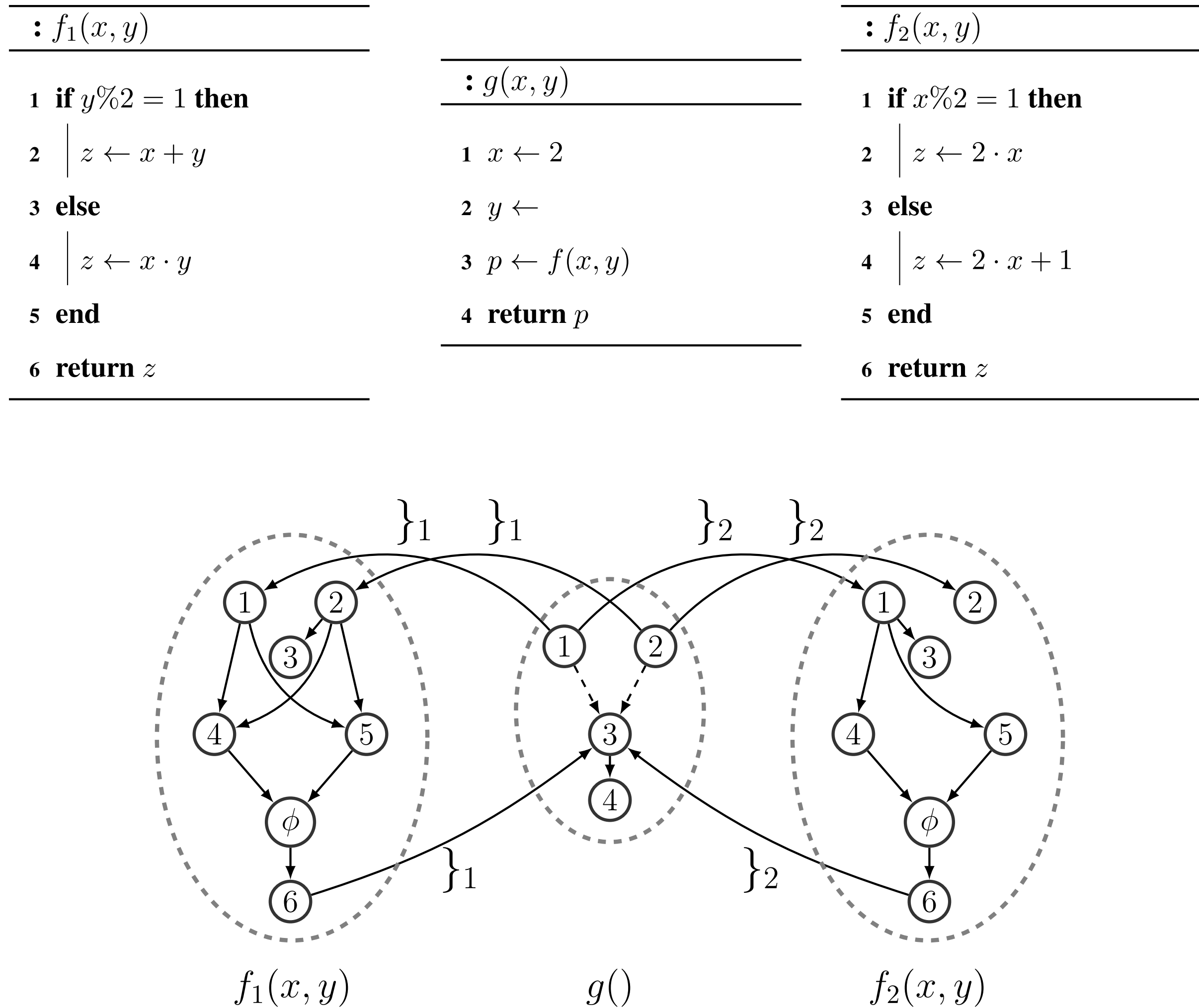


Figure 5.5: Example of a library/client program and the corresponding program-valid data-dependence graph. The library consists of method $g()$ which has a callback function $f(x, y)$. The client implements $f(x, y)$ either as $f_1(x, y)$ or $f_2(x, y)$. The parenthesis-labeled edge model context-sensitive dependencies on parameter passing and return. Note that depending on the implementation of f, there is a data dependence of the variable p on y.

1. We construct a local graph $G_i = (V_i, E_i)$ and the corresponding maximal local graph $G'_i = (V_i, E'_i)$ for each $V_i \in \mathcal{V}$. Recall that G'_i is a conventional graph, since, by definition, E'_i contains only ϵ-labeled edges. Since $\mathcal{V}$ has constant treewidth, each graph G'_i has constant treewidth, and we construct a tree decomposition $\mathrm{Tree}(G'_i)$.

2. We exploit the constant-treewidth property of each G'_i to build a data structure $\mathcal{D}$ which supports the following two operations: (i) Querying whether a node v is reachable from a node u in G'_i, and (ii) Updating G_i by inserting a new edge (x, y). Moreover, each such operation is fast, i.e., it is performed in $O(\log n_i)$ time.

3. Recall that V^1, V^2 are the library and client partitions of G, respectively. In the preprocessing phase, we use the data structure $\mathcal{D}$ to preprocess $G[V^1]$ so that any pair of library nodes that is Dyck-reachable in $G[V^1]$ is discovered and can be queried fast. Hence this library-side reachability information serves as the summary on the library side.

4. In the query phase, we use $\mathcal{D}$ to process the whole graph G, using the summaries computed in the preprocessing phase.

Step 1. Construction of the local graphs G_i and the tree decompositions. The local graphs G_i are extracted from $G[V^1]$ by means of its program-valid partitioning $\mathcal{V}^1 = \{V_1, \dots V_\ell\}$. We consider this partitioning as part of the input, since every local graph G_i in reality corresponds to a unique method of the input program represented by G. Let $n_i = |V_i|$. The maximal local graphs $G'_i = (V_i, E'_i)$ are constructed as defined in Section 5.5.1. Each tree decomposition $\mathrm{Tree}(G'_i)$ is constructed in $O(n_i)$ time using Theorem 2.1. Observe that since $E_i \subseteq E'_i$ (i.e., G_i is a subgraph of its maximal counterpart G'_i), $\mathrm{Tree}(G'_i)$ is also a tree decomposition of G_i. We define $\mathrm{Tree}(G_i) = \mathrm{Tree}(G'_i)$ for all $1 \leqslant i \leqslant \ell$.

Step 2. Description of the data structure $\mathcal{D}$. Here we describe the data structure $\mathcal{D}$, which is built for a conventional graph $G_i = (V_i, E_i)$ (i.e., E_i has only ϵ-labeled edges) and its tree decomposition $\mathrm{Tree}(G_i)$. The purpose of $\mathcal{D}$ is to handle reachability queries on G_i. The data structure supports three operations, given in Algorithm 10, Algorithm 11 and Algorithm 12.

1. The $\mathcal{D}.\mathsf{Preprocess}$ (Algorithm 10) operation builds the data structure for G_i.

2. The $\mathcal{D}.\mathsf{Update}$ (Algorithm 11) updates the graph G_i with a new edge (x, y), provided that there exists a bag B such that $x, y \in B$.

3. The $\mathcal{D}$.Query (Algorithm 12) takes as input a pair of nodes x, y and returns True iff y is reachable from x in G_i, considering all the update operations performed so far.

Algorithm 10: $\mathcal{D}$.Preprocess

Input: A tree-decomposition $\text{Tree}(G_i) = (V_T, E_T)$

```
1 Traverse Tree(G) bottom up
2 foreach encountered bag B do
3     Construct the graph G(B) = (B, R(B))
4     Compute the transitive closure G*(B)
5     foreach (u, v) ∈ B do
6         if u ⇝ v in G*(B) then
7             Insert u, v in R
8     end
9 end
```

Algorithm 11: $\mathcal{D}$.Update

Input: A new edge (x, y)

```
1 Traverse Tree(G) from B_(u,v) to the root
2 foreach encountered bag B do
3     Construct the graph G(B) = (B, R(B))
4     Compute the transitive closure G*(B)
5     foreach u, v ∈ B do
6         if u ⇝ v in G*(B) then
7             Insert (u, v) in R
8     end
9 end
```

The reachability set R. The data structure $\mathcal{D}$ is built by storing a reachability set R between pairs of nodes. The set R has the crucial property that it stores information only between pairs of nodes that appear in some bag of $\text{Tree}(G_i)$ together. That is, $R =\subseteq \bigcup_B B \times B$. Given a bag B, we denote by $R(B)$ the restriction of R to the nodes of B. The reachability set is stored as a collection of $2\sum_i \cdot n_i$ sets $R^F(u)$ and $R^B(u)$, one for every node $u \in V_i$. In turn, the set $R^F(u)$ (resp. $R^B(u)$) will store the nodes in B_u (recall that B_u is the root bag of node u) for which it has been discovered that can be reached from u (resp., that can reach u). It follows directly from the definition of tree decompositions that if $(u, v) \in E_i$ is an edge of G_i then $u \in B_v$ or $v \in B_u$. Hence, given a bag B and nodes $u, v \in B$, querying whether $(u, v) \in R$ reduces to testing whether $v \in R^F(u)$ or $u \in R^B(v)$. Similarly, inserting (u, v) to R reduces to inserting either v to $R^F(u)$ (if $v \in B_u$), or u to $R^B(v)$ (if $u \in B_v$).

Remark 5.6. The map R requires $O(n)$ space. Since each G_i is a constant-treewidth graph, every insert and query operation on R requires $O(1)$ time.

Correctness and complexity of $\mathcal{D}$. Here we establish the correctness and complexity of each operation of $\mathcal{D}$.

It is rather straightforward to see that for every pair of nodes $(u, v) \in R$, we have that v is

reachable from u. The following lemma states a kind of weak completeness: if v is reachable from u via a path of specific type, then $(u, v) \in R$. Although this is different from strong completeness, which would require that $(u, v) \in R$ whenever v is reachable from u, it is sufficient for ensuring completeness of the $\mathcal{D}$.Query algorithm.

Algorithm 12: $\mathcal{D}$.Query

Input: A pair of nodes x, y

```
1  Let X ← {x}, Y ← {y}
2  Traverse Tree(G) from B_x to the root
3  foreach encountered bag B do
4      foreach u, v ∈ B do
5          if u ∈ X and (u, v) ∈ R then
6              Add v to X
7      end
8  end
9  Traverse Tree(G) from B_y to the root
10 foreach encountered bag B do
11     foreach u, v ∈ B do
12         if v ∈ Y and (u, v) ∈ R then
13             Add u to Y
14     end
15 end
16 return True iff X ∩ Y ≠ ∅
```

Algorithm 13: Process

Input: Method graphs

$$G_1 = (V_1, E_1), \ldots, G_\ell = (V_\ell, E_\ell)$$

```
1  foreach 1 ⩽ i ⩽ ℓ do
2      Construct Tree(G_j)
3      Run D.Preprocess on Tree(G_i)
4  end
5  Pool ← {G_1, ... G_ℓ}
6  while Pool ≠ ∅ do
7      Extract G_j from Pool
8      foreach u ∈ V_j ∩ V_e, v ∈ V_j ∩ V_x do
9          if D.Query(u, v) then
10             foreach
                 x : (x, u, α_i) ∈ E, y : (v, y, α_i) ∈ E do
11                 Let G_r = (V_r, E_r) be the graph such
                     that x, y ∈ V_r
12                 if not D.Query(x, y) then
13                     Run D.Update on Tree(G_r) on (x, y)
14                     Insert G_r in Pool
15             end
16     end
17 end
```

Left-right-contained paths. We introduce the notion of left-right contained paths, which is crucial for stating the correctness of the data structure $\mathcal{D}$. Given a bag B of $\mathrm{Tree}(G_i)$, we say that a path $P : x \rightsquigarrow y$ is *left-contained* in B if for every node $w \in P$, if $w \neq x$, we have that $B_w \in T(B)$. Similarly, P is *right-contained* in B if for every node $w \in P$, if $w \neq y$, we have that $B_w \in T(B)$. Finally, P is *left-right-contained* in B if it is both left-contained and right-contained in B.

Lemma 5.10. *The data structure $\mathcal{D}$ maintains the following invariant. For every bag B and pair of nodes $u, v \in B$, if there is a $P_u^v : u \rightsquigarrow v$ which is left-right contained in B, then after* $\mathcal{D}.\mathsf{Preprocess}$ *has processed B, we have $(u, v) \in R$.*

Proof. We prove that the invariant holds (i) at the end of $\mathcal{D}.\mathsf{Preprocess}$, and (ii) after each execution of $\mathcal{D}.\mathsf{Update}$.

1. $\mathcal{D}.\mathsf{Preprocess}$. The proof is given by induction on the sequence of bags processed by $\mathcal{D}.\mathsf{Preprocess}$. The claim is true if B is a leaf of T, as all P_u^v paths considered can only contain nodes from B. Now let B be some non-leaf bag examined by the algorithm, and by the induction hypothesis the claim holds for all children $B^1, \dots B^l$ of B. The claim follows directly from the induction hypothesis if B is not the root bag of any node. Otherwise, let $x_1, \dots x_l$ be the nodes whose root bag is B, and note that any path P_u^v can be decomposed to $P_u^v = P_u^{x_{i_1}} \circ P_{x_{i_1}}^{x_{i_2}} \circ \dots \circ P_{x_{i_r}}^v$, where in all cases P_a^b is a path $a \rightsquigarrow b$, and no node x_i is present in any P_a^b except possibly for the endpoints a and b. By the induction hypothesis, we have $(a, b) \in R$, and thus after the transitive closure $G^*(B)$ is computed, we have that $(u, v) \in R$.

2. $\mathcal{D}.\mathsf{Update}$. The proof is similar to that of $\mathcal{D}.\mathsf{Preprocess}$. The key observation is that if a path $P_u^v : u \rightsquigarrow v$ of interest uses the new edge (x, y), then it must be that u, v appear together in $B_{(x,y)}$ or one of its ancestors, and these are exactly the bags that are processed by $\mathcal{D}.\mathsf{Update}$.

The desired result follows. □

It is rather straightforward that at the end of $\mathcal{D}.\mathsf{Query}$, for every node $w \in X$ (resp. $w \in Y$) we have that w is reachable from x (resp. y is reachable from w). This guarantees that if $\mathcal{D}.\mathsf{Query}$ returns True, then y is indeed reachable from x, via some node $w \in X \cap Y$ (recall that the intersection is not empty, due to Line 16). The following lemma states completeness, namely that if y is reachable from x, then $\mathcal{D}.\mathsf{Query}$ will return True.

Lemma 5.11. *On input x, y, if y is reachable from x, then* $\mathcal{D}.\mathsf{Query}$ *returns* True.

Proof. Let $P : x \rightsquigarrow y$ be a simple path, and $z = \arg_{w \in P} \min \mathsf{Lv}(w)$ be the node of P with the minimum level (possibly $w = x$ or $w = y$). We show that at the end of $\mathcal{D}.\mathsf{Query}$ we have

$w \in X \cap Y$. We only argue that $w \in X$, as the proof of $w \in Y$ is similar.

First, observe that due to Lemma 2.1, B_w must be either B_x or some proper ancestor of B_x. We show the following: for every ancestor bag B of B_x, for every node $z \in B$, if there exists a right-contained path P_x^z in B, then $z \in X$. Note that proving this statement yields that $w \in X$, as by the choice of w we have that P_x^w is a right-contained path in B_w.

We prove the above claim by induction on the ancestors of B_x. The claim is true when $B = B_x$, directly from Lemma 5.10. Now let B be any ancestor of B_x at level i, and by the induction hypothesis the claim holds for the ancestor B' of B at level $i+1$. Examine any path of interest P_x^z, and observe that P_x^z can be decomposed to paths $P_x^y \circ P_y^z$ such that (i) P_x^y is right-contained in the bag B' which is an ancestor of B_x and child of B, and (ii) P_y^z is left-right-contained in B. By the induction hypothesis, we have that $y \in X$. By Lemma 5.10, we have $(y, z) \in R$. Then, in Line 5 will add z in X.

The desired result follows.

□

The following lemma states the complexity of $\mathcal{D}$ operations.

Lemma 5.12. $\mathcal{D}$.Preprocess *requires* $O(n_i)$ *time. Every call to* $\mathcal{D}$.Update *and* $\mathcal{D}$.Query *requires* $O(\log n_i)$ *time.*

Proof. We establish the complexity of each method separately.

1. $\mathcal{D}$.Preprocess. Observe that since the graph has constant treewidth, we have $|B| = 1$ for each encountered bag, and hence the transitive closure is computed in $O(1)$ time. The algorithm will examine each bag once, and since, by Theorem 2.1, there are $O(n)$ bags, the total running time of $\mathcal{D}$.Update is $O(n)$.

2. $\mathcal{D}$.Update. Similarly as before, the transitive closure in each bag requires $O(1)$ time. By Theorem 2.1, $\mathrm{Tree}(G)$ has height $O(\log n)$, and $\mathcal{D}$.Update will examine $O(\log n)$ bags.

3. $\mathcal{D}$.Query. First, note that by Theorem 2.1, $\mathrm{Tree}(G)$ has height $O(\log n)$, and thus the sets X and Y can be implemented as bit-sets of size $O(\log n)$, which allows for $O(1)$-time insertion and querying, and $O(\log n)$ time for computing the intersection.

□

Step 3. Preprocessing the library graph $G[V^1]$. Given the library subgraph $G[V^1]$ and one copy of the data structure $\mathcal{D}$ for each local graph G_i of $G[V^1]$, the preprocessing of the library graph is achieved via the algorithm Process, which is presented in Algorithm 13. In high level, Process initially builds the data structure $\mathcal{D}$ for each local graph G_i using $\mathcal{D}$.Preprocess. Afterwards, it iteratively uses $\mathcal{D}$.Query to test whether there exists a local graph G_j and two nodes $u \in V_j \cap V_e$, $v \in V_j \cap V_x$ such that v is reachable from u in G_j. If so, the algorithm iterates over all nodes x, y such that $(x, u, \alpha_i) \in E$ and $(v, y, \overline{\alpha}_i) \in E$, and uses a $\mathcal{D}$.Query operation to test whether y is reachable from x in their respective local graph G_r. If not, then Process uses a $\mathcal{D}$.Update operation to insert the edge x, y in G_r. Since this new edge might affect the reachability relations among other nodes in V_r, the graph G_r is inserted in Pool for further processing. See Algorithm 13 for a formal description.

The following two lemmas state the correctness and complexity of Process.

Lemma 5.13. *At the end of* Process, *for every graph $G_i = (V_i, E_i)$ and pair of nodes $u, v \in V_i$, we have that v is reachable from u in $G[V^1]$ iff $\mathcal{D}$.*Query *returns* True.

Proof. Given two nodes x, y and a path $P : x \rightsquigarrow y$ such that $\mathcal{S} \vdash \lambda(P)$, we denote by $\mathsf{SH}(P)$ the *stack height* of P, defined as the largest number of consecutive opening parenthesis in $\lambda(P)$. The proof of the lemma then follows by induction on the stack height of the witness path $P : u \rightsquigarrow v$.

In the base case we have $\mathsf{SH}(P)$, and the correctness follows directly from the correctness of $\mathcal{D}$.Query, since $\lambda(P) = \epsilon$ (i.e., P traverses only ϵ-labeled edges).

Now assume that the claim holds for all witness paths with stack height r, and we show that it holds for witness paths of stack height $r + 1$. Indeed, let $P : u \rightsquigarrow v$ be a witness path of stack height $\mathsf{SH}(P) = r + 1$, and $G_r = (V_r, E_r)$ the graph such that $u, v \in V_r$. Then P can be decomposed in the following way:

$$P = u \rightsquigarrow x_1 \rightarrow y_1 \rightsquigarrow w_1 \rightarrow z_1 \rightsquigarrow x_2 \rightarrow y_2 \rightsquigarrow w_2 \rightarrow z_2 \ldots z_q \rightsquigarrow v$$

so that the following hold.

1. For each j there exists $1 \leqslant i \leqslant k$ such that $x_j \in V_c(\alpha_i)$, $y_j \in V_e(\alpha_i)$, $w_j \in V_x(\overline{\alpha}_i)$ and $z_j \in V_r(\overline{\alpha}_i)$.

2. Each path $P_j : y_j \rightsquigarrow w_j$ has stack height $\mathsf{SH}(P_j) \leqslant r$.

3. The paths $u \rightsquigarrow x_1$ and $z_q \rightsquigarrow v$ have stack height 0.

By the induction hypothesis, for each path P_j above, the algorithm will reach a state where $\mathcal{D}$.Query returns True in Line 9, and hence will use $\mathcal{D}$.Update on (x_j, z_j) of the graph G_r in Line 13. When the last such update operation takes place, we have that v is reachable from u in the graph G_r augmented with the edges (x_j, z_j), and by the correctness of the operations of $\mathcal{D}$ in Lemma 5.10, we have that $\mathcal{D}$.Query returns True on query u, v.

The desired result follows.

□

Lemma 5.14. *Let $n = \sum_i n_i$, and k_1 be the number of labels appearing in $E \subseteq V^1 \times V^1 \times \Sigma_k$ (i.e., k_1 is the number of call sites in $G[V^1]$).* Process *requires $O(n + k_1 \cdot \log n)$ time.*

Proof. First, using Theorem 2.1 we obtain that the algorithm spends $\sum_i O(n_i) = O(n)$ time for constructing all tree decompositions in Line 2. Similarly, by Lemma 5.12 the algorithm spends $\sum_i O(n_i) = O(n)$ time for building the data structure $\mathcal{D}$ in Line 3.

We now turn our attention to the main loop in Line 6. We first bound the time taken for processing the graphs inserted in Pool at the beginning of the loop. Since G is b-bounded for $b = O(1)$, for every graph G_j defined in Line 7 there will be $O(b) = O(1)$ executions of $\mathcal{D}$.Query. Hence the total time taken for $\mathcal{D}$.Query operations for the graphs initially in Pool is $\sum_{i=1}^{\ell} O(\log n_i) = O(n)$. For the time spent in $\mathcal{D}$.Query and $\mathcal{D}$.Update due to the loop in Line 10, first note that every $\mathcal{D}$./Query and $\mathcal{D}$.Update require $O(\log n)$ time each. Since the graph G is b-bounded, for constant b, we have $|V_c(\alpha_i)|, |V_r(\overline{\alpha}_i)| \leqslant b = O(1)$, hence summing over all pairs of edges $(x, u, \alpha_i), (v, y, \overline{\alpha}_i)$ in Line 10, we obtain

$$\sum_{i=1}^{k_1} |V_c(\alpha_i)| \cdot |V_r(\overline{\alpha}_i)| \cdot O(\log n) = O(k_1 \cdot b^2 \cdot \log n) = O(k_1 \cdot \log n)$$

Finally, we bound the time spent in the main loop of Line 6 due to graphs added in Pool in Line 14. Since G is b-bounded, the condition in Line 12 can hold true $O(b^2)$ times for each graph. Hence, there will be $O(n)$ graphs added in Pool due to Line 14, and the analysis is similar to the previous paragraph, yielding a $O(k \cdot \log n)$ bound.

The desired result follows. □

Step 4. Library/Client analysis. We are ready to describe the library summarization for Library/Client Dyck reachability. Let $G = (V, E)$ be the program-valid graph representing library and client code, and V^1, V^2 a partitioning of V to library and client nodes.

1. In the preprocessing phase, the algorithm Process is used to preprocess $G[V^1]$. Note that since G is a program valid graph, so is $G[V^1]$, hence Process can execute on $G[V^1]$. The summaries created are in form of $\mathcal{D}$.Update operations performed on edges (x, y).

2. In the querying phase, the set V^2 is revealed, and thus the whole of G. Hence now Process processes G, without using $\mathcal{D}$.Preprocess on the graphs G_i that correspond to library methods, as they have already been processed in step 1. Note that the graphs G_i that correspond to library methods are used for querying and updating.

It follows immediately from Lemma 5.13 that at the end of the second step, for every local graph $G_i = (V_i, E_i)$ of the client graph, for every pair of nodes $u, v \in V_i$, v is Dyck-reachable from u in the program-valid graph G if and only if $\mathcal{D}$.Query returns True on input u, v.

Now we turn our attention to complexity. Let $n_1 = |V^1|$ and $n_2 = |V^2|$. By Lemma 5.14, the time spent for the first step is, $O(n_1 + k_1 \cdot \log n_1)$, and the time spent for the second step is $O(n_2 + k_1 \cdot \log n_1 + k_2 \cdot \log n_2)$.

Constant-time queries. Recall that our task is to support $O(1)$-time queries about the Dyck reachability of pairs of nodes on the client subgraph $G[V^2]$. As Lemma 5.13 shows, after Process has finished, each such query costs $O(\log n_2)$ time. We use existing results for reachability queries on constant-treewidth graphs [Chatterjee *et al.*, 2016e, Theorem 6] which allow us to reduce the query time to $O(1)$, while spending $O(n_2)$ time in total to process all the graphs.

Theorem 5.5. *Consider a Σ_k-labeled program-valid graph $G = (V, E)$ of constant program-valid treewidth, and the library and client subgraphs $G^1 = (V^1, E^1)$ and $G^2 = (V^2, E^2)$. For $i \in \{1, 2\}$ let $n_i = |V^i|$ be the number of nodes, and k_i be the number of call sites in each graph G^i, with $k_1 + k_2 = k$. The algorithm* DynamicDyck *requires*

1. *$O(n_1 + k_1 \cdot \log n_1)$ time and $O(n_1)$ space in the preprocessing phase, and*

2. *$O(n_2 + k_1 \cdot \log n_1 + k_2 \cdot \log n_2)$ time and $O(n_1 + n_2)$ space in the query phase,*

after which pair reachability queries are handled in $O(1)$ time.

5.6 Experimental Results

In this section we report on experimental results obtained for the problems of (i) alias analysis via points-to analysis on SPGs, and (ii) library/client data-dependence analysis.

5.6.1 Alias analysis

Implementation. We have implemented our algorithm BidirectedReach in C++ and evaluated its performance in performing Dyck reachability on bidirected graphs. The algorithm is implemented as presented in Section 5.3, together with the preprocessing step that handles the ϵ-labeled edges. Besides common coding practices we have performed no engineering optimizations. We have also implemented [Zhang *et al.*, 2013, Algorithm 2], including the Fast-Doubly-Linked-List (FDLL), which was previously shown to be very efficient in practice.

Experimental setup. In our experimental setup we used the DaCapo-2006-10-MR2 suit [Blackburn, 2006], which contains 11 real-world benchmarks. We used the tool reported in [Yan *et al.*, 2011b] to extract the Symbolic Points-to Graphs (SPGs), which in turn uses Soot [Vallée-Rai *et al.*, 1999] to process input Java programs. Our approach is similar to the one reported in [Xu *et al.*, 2009a; Yan *et al.*, 2011b; Zhang *et al.*, 2013]. The outputs of the two compared methods were verified to ensure validity of the results. No compiler optimizations were used. All experiments were run on a Windows-based laptop with an Intel Core 2.40 GHz CPU and 16 GB of memory.

SPGs and points-to analysis. For the sake of completeness, we outline the construction of SPGs and the reachability relation they define. A more detailed exposition can be found in [Xu *et al.*, 2009a; Yan *et al.*, 2011b; Zhang *et al.*, 2013]. An SPG is a graph, the node set of which consists of the following three subsets: (i) variable nodes $\mathcal{V}$ that represent variables in the program, (ii) allocation nodes $\mathcal{O}$ that represent objects constructed with the *new* expression, and (iii) symbolic nodes $\mathcal{S}$ that represent abstract heap objects. Similarly, there are three types of edges, as follows, where $\mathsf{Fields} = \{f_i\}_{1\leqslant i\leqslant k}$ denotes the set of all fields of composite data types.

1. Edges of the form $\mathcal{V} \times \mathcal{O} \times \{\epsilon\}$ represent the objects that variables point to.

2. Edges of the form $\mathcal{V} \times \mathcal{S} \times \{\epsilon\}$ represent the abstract heap objects that variables point to.

3. Edges of the form $(\mathcal{O} \cup \mathcal{S}) \times (\mathcal{O} \cup \mathcal{S}) \times \mathsf{Fields}$ represent the fields of objects that other objects point to.

We note that since we focus on context-insensitive points-to analysis, we have not included edges that model calling context in the definition of the SPG. Additionally, only the forward edges labeled with f_i are defined explicitly, and the backwards edges labeled with $\overline{f}_i$ are implicit, since the SPG is treated as bidirected. Memory aliasing between two objects $o_1, o_2 \in \mathcal{S} \cup \mathcal{O}$ occurs when there is a path $o_1 \rightsquigarrow o_2$, such that every opening field access f_i is properly matched by a closing field access $\overline{f}_i$. Hence the Dyck grammar is given by $\mathcal{S} \rightarrow \mathcal{S}\ \mathcal{S} \mid f_i\ \mathcal{S}\ \overline{f}_i \mid \epsilon$. This allows to infer the objects that variable nodes can point to via composite paths that go through many field assignments. See Fig. 5.6 for a minimal example.

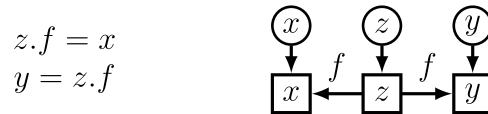


Figure 5.6: A minimal program and its (bidirected) SPG. Circles and squares represent variable nodes and object nodes, respectively. Only forward edges are shown.

Analysis of results. The running times of the compared algorithms are shown in Table 5.3. We can see that the algorithm proposed here for Dyck reachability ion bidirected graphs is much faster than the existing algorithm of [Zhang *et al.*, 2013] in all benchmarks. The highest speedup is achieved in benchmark *luindex*, where our algorithm is 13x times faster. We also see that all times are overall small.

5.6.2 Library/Client data dependence analysis.

Implementation. We have implemented our algorithm DynamicDyck in Java and evaluated its performance in performing Library/Client data-dependency analysis via Dyck reachability. Our algorithm is built on top of Wala [Wal, 2003], and is implemented as presented in Section 5.5. Besides common coding practices we have performed no engineering optimizations. We used

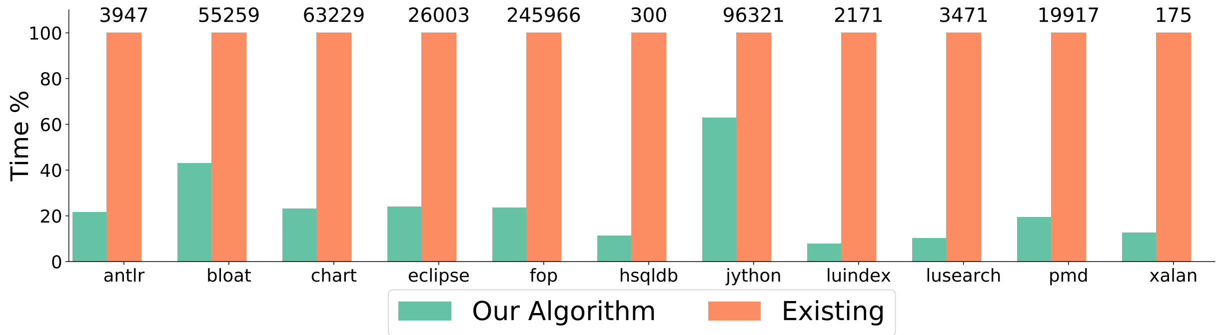


Figure 5.7: Running time of our algorithm vs [Zhang *et al.*, 2013] for context-insensitive field-sensitive points-to analysis on SPGs of various benchmarks. The top row shows the total time (in ms) taken for the slowest method to perform the analysis. The total time is taken as the sum of the time spent in analyzing library and client code. The y-axis shows the percentage of time that each method took as compared to the slowest method.

Benchmark	**Fields**	**Nodes**	**Edges**	**Our Algorithm**	**Existing Algorithm**
antlr	172	13708	23547	0.428783	1.34152
bloat	316	43671	103361	17.7888	34.6012
chart	711	53500	91869	8.99378	34.9101
eclipse	439	34594	52011	3.62835	12.7697
fop	1064	101507	178338	42.5447	148.034
hsqldb	43	3048	4134	0.012899	0.073863
jython	338	56336	167040	40.239	55.3311
luindex	167	9931	14671	0.068013	0.636346
lusearch	200	12837	21010	0.163561	1.12788
pmd	357	31648	58025	2.21662	8.92306
xalan	41	2342	2979	0.006626	0.045144

Table 5.3: Comparison between our algorithm and the existing from [Zhang *et al.*, 2013]. The first three columns contain the number of fields (Dyck parenthesis), nodes and edges in the SPG of each benchmark. The last two columns contain the running times, in seconds.

the LibTW library [van Dijk *et al.*, 2006a] for computing the tree decompositions of the input graphs, under the *greedy degree* heuristic.

Experimental setup. We have used the tool of [Tang *et al.*, 2015] for obtaining the data-dependence graphs of Java programs. In turn, that tool uses Wala [Wal, 2003] to build the graphs, and specifies the parts of the graph that correspond to library and client code. Java programs are suitable for Library/Code analysis, since the ubiquitous presence of callback functions makes the library and client code interdependent, so that the two sides cannot be analyzed in isolation. Our algorithm was compared with the TAL reachability and CFL reachability approach, as already implemented in [Tang *et al.*, 2015]. The comparison was performed in terms of running time and memory usage, first for the analysis of library code to produce the summaries, and then for the analysis of the library summaries with the client code. The outputs of all three methods were compared to ensure validity of the results. The measurements for our algorithm include the time and memory used for computing the tree decompositions. All experiments were run on a Windows-based laptop with an Intel Core 2.40 GHz CPU and 16 GB of memory.

Benchmarks. Our benchmark suit is similar to that of [Tang *et al.*, 2015], consisting of 12 Java programs from SPECjvm2008 [SPE, 2008], together with 4 randomly chosen programs from GitHub [Git, 2008]. We note that as reported in [Tang *et al.*, 2015], they are unable to handle the benchmark *serial* from SPECjvm2008, due to out-of-memory issues when preprocessing the library (recall that the space bound for TAL reachability is $O(n^4)$). In contrast, our algorithm handles *serial* easily, and is thus included in the experiments.

Analysis of results. Our experimental comparison is depicted in Fig. 5.8 and Table 5.4 for running time and Fig. 5.9 and Table 5.5 for memory usage. We briefly discuss our findings.

Treewidth. First, we comment on the treewidth of the obtained data-dependence graphs, which is reported on Table 5.4 and Table 5.5. Recall that our interest is not on the treewidth of the whole data-dependence graph, but on the treewidth of its program-valid partitioning, which yields a subgraph for each method of the input program. In each line of the tables we report the *average* treewidth of each benchmark, averaging over the subgraphs of its program-valid partitioning. We see that the treewidth is typically very small (i.e., in most cases it is 5 or 6) in both library and client code. One exception is the client of mpegaudio, which has large treewidth. Observe that even this corner case of large treewidth was easily handled by our algorithm.

Time. Table 5.4 shows the time spent by each algorithm for analyzing library code and client code separately. We first focus on total time, taken to be the sum of the times spent by each algorithm in the library and client graph of each benchmark. We see that in every benchmark, our algorithm significantly outperforms both TAL and CFL reachability, reaching a 10x-speedup compared to TAL (in *mpegaudio*), and 5x-speedup compared to CFL reachability (in *helloworld*). Note that the benchmark *serial* is missing from the figure, as TAL reachability runs out of memory. The benchmark can be found on Table 5.4, where our algorithm achieves a 630x-speedup compared to CFL reachability.

We now turn our attention to the trade-off between library preprocessing and client querying times. Here, the advantage of TAL over CFL reachability is present for handling client code. However, even for client code our algorithm is faster than TAL in all cases except one, and reaches even a 30x-speedup over TAL (in *sunflow*). Finally, observe that in all cases, the total running time of our algorithm on library and client code combined is much smaller than each of the other methods on library code alone.

Memory. Fig. 5.9 and Table 5.5 compare the total memory used for analyzing library and client code. We see that our algorithm significantly outperforms both TAL and CFL reachability in all benchmarks. Again, TAL uses more memory that CFL in the preprocessing of libraries, but less memory when analyzing client code. However, our algorithm uses even less memory than TAL in all benchmarks. The best performance gain is achieved in *serial*, where TAL runs out of memory after having consumed more than 12 GB. For the same benchmark, CFL reachability uses more than 4.3 GB. In contrast, our algorithm uses only 130 MB, thus achieving a 33x-improvement over CFL, and at least a 90x-improvement over TAL. We stress that for memory usage, these are tremendous gains. Finally, observe that for each benchmark, the maximum memory used by our algorithm for analyzing library and client code is smaller than the minimum memory used between library and client, by each of the other two methods.

Improvement independent of callbacks. We note that in contrast to TAL reachability, the improvements of our algorithm are not restricted to the presence of callbacks. Indeed, the algorithms introduced here significantly outperform the CFL approach even in the presence of no callbacks. This is evident from Table 5.4, which shows that our algorithm processes the library graphs much faster than both CFL and TAL reachability.

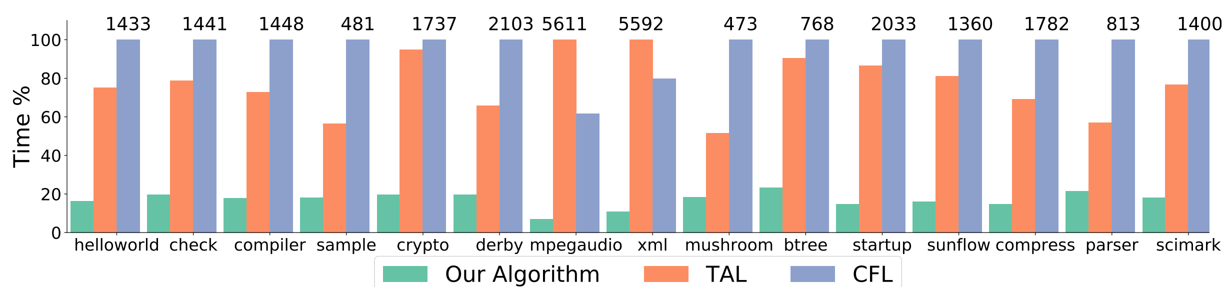


Figure 5.8: Time comparison of our algorithm with TAL and CFL for performing data dependence analysis via CFL reachability. The top row shows the total time (in ms) taken for the slowest method to perform the analysis. The total time is taken as the sum of the time spent in analyzing library and client code. The y-axis shows the percentage of time that each method took as compared to the slowest method. The benchmark *serial* is missing, as TAL mems-out.

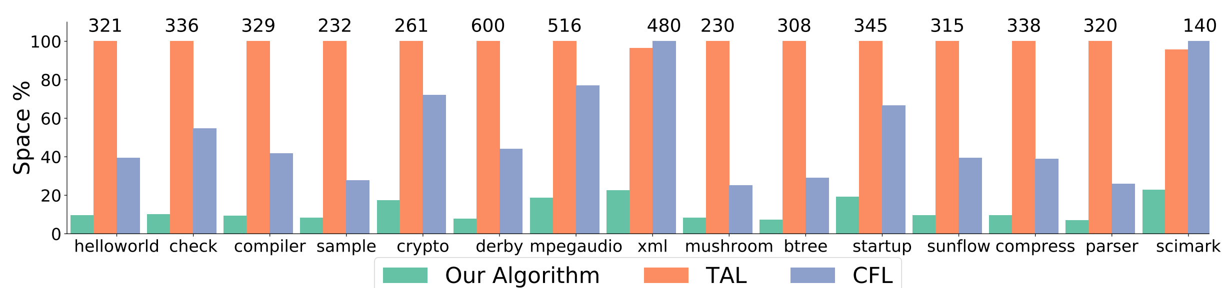


Figure 5.9: Space comparison of our algorithm with TAL and CFL for performing data dependence analysis via CFL reachability. The top row shows the total memory usage (in MB) of the most memory-demanding method to perform the analysis. The total space is taken as the maximum of the space used in analyzing library and client code. The y-axis shows the percentage of memory that each method used as compared to the most memory-demanding method. The benchmark *serial* is missing, as TAL mems-out.

	Nodes		TW		**Our Algorithm**		**TAL**		**CFL**	
Benchmark	Lib.	Cl.	Lib.	Cl.	Lib.	Cl.	Lib.	Cl.	Li.	Cl.
helloworld	16003	296	5	3	229	5	1044	31	855	578
check	16604	3347	5	4	228	54	1062	72	821	620
compiler	16190	536	5	3	248	11	995	57	876	572
sample	3941	28	4	1	86	1	258	14	368	113
crypto	20094	3216	5	5	273	66	1451	196	961	776
derby	23407	1106	6	3	389	22	1301	83	1003	1100
mpegaudio	28917	27576	5	24	204	177	5358	253	1864	1586
xml	71474	2312	5	3	489	115	5492	100	1891	2570
mushroom	3858	7	4	1	86	1	230	14	349	124
btree	6710	1103	4	4	144	34	583	111	571	197
startup	19312	621	5	3	279	17	1651	110	1087	946
sunflow	15615	85	5	2	217	1	1073	31	811	549
compress	16157	1483	5	3	240	23	1119	112	783	999
parser	7856	112	4	1	172	3	443	21	572	241
scimark	16270	2027	5	5	220	34	1004	70	805	595
serial	69999	468	8	3	440	9	MEM-OUT	MEM-OUT	117147	165958

Table 5.4: Running time of our algorithm vs the TAL and CFL approach for data-dependence analysis with library summarization. Times are in milliseconds. MEM-OUT indicates that the algorithm run out of memory. The number of nodes and treewidth reflects the average case among all methods in each benchmark.

	Nodes		TW		**Our Algorithm**		**TAL**		**CFL**	
Benchmark	Lib.	Cl.	Lib.	Cl.	Lib.	Cl.	Lib.	Cl.	Li.	Cl.
helloworld	16003	296	5	3	31	27	321	44	104	126
check	16604	3347	5	4	34	31	336	89	132	184
compiler	16190	536	5	3	31	28	329	44	108	137
sample	3941	28	4	1	19	16	232	59	59	64
crypto	20094	3216	5	5	45	45	261	61	127	188
derby	23407	1106	6	3	46	41	600	88	204	265
mpegaudio	28917	27576	5	24	96	96	516	219	262	397
xml	71474	2312	5	3	108	108	463	153	373	480
mushroom	3858	7	4	1	19	16	230	59	58	58
btree	6710	1103	4	4	22	19	308	65	72	89
startup	19312	621	5	3	66	66	345	92	178	230
sunflow	15615	85	5	2	30	27	315	43	102	124
compress	16157	1483	5	3	32	29	338	50	105	131
parser	7856	112	4	1	22	19	320	64	73	83
scimark	16270	2027	5	5	32	29	134	49	106	140
serial	69999	468	8	3	130	130	MEM-OUT	MEM-OUT	3964	4314

Table 5.5: Memory usage of our algorithm vs the TAL and CFL approach for data-dependence analysis with library summarization. Memory usage is in Megabytes. MEM-OUT indicates that the algorithm run out of memory. The number of nodes and treewidth reflects the average case among all methods in each benchmark.

6 Quantitative Interprocedural Analysis

6.1 Introduction

In this chapter we present the Quantitative Interprocedural Analysis framework. We illustrate how several quantitative problems related to static analysis of recursive programs can be instantiated in this framework, and present some case studies to this direction.

Interprocedural quantitative analysis. Quantitative objectives such as mean-payoff and ratio objectives provide the appropriate framework to express several important system properties such as resource consumption and timeliness. While finite-state systems with mean-payoff objectives have been studied in the literature, the static analysis of RSMs with mean-payoff and ratio objectives has largely been ignored. An interprocedural analysis is *precise* if it provides the "meet-over-all-*valid*-paths" solution (a path is valid if it respects the fact that when a procedure finishes it returns to the site of the most recent call). In the quantitative setting, the problem corresponds to finding the maximal value over all valid paths and to produce a witness (symbolic) path for that value. In this section we consider precise interprocedural quantitative analysis for RSMs with mean-payoff and ratio objectives.

Our contributions. We present a flexible and general modeling framework for quantitative analysis and show how it can be used to reason about quantitative properties of programs and about potential optimizations in the program. We present an efficient polynomial-time algorithm for precise interprocedural quantitative analysis, which is implemented as a tool. We demonstrate the efficiency of the algorithm with three case studies, and show that our approach scales to

programs with thousands of methods.

1. *(Theoretical modeling).* We show that RSMs with mean-payoff and ratio objectives provide a robust framework that naturally captures a wide variety of static program analysis optimization and reasoning problems.

   (a) *(Detecting container usage).* An important problem for performance analysis is detection of runtime bloat that significantly degrades the performance and scalability of programs [Mitchell and Sevitsky, 2007; Xu and Rountev, 2010]. A common source of bloat is inefficient use of containers [Xu and Rountev, 2010]. We show that the problem of detecting usage of containers can be modeled as RSMs with ratio objectives. A good use of a container corresponds to a good event and no use of the container is a bad event, and a misuse is represented as a low ratio of good vs bad events. Hence the container usage problem is naturally modeled as ratio analysis of RSMs. While the problem of detecting container usage was already considered in [Xu and Rountev, 2010], our different approach has the following benefits (see Section 6.5.2 for a comparison). First, our approach can handle recursion ([Xu and Rountev, 2010] does not handle recursion). Second, our approach is sound, and does not yield false positives. Third, the approach of [Xu and Rountev, 2010] ignored DELETE operations and we are able to take into consideration both ADD and DELETE operations (thus provide a more refined analysis). Moreover, our algorithmic approach for analysis of RSMs is polynomial, whereas the algorithmic approach of [Xu and Rountev, 2010] in the worst case can be exponential.

   (b) *(Static profiling of programs).* We use our framework to model a conceptually new way for static profiling of programs for performance analysis. A line in the code (or a code segment) is referred as *hot* if there exists a run of the program where the line of code is frequently executed. For example, a function is referred as *hot* if there exists a run of the program where the function is frequently invoked, i.e., the frequency of calls to the function among all function calls is at least a given threshold. Similarly, a collection of functions is referred as hot if there exists a run of the program where the collection is frequently invoked (note that a collection of methods might be frequently invoked even if each individual method is not). Again this problem is naturally modeled as ratio problem for RSMs, and our approach statically detects

methods that are more frequently invoked. Optimization of frequently executed code would naturally lead to performance improvements and reasoning about hot spots in the code can assist the complier to apply optimization such as function inlining and loop unrolling (see Sections 6.3.2 and 6.3.3 for more details).

(c) *(Other applications).* We show the generality of our framework by demonstrating that it is suitable for the theoretical modeling of diverse applications such as interprocedural worst-case execution time analysis, evaluating speedup in parallel computation, and interprocedural average energy consumption analysis.

2. *(Algorithmic analysis).* The quantitative analysis of RSMs with mean-payoff objectives can be achieved in polynomial time by a reduction to pushdown systems with mean-payoff objectives (which can be solved in polynomial time [Chatterjee and Velner, 2012]). However, the resulting algorithm in the worst case has time complexity that is a polynomial of degree thirteen and space complexity that is a polynomial of degree six (which is prohibitive in practice). We exploit the special theoretical properties of RSMs in order to improve the theoretical upper bound and get an algorithm that in the worst case runs in cubic time and with quadratic space complexity. In addition, we exploit the properties of *real-world* programs and introduce optimizations that give a practical algorithm that is much faster than the theoretical upper bound when the relevant parameters (the total number of entry, exit, call, and returns nodes) are small, which is typical in most applications. Finally we present a linear-time reduction of the quantitative analysis problem with ratio objectives to mean-payoff objectives.

3. *(Tool and experimental results).* We have implemented our algorithm and developed a tool in the Java Soot framework [Vallée-Rai *et al.*, 1999]. We show through two case studies that our approach scales to relatively large programs from well-known benchmarks. The details of the case studies are as follows:

(a) *(Detecting container usage).* Our experimental results show that our tool scales to relatively large benchmarks (DaCapo 2009 [Blackburn *et al.*, 2006]), and discovers relevant and useful information that can be used to optimize performance of the programs. Our tool could analyze all containers in several benchmarks. Our sound approach allows us to avoid false reports and our simple mathematical modelling

even allows us to report misuses that were not reported in other works.

(b) *(Static profiling of programs).* We run an analysis to detect (i) hot methods and (ii) hot collections of methods for various thresholds. Our experimental results on the benchmarks report only a small fraction of the functions as hot for high threshold values, and thus give useful information about potential functions to be optimized for performance gain. In addition we perform a dynamic profiling and mark the top 5% of the most frequently invoked functions as hot. Our experiments show a significant correlation between the results of the static and dynamic analysis. In addition, we show that the sensitivity and specificity of the static classification can be controlled by considering different thresholds, where lower thresholds increase the sensitivity and higher thresholds increase the specificity. We investigate the trade-off curve (ROC curve) and demonstrate the prediction power of our approach.

Thus we show that several conceptually different problems related to program optimizations are naturally modeled in our framework, and demonstrate that we present a flexible and generic framework for quantitative analysis of programs. Moreover, our case studies show that our tool scales to benchmarks from real-world programs.

Related work. We discuss here some related work on the two experimental applications of our approach, namely the detection of container bloat and the static profiling of programs.

Detecting inefficiently-used containers. Bloat detection and detecting inefficiently used containers have been identified in many previous works as a major reason for program inefficiency. Dynamic approaches for the problem were studied in many works such as [Mitchell *et al.*, 2006; Mitchell and Sevitsky, 2007; Dufour *et al.*, 2008; Novark *et al.*, 2009; Shacham *et al.*, 2009; Shankar *et al.*, 2008; Xu *et al.*, 2010b]. A manual approach was proposed in [Mitchell *et al.*, 2006] that uses structuring behavior using flow information, and a way to find data structures that consume excessive memory was presented in [Mitchell and Sevitsky, 2007]. Dufour et al. [Dufour *et al.*, 2008] uses escape analysis to find excessive use of temporary data structures, and there are many dynamic techniques to identify bloats [Novark *et al.*, 2009; Shacham *et al.*, 2009; Shankar *et al.*, 2008; Xu *et al.*, 2009b; Xu *et al.*, 2010b; Xu and Rountev, 2008]. A static approach to analyze the problem was first considered in [Xu and Rountev, 2010], which is the most closely related work to our case study. The work of [Xu and Rountev, 2010] provides

an excellent exposition of the problem with several practical motivations. A big part of the contribution of [Xu and Rountev, 2010] is an automated annotation for the functionality of the containers operation. The main algorithmic approach of [Xu and Rountev, 2010] is to use CFL-reachability (context-free reachability) to identify nesting loop depths and then use this information for detecting misuse of containers.

Static profiling of programs. Static and dynamic profiling of programs is in the heart of program optimization. Static profiling is typically used in branch predictions where the goal is to assign probabilities to branches, and typically require some prior knowledge on the probability of inputs. Static profiling of programs for branch predictions has been considered in [Ball and Larus, 1993; Wu and Larus, 1994; Hennessy and Patterson, 2006; Wagner *et al.*, 1994]. Dynamic profiling has also been used in many applications related to performance optimizations, see [Wikipedia, 2015] for a collection of dynamic profiling tools. Two main drawbacks of dynamic profiling are that it requires inputs, and it cannot be used for compiler optimizations. We use static profiling to determine if a function is invoked frequently along some run of the program, and do not require any prior knowledge on inputs. The techniques used in [Ball and Larus, 1993; Wu and Larus, 1994; Wagner *et al.*, 1994] involve solving linear equations with sparse matrix solvers, whereas our solution method is different (i.e., based on quantitative analysis of RSMs).

Organization. The rest of this chapter is organized as follows.

1. In Section 6.2 we present some additional definitions on RSMs and phrase the quantitative interprocedural analysis problem (QIA).

2. In Section 6.3 we outline several program analysis problems that can be cast in the QIA framework.

3. In Section 6.4 we present a main algorithm for solving the QIA problem, as well as some improved versions.

4. In Section 6.5 we present three experimental studies, regarding (i) potential misuses of containers that might lead to container bloat (ii) static profiling to predict frequently used methods, and (iii) static profiling to predict frequently used collections of methods.

6.2 Definitions

We consider as input an RSM $\mathsf{RSM} = \{A_1, A_2, \dots, A_k\}$, where A_i consists of n_i nodes and b_i boxes, and $n = \sum_i n_i$ and $b = \sum_i b_i$. Recall our definition of an execution path from Section 2.4 as a sequence of configurations $\pi = (\mathcal{C}_1, \dots, \mathcal{C}_\ell)$. For notational convenience, in this section we frequently represent an execution path as an initial configuration followed by a sequence of RSM transitions $\pi = \langle \mathcal{C}_1 t_1 \dots t_{\ell-1} \rangle$. We will use the toy program of Fig. 6.1 as a running example for this section. Our focus is on two related quantitative interprocedural problems: (i) the ratio analysis problem, and (ii) the mean-payoff analysis problem.

The ratio analysis problem. In the *ratio analysis* problem, every transition of a RSM has a label from the set $\{good, bad, neutral\}$. Intuitively, desirable events are labeled as $good$, undesirable events are labeled as bad, and other events are labeled as $neutral$. The ratio analysis problem, given an RSM RSM, a labeling of the events, and a threshold $\lambda > 0$, asks to determine whether there exists an execution path where the ratio of the sum of the weights of $good$ events over the sum of the weights of the bad events is greater than the threshold λ. Formally, we consider a positive integer-weight function wt, that assigns a positive integer-valued weight to every transition of the RSM. For good and bad events, the weight denotes how good or how bad the respective event is. For a finite execution path π we denote by $good(\mathsf{wt}(\pi))$ (resp., $bad(\mathsf{wt}(\pi))$) the sum of weights of the good (resp., bad) events in π. In particular, for the weight function wt that assigns weight 1 to every transition, we have that $good(\mathsf{wt}(\pi))$ (resp. $bad(\mathsf{wt}(\pi))$) represents the number of good (resp. bad) events. We denote $Rat(\mathsf{wt}(\pi)) = \frac{good(\mathsf{wt}(\pi))}{\max\{1, bad(\mathsf{wt}(\pi))\}}$ the ratio of the sum of weights of good and bad events in π (note that in denominator we have $\max\{1, bad(\mathsf{wt}(\pi))\}$ to remove the pathological case of division by zero). For an infinite execution path π we denote by

$$LimRat(\mathsf{wt}(\pi)) = \begin{cases} \liminf\limits_{i \to \infty} Rat(\mathsf{wt}(\pi[1, i])) & \text{if } \pi \text{ has infinitely many} \\ & \text{good or bad events;} \\ 0 & \text{otherwise;} \end{cases}$$

Informally, this represents the ratio as the number of relevant (good/bad) events goes to infinity. Our analysis focuses on execution paths with unbounded number of relevant events, and infinitely many events provide an elegant abstraction for unboundedness. Hence, we investigate the following problem:

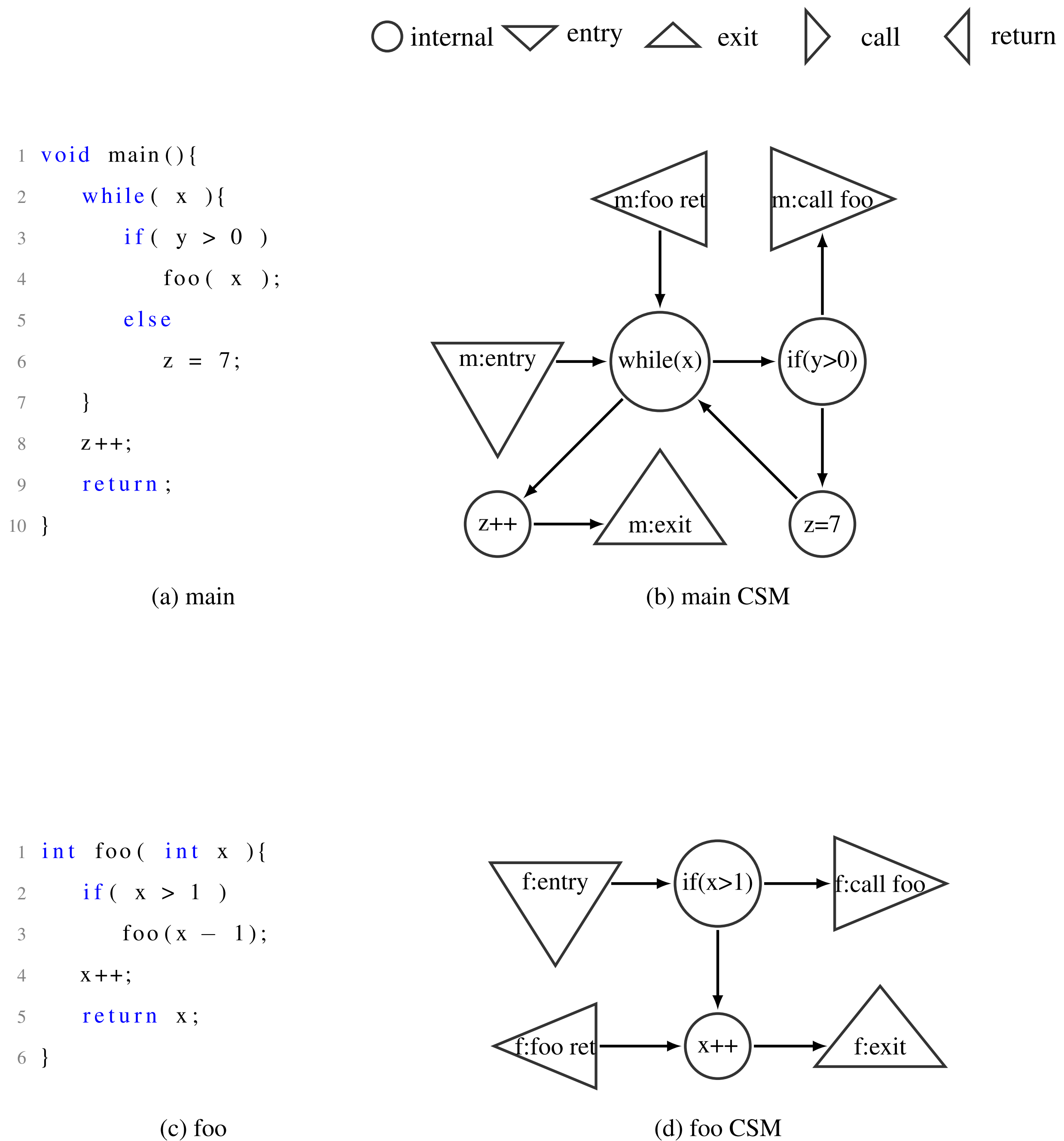


Figure 6.1: An RSM that consists of two CSMs which represent the functions main() and foo().

> Given a RSM with labeling of good, bad, and neutral events, a positive integer weight function wt, and a threshold $\lambda \in \mathbb{Q}$ such that $\lambda > 0$, determine whether there exists an infinite execution path π such that $\mathit{LimRat}(\mathsf{wt}(\pi)) > \lambda$.

Remark 6.1. Our approach can be extended to reason about finite execution paths by a adding an auxiliary transition, labeled as a neutral event, from the final state of the program to its initial state (also see Section 6.3.4).

Mean-payoff analysis problem. In the mean-payoff analysis problem we consider a RSM with a rational-valued weight function wt. For a finite execution path π in a RSM we denote by $\mathsf{wt}(\pi)$ the total weight of the path (i.e., the sum of the weights of the transitions in π), and by $\mathit{Avg}(\mathsf{wt}(\pi)) = \frac{\mathsf{wt}(\pi)}{|\pi|}$ the average of the weights, where $|\pi|$ denotes the length of π. For an infinite execution path π, we denote $\mathit{LimAvg}(\mathsf{wt}(\pi)) = \liminf_{i \to \infty} \mathit{Avg}(\mathsf{wt}(\pi[1, i]))$. The mean-payoff analysis problem asks whether there exists an infinite path π such that $\mathit{LimAvg}(\mathsf{wt}(\pi)) > 0$. In Section 6.4 we show how the ratio analysis problem of RSMs reduces to the mean-payoff analysis problem of RSMs.

Assigning context-dependent and path-dependent weights. In our model the numerical weights are assigned to every transition of a RSM. First, note that since we consider weight functions as an input and allow all weight functions, the weights could be assigned in a dependent way. Second, in general, we can have an RSM, and a finite-state deterministic automaton (such as a deterministic mean-payoff automaton [Chatterjee *et al.*, 2010a]) that assigns weights. The deterministic automaton can assign weights depending on different contexts (or call strings) of invocations, or even independent of the context but dependent on the past few transitions (i.e., path-dependent), i.e., the automaton has the stack alphabet and transition of the RSM as input alphabet and assigns weights depending on the current state of the automaton and an input letter. We call such a weight function *regular weight function*. Given a regular weight function specified by an automaton A and an RSM we can obtain a RSM (that represents the path-dependent weights) by taking their synchronous product, and hence we will focus on RSMs for algorithmic analysis. The regular weight function can also be an *abstraction* of the real weight function, e.g., the regular weight function is an *over-approximation* if the weights that it assigns to the good (resp. bad) events are higher (resp. lower) than the real weights. If the original weight function is bounded, then an over-approximation with a regular weight function can be obtained (which can be refined to be more precise by allowing more states in the automaton of the regular

weight function). Note that the new RSM which is obtained from an RSM and automaton A has a blowup in the number of states of A, and thus there is a tradeoff between the precision of A and the size of the new RSM constructed.

6.3 Applications: Theoretical Modeling

In this section we show that many diverse problems for static analysis can be reduced to ratio analysis of RSMs. We will present experimental results (in Section 6.5) for the problems described in Section 6.3.1 and Section 6.3.2.

6.3.1 Container Analysis

The inefficient use of containers is the cause of many performance issues in Java. An excellent exposition of the problem with several practical motivations is presented in [Xu and Rountev, 2010]. The importance of accurate identification of misuse of containers that minimizes (and ideally eliminates) the number of false warnings was emphasized in [Xu and Rountev, 2010] and much effort was spent to avoid false warnings for real-world programs. We show that the ratio analysis for RSMs provides a mathematically sound approach for the identification of inefficient use of containers.

Two misuses. We aim to capture two common misuses of containers following the definitions in [Xu and Rountev, 2010]. The first inefficient use is an *underutilized container* that always holds very few number of elements. The cause of inefficiency is two-fold: (i) a container is typically created with a default number of slots, and much more memory is allocated than needed; and (ii) the functionality that is associated with the container is typically not specialized to the case that it has only very few elements. The second inefficiency is caused by *overpopulated containers* that are looked up rarely, though potentially they can have many elements. This causes a memory waste and performance penalty for every lookup. Thus we consider the following two cases of misuse:

1. A container is *underutilized* if there exists a constant bound on the number of elements that it holds for all runs of the program.

2. For a threshold λ, a container is *overpopulated* if for all runs of the program the ratio of GET vs ADD operations is less than λ.

We note that our approach is demand-driven (where users can specify to check the misuse of a specific container).

Modeling. The modeling of programs as RSMs is standard. We describe how the weight function and the ratio analysis problem can model the problem of detecting misuses. We abstract the different container operations into GET, ADD, and DELETE operations. For this purpose we require the user to annotate the relevant class methods by GET, ADD, or DELETE; and by a weight function that corresponds to the number of GET, ADD, or DELETE operations that the method does (typically this number is 1). For example, in the class **HashSet**, the **add** method is annotated by ADD, the **contains** method is annotated by GET and the **remove** operation is annotated by DELETE. The **clear** operation which removes all elements from the set is annotated by DELETE but with a large weight (if **clear** appears in a loop, it dominates the add operations of the loop). We note that the annotation can be automated with the approach that is described in [Xu and Rountev, 2010].

1. When detecting underutilized containers we define ADD operations as good events and DELETE operations as bad events, and check for threshold 1. Note that the relevant threshold is 1: if the (long-run) ratio of ADD vs DELETE is not greater than 1, then the total number of elements in the container is bounded by a constant. We remark that we assume that a DELETE operation always succeeds, as our goal is to detect containers that may be underutilized. Additionally, in some popular data-structures, DELETEoperations always succeed as long as the data structure is not empty. This is the case, e.g., with the pop operation in stacks and queues.

2. When detecting overpopulated containers we define GET operations as good events and ADD operations as bad events, and check for the given threshold λ.

In addition, since we wish to analyze heap objects, the allocation of the container is a bad event with a large weight (i.e., similar effect as of **clear**); see Example 6.1. The container is misused iff the answer to the ratio analysis problem is NO (note that in the problem description for container analysis we have quantification over all execution paths and for ratio analysis of RSMs the quantification is existential). The detection is demand-driven and done for an allocated container

```
1  void qux( Queue q, int x ){
2      q.push((x, x/2));
3      if( x > 0 ){
4          qux( q, x/2 );
5      }
6  }
7
8  Queue bar( int x ){
9      return new Queue(x*x);
10 }
```

```
12 void foo( int x ){
13     if( x % 2){
14         Queue q1 = bar(x);
15         qux(q1, x);
16     }
17     else
18     {
19         for( int y = 0 ; y < x ; y++ ){
20             Queue q2 = new Queue(y);
21             q2.push( (y,x) );
22             for( int z = 0 ; z < y ; z++ )
23             {
24                 q2.push((z,y));
25                 ...
26                 q2.pop();
27             }
28         }
29     }
30 }
```

Figure 6.2: An example for underutilized container analysis.

c.

Details of modeling. Intuitively, a transition in the call graph is good if it invokes a functionality that is annotated by a good operation (i.e., ADD operation for the underutilized analysis and GET operation for the overpopulated analysis) and the object that invokes the operation points to container **c**, and it is bad if the invoked operation is annotated as bad. Formally, for a given allocated container **c**: If at a certain line a variable **t** that *may point to* **c** invokes a good functionality, then we denote the transition as good. If **t** *must point to* **c** and invokes a bad functionality, then we denote it as a bad events. All other transitions are neutral. Note that our modeling is conservative. The misuse is detected for the container **c** if all runs of the program have a ratio of good vs bad events that is below the threshold (in other words, the container is not misused if there exists an execution path where the ratio of good vs bad events is above the threshold, and this exactly corresponds to the ratio analysis of RSMs).

Example 6.1 (Underutilized container analysis). We illustrate some important aspects of

the container analysis problem with an example. Consider the program shown in Fig. 6.2. We consider the containers that are allocated in line 9 and in line 20 and analyze them for underutilization. There exist runs that go through line 14 and properly use the container that is allocated in line 9, since the qux method can add unbounded number of elements to the queue (due to its recursive call). However, the container in line 20 is underutilized, since in every run the number of elements is bounded by 2. However, note that if the DELETE operation is not handled, then the container is reported as properly used. We note that since we assign large weights to the allocation of the container, this prevents the analysis from reporting that q2 properly uses the container that is allocated in line 20. In summary, the example illustrates the following important features: (1) the proper usage of the container should be tested also outside of its allocation site [1] (2) sometimes the proper usage of a container is due to recursion; and (3) handling DELETE operations appropriately increases the precision of analysis. While these important features are illustrated with the toy example, such behaviors were also manifested in the programs of the benchmarks (see Section 6.5.2 for details).

6.3.2 Static Profiling of Methods

Finding the most frequently executed lines in the code can help the programmer to identify the critical parts of the program and focus on the optimization of these parts. It can also assist the compiler (e.g., a C compiler) to decide whether it should apply certain optimizations such as *function inlining* (replacing a function call by the body of the called function) and *loop unrolling* (re-write the loop as a repeated sequence of similar independent statements). These optimizations can reduce the running time of the program, but on the other hand, they increase the size of the (binary) code. Hence, knowing whether the function or loop is *hot* (frequently invoked) is important when considering the time vs. code size tradeoff. In this subsection we present the model for profiling the frequency of function calls (which allows finding hot functions), and we note that our profiling technique is generic and can be scaled to detect other hot spots in the code (e.g., hot loops).

Problem description. Given a program with several functions, a function f is called λ-hot, if

[1] E.g., a Queue is allocated in bar function but the proper usage is done outside the allocation site, namely, after the termination of bar.

there exists an (interprocedural) execution path (of unbounded length) of the program where the frequency of calls to f (among all function calls in the execution path) is at least λ. Formally, for an execution path, given a prefix of length i, let $\#f(i)$ denote the number of calls to f and $\#c(i)$ denote the number of function calls in the prefix of length i. The function is λ-hot if there exists an execution path such that $\liminf_{i\to\infty} \frac{\#f(i)}{\#c(i)} > \lambda$.

Modeling. The modeling of programs as RSMs is straightforward. We describe the labeling of events and weight function in RSMs to determine if a function f is λ-hot. First we label call-transitions to f as good events and assign weight 1; then we label all other call-transitions as bad events and assign them weight 1. To ensure that the number of calls to f also appear in the denominator (in the total number of calls) we label transitions from the entry node of f as bad events with weight 1. The function f is λ-hot iff the answer to the ratio analysis problem with threshold λ is YES.

6.3.3 Static Profiling of Libraries

Modern software comprises thousands of methods and classes, which are usually grouped into libraries. In several programming languages (e.g. C++), the developer decides whether libraries will be statically or dynamically linked. While there are several factors that affect this choice (e.g., compatibility issues, licensing restrictions etc.), one consideration is that of performance, as statically linked libraries allow for speedups such as the following.

1. Interprocedural optimizations coming from the compiler by performing interprocedural analysis on the target program together with the statically linked libraries.

2. Avoiding runtime overheads imposed by invocations of methods that lie in dynamically linked libraries.

3. Cache-level optimizations, by allowing the compiler to arrange statically linked libraries in such order to try to minimize cache misses.

On the other hand, infrequently used libraries might better be dynamically linked, (i) to keep the interprocedural optimizations fast, (ii) keep the size of the executable small, and (iii) have fewer

cache misses. Estimating statically the frequently used libraries can assist static vs dynamic linking.

Problem description. Similarly as in Section 6.3.2, given a program that links with several libraries, a library ℓ is called λ-hot, if there exists an (interprocedural) execution path (of unbounded length) of the program where the frequency of calls to methods of ℓ (among all method calls to libraries in the execution path) is at least λ. Formally, for an execution path, given a prefix of length i, let $\#\ell(i)$ denote the number of calls to methods of the library ℓ and $\#t(i)$ denote the number of function calls to library methods in the prefix of length i. The library ℓ is λ-hot if there exists an execution path such that $\liminf_{i\to\infty} \frac{\#\ell(i)}{\#t(i)} > \lambda$.

Modeling. The modeling is similar to that of Section 6.3.2. The program is modeled as an RSM. Call transitions to methods of the library ℓ are labeled as good events, whereas call transitions to methods of other libraries are labeled as bad events. To ensure that the number of calls to methods of ℓ also appear in the denominator (in the total number of library calls) we label transitions from the entry node of every method of ℓ as bad events. All labeled events receive weight 1. The library ℓ is λ-hot iff the answer to the ratio analysis problem with threshold λ is YES.

6.3.4 Estimating Worst-case Execution Time

The approach of [Cerný *et al.*, 2013] for estimating worst-case execution time (WCET) is also naturally captured by ratio analysis. While the intraprocedural problem was considered in [Cerný *et al.*, 2013], our approach allows the more general interprocedural analysis. In this approach, we consider (as in [Cerný *et al.*, 2013]) that each program statement is assigned a cost that corresponds to its running time (e.g., number of CPU cycles).

Modeling. The modeling of WCET analysis of the program is as follows: We add to the RSM of the program a transition from every terminal node to the initial node, and every such transition is a bad event with weight 1. All the other transitions are good events and their weight is their cost (running time). The WCET of the program is at most N cycles if and only if the answer to the ratio analysis problem with threshold N is NO.

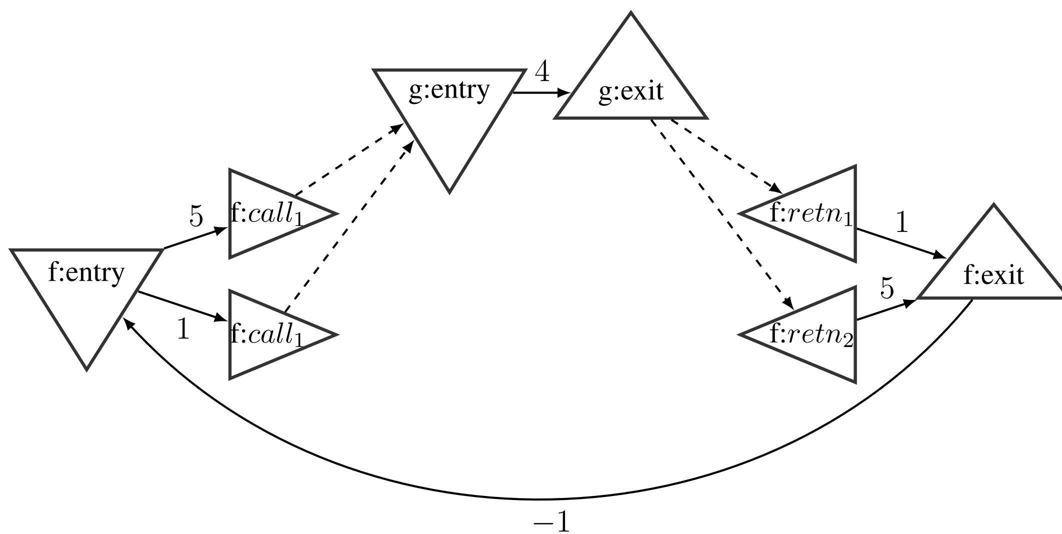


Figure 6.3: Illustration of the Quantitative Interprocedural Analysis problem for capturing the interprocedural WCET. Positive weights indicate good transitions, whereas negative weights indicate bad transitions.

Example 6.2 (Worst-case execution time.). Consider the interprocedural WCET problem shown in Fig. 6.3, where we have a program that consists of two methods f and g. The main method is f, which has two calls to method g. After having executed some intraprocedural analysis locally in each method, we have obtain the respective WCET bounds between various locations in each method, shown with positive integers. In order to reason about the total WCET, a quantitative interprocedural analysis needs to account for execution paths that involve method invocations and returns. Note that a precise interprocedural analysis will return that the WCET is bounded by 10 time units. In contrast, if we apply only intraprocedural analysis and connect (i) every method call to the entry of the invoked method, and (ii) every method return to the exit of the invoked method, then the tightest WCET bound we obtain is 14 time units, i.e., much looser than the bound we obtain from the precise interprocedural analysis.

6.3.5 Evaluating The Speedup in a Parallel Computation

The speed of a parallel computation is limited by the time needed for the sequential fraction of the program. For example, if a program runs for 10 minutes on a single processor core, and a certain part of the program that takes 2 minutes to execute cannot be parallelized, then the minimum execution time cannot be less than two minutes (regardless of how many processors are

devoted to a parallelized execution of this program). Hence, the speedup is at most 5. Amdahl's law [Amdahl, 1967] states that the theoretical speedup that can be obtained by executing a given algorithm on a system capable of executing n threads of execution is at most $\frac{1}{B+\frac{1}{n}(1-B)}$, where B is the fraction of the algorithm that is strictly serial. Our ratio analysis technique can be used to (conservatively) estimate the value of B and thus to evaluate the outcome of Amdahl's law.

Modeling. As in Section 6.3.4, we consider that the cost of every program statement is given, and we add to the RSM of the program a transition from every terminal node to the initial node, this time as a neutral event with weight 0. All the transitions of the code that cannot be parallelized are defined as good events, and the other transitions are defined as bad events. We denote by P the fraction of the code that can be parallelized and by S the fraction of the code that is strictly serial. The value of $\frac{S}{P}$ is at most λ if and only if the answer to the ratio analysis problem with threshold λ is NO. Hence B is bounded by $\frac{1}{1+\frac{1}{\lambda}}$ for which the answer to the ratio analysis problem with threshold λ is NO.

6.3.6 Average Energy Consumption

In the case of many consumer electronics devices, especially mobile phones, battery capacity is severely restricted due to constraints on size and weight of the device. This implies that managing energy well is paramount in such devices. Since most mobile applications are non-terminating (e.g., a web browser), the most important metric for measuring energy consumption is the average memory consumption per time unit [Carroll and Heiser, 2010], e.g., watts per second.

Modeling. We consider that the running time and energy consumption of each statement in the application code is given (or is approximated). In our modeling we split each transition in the RSM into two consecutive transitions, the first is a good event and the next is a bad event. The good event is assigned with a weight that corresponds to the energy consumption of the program statement and the bad event is assigned with a weight that corresponds to the running time of the statement. The average energy consumption of the application is at most λ if and only if the answer to the ratio analysis problem is NO.

6.4 An Algorithm for the Quantitative Interprocedural Analysis Problem

The mean-payoff analysis problem for RSMs can be solved in polynomial time, a result which can be derived from [Chatterjee and Velner, 2012]. In this section we present three results. First, we present an algorithm that significantly improves the current theoretical bound for the problem for RSMs. Second, we present an efficient algorithm that in most practical cases is much faster as compared to the theoretical upper bound. Finally, we present a linear reduction of the ratio analysis problem to the mean-payoff analysis problem for RSMs.

6.4.1 An Improved Algorithm for Mean-payoff Analysis

In this section we first discuss the basic polynomial-time algorithm for mean-payoff analysis of RSMs that can be obtained from the results on pushdown systems shown in [Chatterjee and Velner, 2012].

Results of [Chatterjee and Velner, 2012] and reduction. The results of [Chatterjee and Velner, 2012] show that pushdown systems with mean-payoff objectives can be solved in polynomial time. Given a pushdown system with state space Q and stack alphabet Γ, the polynomial-time algorithm of [Chatterjee and Velner, 2012] can be described as follows. The algorithm is iterative, and in each iteration it constructs a finite graph of size $O(|Q| \cdot |\Gamma|^2)$ and runs a Bellman-Ford style algorithm on the finite graph from each node. The Bellman-Ford algorithm on the finite graph from all nodes in each iteration requires $O(|Q|^3 \cdot |\Gamma|^6)$ time and $O(|Q|^2 \cdot |\Gamma|^4)$ space. The number of iterations required is $O(|Q|^2 \cdot |\Gamma|^2)$. Thus the time and space requirement of the algorithm are $O(|Q|^5 \cdot |\Gamma|^8)$ and $O(|Q|^2 \cdot |\Gamma|^4)$, respectively. An RSM can be interpreted as a pushdown system where N corresponds to Q and $\mathsf{Returns}$ corresponds to Γ.

Theorem 6.1 (**Basic algorithm [Chatterjee and Velner, 2012]).** *The mean-payoff analysis problem for RSMs can be solved in $O(n^5 \cdot b^8)$ time and $O(n^2 \cdot b^4)$ space, respectively.*

Improved algorithm. We will present an improved polynomial-time algorithm for the mean-payoff analysis of RSMs. The improvement relies on the following properties of RSMs:

1. The transitions of a CSM are independent of the stack of a configuration, while in pushdown systems the transitions can depend on the top symbol of the stack. This enables to reduce the size of the finite graphs to be considered in every iteration.

2. Every call node has only one corresponding return node. Therefore, if a CSM A_1 invokes a CSM A_2, then the behavior of A_1 after the termination of A_2 is independent of A_2. This enables us to reduce the number of iterations to $O(b)$.

To present the improved algorithm and its correctness formally, we need a refined analysis and extensions of the results of [Chatterjee and Velner, 2012]. We first describe a key aspect and present an overview of the solution.

Remark 6.2. (Infinite-height lattice). Our algorithm will be an iterative algorithm till some fixpoint is reached. However, for interprocedural analysis with finite-height lattices, fixpoints are guaranteed to exist. Unfortunately in our case for mean-payoff objectives, it is an infinite-height lattice. Thus a fixpoint is not guaranteed. For this reason the analysis for mean-payoff objectives is more involved, and this is even in the case of finite graphs. For example, for reachability objectives in finite graphs linear-time algorithms exist, whereas for finite graphs with mean-payoff objectives the best-known algorithms (for over three decades) are quadratic [Karp, 1978]. This difference is even more pronounced in our case of recursive graphs.

Solution overview. In finite graphs the solution for the mean-payoff analysis is to check whether the graph has a cycle C such that the sum of weights of C is positive. If such a cycle exists, then a *lasso* execution path that leads to the cycle and then follows the cyclic execution path forever has positive mean-payoff value. For RSMs we show that it is enough to find either a loop in the program such that the sum of weights of the loop is positive or a sequence of calls and returns with positive total weight such that the last invoked CSM is the same as the first invoked CSM. For this purpose we compute a *summary function* that finds the maximum weight (according to the sum of weights) execution path between every two statements of a method (i.e., between every two nodes of a CSM). The computation is an extension of the Bellman-Ford algorithm to RSMs. We show that it is enough to compute a summary function for RSMs with a *stack height* that is bounded by some constant, and then all that is left is to mark pairs of nodes such that the weight of a maximal weight execution path between them is unbounded. In finite graphs the maximum weight between two nodes is unbounded only if the graph has a cycle with positive sum of weights (i.e., an execution path with positive total weight that can be pumped). For

RSMs it is also possible to *pump* special types of acyclic execution paths. We first characterize these pumpable execution paths (up to Lemma 6.2). We then show how to compute a bounded summary function (Lemma 6.3 and the paragraph that follows it and Example 6.5). Finally we show how to use the summary function to solve the mean-payoff analysis problem. We start with the basic notions related to stack heights and pumpable execution paths, and their properties which are crucial for the algorithm.

Pumpable pair of execution paths. Let $\pi = \langle \mathcal{C}_1 t_1 t_2 \dots \rangle$ be a finite or infinite execution path (where each t_i is a transition of RSM). A *pumpable pair of execution paths* for π is a pair of non-empty sequences of transitions: $(p_1, p_2) = (t_{i_1} t_{i_1+1} \dots t_{i_1+\ell_1}, t_{i_2} t_{i_2+1} \dots t_{i_2+\ell_2})$, for $\ell_1, \ell_2 \geqslant 0$, $i_1 \geqslant 0$ and $i_2 > i_1 + \ell_1$ such that for every $j \geqslant 0$ the execution path $\pi^j_{(p_1,p_2)}$ obtained by pumping the pair p_1 and p_2 of execution paths j times each is a valid execution path, i.e., for every $j \geqslant 0$ we have

$$\pi^j_{(p_1,p_2)} = \langle \mathcal{C}_1 t_1 \dots t_{i_1-1} (p_1)^j t_{i_1+\ell_1+1} \dots t_{i_2-1} (p_2)^j t_{i_2+\ell_2+1} \dots \rangle$$

is a valid execution path. We illustrate the above definitions with the next example.

Example 6.3 (Pumpable pair of execution paths.). Consider the program from Fig. 6.1 and the corresponding RSM. A possible execution path in the program is

m:entry $\rightarrow$ while(x) $\rightarrow$ if(y>0) $\rightarrow$ m:call foo $\rightarrow$ f:entry $\rightarrow$ if(x>1) $\rightarrow$
f:call foo $\rightarrow$ f:entry $\rightarrow$ if(x>1) $\rightarrow$ x++ $\rightarrow$ f:exit $\rightarrow$ f:foo ret $\rightarrow$ x++ $\rightarrow$ f:exit $\rightarrow$
m:foo ret $\rightarrow$ while(x)

and we denote this execution path with π. Then $\mathsf{ASH}(\pi) = 2$, and the pair of execution paths f:entry $\rightarrow$ if(x>1) $\rightarrow$ f:call foo and f:foo ret $\rightarrow$ x++ $\rightarrow$ f:exit is a pumpable pair of execution paths.

In the next lemmas we first show that every execution path with large additional stack has a pumpable pair of execution paths, and then establish the connection of additional stack height and the existence of pumpable pair of execution paths with positive weights in Lemma 6.2. The key intuition for the proof of the next lemma is that an execution path with $\mathsf{ASH}(\pi) > b + 1$ must contain a recursive call that can be pumped.

Lemma 6.1. *Let π be a finite execution path with $\mathsf{ASH}(\pi) = d > b + 1$. Then π has a pumpable pair of execution paths.*

Proof. Intuitively an execution path with $\mathsf{ASH}(\pi) > b + 1$ must contain a recursive call that can be pumped. We now present the detailed argument. Let $\mathcal{C}_0$ and $\mathcal{C}_j$ be the starting and the end configurations of the finite execution path π, respectively. Let $\ell = \max\{\mathsf{SH}(\mathcal{C}_0), \mathsf{SH}(\mathcal{C}_j)\}$. Given π, let $\mathcal{C}_1$ be the first configuration in π of stack height strictly greater than ℓ and with a call node $u \in \mathsf{Calls}_i$ (for some CSM A_i) such that there exists a configuration $\mathcal{C}_2$ in π with a call node $v \in \mathsf{Calls}_i$ satisfying the following conditions: (i) $u = v$ and (ii) in the execution path segment in π between $\mathcal{C}_1$ and $\mathcal{C}_2$ the stack height is always at least $\mathsf{SH}(\mathcal{C}_1)$. Moreover, let $\mathcal{C}_3$ be the first configuration after $\mathcal{C}_2$ of stack height $\mathsf{SH}(\mathcal{C}_1)$ and with a return node $x \in \mathsf{Returns}_i$. We first justify the existence of these configurations: (i) the existence of $\mathcal{C}_1$ and $\mathcal{C}_2$ follows by the pigeonhole principle and the fact that $\mathsf{ASH}(\pi) > b+1$; and (ii) the existence of $\mathcal{C}_3$ follows because $\mathsf{SH}(\mathcal{C}_1) > \mathsf{SH}(\mathcal{C}_j)$ and hence the call corresponding to $\mathcal{C}_1$ must return in the execution path π. Note that existence of $\mathcal{C}_3$ (i.e., the return of the call of $\mathcal{C}_1$) implies the existence of a configuration $\mathcal{C}_4$ with a return node $y \in \mathsf{Returns}_i$ in the execution path such that $\mathsf{SH}(\mathcal{C}_4) = \mathsf{SH}(\mathcal{C}_2)$, (this corresponds to the return of the call of $\mathcal{C}_2$). Note that since $u = v$, it follows that $x = y$ (as they corresponds to the return of the same call node). The path segment p_1 of π between $\mathcal{C}_1$ and $\mathcal{C}_2$, and the path segment p_2 of π between $\mathcal{C}_4$ and $\mathcal{C}_3$, constitutes a pumpable pair. The result follows. □

Lemma 6.2. *Let $\mathcal{C}_1, \mathcal{C}_2$ be two configurations and $j \in \mathbb{Z}$. Let $d \in \mathbb{N}$ be the minimal additional stack height of all execution paths between $\mathcal{C}_1$ and $\mathcal{C}_2$ with total weight at least j. If $d > b + 1$, then there exists an execution path π^* from $\mathcal{C}_1$ to $\mathcal{C}_2$ with additional stack height d that has a pumpable pair (p_1, p_2) with $\mathsf{wt}(p_1) + \mathsf{wt}(p_2) > 0$.*

Proof. Let us consider the set of execution paths Π between $\mathcal{C}_1$ and $\mathcal{C}_2$ with total weight at least j, and let $\Pi_{\min}$ be the subset of Π that has minimal additional stack height. The proof is by induction on the length of execution paths in $\Pi_{\min}$. Consider an execution path π from $\Pi_{\min}$ that has the shortest length among all execution paths in $\Pi_{\min}$. Since $\mathsf{ASH}(\pi) = d > b + 1$, then by Lemma 6.1 it contains a pumpable pair. Let us consider the path π_1 obtained from π by pumping the pumpable pair zero times (i.e., the pumpable pair is removed). Since we remove a part of the execution path we have that $\mathsf{ASH}(\pi_1) \leqslant \mathsf{ASH}(\pi)$. If $\mathsf{wt}(\pi_1) \geqslant \mathsf{wt}(\pi)$, then we obtain an execution path π_1 with weight at least j, with either smaller additional stack height than π, or of shorter length, contradicting that π is the shortest length minimal additional stack height execution path with weight at least j. Hence we must have $\mathsf{wt}(\pi_1) < \mathsf{wt}(\pi)$, and hence

the pumpable pair has positive weight. Now for an arbitrary execution path π in $\Pi_{\min}$ we obtain that it has a pumpable pair. Either the pumpable pair has positive weight and we are done, else removing the pumpable pair we obtain a shorter length execution path of the same stack height, and the result follows by inductive hypothesis on the length of execution paths. □

Example 6.4 (Illustration of Lemma 6.2). We illustrate Lemma 6.2 on our running example. Consider again the program from Fig. 6.1 and the corresponding RSM. Additionally, consider a weight function that assigns -1 to the transition that foo calls recursively itself, +2 to the transitions to the exit node of foo (i.e., when foo returns), and 0 to every other transition. Examine two configurations $\mathcal{C}_1 = (\epsilon, f : if(x > 1))$, $\mathcal{C}_2 = (\epsilon, f : exit)$, and note that for $j = 4$, the minimal additional stack height d of all execution paths from $\mathcal{C}_1$ to $\mathcal{C}_2$ with total weight at least j is $d = 4$, as they all have to make at least 4 recursive calls to foo to witness a weight of at least 4 (in particular, i invocations and returns to and from foo contribute a weight of $i \cdot (-1 + 2) = i$). Since there are only 2 calls in total, Lemma 6.2 identifies a pumpable pair of execution paths f:entry $\rightarrow$ if(x>1) $\rightarrow$ f:call foo and f:foo ret $\rightarrow$ x++ $\rightarrow$ f:exit with positive total weight. Indeed, the weight is $-1 + 2 > 0$, and the pair of execution paths is pumpable due to recursion, as pointed in Example 6.3. Observe that this conclusion cannot be made using Lemma 6.2 with e.g. $j = 1$, as in that case there is an execution path from $\mathcal{C}_1$ to $\mathcal{C}_2$ that witnesses weight at least j and has additional stack height only 1 (i.e., the execution path that only calls foo recursively once), which is less than the number of calls in the program.

Our algorithm for the mean-payoff analysis problem is based on detecting the existence of certain *non-decreasing* execution paths with positive weight. The maximal weights of such non-decreasing execution paths between node pairs are captured with the notion of a *summary function* and *bounded summary functions* (with bounded additional stack height). We now define them, and establish the lemma related to the number of bounded summary functions to be computed.

Local minimum and non-decreasing execution paths. A configuration $\mathcal{C}_i$ in an execution path $\pi = \langle \mathcal{C}_1, \dots, \mathcal{C}_\ell \rangle$ is a *local minimum* if the stack height of $\mathcal{C}_i$ is minimal in π, i.e., $|\alpha_i| = \min(|\alpha_1|, \dots, |\alpha_\ell|)$. An execution path from configuration (u, α) to $(v, \alpha\beta)$ is a *non-decreasing α-execution path* if (u, α) is a local minimum. Note that if a sequence of transitions is a non-decreasing α-execution path for some $\alpha \in \mathsf{Returns}^*$, then the same sequence of transitions is a non-decreasing γ-path for every $\gamma \in \mathsf{Returns}^*$. Hence, we say that π is a non-decreasing

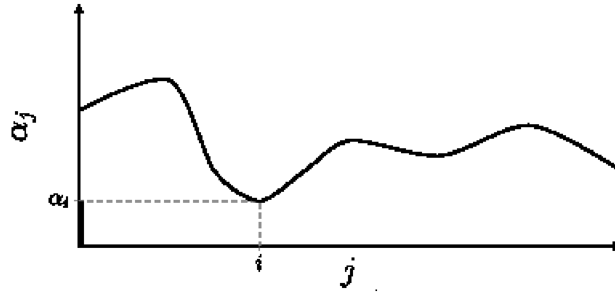


Figure 6.4: Example of an execution path $\pi = \langle \mathcal{C}_1, \ldots, \mathcal{C}_\ell \rangle$, where j indexes the j-th configuration of π, and α_j is the stack of $\mathcal{C}_j$. The configuration $\mathcal{C}_i$ is a local minimum of π. The suffix π_i of π starting at the i-th configuration is a non-decreasing execution path, as the top symbol of α_i is never popped.

execution path if there exists $\alpha \in \mathsf{Returns}^*$ such that π is a non-decreasing α-execution path. Fig. 6.4 illustrates the concepts of local minimum and non-decreasing execution paths.

Summary function. Given the RSM RSM and $\alpha \in \mathsf{Returns}^*$, we define a summary function $s_\alpha : \bigcup_{1 \leqslant i \leqslant k}(N_i \times N_i) \to \{-\infty\} \cup \mathbb{Z} \cup \{\omega\}$ as:

1. $s_\alpha(u, v) = z \in \mathbb{Z}$ iff the weight of the maximum weight non-decreasing execution path from configuration (u, α) to configuration (v, α) is z.

2. $s_\alpha(u, v) = \omega$ iff for all $j \in \mathbb{N}$ there exists a non-decreasing execution path from (u, α) to (v, α) with weight at least j.

3. $s_\alpha(u, v) = -\infty$ iff there is no non-decreasing execution path from (u, α) to (v, α).

We note that for every $\alpha, \beta \in \mathsf{Returns}^*$ it holds that $s_\alpha \equiv s_\beta$. Hence, we consider only $s \equiv s_\epsilon$ (where ϵ is the empty string and corresponds to empty stack). The computation of the summary function is done by considering stack height bounded summary functions defined below.

Stack height bounded summary function. For every $d \in \mathbb{N}$, the *stack height bounded summary function* $s_d : \bigcup_{1 \leqslant i \leqslant k}(N_i \times N_i) \to \{-\infty\} \cup \mathbb{Z} \cup \{\omega\}$ is defined as follows: (i) $s_d(u, v) = z \in \mathbb{Z}$ iff the weight of the maximum weight non-decreasing execution path from (u, ϵ) to (v, ϵ) with additional stack height at most d is z; (ii) $s_d(u, v) = \omega$ iff for all $j \in \mathbb{N}$ there exists a non-decreasing execution path from (u, ϵ) to (v, ϵ) with weight at least j and additional stack height at most d; and (iii) $s_d(u, v) = -\infty$ iff there is no non-decreasing execution path with additional stack height at most d from (u, ϵ) to (v, ϵ).

Facts of summary functions. We have the following facts: (i) for every $d \in \mathbb{N}$, we have $s_{d+1} \geqslant s_d$ (monotonicity); and (ii) s_{d+1} is computable from s_d and RSM. By the above facts we get that if $s_d \equiv s_{d+1}$, i.e., if a fix point is reached, then $s \equiv s_d$. For interprocedural analysis with finite-height lattices, fix points are guaranteed to exist. Unfortunately in our case, the image of s_ℓ is infinite and moreover, it is an infinite-height lattice. Thus a fix point is not guaranteed. The next lemma shows that we can compute all the non-ω values of s with the bounded summary function.

Lemma 6.3. *Let $d = b + 1$. For all $u, v \in N$, if $s(u, v) \in \mathbb{Z} \cup \{-\infty\}$, then $s(u, v) = s_d(u, v)$.*

Proof. Obviously $s(u, v) \geqslant s_d(u, v)$. If $s_d(u, v) < s(u, v)$ it follows that there exists a non-decreasing execution path π from u to v with $\mathsf{wt}(\pi) > s_d(u, v)$. By the definition of the bounded-height summary function it follows that $\mathsf{ASH}(\pi) > d$, and w.l.o.g we assume that π has the minimal additional stack height among all non-decreasing execution paths from u to v with weight $\mathsf{wt}(\pi)$. Then by Lemma 6.2 it follows that π has a pumpable pair (p_1, p_2) with $\mathsf{wt}(p_1) + \mathsf{wt}(p_2) = w_p > 0$. Hence, for every $j \geqslant 0$ the execution path π^j that is obtained from π by pumping the pair (p_1, p_2) exactly j times has weight $\mathsf{wt}(\pi^j) = \mathsf{wt}(\pi) + (j - 1) \cdot w_p$, and it is a valid non-decreasing execution path from u to v. Hence, for every $\ell \in \mathbb{N}$ the execution path π^j for $j = \lceil \frac{\ell - \mathsf{wt}(\pi)}{w_p} + 1 \rceil$ satisfies $\mathsf{wt}(\pi^j) \geqslant \ell$ (if $\ell \leqslant \mathsf{wt}(\pi)$, then we set $j = 1$). By definition we get that $s(u, v) = \omega$, and this completes the proof. $\square$

By Lemma 6.3 we get that if $s_{d+1}(u, v) > s_d(u, v)$ (for $d = b + 1$), then $s(u, v) = \omega$. Hence, the summary function s is obtained by the fix point of the following computation: (i) Compute $s_{\ell+1}$ from s_ℓ up to s_d for $d = b+1$; (ii) for $i \geqslant b+1$, if $s_{\ell+1}(u, v) > s_\ell(u, v)$, then set $s_{\ell+1}(u, v) = \omega$; (iii) a fix point is reached after at most $O(b)$ iterations (say j iterations), and then we set $s \equiv s_j$. This establishes that we require only $O(b)$ iterations as compared to $O(n^2 \cdot b^2)$ iterations. The number of returns and calls are the same and thus we significantly improve the number of iterations required from the quartic worst-case bound to linear bound. We now describe the computation of every iteration to obtain $s_{\ell+1}$ from s_ℓ.

Computation of $s_{\ell+1}$ from s_ℓ. We first compute a partial function, namely, $s'_{i+1} : \mathsf{En} \times \mathsf{Ex} \to \{-\infty, \omega\} \cup \mathbb{Z}$ that satisfies $s'_{\ell+1}(u, v) = s_{\ell+1}(u, v)$ for every $u \in \mathsf{En}$ and $v \in \mathsf{Ex}$. We initialize

$s'_0(u,v) = s_0(u,v)$. For every CSM A_i we construct the weighted graph G_i^ℓ by taking all the nodes and transitions of A_i and by adding a transition between every call node and its corresponding return node. For every transition between a pair of nodes $u, v \in N_i \setminus (\mathsf{Calls}_i \cup \mathsf{Returns}_i)$ we assign the weight according to the original weight in RSM. For every transition between a call node that invokes CSM A_p and a corresponding return node we assign the weight $s'_i(\mathsf{en}_p, \mathsf{ex}_p)$. To compute s'_{i+1} for CSM A_i we run one Bellman-Ford iteration over G_i^ℓ for source node en_i and target node ex_i. We observe the next two key properties of s'_i:

- For every iteration i, a CSM A_i, and pair of nodes $u, v \in N_i$ we have that the weight of the maximum weight execution path between u and v in G_i^ℓ is exactly $s_{\ell+1}(u,v)$ (the proof is by a simple induction over i).
- If $s'_{i+1} \equiv s'_i$, then $s_{\ell+1} \equiv s_\ell$ (follows from the first key property).

Hence, to compute s we compute s'_{i+1} from s'_i until we get $s'_{i+1} \equiv s'_i$, and then we compute all pairs maximum weight execution paths (e.g., by the Floyd-Warshall algorithm) over every G_i^ℓ and get $s_{\ell+1}$ (and $s_{\ell+1} \equiv s$). The Floyd-Warshall algorithm has a cubic time complexity and quadratic space complexity [Cormen *et al.*, 2009]. Therefore, the time complexity for computing every iteration of s_ℓ is $O(\sum_i n_i^2)$ and the complexity of the last step is $O(\sum_i n_i^3)$. The space complexity of the last step is $O(\max_i n_i^2)$, but to store s_ℓ we require $O(\sum_i n_i^2)$ space.

Summary graph. Given the RSM RSM with a summary function s, we construct the *summary graph* $\mathsf{Gr}(\mathsf{RSM}, s) = (V_s, E_s, \mathsf{wt}_s)$ of RSM with weight function $\mathsf{wt}_s : E_s \to \mathbb{Z} \cup \{\omega\}$ as follows: (i) $V_s = N \setminus (\mathsf{Ex} \cup \mathsf{Returns})$; and (ii) $E_s = E_{internal} \cup E_{calls}$ where $E_{internal} = \{(u,v) \mid u,v \in N_i$ for some i, and $s(u,v) > -\infty\}$ contains the transitions in the same CSM and $E_{calls} = \{(u,v) \mid u \in \mathsf{Calls}$ and $v \in \mathsf{En}$ and u is a call to a CSM with entry node $v\}$ contains the call transitions. The weights of $E_{internal}$ are according to the summary function s and the weights of E_{Calls} are according to the weights of these transitions in RSM (i.e., according to the weight function wt of RSM). A simple cycle in $\mathsf{Gr}(\mathsf{RSM}, s)$ is a *positive simple cycle* iff one of the following conditions hold: (i) the cycle contains an ω edge; or (ii) the sum of the weights of the cycles according to the weights of the summary graph is positive. Lemma 6.4 shows the equivalence of the mean-payoff analysis problem and positive cycles in the summary graph.

Lemma 6.4. *The RSM* RSM *has an execution path* π *with* $LimAvg(\mathsf{wt}(\pi)) > 0$ *iff the summary graph* $\mathsf{Gr}(\mathsf{RSM}, s)$ *has a (reachable) positive cycle.*

Proof. If $\mathsf{Gr}(\mathsf{RSM}, s)$ does not contain a positive cycle, then it follows that the weight of every non-decreasing execution path in RSM is bounded by the weight of the maximum weight execution path in $\mathsf{Gr}(\mathsf{RSM}, s)$. Hence, for every infinite execution path π we get that every prefix of π is a non-decreasing execution path from the initial configuration with bounded weight (sum of weights bounded from above), and therefore $LimAvg(\mathsf{wt}(\pi)) \leqslant 0$. Conversely, if $\mathsf{Gr}(\mathsf{RSM}, s)$ has a positive cycle, then it follows that there is an execution path $\pi_0\pi_1$ in $\mathsf{Gr}(\mathsf{RSM}, s)$ such that π_0 and π_1 are non-decreasing paths, π_1 begins and ends in the same node (possibly at higher stack height) and $\mathsf{wt}(\pi_1) > 0$. Hence, the path $\pi_0\pi_1^\omega$ is a valid execution path and satisfies $LimAvg(\mathsf{wt}(\pi_0\pi_1^\omega)) = \frac{\mathsf{wt}(\pi_1)}{|\pi_1|} > 0$, where $\pi_1^\omega = \pi_1 \cdot \pi_1 \cdot \pi_1 \ldots$ is the infinite concatenation of the finite execution path π_1. The desired result follows. □

Algorithm and analysis. Algorithm 14 solves the mean-payoff analysis problem for RSMs. The computation of the summary function requires $O(b)$ computations of the partial summary function s'_i (Line 4). Each such computation requires k runs of Bellman-Ford algorithm (Line 6), each run over a graph of n_i nodes and m_i edges (hence, each run takes $O(n_i \cdot m_i)$ time). In addition the computation requires k runs of all pairs maximum weight path (Floyd-Warshall) algorithm (Line 17). Each run is over a graph of $O(n_i)$ nodes (hence, each run takes $O(n_i^3)$ time and $O(n_i^2)$ space). Finally we detect positive cycles by running Bellman-Ford algorithm once over the summary graph (Line 19), which takes $O(n \cdot m)$ time and $O(n)$ space. Thus we obtain the following result.

Theorem 6.2 (**Improved algorithm).** *Algorithm 14 solves the mean-payoff analysis problem for RSMs in* $O\left((b \cdot (\sum_i n_i \cdot m_i)) + (\sum_i n_i^3) + n \cdot m\right)$ *time and* $O(\sum_i n_i^2)$ *space.*

Note that in the worst case the running time of Algorithm 14 is cubic and the space requirement is quadratic. The next example is an illustration of a run of Algorithm 14.

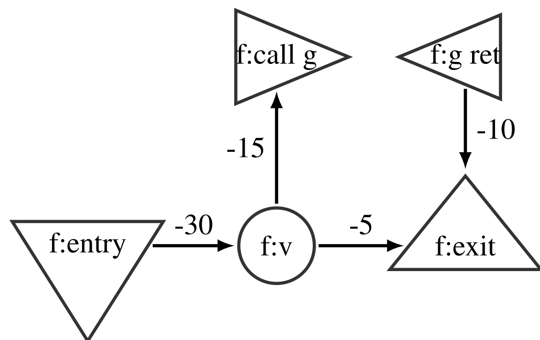
Example 6.5 (**Mean-payoff analysis).** Consider the RSM of Fig. 6.5 that consists of CSMs f and g and the entry of f is the initial entry of the program. We now describe the run of Algorithm 14 over the RSM. For simplicity, we denote the graph of f by F and the graph of g by G (and not by G_1 and G_2). Note that the number of call nodes is 3.

We first compute the summary function s' and the first step is to compute s'_0. We have s'_0(f:entry,f:exit) $= -35$, and s'_0(g:entry,g:exit) $= -25$.

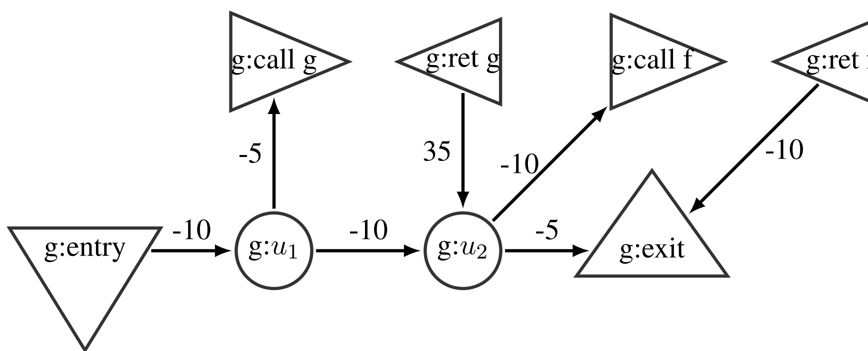
In order to compute s'_1(f:entry,f:exit) we construct a graph F^0 from F by adding a transition

Algorithm 14: Mean-payoff RSM Analysis

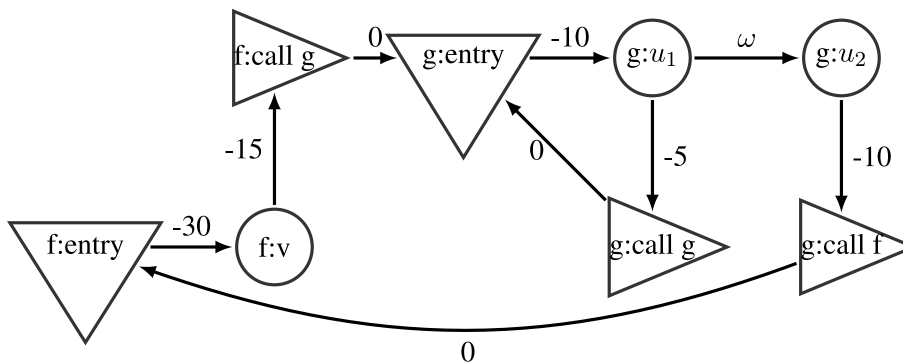
```
   Input:
   Output:
1  for i ← 1 to k do
   |  // Compute s'_0 by running the Bellman-Ford algorithm on G_i
2  |  s'_0(en_i, ex_i) ← BELLMAN-FORD(G_i)
3  end
4  ℓ ← 1 while True do
5  |  for i ← 1 to k do
6  |  |  Construct G_i^{ℓ−1} according to s'_{ℓ−1}
   |  |  // Compute s'_ℓ by running the Bellman-Ford algorithm over G_i^{ℓ−1}
7  |  |  s'_ℓ(en_i, ex_i) ← BELLMAN-FORD(G_i^{ℓ−1})
8  |  end
9  |  if s'_ℓ ≡ s'_{ℓ−1} then
10 |  |  break
11 |  if ℓ > b + 1 then
12 |  |  for i ← 1 to k do
13 |  |  |  if s'_ℓ(en_i, ex_i) > s'_{ℓ−1}(en_i, ex_i) then
14 |  |  |  |  s'_ℓ(en_i, ex_i) = ω
15 |  |  end
16 |  ℓ ← i + 1
17 end
   /* Compute s by executing Floyd-Warshall on each G_i wrt s'_ℓ    */
18 s ← FLOYD-WARSHALL(s'_ℓ)
19 Construct Gr(RSM, s) from s
   // Execute the Bellman-Ford algorithm on Gr(RSM, s)
20 BELLMAN-FORD(Gr(RSM, s))
21 if Gr(RSM, s) has a positive cycle then
22 |  return Yes
23 else
24 |  return No
25 end
```



(a) CSM f



(b) CSM g



(c) Summary graph of f and g

Figure 6.5: Two CSMs f and g and their summary graph.

from the node f:call g to the node f:ret g with weight s'_0(g:entry,g:exit) and find the maximum weight path from f:entry to f:exit in F^0. We get s'_1(f:entry,f:exit) $= -35$. In order to compute s'_1(g:entry,g:exit) we construct a graph G^0 from G by adding a transition from g:call g to g:ret g with weight s'_0(g:entry,g:exit) and a transition from g:call f to g:ret f with weight s'_0(f:entry,f:exit) and find the maximum weight path from g:entry to g:exit in G^0. We get s'_1(g:entry,g:exit) $= -10$.

Since $s'_1 \neq s'_0$, we continue to compute s'_2. We construct F^1 and G^1 in the same manner as we constructed F^0 and G^0 (but take the values of s'_1 instead of s'_0) and get s'_2(f:entry,f:exit) $= -35$, s'_2(g:entry,g:exit) $= 5$. For $i = 3$ we get s'_3(f:entry,f:exit) $= -35$, s'_3(g:entry,g:exit) $= 20$. For $i = 4$, s'_4(f:entry,f:exit) $= -35$, s'_4(g:entry,g:exit) $= 35$.

For $i = 5$ we get s'_5(f:entry,f:exit) $= -20$ and s'_5(g:entry,g:exit) $= 50$. Since $i > |\mathsf{Calls}| + 1$ and s'_5(f:entry,f:exit) $> s'_4$(f:entry,f:exit) and s'_5(g:entry,g:exit) $> s'_4$(g:entry,g:exit) we assign assign s'_5(f:entry,f:exit) $= \omega$ and s'_5(g:entry,g:exit) $= \omega$. In the sixth iteration we get a fix point (that is, $s'_6 \equiv s'_5$) and exit the loop block.

From F^5 and G^5 we compute the summary function s. For example s(g:entry,g:u_1) $= \omega$ and s(f:entry,f:v) $= -30$. Finally, we construct the summary graph (see Fig. 6.5c) and check whether a positive cycle exists. The cycle f:entry$\to$f:$v$$\to$f:call g$\to$g:entry$\to$g:$u_1$$\to$g:$u_2$$\to$g:call f$\to$f:entry contains an ω-edge and thus, it is a positive cycle. Hence algorithm returns YES.

6.4.2 An Efficient Algorithm for Mean-payoff Analysis

Here we further improve the algorithm for the mean-payoff analysis problem for RSMs, and the improvement depends on the fact that typically the number of entry, exit, call, and returns nodes is much smaller than the size of the RSMs. Formally, in most typical cases we have $|\mathsf{Ex} \cup \mathsf{Returns} \cup \mathsf{Calls} \cup \mathsf{En}| << n$. Let $X_i = \{\mathsf{ex}_i, \mathsf{en}_i\} \cup \mathsf{Returns}_i \cup \mathsf{Calls}_i$ and $X = \bigcup_i X_i$. We present an improvement that enables us to construct the summary function over graphs of size $O(|X_i|)$ (instead of graphs of size $O(n_i)$ of Section 6.4.1), and with at most $O(b)$ iterations. Hence, the algorithm in most typical cases will be much faster and require much smaller space.

Compact representation. The key idea for the improvement is to represent the CSMs in *compact form*. The compact form of a CSM A_i, denoted by $\mathsf{Comp}(A_i)$, is a graph which consists of the

entry, exit, call, and returns node of A_i. There is transition between every node in $\mathsf{Comp}(A_i)$, and the weight of each transition is the maximum weight execution path between the nodes with additional stack height 0 (and if there is no such execution path, then the weight is $-\infty$). Formally, $\mathsf{Comp}(A_i) = (V_i^c, E_i^c)$; where $V_i^c = X_i$; $E_i^c = V_i^c \times V_i^c$, and $\mathsf{wt}_i(v_1, v_2) = s_0(v_1, v_2)$ (where s_0 is the bounded-height summary function of height 0). If in $\mathsf{Comp}(A_i)$ there is a cycle with positive weight that is reachable from the entry node, then we say that A_i is a *positive mean-payoff witness*. The computation of the compact form for a CSM A_i requires $O(|X_i| \cdot n_i \cdot m_i)$ time and $O(n_i)$ space (running Bellman-Ford on each $\mathsf{Comp}(A_i)$), and thus the compact form for all CSMs can be computed in $O(\sum_i |X_i| \cdot n_i \cdot m_i)$ time and $O(\max_i n_i)$ space (note that the space can be reused).

Witness in summary graph of compact forms. After constructing the compact forms, we compute a summary function for $\mathsf{Comp}(A_1), \ldots, \mathsf{Comp}(A_k)$, and a corresponding summary graph. We say that there is an execution path with positive mean-payoff iff there exists a positive cycle in the summary graph or there exists an execution path to the entry node of a positive mean-payoff witness. The correctness of the algorithm relies on the next lemma.

Lemma 6.5. *Let* $\mathsf{RSM} = \langle A_1, \ldots, A_k \rangle$ *be an RSM, let* $\mathsf{Gr}(\mathsf{RSM}, s)$ *be its summary graph and let* $\mathsf{Comp}(\mathsf{Gr}(\mathsf{RSM}, s))$ *be the summary graph that is formed by* $\mathsf{Comp}(A_1), \ldots, \mathsf{Comp}(A_k)$. *The following assertions are equivalent:*

1. $\mathsf{Gr}(\mathsf{RSM}, s)$ *has a (reachable) positive cycle.*

2. $\mathsf{Comp}(\mathsf{Gr}(\mathsf{RSM}, s))$ *has a (reachable) positive cycle or a positive mean-payoff witness.*

Proof. We first observe that every node in $\mathsf{Comp}(\mathsf{Gr}(\mathsf{RSM}, s))$ exists also in $\mathsf{Gr}(\mathsf{RSM}, s)$ and that the weight of every execution path in $\mathsf{Comp}(\mathsf{Gr}(\mathsf{RSM}, s))$ has the same weight for the corresponding execution path in $\mathsf{Gr}(\mathsf{RSM}, s)$ (this can be easily shown by induction over the number of iterations that are needed to obtain a fix point in the bounded-height summary function). Hence, if $\mathsf{Gr}(\mathsf{RSM}, s)$ has a positive simple cycle that contains a call node c for $c \in \mathsf{Calls}$, then c is a node also in $\mathsf{Comp}(\mathsf{Gr}(\mathsf{RSM}, s))$ and by the observation above, c is part of a positive cycle in $\mathsf{Comp}(\mathsf{Gr}(\mathsf{RSM}, s))$. Therefore, $\mathsf{Comp}(\mathsf{Gr}(\mathsf{RSM}, s))$ has a positive cycle. Otherwise, $\mathsf{Gr}(\mathsf{RSM}, s)$ has a positive simple cycle that does not contain a call node. Hence, there is a CSM A_i with a reachable positive simple cycle that has additional stack height 0. Therefore

$\mathsf{Comp}(\mathsf{Gr}(\mathsf{RSM}, s))$ has a positive cycle or a positive mean-payoff witness. This concludes the proof of one direction and the proof for the converse direction is trivial. □

The above lemma establishes the correctness of the computation on compact form graphs, and gives us the following result.

Theorem 6.3 (Efficient algorithm). *The mean-payoff analysis problem for RSMs can be solved in $O\left((b \cdot (\sum_i |X_i|^3)) + |X|^3\right)$ time and $O(\sum_i |X_i|^2 + \max_i n_i)$ space, where $X_i = \{\mathsf{ex}_i, \mathsf{en}_i\} \cup \mathsf{Returns}_i \cup \mathsf{Calls}_i$ and $X = \bigcup_i X_i$.*

6.4.3 An Improved Algorithm for Mean-payoff Analysis on RSMs of Constant Treewidth

Here we sketch an improved solution the mean-payoff analysis problem for RSMs of constant treewidth. Note that the while loop in Line 4 of Algorithm 14 is similar to Algorithm 6 from Section 4.2 for computing bounded stack-height distances. The modification then consists of the following steps:

1. First, we execute Algorithm 6 for stack height $h = b + 1$.
2. Then, we perform an all-pairs distance computation in each control-flow graph G_i of CSM A_i, by breaking it down to n_i single-source distance queries using the linear single-source query time algorithm from Section 3.2.
3. Finally, we proceed from Line 18 of Algorithm 14 to detect a positive cycle on the summary graph.

The time required for Algorithm 6 is $O(n)$ for the preprocessing, plus $O(b \cdot \log n)$ time for updating the weight of call-to-return edges for each of the $O(b)$ iterations. The all-pairs distance computation of step 2 requires $O(\sum_i n_i^2)$ time. Finally, Algorithm 14 requires $O(n \cdot m)$ time for detecting a positive cycle on the summary graph. The space required is dominated by the $O(\sum_i n_i^2)$ cost for storing the all-pairs distances in each control-flow graph G_i.

Theorem 6.4 (Algorithm for constant-treewidth RSMs). *Given an RSM* RSM *of constant treewidth, the mean-payoff analysis problem for* RSM *can be solved in $O\left(b \cdot \sum_i b \cdot \log n_i + n \cdot m\right)$ time and $O(\sum_i n_i^2)$ space.*

6.4.4 Reduction: Ratio Analysis to Mean-payoff Analysis

We now establish a linear reduction of the ratio analysis problem to the mean-payoff analysis problem. Given an RSM RSM with labeling of good, bad, and neutral events, a positive integer weight function w, and rational threshold $\lambda > 0$, the reduction of the ratio analysis problem to the mean-payoff analysis problem is as follows. We consider an RSM RSM' with weight function w_λ for the mean-payoff objective defined as follows: for a transition e we have

$$\mathsf{wt}_\lambda(e) = \begin{cases} \mathsf{wt}(e) & \text{if } e \text{ is labelled with } \mathit{good} \\ -\lambda \cdot \mathsf{wt}(e) & \text{if } e \text{ is labelled with } \mathit{bad} \\ 0 & \text{otherwise (if } e \text{ is labelled with } \mathit{neutral}) \end{cases}$$

The next lemma establishes the correctness of the reduction.

Lemma 6.6. *Given an RSM* RSM *with labeling of good, bad, and neutral events, a positive integer weight function* w*, and rational threshold* $\lambda > 0$*, let* RSM' *be the RSM with weight function* w_λ*. There exists an execution path* π *in* RSM *with* $\mathit{LimRat}(\mathsf{wt}(\pi)) > \lambda$ *iff there exists an execution path* π *in* RSM' *with* $\mathit{LimAvg}(w_\lambda(\pi)) > 0$*.*

Proof. Observe that by the definition of w_λ we have that for every $\epsilon > 0$ and a finite execution path π:

$$\mathit{Rat}(\mathsf{wt}(\pi)) \geqslant \lambda + \epsilon \text{ iff } \mathit{Avg}(w_\lambda(\pi)) \geqslant \epsilon.$$

LimAvg *implies* LimRat. Consider an infinite execution path π. If $\mathit{LimAvg}(\mathsf{wt}_\lambda(\pi)) > 0$, then by definition there is an $\epsilon > 0$ and $m_0 \in \mathbb{N}$ such that for every $m \geqslant m_0$ we have $\mathit{Avg}(\mathsf{wt}_\lambda(\pi[1, m])) \geqslant \epsilon$. Hence by the above observation there exist $\epsilon > 0$ and $m_0 \in \mathbb{N}$ such that for every $m \geqslant m_0$ we have $\mathit{Rat}(\mathsf{wt}(\pi[1, m])) \geqslant \lambda + \epsilon$. Moreover, it follows that in π there are infinitely many edges with positive weights (according to w_λ) and hence π has infinitely many good events. Hence we get that $\mathit{LimRat}(\mathsf{wt}(\pi)) > \lambda$.

LimRat *implies* LimAvg. The proof for the converse direction is less trivial and relies on properties of RSMs that we established. Suppose that there is an infinite execution path π with $\mathit{LimRat}(\mathsf{wt}(\pi)) > \lambda$. We note that every infinite execution path has infinitely many local minima and let $\mathcal{C}_1, \mathcal{C}_2, \ldots$ be an infinite sequence of local minima in π. We have the following facts:

1. The segment between every two such configurations $\mathcal{C}_i$ and $\mathcal{C}_j$ for $i < j$ is a non-decreasing execution path (since each $\mathcal{C}_i$ is a local minimum).

2. There is a configuration $\mathcal{C}_p$ with a node u_p such that for every $\ell \in \mathbb{N}$ there exists a configuration $\mathcal{C}_j$ (for $p < j$) with node u_j such that the segment between $\mathcal{C}_p$ and $\mathcal{C}_j$ is of length greater than ℓ and $u_p = u_j$, i.e., $\mathcal{C}_p$ and $\mathcal{C}_j$ have the same node (follows from the pigeonhole principle, since the number of local minima is infinite and we have finitely many nodes).

We claim that there exists a non-decreasing finite execution path π^* that is a segment of π, which begins at $\mathcal{C}_p$ and ends at a configuration that has the same node (possibly at different stack height), and we have $\mathit{Rat}(\mathsf{wt}(\pi^*)) > \lambda$. Assume towards the contradiction of the claim that for every configuration $\mathcal{C}_j$ with $u_p = u_j$, with $p < j$, we have $\mathit{Rat}(\mathsf{wt}(\pi^*)) \leqslant \lambda$. If π has only finitely many good or bad events, then $\mathit{LimRat}(\mathsf{wt}(\pi)) = 0 < \lambda$. Else we consider the following sequence of prefixes of π: π_0 is the prefix of π that ends in $\mathcal{C}_p$; and π_i is the segment that starts in $\mathcal{C}_p$ and ends in the i-th local minimum after $\mathcal{C}_p$ that has the same node u_p. Then we have

$$\mathit{Rat}(\pi_0 \cdot \pi_i) \leqslant \lambda + \frac{\mathsf{wt}(\pi_0)}{i};$$

since the length of π_i is at least i. Hence, by definition $\mathit{LimRat}(\mathsf{wt}(\pi)) \leqslant \lambda$, which establishes the desired contradiction. Thus we have that $\mathit{Rat}(\mathsf{wt}(\pi^*)) > \lambda$ and therefore $\mathit{Avg}(\mathsf{wt}_\lambda(\pi^*)) > 0$ and the execution path $\pi' = \pi_0(\pi^*)^\omega$ is a valid path (since π^* is a non-decreasing execution path that begins and ends in the same node) with $\mathit{Avg}(\mathsf{wt}_\lambda(\pi')) > 0$. The desired result follows. □

Remark 6.3. Note that in our reduction from ratio analysis to mean-payoff analysis we do not change the RSM, but only change the weight function. Thus our algorithms from Theorems 6.2 and 6.3 can also solve the ratio analysis problem for RSMs. Moreover, our proof of Lemma 6.6 shows that for all execution paths π, if we have $\mathit{LimAvg}(\mathsf{wt}_\lambda(\pi)) > 0$, then we also have $\mathit{LimRat}(\mathsf{wt}(\pi)) > \lambda$, i.e., any witness for the mean-payoff analysis is also a witness for the ratio analysis.

6.5 Experimental Results: Three Case Studies

In this section we present our experimental results on three case studies described in Section 6.3.1 and Section 6.3.2. We run our case studies on several benchmarks in Java, including DaCapo

2009 benchmarks [Blackburn *et al.*, 2006], and we use [Bodden *et al.*, 2011; Lhoták and Hendren, 2003] to assist Soot for the construction of the control-flow graphs. First we present some optimizations that proved useful for speed-up in the benchmarks.

6.5.1 Optimizations for the Case Studies

We present four optimizations for the case studies: the first two are general, and the last two are specific to our case studies. All optimizations presented here have provided significant speedups, and are orthogonal to each other (i.e., they can be applied independently, and each optimization results in approximately in a speedup of the same order independent of the other optimizations that are present).

Faster computation of stack height bounded summary function. We note that if CSM A_ℓ invokes only CSMs $A_{j_1}, \dots, A_{j_k}$, and $s'_\ell(\mathsf{en}_{j_h}, \mathsf{ex}_{j_h}) = s'_{\ell-1}(\mathsf{en}_{j_h}, \mathsf{ex}_{j_h})$ for all $h \in \{1, \dots, k\}$, then $s'_\ell(\mathsf{en}_i, \mathsf{ex}_i) = s'_{\ell-1}(\mathsf{en}_i, \mathsf{ex}_i)$. Hence, when computing s'_ℓ, we maintain a set $\mathcal{L}^\ell = \{i \mid s'_\ell(\mathsf{en}_i, \mathsf{ex}_i) > s'_{\ell-1}(\mathsf{en}_i, \mathsf{ex}_i)\}$, and in the next iteration we run Bellman-Ford algorithm only for the CSMs that invoke CSMs from $\mathcal{L}^\ell$.

Reducing the number of iterations for fix point. We now present an optimization that allows us to reduce the number of bounded-height summary functions from $O(b)$ to a practically constant number. We note that the $O(b)$ theoretical bound is tight. However, only pathological cases can reach even a fraction of this bound. We note that in typical programs the average nesting of function calls is practically constant (say 10). So if we do not get a fix point after 10 iterations (i.e., $s'_{11} > s'_{10}$), then it is probably because there is a recursive call with positive weight. If this is the case, then if we build the summary graph according to s_{11}, we will get a positive cycle in the summary graph, that is, we will get a witness for a path with a positive mean-payoff, and we can stop the computation (since by definition $s \geqslant s_{11}$, we get that this witness is valid). Hence, our optimized algorithm is to compute the bounded-height summary function s'_ℓ and if $s'_\ell > s'_{\ell-1}$ and $\ell = 10, 20, 30, \dots$, then we construct the summary graph and look for a witness path. If a path is found, then we are done. Otherwise we continue and compute $s'_{\ell+1}$.

Removing redundant CSMs. Consider an RSM $\mathsf{RSM} = \langle A_1, \dots, A_k \rangle$ in which every node is

reachable from the program entry (the entry node of the main method). We say that CSM A_i is *non-redundant* if (i) the CSM has non-zero weight transitions (good or bad events); or (ii) it invokes a non-redundant CSM, and is called *redundant* otherwise. Let A_i be a redundant CSM. For every path π that contains a transition to en_i (an invocation of A_i), the segment of π between that transition and the first transition to ex_i contains only neutral transitions. Because all nodes of RSM are reachable, we can safely replace each call node that invokes A_i by an internal node that leads to the corresponding return node, and label it as a neutral event. Our optimization then consists of removing redundant CSMs, as follows:

1. First, we perform a single-source interprocedural reachability from the program entry, which requires linear time ([Reps *et al.*, 1995a]), and discard all non-reachable nodes in all CSMs.

2. Then, we perform a backwards reachability computation on the call graph of RSM, starting from the set of all CSMs that contain non-zero weight transitions. All detected redundant CSMs are discarded, and calls to them are replaced according to the above description.

Hence, when computing the bounded-height summary function, the size of the graph is smaller and the Bellman-Ford algorithm takes less time. Additionally, the number of calls b decreases, which reduces the number of iterations required in the main loop of Algorithm 14. In the first case study, typically more than half of the methods are eliminated in this process.

Incremental computation of summary functions. We present the final optimization which is relevant for our second and third case studies. Let RSM^1 be an RSM and let RSM^2 be an RSM that is obtained from RSM^1 only by increasing some of the transitions weights. Let s^1 be the summary function of RSM^1. Then we can compute the summary function of RSM^2 by setting $s'^2_\ell \equiv s^1$ and by computing s'^2_ℓ from $s'^2_{\ell+1}$ in the usual way. The correctness is almost trivial. Since the weights of RSM^2 are at least large as the weights of RSM^1, we get that if we conceptually add a transition (n_1, n_2) with weight $s^1(n_1, n_2)$ for every two nodes (in the same CSM) in RSM^2, then the weights of the paths with the maximal weight in RSM^2 remain the same. By assigning $s'^2_0 = s^1$ we only add such conceptual transitions. Hence, the correctness follows. We now describe how this optimization speed up the analysis of the second case study. In the static profiling for function frequencies, we need to build a summary graph for every function f, and then run the mean-payoff analysis for every such graph. Given this optimization, we

can first compute (only once) a summary graph for the case that all method invocations are bad events. We denote this RSM by RSM^* and the corresponding summary function by s^*. Note that in RSM^* all weights are negative, and the mean-payoff analysis answer is trivially NO. But still the summary function computation, which computes the quantitative information about the maximum weight context-free paths, provides useful information and saves recomputation. To determine the frequency of f we assign weights to RSM and get RSM^f, and the difference between RSM^f and RSM^* is only in the weight that is assigned to the invocation of f. We then compute the summary function s^f for RSM^f by first assigning $s_0'^f = s^*$. In practical cases, programs can have thousands of methods, but only small portion of them will have a path to f. So along with the previous optimizations we get that only few Bellman-Ford runs are required to compute s^f. Overall, the computation of s^* is expensive, and may take several minutes for a large program, but it is done only once, and then the computation of each s^f is much faster.

6.5.2 Container Analysis

Technical details about experimental results. We discuss a few relevant details about our experiments and results.

- We use the points-to analysis tool of [Sridharan and Bodík, 2006b]. This tool provides interprocedural on demand analysis for a may-alias relationship of two variables. We say that a variable may point to an allocated container if it may-alias the container, and a variable must point to an allocated container if it may-alias only one allocated container [2].
- For the underutilized containers the threshold is 1, and for the analysis of overpopulated containers we set a threshold of 0.1 for our experimental results. That is, if the ratio between the number of added elements to the number of lookup operations is more than 10, then the container is overpopulated.

Experimental results. Our experimental results on the benchmarks are reported in Table 6.1. In the table, **# M** and **# CO** represent the number of methods and containers that are reachable from the main entry of the program, respectively; **# OP** and **# UC** represent the number of

[2] In general, a singleton may-alias set does not establish must-alias relationship. However, this rule provides a good approximation, as it is very rare that a variable does not point to any object in all of the program executions.

overpopulated and underutilized containers discovered by our tool, respectively; and **TA(s)** and **TQ(s)** represent the time required for alias analysis and the time required for the quantitative analysis of RSMs (in seconds), respectively; and the entries of the respective columns represent the time for overpopulated/underutilized container analysis. We now highlight some interesting aspects of our experimental results. First, our approach for container analysis discovers containers that are overpopulated or underutilized, while maintaining soundness. Second, the cases that we identify reveal useful information for optimization, for example, in the first (batik-rasterizer) and the second (batik-svgpp) benchmarks we identify containers that always have a small bounded number of elements.

Benchmark	# M	# CO	# OP	# UC	TA(s)	TQ(s)
batik-rasterizer	21433	9	1	2	124/125	144/143
batik-svgpp	7859	3	0	3	20/20	14/13
mrt	9798	10	1	0	70/13	41/59
java_cup	8173	10	0	0	19/19	25/22
xalan	8729	6	3	2	5/5	41/43
polyglot	8068	8	2	2	0/0	17/17
antlr	8607	15	5	2	11/12	25/24
jflex	21852	43	3	6	2473/2614	178/210
avrora	13331	75	9	9	145/141	111/113
muffin	22503	50	3	5	2500/157	352/173
bloat06	10675	211	32	14	399/250	2241/2165
eclipse06	9335	74	8	4	37/22	222/164
jython06	12210	66	9	5	154/68	13593/8376

Table 6.1: Experimental results for container usage analysis

With our approach we were able to fully analyze all the containers in all benchmarks. Below we present example snippets of code from the benchmarks where our analysis gives different results from the analysis of the existing tool of [Xu and Rountev, 2010] with which we have compared our results. The example in Fig. 6.7 shows that handling DELETE operations leads to more refined analysis: in the example, if DELETE operations are not handled, then the misuse is not detected. The example in Fig. 6.8 shows that the proper utilization of containers might depend on the recursive calls. Finally, the example in Fig. 6.9 illustrates that the proper use of

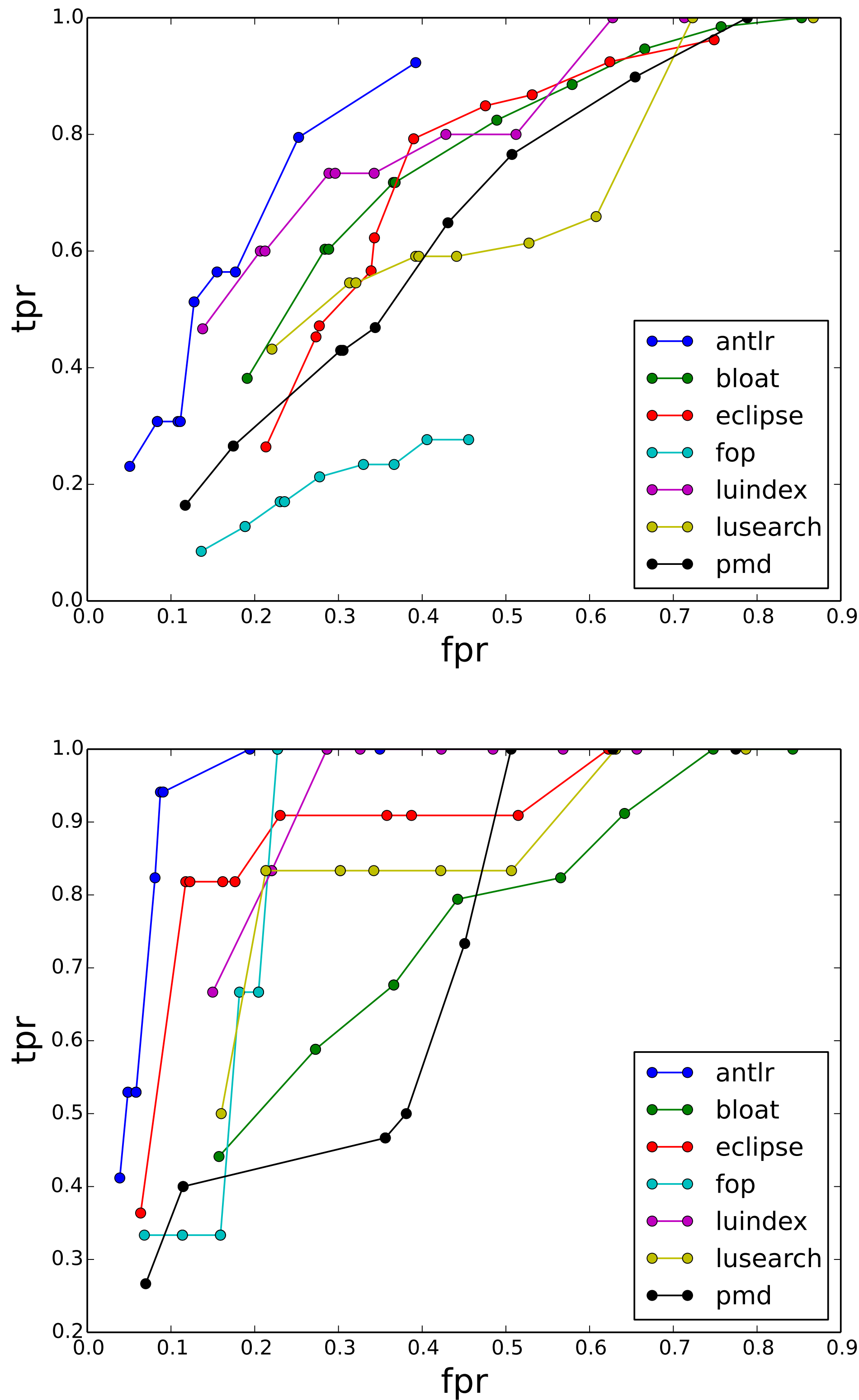


Figure 6.6: The ROC curves for the analysis of frequently invoked methods. The left plot shows the results when all methods are analyzed. The right plot shows the results when only the active methods are analyzed. The different dots of each curve correspond to different threshold values λ.

Benchmark	# M	# I	T
antlr	768	326	1.2
bloat	2576	676	30.8
eclipse	1056	215	2.3
fop	429	47	0.4
luindex	567	239	0.7
lusearch	842	237	2.5
pmd	2547	589	11.5

Table 6.2: Experimental results for frequency of functions

containers can be outside its allocation site, and thus detecting proper use requires a quantitative interprocedural framework, such as the one proposed here.

6.5.3 Static Profiling: Frequency of Function Calls

We have examined ten thresholds, namely $\frac{1}{30}, \frac{2}{30}, \frac{3}{30}, \ldots, \frac{10}{30}$, and for each threshold λ we say that a method is *statically hot* if it is λ-hot (according to the definition in Section 6.3.2). We compared the results to dynamic profiling from the DaCapo benchmarks [Blackburn *et al.*, 2006]. In the dynamic profiling we define the top 5% of the most frequently invoked functions as *dynamically hot*. For example, if a program has 1000 functions, and in the benchmark 500 functions were invoked at least once, then the 25 most frequently invoked functions are dynamically hot. We note that theoretically speaking, the definitions of dynamic and static hotness are incomparable (basically for any λ), but our experimental results show a good correlation between the two notions. To illustrate the correlation we treat our static analysis as a *classifier* of hot methods, and the *specificity* and *sensitivity* of the classifier are controlled by the threshold λ. The sensitivity of a classifier is measured by the *true positive rate* (tpr), which is

$$\frac{\text{\#dynamically hot methods that are reported as statically hot}}{\text{\#dynamically hot methods}}$$

The specificity is determined by the *false positive rate* (fpr):

$$\frac{\text{\#non-dynamically hot methods that are reported as statically hot}}{\text{\#methods}}$$

For high values of λ, the classifier is expected to capture only dynamically hot methods (but it will miss most of the dynamically hot methods), and thus it will have very high fpr but very

```java
public void cleared() {
    ...
    if (list != null){
        ...
    }
    else{
        Object o = elementsById.remove(id);
        if (o != this) // oops not us!
            elementsById.put(id, o);
    }
}
```

```java
public void run() {
    while(true) {
        ...
        if (...) {
            ...
            rc.cleared();
        }
        ...
    }
    ...
}
```

Figure 6.7: An example from benchmark batik. The method **run** invokes **cleared** in a loop, and in every invocation, one element of elementsById is removed and one element is added. Thus in this loop the total number of elements in elementsById is bounded.

low tpr. For very low values of λ the classifier will report most of the methods as hot, so most of the hot methods will be reported as hot, and we will have very high tpr but very low fpr. A fundamental metric for classifier evaluation is a *receiver operating characteristic (ROC) graph*. A ROC graph is a plot with the false positive rate on the X axis and the true positive rate on the Y axis. The point (0,1) is the perfect classifier, and the area beneath an ROC curve can be used as a measure of accuracy of the classifier.

In our experimental evaluation we only considered application functions (and not library functions), and the results are presented in Table 6.2. In the table, **# M** represents the number of application methods (that are reachable from the main entry of the program), **# I** represents the number of application methods that were actually invoked in the execution of the benchmark, **T** represents the average running time for the static analysis of a single method (i.e., to check whether a single method is λ-hot for a fixed λ) (in seconds). For each λ we present the tpr and fpr values of the classifications. We present an evaluation for two cases. In the first case we statically analyze all the methods and calculate the tpr and fpr accordingly. In the second case we consider only the *active methods*, namely, the methods that were invoked at least once in the program, and we remove all the other methods from the program control flow graph. This simulates a typical case where the programmer has prior knowledge on methods that are definitely not hot and can instruct the static analysis to ignore them. The ROC curves are presented in Fig. 6.6, where the

```
1  void addCNAME(CNAMERecord cname) {
2      if (backtrace == null)
3          backtrace = new Vector();
4      backtrace.insertElementAt(cname, 0);
5  }
6
7  public SetResponse findRecords(Name name, short type) {
8      ...
9      if (type != Type.CNAME && type != Type.ANY && rrset.getType() == Type.CNAME)
10     {
11         zr = findRecords(cname.getTarget(), type);
12         zr.addCNAME(cname);
13         ...
14     }
15     ...
16     return zr;
17 }
```

Figure 6.8: An example from benchmark muffin. The method **findRecords** has a recursive call, and method **addCNAME** adds an element to vector backtrace. A path with recursion depth n adds n elements to backtrace. Hence, backtrace may have unbounded number of elements and it is not underutilized.

```
1  Hashtable cgi(Request request)
2  {
3      Hashtable attrs = new Hashtable(13);
4      ...
5      if (query != null)
6      {
7          StringTokenizer st = new StringTokenizer(decode(query), "&");
8          while (st.hasMoreTokens())
9          {
10             ...
11             attrs.put(key, value);
12         }
13     }
14     ...
15     return attrs;
16 }
17
18 public Reply recvReply(Request request)
19 {
20     ...
21     else if (request.getPath().equals("/admin/set"))
22     {
23         Hashtable attrs = cgi(request);
24         ...
25         for (int i = 0; i < enabled.size(); i++)
26         {
27             ...
28             prefs.put(key, (String) attrs.get(key));
29         }
30         ...
31     }
32     ...
33     return reply;
34 }
```

Figure 6.9: An example from benchmark muffin. The method **cgi** allocates the container attrs and potentially adds it many elements. The method **recvReply** performs a **get** operation over attrs in a loop. Since we analyze not only the operations that are nested in the allocation site, we detect that attrs is not overpopulated.

most left points on the graph are for $\lambda = \frac{10}{30}$ and the fpr and tpr increases as λ decreases (until it finally reaches $\frac{1}{30}$). In general, for most of the programs the static analysis gives useful and quite accurate information. Specifically, the threshold $\lambda = \frac{7}{30}$ captures more than half of the hot methods for most benchmarks (i.e., except fop and antlr) and with a fpr less than 0.3 which means that if a method was statically reported as not hot, then with probability 0.7 it is really not hot. We note that the analysis over fop gives quite poor results because only 10% of the methods were active. However, when we analyzed only the active methods we get better results for fop, see the right hand graph in Fig. 6.6. When we only consider the active methods, the threshold $\lambda = \frac{9}{30}$ captures most of the dynamically hot methods while maintaining a fpr less than 0.1 (for most programs).

6.5.4 Static Profiling: Frequency of Class Method Calls

We have experimented with the potential of our framework to statically predict collections of methods, where the set will be frequently invoked. As discussed in Section 6.3.3, when such sets of methods are software libraries, good predictions can assist static vs dynamic library linking. As our prototype implementation has focused on Java programs, where static linking does not apply (the JVM dynamically loads classes as they are referenced), we have grouped methods based on the class they belong to. Then we applied the modeling of Section 6.3.3, where a Java class is used as a substitute for a library.

We examined the thresholds $\frac{1}{30}, \frac{2}{30}, \frac{3}{30}, ..., \frac{16}{30}$. Similarly to the case of hot methods, for every threshold λ we say that a class is *statically hot* if the collection of its methods is λ-hot (according to the definition in Section 6.3.3). Again, we compared the results to dynamic profiling from the DaCapo benchmarks [Blackburn *et al.*, 2006]. In the dynamic profiling we define the top 5% of the most frequently invoked classes as *dynamically hot*. Table 6.3 presents a summary of the analysis, where **# C** represents the number of classes (that are reachable from the main entry of the program), **# I** represents the number of classes that were actually invoked in the execution of the benchmark, **T** represents the average running time for the static analysis of a single class (i.e., to check whether the collection of methods of a whole class is λ-hot for a fixed λ) (in seconds).

Fig. 6.10 shows the corresponding ROC curves. The leftmost point of each curve corresponds to $\lambda = \frac{16}{30}$, and the fpr and tpr increase as we move to the right. We see that the static analysis

Benchmark	# C	# I	T
antlr	103	79	8.3
bloat	284	168	29.7
eclipse	187	69	4.2
fop	197	24	0.2
luindex	101	64	3.1
lusearch	164	69	9.0
pmd	379	130	10.5

Table 6.3: Experimental results for frequency of classes

provides useful information, as for most benchmarks it achieves relatively large tpr while keeping the fpr reasonably low. The static analysis performs poorly in the case of fop, for which we have seen in Section 6.5.3 that the overlap between methods discovered in the static and dynamic analysis is very small.

Hot methods vs hot collections of methods. We clarify some differences between the concepts of hot methods and hot collections of methods. First, we note that frequently invoked collections of methods cannot be detected simply by detecting frequently invoked methods, and a separate analysis is required. Fig. 6.11 illustrates the difference on a small example. We have used a larger threshold range in our analysis of hot classes, and thus groups of functions are in general expected to meet larger thresholds than individual functions. In our experimental results, the information computed is not of direct use, as all classes are loaded on runtime by the JVM. However, it is demonstrated that our framework can statically analyze programs and provide meaningful suggestions as to which groups of methods (e.g., classes, libraries) will be frequently used.

Remarks. We run the experiments on a single thread Intel Pentium 3.80GHz. For Table 6.1 results, the alias analysis did not complete for some benchmarks (e.g. fop, pmd). In Table 6.2 we only show benchmarks for which we managed to obtain dynamic profiles. For a few benchmarks (e.g. jython) the quantitative analysis took too long for the entire benchmark. In such cases, our tool could be used to focus on specific functions.

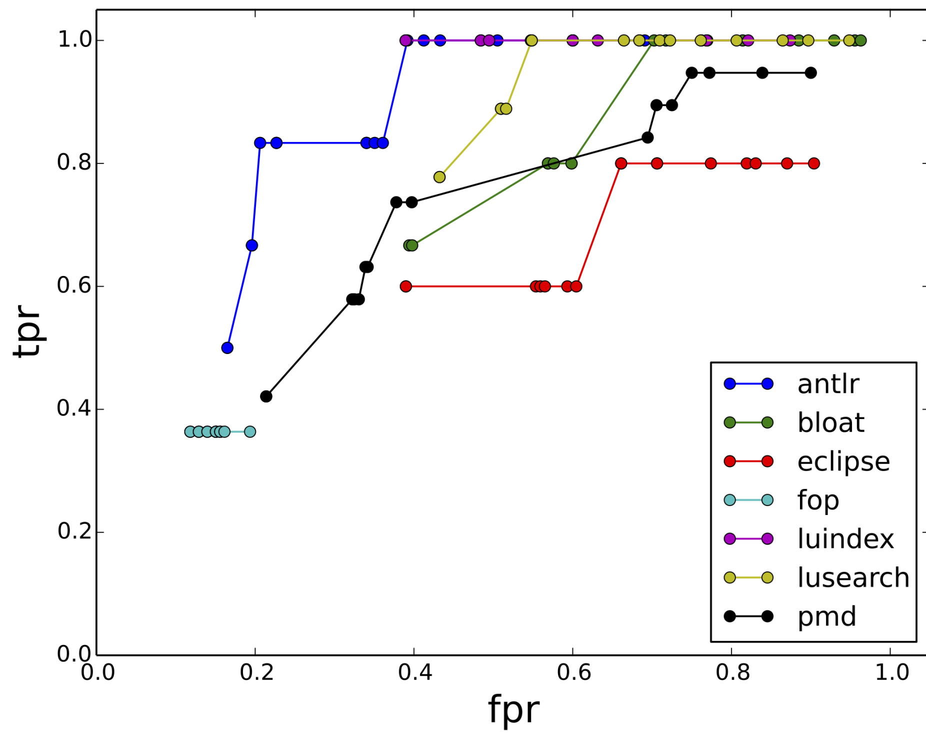


Figure 6.10: The ROC curves for the analysis of frequently invoked classes. The different dots of each curve correspond to different threshold values λ.

```
1  class A:
2  {
3      public void f1()
4      {
5          ...
6      }
7      public void f2()
8      {
9          ...
10     }
11     public void f3()
12     {
13         ...
14     }
15 }
16
17 void main()
18 {
19     A a = new A();
20     while(1)
21     {
22         a.f1();
23         a.f2();
24         a.f3();
25     }
26 }
```

Figure 6.11: Illustration of the difference between hot methods and hot collections of methods. Here, the collection of the methods of class A is λ-hot for any threshold $\lambda \leqslant 1$. However, each of the methods $f1()$, $f2()$ and $f3()$ of A are at most $\frac{1}{3}$-hot. Hence determining frequently invoked collections of methods requires separate analysis, and cannot rely on simply detecting frequently invoked methods.

7 Semiring Distances on Concurrent Systems of Constant-treewidth Components

7.1 Introduction

In this section we consider concurrent finite-state systems where each component is a constant-treewidth graph, and the algorithmic question is to determine the semiring distance between pairs of nodes in the system. Our main contributions are algorithms which significantly improve the worst-case running time of the existing algorithms. We establish conditional optimality results for some of our algorithms in the sense that they cannot be improved without achieving major breakthroughs in the algorithmic study of graph problems. Finally, we provide a prototype implementation of our algorithms which significantly outperforms the existing algorithmic methods on several benchmarks.

Concurrency and algorithmic approaches. The analysis of concurrent systems is one of the fundamental problems in computer science in general, and programming languages in particular. A finite-state concurrent system consists of several components, each of which is a finite-state graph, and the whole system is a composition of the components. Since errors in concurrent systems are hard to reproduce by simulations due to combinatorial explosion in the number of interleavings, formal methods are necessary to analyze such systems. In the heart of the formal approaches are graph algorithms, which provide the basic search procedures for the problem. The basic graph algorithmic approach is to construct the product graph (i.e., the product of the

component systems) and then apply the best-known graph algorithms on the product graph. While there are many practical approaches for the analysis of concurrent systems, a fundamental theoretical question is whether special properties of graphs that arise in analysis of programs can be exploited to develop asymptotically faster algorithms as compared to the basic approach.

The algorithmic problem. In graph theoretic parlance, graph algorithms typically consider two types of queries: (i) a *pair query* given nodes u and v (called (u, v)-pair query) asks for the semiring distance from u to v; and (ii) a *single-source* query given a node u asks for the answer of (u, v)-pair queries for all nodes v. In the context of concurrency, in addition to the classic pair and single-source queries, we also consider *partial* queries. Given a concurrent system with k components, a node in the product graph is a tuple of k component nodes. A *partial* node $\overline{u}$ in the product only specifies nodes of a nonempty strict subset of all the components. Our work also considers partial pair and partial single-source queries, where the query nodes are partial nodes. Queries on partial nodes are very natural, as they capture properties between local locations in a component, that are shaped by global paths in the whole concurrent system. For example, constant propagation and dead code elimination are local properties in a program, but their analysis requires analyzing the concurrent system as a whole.

	Preprocess		**Query time**			
	Time	Space	Single-source	Pair	Partial single-source	Partial pair
Previous results [Lehmann, 1977; Floyd, 1962] [Warshall, 1962; Kleene, 1956]	$O(n^6)$	$O(n^4)$	$O(n^2)$	$O(1)$	$O(n^2)$	$O(1)$
Corollary 7.2 ($\epsilon > 0$)	$O(n^3)$	$O(n^{2+\epsilon})$	$O(n^{2+\epsilon})$	$O(n^2)$	$O(n^{2+\epsilon})$	$O(n^2)$
Theorem 7.2 ($\epsilon > 0$)	$O(n^{3+\epsilon})$	$O(n^3)$	$O(n^{2+\epsilon})$	$O(n)$	$O(n^2)$	$O(1)$
Corollary 7.3 ($\epsilon > 0$)	$O(n^{4+\epsilon})$	$O(n^4)$	$O(n^2)$	$O(1)$	$O(n^2)$	$O(1)$

Table 7.1: The algorithmic complexity for computing algebraic path queries wrt a closed, complete semiring on a concurrent graph G which is the product of two constant-treewidth graphs G_1, G_2, with n nodes each.

Previous results. In this section we consider finite-state concurrent systems, where each component graph has constant treewidth, and the trace properties are specified as semiring distances.

Our framework can model a large class of problems: typically the control-flow graphs of programs have constant treewidth [Thorup, 1998; Gustedt *et al.*, 2002; Burgstaller *et al.*, 2004], and if there is a constant number of synchronization variables with constant-size domains, then each component graph has constant treewidth. Note that this imposes little practical restrictions, as typically synchronization variables, such as locks, mutexes and condition variables have small (even binary) domains (e.g. locked/unlocked state). The best-known graph algorithm for the algebraic path problem is the classic Warshall-Floyd-Kleene [Lehmann, 1977; Floyd, 1962; Warshall, 1962; Kleene, 1956] style dynamic programming, which requires cubic time. Two well-known special cases of the algebraic paths problem are (i) computing the shortest path from a source to a target node in a weighted graph, and (ii) computing the regular expression from a source to a target node in an automaton whose edges are labeled with letters from a finite alphabet. In the first case, the best-known algorithm is the Bellman-Ford algorithm with time complexity $O(n \cdot m)$. In the second case, the well-known construction of Kleene's [Kleene, 1956] theorem requires cubic time. The only existing algorithmic approach for the problem we consider is to first construct the product graph (thus if each component graph has size n, and there are k components, then the product graph has size $O(n^k)$), and then apply the best-known graph algorithm (thus the overall time complexity is $O(n^{3 \cdot k})$ for general semiring distances). Hence for the important special case of two components we obtain a hexic-time (i.e., $O(n^6)$) algorithm. Moreover, the current best-known algorithms for the algebraic path problem event for one pair query (or one single-source query) computes the entire transitive closure. Hence the existing approach does not allow a tradeoff of preprocessing and query as even for one query the entire transitive closure is computed.

Our contributions. Our main contributions are improved algorithmic upper bounds, proving several optimality results of our algorithms, and experimental results. Below all the complexity measures (time and space) are in the number of basic machine operations and number of semiring operations. We elaborate our contributions below.

1. *Improved upper bounds.* We present improved upper bounds both for generally k components, and the important special case of two components.

   - *General case.* We show that for $k \geqslant 3$ components with n nodes each, after $O(n^{3 \cdot (k-1)})$ preprocessing time, we can answer (i) single-source queries in $O(n^{2 \cdot (k-1)})$ time, (ii) pair queries in $O(n^{k-1})$ time, (iii) partial single-source queries

in $O(n^k)$ time, and (iv) partial pair queries in $O(1)$ time; while using at all times $O(n^{2 \cdot k-1})$ space. In contrast, the existing methods [Lehmann, 1977; Floyd, 1962; Warshall, 1962; Kleene, 1956] compute the transitive closure even for a single query, and thus require $O(n^{3 \cdot k})$ time and $O(n^{2 \cdot k})$ space.

- *Two components.* For the important case of two components, the existing methods require $O(n^6)$ time and $O(n^4)$ space even for one query. In contrast, we establish a variety of tradeoffs between preprocessing and query times, and the best choice depends on the number of expected queries. In particular, for any fixed $\epsilon > 0$, we establish the following three results.

  Three results. First, we show (Corollary 7.2) that with $O(n^3)$ preprocessing time and using $O(n^{2+\epsilon})$ space, we can answer single-source queries in $O(n^{2+\epsilon})$ time, and pair and partial pair queries require $O(n^2)$ time. Second, we show (Theorem 7.2) that with $O(n^{3+\epsilon})$ preprocessing time and using $O(n^3)$ space, we can answer pair and partial pair queries in time $O(n)$ and $O(1)$, respectively. Third, we show (Corollary 7.3) that the transitive closure can be computed using $O(n^{4+\epsilon})$ preprocessing time and $O(n^4)$ space, after which single-source queries require $O(n^2)$ time, and pair and partial pair queries require $O(1)$ time (i.e., all queries require linear time in the size of the output).

  Tradeoffs. Our results provide various tradeoffs: The first result is best for answering $O(n^{1+\epsilon})$ pair and partial pair queries; the second result is best for answering between $\Omega(n^{1+\epsilon})$ and $O(n^{3+\epsilon})$ pair queries, and $\Omega(n^{1+\epsilon})$ partial pair queries; and the third result is best when answering $\Omega(n^{3+\epsilon})$ pair queries. Observe that the transitive closure computation is preferred when the number of queries is large, in sharp contrast to the existing methods that compute the transitive closure even for a single query. Our results are summarized in Table 7.1 and the tradeoffs are pictorially illustrated in Fig. 7.1.

2. *Optimality of our results.* Given our significant improvements for the case of two components, a very natural question is whether the algorithms can be improved further. While presenting matching bounds for polynomial-time graph algorithms to establish optimality is very rare in the whole of computer science, we present *conditional lower bounds*

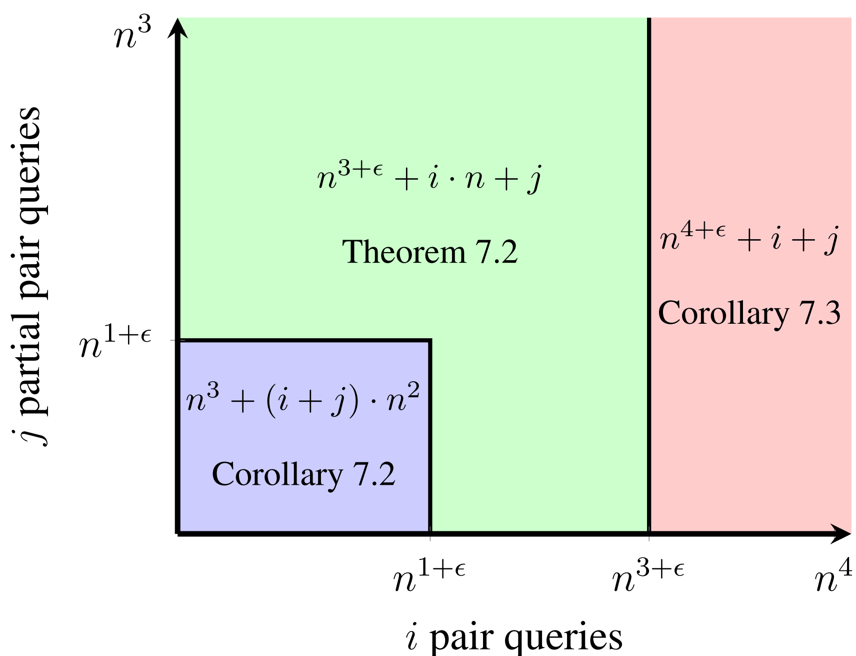


Figure 7.1: Given a concurrent graph G of two constant-treewidth graphs of n nodes each, the figure illustrates the time required by the variants of our algorithms to preprocess G, and then answer i pair queries and j partial pair queries. The different regions correspond to the best variant for handling different number of such queries. In contrast, the current best solution requires $O(n^6 + i + j)$ time. For ease of presentation we omit the $O(\cdot)$ notation.

which show that our combined preprocessing and query time cannot be improved without achieving a major breakthrough in graph algorithms.

- *Almost optimality.* First, note that in the first result (obtained from Corollary 7.2) our space usage and single-source query time are arbitrarily close to optimal, as both the input and the output have size $\Theta(n^2)$. Moreover, the result is achieved with preprocessing time asymptotically less than $\Omega(n^4)$, which is a lower bound for computing the transitive closure (which has n^4 entries). Furthermore, in our third result (obtained from Corollary 7.3) the $O(n^{4+\epsilon})$ preprocessing time is arbitrarily close to optimal, and the $O(n^4)$ preprocessing space is indeed optimal, as the transitive closure computes the distance among all n^4 pairs of nodes (which requires $\Omega(n^4)$ time and space).

- *Conditional lower bound.* In recent years, the conditional lower bound problem has received vast attention in complexity theory, where under the assumption that certain problems (such as matrix multiplication, all-pairs shortest path) cannot be solved faster than the existing upper bounds, lower bounds for other problems (such as dynamic graph algorithms) are obtained [Abboud and Williams, 2014; Abboud *et al.*, 2015; Henzinger *et al.*, 2015]. The current best-known algorithm for the algebraic path problem on general (not constant-treewidth) graphs is cubic in the number of nodes. Even for the special case of shortest paths with positive and negative weights, the best-known algorithm (which has not been improved over five decades) is $O(n \cdot m)$, where m is the number of edges. Since m can be $\Omega(n^2)$, the current best-known worst-case complexity is cubic in the number of nodes. We prove that pair queries require more time in a concurrent graph of two constant-treewidth graphs, with n nodes each, than in general graphs with n nodes. This implies that improving the $O(n^3)$ combined preprocessing and query time over our result (from Corollary 7.2) for answering r queries, for $r = O(n)$, would yield the same improvement over the $O(n^3)$ time for answering r pair queries in general graphs. That is, the combination of our preprocessing and query time (from Corollary 7.2) cannot be improved without equal improvement on the long standing cubic bound for the shortest path and the algebraic path problems in general graphs. Additionally, our result (from Theorem 7.2) cannot be improved much further even for n^2 queries,

as the combined time for preprocessing and answering n^2 queries is $O(n^{3+\epsilon})$ using Theorem 7.2, while the existing bound is $O(n^3)$ for general graphs.

3. *Experimental results.* We provide a prototype implementation of our algorithms which significantly outperforms the baseline methods on several benchmarks.

Technical contributions. The results of this chapter rely on several novel technical contributions.

1. *Upper bounds*. Our upper bounds depend on a series of technical results.

   (a) Given a concurrent graph G obtained from k constant-treewidth graphs G_i, we show how a tree-decomposition of G can be constructed from the strongly balanced tree-decompositions T_i of the components G_i, in time that is linear in the size of the output. We note that G can have large treewidth, and thus determining the treewidth of G can be computationally expensive. Instead, our construction avoids computing the treewidth of G, and directly constructs a tree-decomposition of G from the strongly balanced tree decompositions T_i. Although the resulting tree decomposition is not optimal, it suffices for obtaining significant speedups on the algebraic path problem.

   (b) Given the above tree-decomposition algorithm for concurrent graphs G, in Section 7.5 we present the algorithms for handling algebraic path queries. In particular, we introduce the *partial expansion* $\overline{G}$ of G for additionally handling partial queries, and describe the algorithms for preprocessing and querying $\overline{G}$ in the claimed time and space bounds.

2. *Lower bound.* Given an arbitrary graph G (not of constant treewidth) of n nodes, we show how to construct a constant-treewidth graph G'' of $2 \cdot n$ nodes, and a graph G' that is the product of G'' with itself, such that algebraic path queries in G coincide with such queries in G'. This construction requires quadratic time on n. The conditional optimality of our algorithms follows, as improvement over our algorithms must achieve the same improvement for the algebraic path problem on arbitrary graphs.

Organization. The rest of this chapter is organized as follows.

1. In Section 7.2 we present some definitions regarding concurrent graphs and extensions of the algebraic path problem in the concurrent setting.

2. In Section 7.4 we present a method for constructing a tree decomposition of a concurrent graph, given a strongly-balanced tree decomposition for each component graph.

3. In Section 7.5 we present a data structure for preprocessing and querying a concurrent graph that is the composition of k constant-treewidth graphs.

4. In Section 7.6 we present some (conditional, in cases) lower bounds which prove the optimality of our data structure in handling algebraic path queries in several cases.

5. In Section 7.3 we sketch a modeling example where algebraic paths can be used for the static analysis of a concurrent program.

6. In Section 7.7 we present an experimental evaluation of our data structure on some real-world concurrent programs.

7.2 Definitions

Product graphs. A graph $G_{\mathsf{p}} = (V_{\mathsf{p}}, E_{\mathsf{p}})$ is said to be the *product graph* of k graphs $(G_i = (V_i, E_i))_{1 \leqslant i \leqslant k}$ if $V_{\mathsf{p}} = \prod_i V_i$ and E_{p} is such that for all $u, v \in V_{\mathsf{p}}$ with $u = \langle u_i \rangle_{1 \leqslant i \leqslant k}$ and $v = \langle v_i \rangle_{1 \leqslant i \leqslant k}$, we have $(u, v) \in E_{\mathsf{p}}$ iff there exists a set $\mathcal{I} \subseteq [k]$ such that (i) $(u_i, v_i) \in E_i$ for all $i \in \mathcal{I}$, and (ii) $u_i = v_i$ for all $i \notin \mathcal{I}$. In words, an edge $(u, v) \in E_{\mathsf{p}}$ is formed in the product graph by traversing a set of edges $\{(u_i, v_i) \in E_i\}_{i \in \mathcal{I}}$ in some component graphs $\{G_i\}_{i \in \mathcal{I}}$, and traversing no edges in the remaining $\{G_i\}_{i \notin \mathcal{I}}$. We say that G_{p} is the k*-self-product* of a graph G' if $G_i = G'$ for all $1 \leqslant i \leqslant k$.

Concurrent graphs. A graph $G = (V, E)$ is called a *concurrent graph* of k graphs $(G_i = (V_i, E_i))_{1 \leqslant i \leqslant k}$ if $V = V_{\mathsf{p}}$ and $E \subseteq E_{\mathsf{p}}$, where $G_{\mathsf{p}} = (V_{\mathsf{p}}, E_{\mathsf{p}})$ is the product graph of $(G_i)_i$. Given a concurrent graph $G = (V, E)$ and a node $u \in V$, we will denote by u_i the i-th constituent of u. We say that G is a k*-self-concurrent* of a graph G' if G_{p} is the k*-self-product* of G'.

Various notions of composition. The framework we consider is quite general as it captures various different notions of concurrent composition. Indeed, the edge set of the concurrent graph is any possible subset of the edge set of the corresponding product graph. Then, two well-known

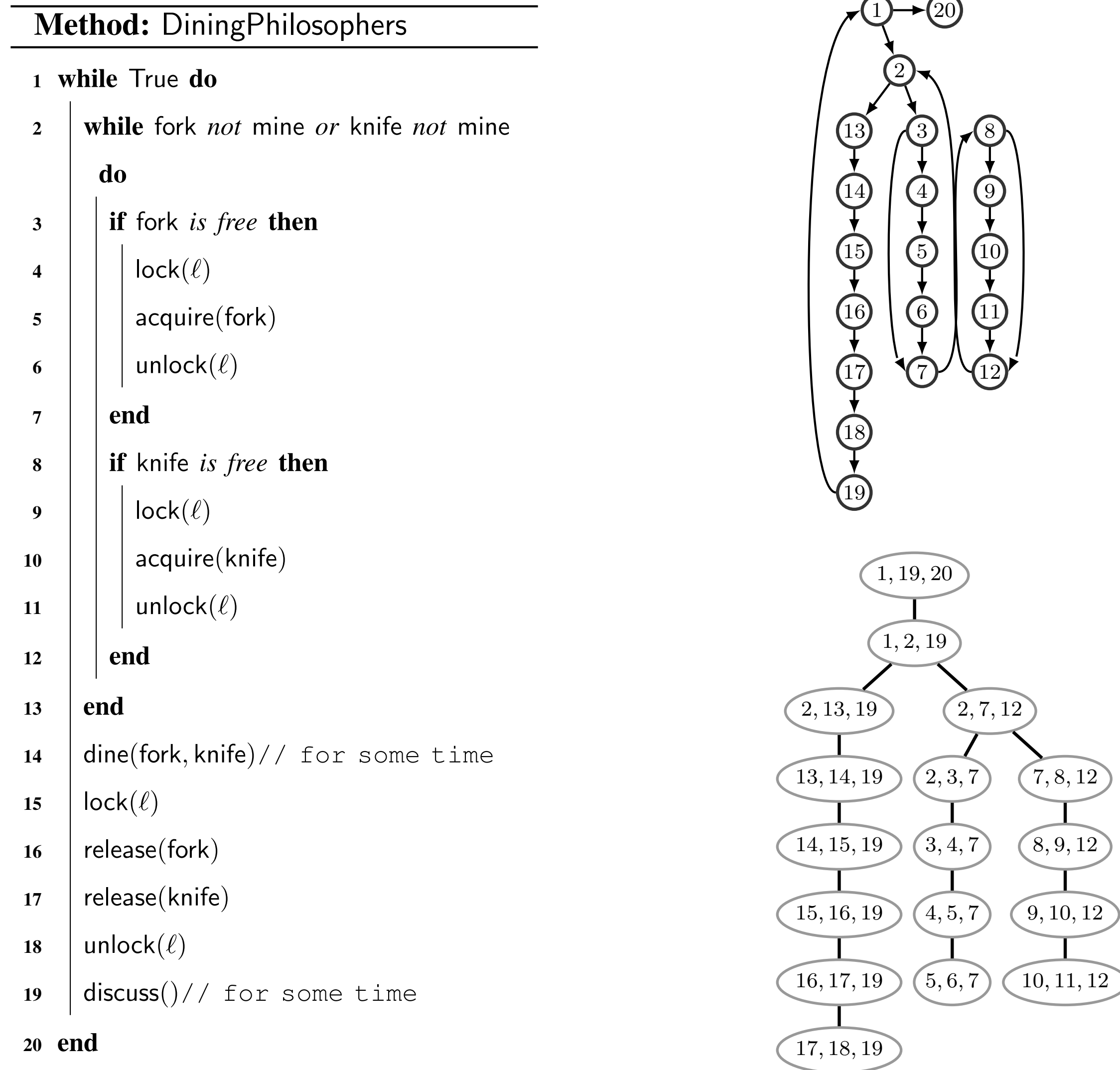


Figure 7.2: A concurrent program, its control flow graph, and a tree decomposition of the control-flow graph.

composition notions can be modeled as follows. For any edge $(u, v) \in E$ of the concurrent graph G, let $\mathcal{I}_{u,v} = \{i \in [k] : (u_i, v_i) \in E_i\}$ denote the components that execute a transition in (u, v).

1. In *synchronous composition* at every step all components make one move each, simultaneously. This is captured by $\mathcal{I}_{u,v} = [k]$ for all $(u, v) \in E$.

2. In *asynchronous composition* at every step only one component makes a move. This is captured by $|\mathcal{I}_{u,v}| = 1$ for all $(u, v) \in E$.

Partial nodes of concurrent graphs. A *partial node* $\overline{u}$ of a concurrent graph G is an element of $\prod_i (V_i \cup \{\bot\})$, where $\bot \notin \bigcup_i V_i$. Intuitively, $\bot$ is a fresh symbol to denote that a component

is unspecified. A partial node $\overline{u}$ is said to *refine* a partial node $\overline{v}$, denoted by $\overline{u} \sqsubseteq \overline{v}$ if for all $1 \leqslant i \leqslant k$ either $\overline{v}_i = \bot$ or $\overline{v}_i = \overline{u}_i$. We say that the partial node $\overline{u}$ *strictly refines* $\overline{v}$, denoted by $\overline{u} \sqsubset \overline{v}$, if $\overline{u} \sqsubseteq \overline{v}$ and $\overline{u} \neq \overline{v}$ (i.e., for at least one constituent i we have $\overline{v}_i = \bot$ but $\overline{u}_i \neq \bot$). A partial node $\overline{u}$ is called *strictly partial* if it is strictly refined by some node $u \in V$ (i.e., $\overline{u}$ has at least one $\bot$). The notion of semiring distances is extended to partial nodes, and for partial nodes $\overline{u}, \overline{v}$ of G we define the semiring distance from $\overline{u}$ to $\overline{v}$ as

$$d(\overline{u}, \overline{v}) = \bigoplus_{u \sqsubseteq \overline{u}, v \sqsubseteq \overline{v}} d(u, v)$$

where $u, v \in V$. In the sequel, a partial node $\overline{u}$ will be either (i) a node of V, or (ii) a strictly partial node. We refer to nodes of the first case as actual nodes, and write u (i.e., without the bar). Distances where one endpoint is a strictly partial node $\overline{u}$ succinctly quantify over all nodes of all the components for which the corresponding constituent of $\overline{u}$ is $\bot$. Observe that the distance still depends on the unspecified components.

Semiring distance queries on concurrent graphs of constant-treewidth components. In this chapter we are interested in the following problem. Let $G = (V, E)$ be a concurrent graph of $k \geqslant 2$ constant-treewidth graphs $(G_i = (V_i, E_i))_{1 \leqslant i \leqslant k}$, and $\mathsf{wt} : E \to \Sigma$ be a weight function that assigns to every edge of G a weight from a set Σ that forms a complete, closed semiring $S = (\Sigma, \oplus, \otimes, \overline{\mathbf{0}}, \overline{\mathbf{1}})$. The *semiring distance problem* on G asks the following types of queries:

1. *Single-source query.* Given a partial node $\overline{u}$ of G, return the distance $d(\overline{u}, v)$ to every node $v \in V$. When the partial node $\overline{u}$ is an actual node of G, we have a traditional single-source query.

2. *Pair query.* Given two nodes $u, v \in V$, return the distance $d(u, v)$.

3. *Partial pair query.* Given two partial nodes $\overline{u}, \overline{v}$ of G where at least one is strictly partial, return the distance $d(\overline{u}, \overline{v})$.

Fig. 7.2 presents the notions introduced in this section on a toy example on the dining philosophers program. See Section 7.3 for an example on pair and partial pair queries in the analysis of the dining philosophers program.

Input parameters. For technical convenience, we consider a uniform upper bound n on the number of nodes of each G_i (i.e. $|V_i| \leqslant n$). Similarly, we let $t = O(1)$ be an upper bound on the

treewidth of each G_i. The number k is taken to be fixed and independent of n. The input of the problem consists of the graphs $(G_i)_{1\leqslant i\leqslant k}$, together with some representation of the edge relation E of G.

Complexity measures. The complexity of our algorithms is measured as a function of n. In particular, we ignore the size of the representation of E when considering the size of the input. This has the advantage of obtaining complexity bounds that are independent of the representation of E, which can be represented implicitly (such as synchronous or asynchronous composition) or explicitly, depending on the modeling of the problem under consideration. The time complexity of our algorithms is measured in number of operations, with each operation being either a basic machine operation, or an application of one of the operations of the semiring.

7.3 Modeling Example

Fig. 7.2 illustrates the introduced notions in a small example of the well-known k dining philosophers problem. For the purpose of the example, lock is considered a blocking operation. Consider the case of $k = 2$ threads being executed in parallel. The graphs G_1 and G_2 that correspond to the two threads have nodes of the form (i, ℓ), where $i \in [20]$ is a node of the control flow graph, and $\ell \in [3]$ denotes the thread that controls the lock ($\ell = 3$ denotes that ℓ is free, whereas $\ell = i \in [2]$ denotes that it is acquired by thread i). The concurrent graph G is taken to be the asynchronous composition of G_1 and G_2, and consists of nodes $\langle x, y\rangle$, where x and y is a node of G_1 and G_2 respectively, such that x and y agree on the value of ℓ (all other nodes can be discarded). For brevity, we represent nodes of G as triplets $\langle x, y, \ell\rangle$ where now x and y are nodes in the control flow graphs G_1 and G_2 (i.e., without carrying the value of the lock), and ℓ is the value of the lock. A transition to a node $\langle x, y, \ell\rangle$ in which one component G_i performs a lock is allowed only from a node where $\ell = 3$, and sets $\ell = i$ in the target node (i.e., $\langle x, y, i\rangle$). Similarly, a transition to a node $\langle x, y, \ell\rangle$ in which one component G_i performs an unlock is allowed from a node where $\ell = i$, and sets $\ell = 3$ in the target node(i.e., $\langle x, y, 3\rangle$).

Suppose that we are interested in determining (1) whether the first thread can execute dine(fork, knife) without owning fork or knife, and (2) whether a deadlock can be reached in which each thread owns one resource. These questions naturally correspond to partial pair

and pair queries respectively, as in case (1) we are interested in a local property of G_1, whereas in case (2) we are interested in a global property of G. We note, however, that case (1) still requires an analysis on the concurrent graph G. In each case, the analysis requires a set of data facts D, along with dataflow functions $f : 2^D \to 2^D$ that mark each edge. These functions are distributive, in the sense that $f(A) = \bigcup_{a \in A} f(\{a\})$. Hence, with a slight abuse of notation, we can define f as functions $f : D \to D$, and their extension to $2^D \to 2^D$ is according to the distributivity property.

Local property as a partial pair query. Assume that we are interested in determining whether the first thread can execute $\mathsf{dine}(\mathsf{fork}, \mathsf{knife})$ without owning fork or knife. A typical data fact set is $D = \{\mathsf{fork}, \mathsf{knife}, \mathsf{null}\}$, where each data fact denotes that the corresponding resource must be owned by the first thread. The concurrent graph G is associated with a weight function wt of dataflow functions $f : 2^D \to 2^D$. The dataflow function $\mathsf{wt}(e)$ along an edge e behaves as follows on input data fact F (we only describe the case where $F = \mathsf{fork}$, as the other case is symmetric).

1. If e transitions to a node in which the second thread acquires fork or the first thread releases fork, then $\mathsf{wt}(e)(\mathsf{fork}) \to \mathsf{null}$ (i.e., fork is removed from the data facts).

2. Else, if e transitions to a node in which the first thread acquires fork, then $\mathsf{wt}(e)(\mathsf{null}) \to \mathsf{fork}$ (i.e., fork is inserted to the data facts).

Similarly for the $F = \mathsf{knife}$ data fact. The "meet-over-all-paths" operation is set intersection. Then the question is answered by testing whether $d(\langle 1, 1, 3\rangle, \langle 14, \bot, 3\rangle) = \{\{\mathsf{fork}, \mathsf{knife}\}\}$, i.e., by performing a *partial pair query*, in which the node of the second thread is unspecified.

Global property as a pair query. Assume that we are interested in determining whether the two threads can cause a deadlock. Because of symmetry, we look for a deadlock in which the first thread may hold the fork, and the second thread may hold the knife. A typical data fact set is $D = 2^{\{\mathsf{fork}, \mathsf{knife}\}}$. For a data fact $F \in D$ we have

1. $\mathsf{fork} \in F$ if fork may be acquired by the *first* thread.

2. $\mathsf{knife} \in F$ if knife may be acquired by the *second* thread.

The concurrent graph G is associated with a weight function wt of dataflow functions $f : 2^D \to$

2^D. The dataflow function $\mathsf{wt}(e)$ along an edge e behaves as follows on input data fact F.

1. If e transitions to a node in which the second thread acquires fork or the first thread releases fork, then $\mathsf{wt}(e)(F) \to F \setminus \{\mathsf{fork}\}$ (i.e., the first thread no longer owns fork).

2. If e transitions to a node in which the first thread acquires fork, then $\mathsf{wt}(e)(F) \to F \cup \{\mathsf{fork}\}$ (i.e., the first thread now owns fork).

3. If e transitions to a node in which the first thread acquires knife or the second thread releases knife, then $\mathsf{wt}(e)(F) \to F \setminus \{\mathsf{knife}\}$ (i.e., the second thread no longer owns knife).

4. If e transitions to a node in which the second thread acquires knife, then $\mathsf{wt}(e)(F) \to F \cup \{\mathsf{knife}\}$ (i.e., the second thread now owns knife).

The "meet-over-all paths" operation is set union. Then the question is answered by testing whether $\{\mathsf{fork}, \mathsf{knife}\} \in d(\langle 1,1,3\rangle, \langle 2,2,3\rangle)$, i.e., by performing a *pair query*, and finding out whether the two threads can start the $while$ loop with each one holding one resource. Alternatively, we can answer the question by performing a single-source query from $\langle 1,1,3\rangle$ and finding out whether there exists any node in the concurrent graph G in which every thread owns one resource (i.e., its distance contains $\{\mathsf{fork}, \mathsf{knife}\}$).

7.4 Concurrent Tree Decomposition

In this section we present the construction of a tree-decomposition $\mathrm{Tree}(G)$ of a concurrent graph $G = (V, E)$ of k constant-treewidth graphs. In general, G can have treewidth which depends on the number of its nodes (e.g., G can be a grid, which has treewidth n, obtained as the product of two lines, which have treewidth 1). While the treewidth computation for constant-treewidth graphs is linear time [Bodlaender, 1996], it is NP-complete for general graphs [Bodlaender, 1993]. Hence computing a tree decomposition that achieves the treewidth of G can be computationally expensive (e.g., exponential in the size of G). Here we develop an algorithm ConcurTree which constructs a tree-decomposition $\mathsf{ConcurTree}(G)$ of G, given an (α, β, γ) tree-decomposition of the components, in $O(n^k)$ time and space (i.e., linear in the size of G), such that the following properties hold: (i) the width is $O(n^{k-1})$; and (ii) for every bag in level at least $i \cdot \gamma$, the size of the bag is $O(n^{k-1} \cdot \beta^i)$ (i.e., the size of the bags decreases geometrically along the levels).

Algorithm $\mathsf{ConcurTree}$ **for concurrent tree decomposition.** Let G be a concurrent graph of k graphs $(G_i)_{1\leqslant i\leqslant k}$. The input consists of a full binary tree-decomposition T_i of constant width for every graph G_i. In the following, B_i ranges over bags of T_i, and we denote by $B_{i,r}$, with $r \in [2]$, the r-th child of B_i. We construct the *concurrent tree-decomposition* $T = \mathsf{ConcurTree}(G) = (V_T, E_T)$ of G using the recursive procedure $\mathsf{ConcurTree}$, which operates as follows. On input $(T_i(B_i))_{1\leqslant i\leqslant k}$, return a tree decomposition where

1. The root bag B is

$$B = \bigcup_{1\leqslant i\leqslant k} \left(\left(\prod_{j<i} V_{T_j}(B_j) \right) \times B_i \times \left(\prod_{j>i} V_{T_j}(B_j) \right) \right) \tag{7.1}$$

2. If every B_i is a non-leaf bag of T_i, for every choice of $\langle r_1, \dots, r_k \rangle \in [2]^k$, repeat the procedure for $(T_i(B_{i,r_i}))_{1\leqslant i\leqslant k}$, and let B' be the root of the returned tree. Make B' a child of B.

3. If some B_i is a leaf bag of T_i, then the algorithm terminates.

See Algorithm 16 for the formal description.

Algorithm 16: $\mathsf{ConcurTree}$

Input: Tree-decompositions $T_i = (V_{T_i}, E_{T_i})_{1\leqslant i\leqslant k}$ with root bags $(B_i)_{1\leqslant i\leqslant k}$.

Output: A tree decomposition T of the concurrent graph

```
1  Assign B ← ∅
2  Assign T ← a tree with the single bag B as its root
3  for i ∈ [k] do
4  |  Assign B ← B ∪ (∏_{1⩽j<i} V_{T_j}(B_j) × B_i × ∏_{i<j⩽k} V_{T_j}(B_j))
5  end
6  if none of the B_i's is a leaf in its respective T_i then
7  |  for every sequence of bags B'_1, ..., B'_k such that each B'_i is a child of B_i in T_i do
8  |  |  Assign T'_i ← ConcurTree(T_1(B'_1), ..., T_k(B'_k))
9  |  |  Add T'_i to T_i, setting the root of T'_i as a new child of B
10 |  end
11 end
12 return T
```

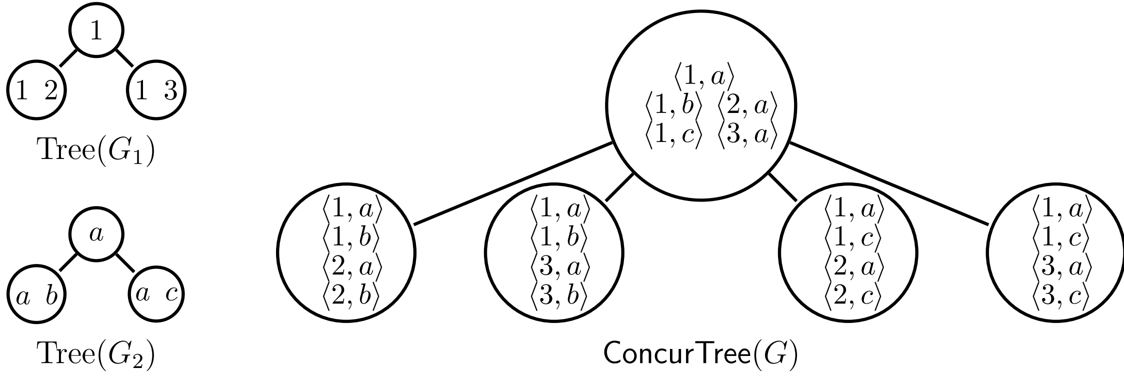


Figure 7.3: The tree-decomposition $\mathsf{ConcurTree}(G)$ of a concurrent graph G of two constant-treewidth graphs G_1 and G_2.

Let B_i be the root of the tree-decomposition T_i. We denote by $\mathsf{ConcurTree}(G)$ the application of the recursive procedure $\mathsf{ConcurTree}$ on $(T_i(B_i))_{1\leqslant i\leqslant k}$. Fig. 7.3 provides an illustration.

Remark 7.1. Recall that for any bag B_j of a tree-decomposition T_j, we have $V_{T_j}(B_j) = \bigcup_{B'_j} B'_j$, where B'_j ranges over bags in $T_j(B_j)$. Then, for any two bags B_{i_1}, B_{i_2}, of tree-decompositions T_{i_1} and T_{i_2} respectively, we have

$$V_{T_{i_1}}(B_{i_1}) \times V_{T_{i_2}}(B_{i_2}) = \bigcup_{B'_{i_1}, B'_{i_2}} \left(B'_{i_1} \times B'_{i_2}\right)$$

where B'_{i_1} and B'_{i_2} range over bags in $T_{i_1}(B_{i_1})$ and $T_{i_2}(B_{i_2})$ respectively. Since each tree-decomposition T_i has constant width, it follows that $|V_{T_{i_1}}(B_{i_1}) \times V_{T_{i_2}}(B_{i_2})| = O(|T_{i_1}(B_{i_1})| \cdot |T_{i_2}(B_{i_2})|)$. Thus, the size of each bag B of $\mathsf{ConcurTree}(G)$ constructed in Eq. (7.1) on some input $(T_i(B_i))_i$ is $|B| = O(\sum_i \prod_{j\neq i} n_j)$, where $n_i = |T_i(B_i)|$.

In view of Remark 7.1, the time and space required by $\mathsf{ConcurTree}$ to operate on input $(T_i(B_i))_{1\leqslant i\leqslant k}$ where $|T_i(B_i)| = n_i$, is given, up to constant factors, by

$$\mathcal{T}(n_1, \dots, n_k) \leqslant \sum_{1\leqslant i\leqslant k} \prod_{j\neq i} n_j + \sum_{(r_i)_i \in [2]^k} \mathcal{T}(n_{1,r_1}, \dots, n_{k,r_k}) \tag{7.2}$$

such that for every i we have that $\sum_{r_i\in[2]} n_{i,r_i} \leqslant n_i$.

The following lemma establishes the correctness of the construction.

Lemma 7.1. $\mathsf{ConcurTree}(G)$ *is a tree decomposition of* G.

Proof. We show that T satisfies the three conditions of a tree decomposition.

C1 For each node $u = \langle u_i \rangle_{1 \leqslant i \leqslant k}$, let $j = \arg\min_i \mathsf{Lv}(u_i)$. Then $u \in B$, where B is the bag constructed by step 1 of ConcurTree when it operates on input $(T_i(B_i))_{1 \leqslant i \leqslant k}$, where each T_i is a tree decomposition, and additionally $B_j = B_{u_j}$ (i.e., B_j is the root bag of u_j in T_j).

C2 Similarly, for each edge $(u, v) \in E$ with $u = \langle u_i \rangle_{1 \leqslant i \leqslant k}$ and $v = \langle v_i \rangle_{1 \leqslant i \leqslant k}$, let $j = \arg\min_i(\max(\mathsf{Lv}(u_i), \mathsf{Lv}(v_i)))$, where $\arg\min_i f(i)$ returns the value of i that minimizes f. Then $(u, v) \in B$, where B is a bag similar to C1.

C3 For any node $u = \langle u_i \rangle_{1 \leqslant i \leqslant k}$ and path $P : B \rightsquigarrow B'$ with $u \in B \cap B'$, let B'' be any bag of P. Since at least one of B, B' is a descendant of B'', we have $V_T(B) \subseteq V_T(B'')$ or $V_T(B') \subseteq V_T(B'')$, and because $u \in B \cap B'$, if B'' was constructed on input $(T_i(B''_i))_{1 \leqslant i \leqslant k}$, where each T_i is a tree decomposition, we have $u_i \in V_{T_i}(B''_i)$. Let $(T_i(B_i))_{1 \leqslant i \leqslant k}$ and $(T_i(B'_i))_{1 \leqslant i \leqslant k}$ be the inputs to the algorithm when B and B' were constructed, and it follows that for some $1 \leqslant j \leqslant k$ we have $u_j \in B_j \cap B'_j$. Then B''_j is an intermediate bag in the path $P_j : B_j \rightsquigarrow B'_j$ in T_j, thus $u_j \in B''_j$ and hence $u \in B''$.

The desired result follows. □

We now turn our attention to the complexity. We start with analyzing the following recurrence, which will be useful in the complexity analysis afterwards.

Lemma 7.2. *Consider the following recurrence.*

$$\mathcal{T}(n_1, \ldots, n_k) \leqslant \sum_{1 \leqslant i \leqslant k} \prod_{j \neq i} n_j + \sum_{(r_i)_i \in [2]^k} \mathcal{T}(n_{1,r_1}, \ldots, n_{k,r_k}) \tag{7.3}$$

such that for every i we have that $n_{i,1}, n_{i,2} \geqslant 1$ and $\sum_{r_i \in [2]} n_{i,r_i} \leqslant n_i$ and as the base case we have that if $n_i = 1$ for some i, then

$$\mathcal{T}(n_1, \ldots, n_k) \leqslant \sum_{1 \leqslant i \leqslant k} \prod_{j \neq i} n_j \tag{7.4}$$

Then Eq. 7.3 has the solution

$$\mathcal{T}(n_1, \ldots, n_k) \leqslant 2 \cdot k \cdot \prod_{1 \leqslant i \leqslant k} n_i - \sum_{1 \leqslant i \leqslant k} \prod_{j \neq i} n_j. \tag{7.5}$$

Proof. Observe that the right hand side of Eq. 7.5 is always larger than the right hand side of Eq.7.4. Hence, in order to verify that Eq. 7.3 has Eq. 7.5 as a solution, it suffices to substitute

Eq. 7.5 in Eq. 7.3 (i.e., we take Eq. 7.5 also as the base case solution). Indeed, substituting Eq. 7.5 to the recurrence Eq. 7.3 we have

$$\begin{aligned}\mathcal{T}(n_1,\dots,n_k) &\leqslant \sum_{1\leqslant i\leqslant k}\prod_{j\neq i} n_j + \sum_{(r_i)_i\in[2]^k}\left(2\cdot k\cdot \prod_{1\leqslant i\leqslant k} n_{i,r_i} - \sum_{1\leqslant i\leqslant k}\prod_{j\neq i} n_{j,r_j}\right)\\ &= \sum_{1\leqslant i\leqslant k}\prod_{j\neq i} n_j + 2\cdot k\cdot X - Y \qquad (7.6)\end{aligned}$$

where

$$X = \sum_{(r_i)_i\in[2]^k}\left(\prod_{1\leqslant i\leqslant k} n_{i,r_i}\right) \qquad \text{and} \qquad Y = \sum_{(r_i)_i\in[2]^k}\left(\sum_{1\leqslant i\leqslant k}\prod_{j\neq i} n_{j,r_j}\right)$$

We compute X and Y respectively.

$$X = \sum_{(r_i)_i\in[2]^k}\left(\prod_{1\leqslant i\leqslant k} n_{i,r_i}\right) = \sum_{r_1\in[2]} n_{1,r_1}\cdot\left(\sum_{r_2\in[2]} n_{2,r_2}\cdot\left(\cdots\sum_{r_k\in[2]} n_{k,r_k}\right)\right) \leqslant \prod_{1\leqslant i\leqslant k} n_i \quad (7.7)$$

by factoring out every n_{i,r_i} of the sum. Similarly,

$$\begin{aligned}Y = &\sum_{(r_i)_i\in[2]^k}\left(\sum_{1\leqslant i\leqslant k}\prod_{j\neq i} n_{j,r_j}\right) = \sum_{1\leqslant i\leqslant k}\left(\sum_{(r_i)_i\in[2]^k}\prod_{j\neq i} n_{j,r_j}\right)\\ =&2\cdot\sum_{1\leqslant i\leqslant k}\left(\sum_{r_1\in[2]} n_{1,r_1}\cdot\ldots(\sum_{r_{i-1}\in[2]} n_{i-1,r_{i-1}}\cdot(\sum_{r_{i+1}\in[2]} n_{i+1,r_{i+1}}\cdot\ldots(\sum_{r_k\in[2]} n_{k,r_k})))\right)\\ \geqslant&2\cdot\sum_{1\leqslant i\leqslant k}\prod_{j\neq i} n_j \qquad (7.8)\end{aligned}$$

The second equality is obtained by swapping the inner with the outer sum. The third equality follows by expanding the sum over $(r_i)_i \in [2]^k$. The final inequality is obtained since for all $1 \leqslant i \leqslant k$ we have $n_{i,1} + n_{i,2} \leqslant n_i$. Substituting inequalities Eq. 7.7 and 7.8 to Eq. 7.6 we obtain

$$\mathcal{T}(n_1,\dots,n_k) \leqslant \sum_{1\leqslant i\leqslant k}\prod_{j\neq i} n_j + 2\cdot k\cdot X - Y \leqslant 2\cdot k\cdot\prod_{1\leqslant i\leqslant k} n_i - \sum_{1\leqslant i\leqslant k}\prod_{j\neq i} n_j$$

as desired. □

Lemma 7.3. ConcurTree *requires $O(n^k)$ time and space.*

Proof. It is easy to verify that ConcurTree(G) performs a constant number of operations per node per bag in the returned tree decomposition. Hence we will bound the time taken by bounding the size of ConcurTree(G). Consider a recursion step of ConcurTree on input $(T_i(B_i))_{1\leqslant i\leqslant k}$. Let

$n_i = |T_i(B_i)|$ for all $1 \leqslant i \leqslant k$, and $n_{i,r_i} = |T_i(B_{i,r_i})|$, $r_i \in [2]$, where B_{i,r_i} is the r_i-th child of B_i. In view of Remark 7.1, the time required by ConcurTree on this input is given by the recurrence in Eq. (7.3), up to a constant factor. The desired result follows from Lemma 7.2. □

We summarize the results of this section with the following theorem.

Theorem 7.1. *Let $G = (V, E)$ be a concurrent graph of k constant-treewidth graphs $(G_i)_{1 \leqslant i \leqslant k}$ of n nodes each. Let a binary (α, β, γ) tree-decomposition T_i for every graph G_i be given, for some constant α.* ConcurTree *constructs a 2^k-ary tree-decomposition* ConcurTree(G) *of G in $O(n^k)$ time and space, with the following property. For every $i \in \mathbb{N}$ and bag B at level* Lv$(B) \geqslant i \cdot \gamma$, *we have $|B| = O(n^{k-1} \cdot \beta^i)$.*

Proof. Lemma 7.1 proves the correctness and Lemma 7.3 the complexity. Here we focus on bounding the size of a bag B with Lv$(B) \leqslant i \cdot \gamma$. Let $(T_i(B_i))_{1 \leqslant i \leqslant k}$ be the input on ConcurTree when it constructed B using Eq. (7.1) and $n_i = |T_i(B_i)|$. Observe that Lv$(B) =$ Lv(B_i) for all i, and since each T_i is (β, γ)-balanced, we have that $n_i \leqslant O(n \cdot \beta^i)$. Since each T_i is α-approximate, $|B_i| = O(1)$ for all i. It follows from Eq. (7.1) and Remark 7.1 that $|B| = O(n^{k-1} \cdot \beta^i)$. □

7.5 Semiring Distances on Concurrent Graphs

We now turn our attention to the core algorithmic problem of this chapter, namely answering semiring distance queries in a concurrent graph G of k constant-treewidth graphs $(G_i)_{1 \leqslant i \leqslant k}$. To this direction, we develop a data structure ConcurSD (for *concurrent semiring distances*) which will preprocess G and afterwards support single-source, pair, and partial pair queries on G.

Informal description of the preprocessing. The preprocessing phase of ConcurSD is handled by algorithm ConcurPreprocess, which performs the following steps.

1. First, the *partial expansion* $\overline{G}$ of G is constructed by introducing a pair of strictly partial nodes $\overline{u}^1$, $\overline{u}^2$ for every strictly partial node $\overline{u}$ of G, and edges between strictly partial nodes and the corresponding nodes of G that refine them.

2. Second, the concurrent tree-decomposition $T =$ ConcurTree(G) of G is constructed, and modified to a tree-decomposition $\overline{T}$ of the partial expansion graph $\overline{G}$.

3. Third, a standard, two-way pass of $\overline{T}$ is performed to compute *local distances*. In this step, for every bag $\overline{B}$ in $\overline{T}$ and all partial nodes $\overline{u}, \overline{v} \in \overline{B}$, the distance $d(\overline{u}, \overline{v})$ is computed (i.e., all-pair distances in $\overline{B}$). Since we compute distances between nodes that are *local* in a bag, this step is called local distance computation. This information is used to handle (i) single-source queries and (ii) partial pair queries in which both nodes are strictly partial.

4. Finally, a top-down pass of $\overline{T}$ is performed in which for every node u and partial node $\overline{v} \in \overline{\mathcal{V}}_{\overline{T}}(\overline{B}_u)$ (i.e., $\overline{v}$ appears in some ancestor of $\overline{B}_u$) the distances $d(u, \overline{v})$ and $d(\overline{v}, u)$ are computed. This information is used to handle pair queries in which at least one node is a node of G (i.e., not strictly partial).

Bottom-up and top-down traversals. In the description of the preprocessing algorithm ConcurPreprocess, we make use of two types of traversals of the tree decomposition. A *bottom-up traversal* is any traversal of the tree in which a bag B is visited after all children of B have been visited. A *top-down traversal* is any traversal of the tree in which a bag B is visited after the parent of B has been visited.

Algorithm ConcurPreprocess**.** We now formally describe algorithm ConcurPreprocess for preprocessing the concurrent graph $G = (V, E)$ for the purpose of answering semiring distance queries. For any desired $0 < \epsilon \leqslant 1$, we choose appropriate constants α, β, γ, which will be defined later for the complexity analysis. On input $G = (V, E)$, where G is a concurrent graph of k constant-treewidth graphs $(G_i = (V_i, E_i))_{1 \leqslant i \leqslant k}$, and a weight function $\mathsf{wt} : E \to \Sigma$, ConcurPreprocess operates as follows:

1. Construct the *partial expansion* $\overline{G} = (\overline{V}, \overline{E})$ of G together with an extended weight function $\overline{\mathsf{wt}} : \overline{E} \to \Sigma$ as follows.

   (a) The node set is $\overline{V} = V \cup \{\overline{u}^1, \overline{u}^2 : \exists u \in V \text{ s.t. } u \sqsubset \overline{u}\}$; i.e., $\overline{V}$ consists of nodes in V and two copies for every partial node $\overline{u}$ that is strictly refined by a node u of G.

   (b) The edge set is $\overline{E} = E \cup \{(\overline{u}^1, u), (u, \overline{u}^2) : \overline{u}^1, \overline{u}^2 \in \overline{V} \text{ and } u \in V \text{ s.t. } u \sqsubset \overline{u}^1, \overline{u}^2\}$, i.e., along with the original edges E, the first (resp. second) copy of every strictly partial node has outgoing (resp. incoming) edges to (resp. from) the nodes of G that refine it.

   (c) For the weight function we have $\overline{\mathsf{wt}}(\overline{u}, \overline{v}) = \mathsf{wt}(\overline{u}, \overline{v})$ if $\overline{u}, \overline{v} \in V$, and $\overline{\mathsf{wt}}(\overline{u}, \overline{v}) = \overline{\mathbf{1}}$

otherwise. That is, the original weight function is extended with value $\overline{\mathbf{1}}$ (which is neutral for semiring multiplication) to all new edges in $\overline{G}$.

2. Construct the tree-decomposition $\overline{T} = (\overline{V}_T, \overline{E}_T)$ of $\overline{G}$ as follows.

   (a) Obtain an (α, β, γ) tree-decomposition $T_i = \mathrm{Tree}(G_i)$ of every graph G_i using Theorem 2.2.

   (b) Construct the concurrent tree-decomposition $T = \mathsf{ConcurTree}(G)$ of G using $(T_i)_{1 \leqslant i \leqslant k}$.

   (c) Let $\overline{T}$ be identical to T, with the following exception: For every bag B of T and $\overline{B}$ the corresponding bag in $\overline{T}$, for every node $u \in B$, insert in $\overline{B}$ all strictly partial nodes $\overline{u}^1, \overline{u}^2$ of $\overline{V}$ that u refines. Formally, set $\overline{B} = B \cup \{\overline{u}^1, \overline{u}^2 : \exists u \in B \text{ s.t. } u \sqsubset \overline{u}\}$. Note that also $u \in \overline{B}$.

   Observe that the root bag of $\overline{T}$ contains all strictly partial nodes.

3. Perform the *local distance computation* on $\overline{T}$ as follows. For every partial node $\overline{u}$, maintain two map data structures $\mathrm{F{\scriptstyle WD}}_{\overline{u}}, \mathrm{B{\scriptstyle WD}}_{\overline{u}} : \overline{B}_{\overline{u}} \to \Sigma$. Intuitively, $\mathrm{F{\scriptstyle WD}}_{\overline{u}}$ (resp. $\mathrm{B{\scriptstyle WD}}_{\overline{u}}$) aims to store the forward (resp., backward) distance, i.e., distance from (resp., to) $\overline{u}$ to (resp. from) nodes in $\overline{B}_{\overline{u}}$. Initially set $\mathrm{F{\scriptstyle WD}}_{\overline{u}}(\overline{v}) = \overline{\mathsf{wt}}(\overline{u}, \overline{v})$ and $\mathrm{B{\scriptstyle WD}}_{\overline{u}}(\overline{v}) = \overline{\mathsf{wt}}(\overline{v}, \overline{u})$ for all partial nodes $\overline{v} \in \overline{B}_{\overline{u}}$ (and $\mathrm{F{\scriptstyle WD}}_{\overline{u}}(\overline{v}) = \mathrm{B{\scriptstyle WD}}_{\overline{u}}(\overline{v}) = \overline{\mathbf{0}}$ if $(\overline{u}, \overline{v}) \notin \overline{E}$). At any point in the computation, given a bag $\overline{B}$ we denote by $wt_{\overline{B}} : \overline{B} \times \overline{B} \to \Sigma$ a map data structure such that for every pair of partial nodes $\overline{u}, \overline{v}$ with $\mathsf{Lv}(\overline{v}) \leqslant \mathsf{Lv}(\overline{u})$ we have $wt_{\overline{B}}(\overline{u}, \overline{v}) = \mathrm{F{\scriptstyle WD}}_{\overline{u}}(\overline{v})$ and $wt_{\overline{B}}(\overline{v}, \overline{u}) = \mathrm{B{\scriptstyle WD}}_{\overline{u}}(\overline{v})$.

   (a) Traverse $\overline{T}$ bottom-up, and for every bag $\overline{B}$, execute an all-pairs semiring distance computation on $\overline{G}[\overline{B}]$ with weight function $wt_{\overline{B}}$. This is done using standard algorithms for the transitive closure, e.g. [Lehmann, 1977; Floyd, 1962; Warshall, 1962; Kleene, 1956]. For every pair of partial nodes $\overline{u}, \overline{v}$ with $\mathsf{Lv}(\overline{v}) \leqslant \mathsf{Lv}(\overline{u})$, set $\mathrm{B{\scriptstyle WD}}_{\overline{u}}(\overline{v}) = d'(\overline{v}, \overline{u})$ and $\mathrm{F{\scriptstyle WD}}_{\overline{u}}(\overline{v}) = d'(\overline{u}, \overline{v})$, where $d'(\overline{u}, \overline{v})$ and $d'(\overline{v}, \overline{u})$ are the computed distances in $\overline{G}[\overline{B}]$.

   (b) Traverse $\overline{T}$ top-down, and for every bag $\overline{B}$ perform the computation of Item 3a.

4. Perform the *ancestor distance computation* on $\overline{T}$ as follows. For every node u, maintain

two map data structures $\mathsf{F}_u, \mathsf{T}_u : \overline{\mathcal{V}}_{\overline{T}}(\overline{B}_u) \to \Sigma$ from partial nodes that appear in the ancestor bags of $\overline{B}_u$ to Σ. These maps aim to capture distances between the node u and nodes in the ancestor bags of $\overline{B}_u$ (in contrast to $\textsc{Fwd}_u$ and $\textsc{Bwd}_u$ which store distances only between u and nodes in $\overline{B}_u$). Initially, set $\mathsf{F}_u(\overline{v}) = \textsc{Fwd}_u(\overline{v})$ and $\mathsf{T}_u(\overline{v}) = \textsc{Bwd}_u(\overline{v})$ for every partial node $\overline{v} \in \overline{B}_u$. Given a pair of partial nodes $\overline{u}, \overline{v}$ with $\mathsf{Lv}(\overline{v}) \leqslant \mathsf{Lv}(\overline{u})$ we denote by $wt^+(\overline{u}, \overline{v}) = \mathsf{F}_{\overline{u}}(\overline{v})$ and $wt^+(\overline{v}, \overline{u}) = \mathsf{T}_{\overline{u}}(\overline{v})$. Traverse $\overline{T}$ via a DFS starting from the root, and for every encountered bag $\overline{B}$ with parent $\overline{B}'$, for every node u such that $\overline{B}$ is the root bag of u, for every partial node $\overline{v} \in \overline{\mathcal{V}}_{\overline{T}}(\overline{B}_u)$, assign

$$\mathsf{F}_u(\overline{v}) = \bigoplus_{x \in \overline{B} \cap \overline{B}'} \textsc{Fwd}_u(x) \otimes wt^+(x, \overline{v}) \tag{7.9}$$

$$\mathsf{T}_u(\overline{v}) = \bigoplus_{x \in \overline{B} \cap \overline{B}'} \textsc{Bwd}_u(x) \otimes wt^+(\overline{v}, x) \tag{7.10}$$

If $\overline{B}$ is the root of $\overline{T}$, simply initialize the maps F_u and T_u according to the corresponding maps $\textsc{Fwd}_u$ and $\textsc{Bwd}_u$ constructed from Item 3.

5. Preprocess $\overline{T}$ to answer LCA queries in $O(1)$ time [Harel and Tarjan, 1984].

See Algorithms 17 to 20 for the formal description of the above steps.

The following claim states that the first (resp. second) copy of each strictly partial node inserted in Item 1 captures the distance from (resp. to) the corresponding strictly partial node of $\overline{G}$.

Claim 7.1. *For every partial node $\overline{u}$ and strictly partial node $\overline{v}$ we have $d(\overline{u}, \overline{v}) = d(\overline{u}, \overline{v}^2)$ and $d(\overline{v}, \overline{u}) = d(\overline{v}^1, \overline{u})$.*

Proof. By construction, for every node $v \in V$ that strictly refines $\overline{v}$ (i.e., $v \sqsubset \overline{v}$), we have $\overline{\mathsf{wt}}(\overline{v}^1, v) = d(\overline{v}^1, v) = \overline{\mathbf{1}}$ and $\overline{\mathsf{wt}}(v, \overline{v}^2) = d(v, \overline{v}^2) = \overline{\mathbf{1}}$, i.e., every such v can reach (resp. be reached from) $\overline{v}^2$ (resp. $\overline{v}^1$) without changing the distance from $\overline{u}$. The claim follows easily. □

Key novelty and insights. The key novelty and insights of our algorithm are as follows:

1. A partial pair query can be answered by breaking it down to several pair queries. Instead, preprocessing the partial expansion of the concurrent graph allows to answer partial pair queries directly. Moreover, the partial expansion does not increase the asymptotic complexity of the preprocessing time and space.

Algorithm 17: ConcurPreprocess Item 1

Input: Graphs $(G_i = V_i, E_i)_{1 \leqslant i \leqslant k}$, a concurrent graph $G(V, E)$ of G_i's and a weight function $\mathsf{wt} : E \to \Sigma$

```
/* Construct the partial expansion G̅ of G                    */
```

1 Assign $\overline{V} \leftarrow V$

2 Assign $\overline{E} \leftarrow E$

3 Create a map $\overline{\mathsf{wt}} : \overline{E} \to \Sigma$

4 Assign $\overline{\mathsf{wt}} \leftarrow \mathsf{wt}$

5 **foreach** $u' \in \prod_i (V_i \cup \{\perp\})$ **do**

6   Let $u \in V$ such that $u \sqsubset u'$

7   Assign $\overline{V} \leftarrow \overline{V} \cup \left\{\overline{u}^1, \overline{u}^2\right\}$

8   Assign $\overline{E} \leftarrow \overline{E} \cup \left\{(\overline{u}^1, u), (u, \overline{u}^2)\right\}$

9   Set $\overline{\mathsf{wt}}(\overline{u}^1, u) \leftarrow \overline{\mathbf{1}}$

10   Set $\overline{\mathsf{wt}}(u, \overline{u}^2) \leftarrow \overline{\mathbf{1}}$

11 **end**

12 **return** $\overline{G} = (\overline{V}, \overline{E})$ and $\overline{\mathsf{wt}}$

Algorithm 18: ConcurPreprocess Item 2

Input: A tree-decomposition $T = \mathrm{Tree}(G) = (V_T, E_T)$ and the partial expansion $\overline{G} = (\overline{V}, \overline{E})$

```
/* Construct the tree-decomposition T̅ of G̅                   */
```

1 Assign $V_{\overline{T}} \leftarrow \varnothing$

2 **foreach** *bag* $B \in V_T$ **do**

3   Assign $\overline{B} \leftarrow B$ **foreach** $u \in B$ **do**

4   **foreach** $\overline{u} \in \overline{V}$ *such that* $u \sqsubset \overline{u}$ **do**

5   Assign $\overline{B} \leftarrow \overline{B} \cup \left\{\overline{u}^1, \overline{u}^2\right\}$

6   **end**

7   **end**

8   Assign $V_{\overline{T}} \leftarrow V_{\overline{T}} \cup \{\overline{B}\}$

9 **end**

10 **return** $\overline{T} = (V_{\overline{T}}, E_{\overline{T}})$

Algorithm 19: ConcurPreprocess Item 3

Input: The partial expansion tree-decomposition $\overline{T} = (\overline{V}_{\overline{T}}, \overline{E}_{\overline{T}})$, and weight function $\overline{\mathsf{wt}}$

```
/* Local distance computation */
```

1 **foreach** *partial node* $\overline{u}$ **do**
2   Create two maps $\textsc{Fwd}_{\overline{u}}, \textsc{Bwd}_{\overline{u}} : \overline{B}_{\overline{u}} \to \Sigma$
3   **for** $\overline{v} \in \overline{B}_{\overline{u}}$ **do**
4   Assign $\textsc{Fwd}_{\overline{u}}(\overline{v}) \leftarrow \overline{\mathsf{wt}}(\overline{u}, \overline{v})$
5   Assign $\textsc{Bwd}_{\overline{u}}(\overline{v}) \leftarrow \overline{\mathsf{wt}}(\overline{v}, \overline{u})$
6   **end**
7 **end**
8 **foreach** *bag* $\overline{B}$ *of* $\overline{T}$ *in bottom-up order* **do**
9   Assign $d' \leftarrow$ the transitive closure of $\overline{G}[\overline{B}]$ wrt $wt_{\overline{B}}$
10   **foreach** $\overline{u}, \overline{v} \in \overline{B}$ **do**
11   **if** $\mathsf{Lv}(\overline{v}) \leqslant \mathsf{Lv}(\overline{u})$ **then**
12   Assign $\textsc{Bwd}_{\overline{u}}(\overline{v}) \leftarrow d'(\overline{v}, \overline{u})$
13   Assign $\textsc{Fwd}_{\overline{u}}(\overline{v}) \leftarrow d'(\overline{u}, \overline{v})$
14   **end**
15   **end**
16 **end**
17 **foreach** *bag* $\overline{B}$ *of* $\overline{T}$ *in top-down order* **do**
18   Assign $d' \leftarrow$ the transitive closure of $\overline{G}[\overline{B}]$ wrt $wt_{\overline{B}}$
19   **foreach** $\overline{u}, \overline{v} \in \overline{B}$ **do**
20   **if** $\mathsf{Lv}(\overline{v}) \leqslant \mathsf{Lv}(\overline{u})$ **then**
21   Assign $\textsc{Bwd}_{\overline{u}}(\overline{v}) \leftarrow d'(\overline{v}, \overline{u})$
22   Assign $\textsc{Fwd}_{\overline{u}}(\overline{v}) \leftarrow d'(\overline{u}, \overline{v})$
23   **end**
24   **end**
25 **end**

Algorithm 20: ConcurPreprocess Item 4

Input: The partial expansion tree-decomposition $\overline{T} = (\overline{V}_{\overline{T}}, \overline{E}_{\overline{T}})$ and maps $\textsc{Fwd}_{\overline{u}}, \textsc{Bwd}_{\overline{u}} : \overline{B}_u \to \Sigma$ for every partial node $\overline{u}$

```
/* Ancestor distance computation                                    */
```

1 **foreach** *node* $u \in V$ **do**
2   Create two maps $\mathsf{F}_u, \mathsf{T}_u : \overline{\mathcal{V}}_{\overline{T}}(\overline{B}_u) \to \Sigma$
3 **end**
4 **foreach** *bag* $\overline{B}$ *of* $\overline{T}$ *in DFS order starting from the root* **do**
5   Let $\overline{B}'$ be the parent of $\overline{B}$
6   **foreach** *node* $u \in \overline{B} \cap V$ *such that* $\overline{B}$ *is the root of* u **do**
7   **foreach** $\overline{v} \in \overline{\mathcal{V}}_{\overline{T}}(\overline{B}_u)$ **do**
8   Assign $\mathsf{F}_u(\overline{v}) \leftarrow \bigoplus_{x \in \overline{B} \cap \overline{B}'} \textsc{Fwd}_u(x) \otimes wt^+(x, \overline{v})$
9   Assign $\mathsf{T}_u(\overline{v}) \leftarrow \bigoplus_{x \in \overline{B} \cap \overline{B}'} \textsc{Bwd}_u(x) \otimes wt^+(\overline{v}, x)$
10   **end**
11   **end**
12 **end**

2. ConcurPreprocess computes the transitive closure only during the local distance computation in each bag (Item 3 above), instead of a global computation on the whole graph. The key reason of our algorithmic improvement lies on the fact that the local computation is cheaper than the global computation, and is also sufficient to handle queries fast.

3. The third key aspect of our algorithm is the strongly balanced tree decomposition, which is crucially used in Theorem 7.1 to construct a tree decomposition for the concurrent graph such that the size of the bags decreases geometrically along the levels. By using the cheaper local distance computation (as opposed to the transitive closure globally) and recursing on a geometrically decreasing series we obtain the desired complexity bounds for our algorithm. Both the strongly balanced tree decomposition and the fast local distance computation play important roles in our algorithmic improvements.

We now turn our attention to the analysis of ConcurPreprocess.

Lemma 7.4. *$\overline{T}$ is a tree decomposition of the partial expansion $\overline{G}$.*

Proof. By Theorem 7.1, $\mathsf{ConcurTree}(G)$ is a tree decomposition of G. To show that $\overline{T}$ is a tree decomposition of the partial expansion $\overline{G}$, it suffices to show that the conditions C1-C3 are met for every pair of nodes $\overline{u}^1$, $\overline{u}^2$ that correspond to a strict partial node $\overline{u}$ of $\overline{G}$. We only focus on $\overline{u}^1$, as the other case is similar.

C1 This condition is met, as $\overline{u}^1$ appears in every bag of $\overline{T}$ that contains a node u that refines $\overline{u}^1$.

C2 Since every node $\overline{u}^1$ is connected only to nodes u of G that refine $\overline{u}$, this condition is also met.

C3 First, observe that $\overline{u}^1$ appears in the root bag $\overline{B}$ of $\overline{T}$. Then, for every simple path $P : \overline{B} \rightsquigarrow \overline{B}'$ from the root to some leaf bag $\overline{B}'$, if $\overline{B}''$ is the first bag in P where $\overline{u}^1$ does not appear, then some non-$\bot$ constituent of u does not appear in bags of $\overline{T}_{\overline{B}''}$, hence neither does $\overline{u}^1$. Thus, $\overline{u}^1$ appears in a contiguous subtree of $\overline{T}$.

The desired result follows. □

In Lemma 7.5 we establish that the forward and backward maps computed by $\mathsf{ConcurPreprocess}$ store the distances between nodes.

Lemma 7.5. *At the end of* $\mathsf{ConcurPreprocess}$*, the following assertions hold:*

1. *For all nodes $u, v \in V$ such that $\overline{B}_u$ appears in $\overline{T}(\overline{B}_v)$, we have $\mathsf{F}_u(v) = d(u, v)$ and $\mathsf{T}_u(v) = d(v, u)$.*

2. *For all strictly partial nodes $\overline{v} \in \overline{V}$ and nodes $u \in V$ we have $\mathsf{F}_u(\overline{v}^2) = d(u, \overline{v})$ and $\mathsf{T}_u(\overline{v}^1) = d(\overline{v}, u)$.*

3. *For all strictly partial nodes $\overline{u}, \overline{v} \in \overline{V}$ we have $\textsc{Fwd}_{\overline{u}^1}(\overline{v}^2) = d(\overline{u}, \overline{v})$ and $\textsc{Bwd}_{\overline{u}^2}(\overline{v}^1) = d(\overline{v}, \overline{u})$.*

Proof. We describe the key invariants that hold during the traversals of $\overline{T}$ by $\mathsf{ConcurPreprocess}$ in Items 3a, 3b and 4, after the algorithm processes a bag $\overline{B}$.

Item 3a For every pair of partial nodes $\overline{u}, \overline{v} \in \overline{B}$ such that $\mathsf{Lv}(\overline{v}) \leqslant \mathsf{Lv}(\overline{u})$ we have $\textsc{Fwd}_{\overline{u}}(\overline{v}) \preceq \bigoplus_{P_1} \otimes(P_1)$ and $\textsc{Bwd}_{\overline{u}}(\overline{v}) \preceq \bigoplus_{P_2} \otimes(P_2)$ where P_1 and P_2 are $\overline{u} \rightsquigarrow \overline{v}$ and $\overline{v} \rightsquigarrow \overline{u}$ paths respectively that only traverse nodes in $\overline{V}_{\overline{T}}(\overline{B})$. The statement follows by a straightforward

induction on the levels processed by the algorithm in the bottom-up pass. Note that if $\overline{u}$ and $\overline{v}$ are partial nodes in the root of $\overline{T}$, the statement yields $\textsc{Fwd}_{\overline{u}}(\overline{v}) = d(u, v)$ and $\textsc{Bwd}_{\overline{u}}(\overline{v}) = d(v, u)$.

Item 3b The invariant is similar to the previous, except that P_1 and P_2 range over all $\overline{u} \rightsquigarrow \overline{v}$ and $\overline{v} \rightsquigarrow \overline{u}$ paths in $\overline{G}$ respectively. Hence now $\textsc{Fwd}_{\overline{u}}(\overline{v}) = d(\overline{u}, \overline{v})$ and $\textsc{Bwd}_{\overline{u}}(\overline{v}) = d(\overline{v}, \overline{u})$. The statement follows by a straightforward induction on the levels processed by the algorithm in the top-down pass. Note that the base case on the root follows from the previous item, where the maps $\textsc{Bwd}$ and $\textsc{Fwd}$ store actual distances.

Item 4 For every node $u \in \overline{B}$ and partial node $\overline{v} \in \overline{\mathcal{V}}_{\overline{T}}(\overline{B})$ we have $\mathsf{F}_u(\overline{v}) = d(u, \overline{v})$ and $\mathsf{T}_u(\overline{v}) = d(\overline{v}, u)$. The statement follows from Lemma 2.3 and a straightforward induction on the length of the path from the root of $\overline{T}$ to the processed bag $\overline{B}$. Indeed, the statement is true when $\overline{B}$ is the root of the tree decomposition, which serves as the basis of the induction. This follows from the correctness Item 3b, as at this point the maps F and T restricted to nodes of $\overline{B}$ are identical to the maps $\textsc{Fwd}$ and $\textsc{Bwd}$ restricted to nodes of $\overline{B}$. For the inductive step, consider any bag $\overline{B}$, and assume that the statement holds for the parent bag $\overline{B}'$ of $\overline{B}$. Lemma 2.3 yields that for bag $\overline{B}'' \in \overline{\mathcal{V}}_{\overline{T}}(\overline{B})$, for every pair of partial nodes $\overline{u}, \overline{v}$ such that $\overline{u} \in \overline{B}$ and $\overline{v} \in \overline{B}''$, we have that

$$d(\overline{u}, \overline{v}) = \bigoplus_{\overline{w} \in \overline{B}'} \left(d(\overline{u}, \overline{w}) \otimes d(\overline{w}, \overline{v})\right)$$

By the induction hypothesis, the distances $d(\overline{w}, \overline{v})$ are found in the map F_u, whereas the distances $d(\overline{u}, \overline{w})$ are found, by the correctness of Item 3b in the maps $\textsc{Fwd}_u$ and $\textsc{Bwd}_u$. It follows that the algorithm combines the distances computed in these maps to compute the distance $d(\overline{u}, \overline{v})$.

Statement 1 of the lemma follows from Item 4. Similarly for statement 2, together with the observation that every strictly partial node $\overline{v}$ appears in the root of $\overline{T}$, and thus $\overline{v} \in \overline{\mathcal{V}}_{\overline{T}}(\overline{B}_u)$. Finally, statement 3 follows again from the fact that all strictly partial nodes appear in the root bag of $\overline{T}$. The desired result follows. $\square$

Complexity analysis. We now consider the complexity analysis of $\mathsf{ConcurPreprocess}$. Recall that $\mathsf{ConcurPreprocess}$ takes as part of its input a desired constant $0 < \epsilon \leqslant 1$. We choose a $\lambda \in \mathbb{N}$ and $\delta \in \mathbb{R}$ such that $\lambda \geqslant 4/\epsilon$ and $\delta \leqslant \epsilon/18$. Additionally, we set $\alpha = \frac{4 \cdot \lambda}{\delta}$, $\beta = \left(\frac{1+\delta}{2}\right)^{\lambda - 1}$

and $\gamma = \lambda$, which are the constants used for constructing an (α, β, γ) tree-decomposition $T_i = \mathrm{Tree}(G_i)$ in Item 2a of ConcurPreprocess. We start with a technical lemma on two recurrence relations, $\mathcal{T}_k$ and $\mathcal{S}_k$, which are parameterized by k, and will help us bound the time and space, respectively, spent by ConcurPreprocess.

Lemma 7.6. *Consider the recurrences in Eqs.* (7.11) *and* (7.12)*.*

$$\mathcal{T}_k(n) \leqslant n^{3\cdot(k-1)} + 2^{\lambda\cdot k} \cdot \mathcal{T}_k\left(n \cdot \left(\frac{1+\delta}{2}\right)^{\lambda-1}\right) \tag{7.11}$$

$$\mathcal{S}_k(n) \leqslant n^{2\cdot(k-1)} + 2^{\lambda\cdot k} \cdot \mathcal{S}_k\left(n \cdot \left(\frac{1+\delta}{2}\right)^{\lambda-1}\right) \tag{7.12}$$

Then

1. $\mathcal{T}_k(n) = O(n^{3\cdot(k-1)})$*, and*

2. *(i)* $\mathcal{S}_k(n) = O(n^{2\cdot(k-1)})$ *if* $k \geqslant 3$*, and (ii)* $\mathcal{S}_2(n) = O(n^{2+\epsilon})$*.*

Proof. We analyze each recurrence separately. First we consider Eq. (7.11). Note that

$$\left(n \cdot \left(\frac{1+\delta}{2}\right)^{\lambda-1}\right)^{3\cdot(k-1)} = \left(\frac{1+\delta}{2}\right)^{3\cdot(\lambda-1)\cdot(k-1)} \cdot n^{3\cdot(k-1)} \tag{7.13}$$

and

$$2^{\lambda\cdot k} \cdot \left(\frac{1+\delta}{2}\right)^{3\cdot(\lambda-1)\cdot(k-1)} = \frac{(1+\delta)^{3\cdot(\lambda-1)\cdot(k-1)}}{2^{2\cdot k\cdot\lambda+3\cdot(k+\lambda-1)}} \tag{7.14}$$

and since $\log(1+\delta) = \frac{\ln(1+\delta)}{\ln 2} < \frac{\delta}{\ln 2} < 2\cdot\delta$, we have

$$(1+\delta)^{3\cdot(\lambda-1)\cdot(k-1)} = 2^{\log(1+\delta)\cdot 3\cdot(\lambda-1)\cdot(k-1)} < 2^{6\cdot\delta\cdot(\lambda-1)\cdot(k-1)}$$

Hence the expression in Eq. (7.14) is bounded by 2^x with

$$\begin{aligned} x &\leqslant 6\cdot\delta\cdot(\lambda-1)\cdot(k-1) - 2\cdot k\cdot\lambda + 3\cdot(\lambda+k-1) \\ &= -2\cdot\lambda\cdot k\cdot(1-3\cdot\delta) + 3\cdot(\lambda+k-1)\cdot(1-2\cdot\delta) \end{aligned}$$

Let $f(k) = -2\cdot\lambda\cdot k\cdot(1-3\cdot\delta) + 3\cdot(\lambda+k-1)\cdot(1-2\cdot\delta)$ and note that since $\lambda \geqslant \frac{4}{\epsilon} \geqslant 4$ and $\delta \leqslant \frac{\epsilon}{18} \leqslant \frac{1}{18}$, $f(k)$ is decreasing, and thus maximized for $k = 2$, for which we obtain $f(2) = -4\cdot\lambda\cdot(1-3\cdot\delta) + 3\cdot(\lambda+1)\cdot(1-2\cdot\delta) = -\lambda\cdot(1-6\cdot\delta) \leqslant 0$ as $\delta \leqslant \frac{1}{18}$. It follows that there exists a constant $c < 1$ for which

$$2^{\lambda\cdot k} \cdot \mathcal{T}_k\left(n \cdot \left(\frac{1+\delta}{2}\right)^{\lambda-1}\right) \leqslant c \cdot n^{3\cdot(k-1)}$$

which yields that Eq. (7.11) follows a geometric series, and thus $\mathcal{T}_k(n) = O(n^{3\cdot(k-1)})$.

We now turn our attention to Eq. (7.12). When $k \geqslant 3$, an analysis similar to Eq. (7.11) yields the bound $O(n^{2\cdot(k-1)})$. When $k = 2$, since $\epsilon > 0$, we write Eq. (7.12) as

$$\mathcal{S}_2(n) \leqslant n^{2+\epsilon} + 2^{2\cdot\lambda} \cdot \mathcal{S}_2\left(n \cdot \left(\frac{1+\delta}{2}\right)^{\lambda-1}\right) \tag{7.15}$$

Similarly as above, we have

$$\left(n \cdot \left(\frac{1+\delta}{2}\right)^{\lambda-1}\right)^{2+\epsilon} = \left(\frac{1+\delta}{2}\right)^{(2+\epsilon)\cdot(\lambda-1)} \cdot n^{2+\epsilon} \tag{7.16}$$

and

$$2^{2\cdot\lambda} \cdot \left(\frac{1+\delta}{2}\right)^{(2+\epsilon)\cdot(\lambda-1)} = \frac{(1+\delta)^{(2+\epsilon)\cdot(\lambda-1)}}{2^{-2+\epsilon\cdot(\lambda-1)}} \tag{7.17}$$

and since $\log(1+\delta) = \frac{\ln(1+\delta)}{\ln 2} < \frac{\delta}{\ln 2} < 2\cdot\delta$, we have

$$(1+\delta)^{(2+\epsilon)\cdot(\lambda-1)} < 2^{2\cdot\delta\cdot(2+\epsilon)\cdot(\lambda-1)}$$

Hence the expression in Eq. (7.17) is bounded by 2^x with

$$\begin{aligned}
x &\leqslant 2\cdot\delta\cdot(2+\epsilon)\cdot(\lambda-1) + 2 - \epsilon\cdot(\lambda-1)\\
&= (\lambda-1)\cdot(2\cdot\delta\cdot(2+\epsilon)-\epsilon)+2\\
&\leqslant (\lambda-1)\cdot\frac{4\cdot\epsilon+2\cdot\epsilon^2-18\cdot\epsilon}{18}+2\\
&\leqslant (1-\lambda)\cdot\epsilon\cdot\frac{2}{3}+2\\
&\leqslant -(4-\epsilon)\cdot\frac{2}{3}+2 \leqslant 0
\end{aligned}$$

since $\delta \leqslant \frac{\epsilon}{18}$ and $\lambda \geqslant \frac{4}{\epsilon}$ and $\epsilon \leqslant 1$. It follows that there exists a constant $c < 1$ for which

$$2^{2\cdot\lambda}\cdot\mathcal{S}_2(n) \leqslant c\cdot n^{2+\epsilon}$$

which yields that Eq. (7.15) follows a geometric series, and thus $\mathcal{S}_2(n) = O(n^{2+\epsilon})$. □

The following lemma analyzes the complexity of ConcurPreprocess, and makes use of the above recurrences.

Lemma 7.7. ConcurPreprocess *requires* $O(n^{2\cdot k-1})$ *space and*

1. $O(n^{3\cdot(k-1)})$ *time if* $k \geqslant 3$*, and*

2. *$O(n^{3+\epsilon})$ time if $k=2$.*

Proof. We examine each step of the algorithm separately.

1. The time and space required for this step is bounded by the number of nodes introduced in the partial expansion $\overline{G}$, which is $2 \cdot \sum_{i<k} \binom{n}{i} = O(n^{k-1})$.

2. By Theorem 7.1, $\mathsf{ConcurTree}(G)$ is constructed in $O(n^k)$ time and space. In $\overline{T}$, the size of each bag $\overline{B}$ is increased by constant factor, hence this step requires $O(n^k)$ time and space.

3. In each pass, $\mathsf{ConcurPreprocess}$ spends $|\overline{B}|^3$ time to perform an all-pairs semiring distance computation in each bag $\overline{B}$ of $\overline{T}$ [Lehmann, 1977; Floyd, 1962; Warshall, 1962; Kleene, 1956]. The space usage for storing all maps $\textsc{Fwd}_{\overline{u}}$ and $\textsc{Bwd}_{\overline{u}}$ for every node $\overline{u}$ whose root bag is $\overline{B}$ is $O(|\overline{B}|^2)$, since there are at most $|\overline{B}|$ such nodes $\overline{u}$, and each map has size $|\overline{B}|$. By the previous item, we have $|\overline{B}| = O(|B|)$, where B is the corresponding bag of T before the partial expansion of G. By Theorem 7.1, we have $|B| = O(n^{k-1} \cdot \beta^i)$, where $\mathsf{Lv}(B) \geqslant i \cdot \gamma = i \cdot \lambda$, and $\beta = \left(\frac{1+\delta}{2}\right)^{\lambda-1}$. Then, since $\overline{T}$ is a full 2^k-ary tree, the time and space required for preprocessing every $\gamma = \lambda$ levels of $\overline{T}$ is given by the following recurrences respectively (ignoring constant factors for simplicity).

$$\mathcal{T}_k(n) \leqslant n^{3\cdot(k-1)} + 2^{\lambda \cdot k} \cdot \mathcal{T}_k\left(n \cdot \left(\frac{1+\delta}{2}\right)^{\lambda-1}\right)$$
$$\mathcal{S}_k(n) \leqslant n^{2\cdot(k-1)} + 2^{\lambda \cdot k} \cdot \mathcal{S}_k\left(n \cdot \left(\frac{1+\delta}{2}\right)^{\lambda-1}\right)$$

By the analysis of Eqs. (7.11) and (7.12) of Lemma 7.6, we have that $\mathcal{T}_k(n) = O(n^{3\cdot(k-1)})$ and (i) $\mathcal{S}_k(n) = O(n^{2\cdot(k-1)})$ if $k \geqslant 3$, and (ii) $\mathcal{S}_2(n) = O(n^{2+\epsilon})$.

4. We first focus on the space usage. Let $\overline{B}_u^i$ denote the ancestor bag of $\overline{B}_u$ at level i. We have

$$|\overline{\mathcal{V}}_{\overline{T}}(\overline{B}_u)| \leqslant \sum_i |\overline{B}_u^i| \leqslant c_1 \cdot \sum_i |\overline{B}_u^{\lfloor \frac{i}{\gamma} \rfloor}| \leqslant c_2 \cdot \sum_i |B_u^{\lfloor \frac{i}{\gamma} \rfloor}|$$
$$\leqslant c_3 \cdot \sum_i \left(n^{k-1} \cdot \beta^i\right) = O(n^{k-1})$$

for some constants c_1, c_2, c_3. The first inequality comes from expressing the size of all (constantly many) ancestors $\overline{B}_u^i$ with $\lfloor \frac{i}{\gamma} \rfloor = j$ as a constant factor the size of $\overline{B}_u^{\lfloor \frac{i}{\gamma} \rfloor}$. The

second inequality comes from Item 1 of this lemma, which states that $O(|\overline{B}|) = O(|B|)$ for every bag $\overline{B}$. The third inequality comes from Theorem 7.1. The final equality holds because β is a constant, and thus the sum forms a geometric series. By Item 2, there are $O(n^k)$ such nodes u in $\overline{T}$, hence the space required is $O(n^{2\cdot k-1})$.

We now turn our attention to the time requirement. For every bag $\overline{B}$, the algorithm requires $O(|\overline{B}|^2)$ time to iterate over all pairs of nodes u and x in Eqs. (7.9) and (7.10) to compute the values $\mathsf{F}_u(\overline{v})$ and $\mathsf{T}_u(\overline{v})$ for every $\overline{v} \in \overline{\mathcal{V}}_{\overline{T}}(\overline{B})$. Hence the time required for all nodes u and one partial node $\overline{v} \in \overline{\mathcal{V}}_{\overline{T}}(\overline{B})$ to store the maps values $\mathsf{F}_u(\overline{v})$ and $\mathsf{T}(\overline{v})$ is given by the recurrence

$$\mathcal{T}_k(n) \leqslant n^{2\cdot(k-1)} + 2^{\lambda\cdot k} \cdot \mathcal{T}_k\left(n \cdot \left(\frac{1+\delta}{2}\right)^{\lambda-1}\right)$$

The analysis of Eqs. (7.11) and (7.12) of Lemma 7.6 gives $\mathcal{T}_k(n) = O(n^{2\cdot(k-1)})$ for $k \geqslant 3$ and $\mathcal{T}_2(n) = O(n^{2+\epsilon})$ (i.e., the above time recurrence is analyzed as the recurrence for $\mathcal{S}_k$ of Lemma 7.6). From the space analysis we have that there exist $O(n^{k-1})$ partial nodes $\overline{v} \in \overline{\mathcal{V}}_{\overline{T}}(\overline{B})$ for every node u whose root bag is $\overline{B}$. Hence the total time for this step is $O(n^{3\cdot(k-1)})$ for $k \geqslant 3$, and $O(n^{3+\epsilon})$ for $k = 2$.

5. This step requires time linear in the size of $\overline{T}$ [Harel and Tarjan, 1984].

The desired result follows. □

Algorithm ConcurQuery**.** In the query phase, ConcurSD answers distance queries using the algorithm ConcurQuery. We distinguish three cases, according to the type of the query.

1. *Single-source query.* Given a source node u, initialize a map data structure $A : V \to \Sigma$, and initially set $A(v) = \text{FWD}_u(v)$ for all $v \in \overline{B}_u$, and $A(v) = \overline{\mathbf{0}}$ for all other nodes $v \in V \setminus \overline{B}_u$. Perform a BFS on $\overline{T}$ starting from $\overline{B}_u$, and for every encountered bag $\overline{B}$ and nodes $x, v \in \overline{B}$ with $\mathsf{Lv}(v) \leqslant \mathsf{Lv}(x)$, set $A(v) = A(v) \oplus (A(x) \otimes \text{FWD}_x(v))$. Return the map A.

2. *Pair query.* Given two nodes $u, v \in V$, find the LCA $\overline{B}$ of bags $\overline{B}_u$ and $\overline{B}_v$. Return $\bigoplus_{x\in\overline{B}\cap V}(\mathsf{F}_u(x) \otimes \mathsf{T}_v(x))$.

3. *Partial pair query.* Given two partial nodes $\overline{u}, \overline{v}$,

   (a) If both $\overline{u}$ and $\overline{v}$ are strictly partial, return $\text{FWD}_{\overline{u}^1}(\overline{v}^2)$, else

(b) If $\overline{u}$ is strictly partial, return $\mathsf{T}_v(\overline{u}^1)$, else

(c) Return $\mathsf{F}_u(\overline{v}^2)$.

See Algorithms 21 to 23 for the formal description of the above steps. We thus establish the following theorem.

Algorithm 21: ConcurQuery Single-source query

Input: A source node $u \in V$

Output: A map $A : V \to \Sigma$ that contains distances of nodes from u

1 Create a map $A : V \to \Sigma$
2 **for** $v \in V$ **do**
3   Assign $A(v) \leftarrow \overline{\mathbf{0}}$
4 **end**
5 **for** *every bag* $\overline{B}$ *of* $\overline{T}$ *in BFS order starting from* $\overline{B}_u$ **do**
6   **for** $x, v \in \overline{B} \cap V$ **do**
7   **if** $\mathsf{Lv}(v) \leqslant \mathsf{Lv}(x)$ **then**
8   Assign $A(v) \leftarrow A(v) \oplus A(x) \otimes \textsc{Fwd}_x(v)$
9   **end**
10   **end**
11 **end**
12 **return** A

Algorithm 22: ConcurQuery Pair query

Input: Two nodes $u, v \in V$

Output: The distance $d(u, v)$

1 Let $\overline{B} \leftarrow$ the LCA of $\overline{B}_u$ and $\overline{B}_v$ in $\overline{T}$
2 Assign $d \leftarrow \overline{\mathbf{0}}$
3 **for** $x \in \overline{B} \cap V$ **do**
4   Assign $d \leftarrow d \oplus \textsc{Fwd}^+_u(x) \otimes \textsc{Bwd}^+_v(x)$
5 **end**
6 **return** d

Theorem 7.2. *Let* $G = (V, E)$ *be a concurrent graph of* k *constant-treewidth graphs* $(G_i)_{1 \leqslant i \leqslant k}$, *and* $\mathsf{wt} : E \to \Sigma$ *a weight function of* G. *For any fixed* $\epsilon > 0$, *the data structure* ConcurSD *correctly answers single-source and pair semiring distance queries and requires:*

Algorithm 23: ConcurQuery Partial pair query

Input: Two partial nodes $\overline{u}, \overline{v} \in \overline{V}$, at least one of which is strictly partial

Output: The distance $d(\overline{u}, \overline{v})$

1 **if** *both* $\overline{u}$ *and* $\overline{v}$ *are strictly partial* **then**
2   **return** $\text{FWD}_{\overline{u}^1}(\overline{v}^2)$
3 **else if** $\overline{u}$ *is strictly partial* **then**
4   **return** $\text{BWD}^+_v(\overline{u}^1)$
5 **else**
6   **return** $\text{FWD}^+_u(\overline{v}^2)$
7 **end**

1. *Preprocessing time*
   (a) $O(n^{3\cdot(k-1)})$ *if* $k \geqslant 3$*, and*
   (b) $O(n^{3+\epsilon})$ *if* $k = 2$.
2. *Preprocessing space* $O(n^{2\cdot k-1})$.
3. *Single-source query time*
   (a) $O(n^{2\cdot(k-1)})$ *if* $k \geqslant 3$*, and*
   (b) $O(n^{2+\epsilon})$ *if* $k = 2$.
4. *Pair query time* $O(n^{k-1})$.
5. *Partial pair query time* $O(1)$.

Proof. The correctness of ConcurQuery for handling all queries follows from Lemma 2.3 and the properties of the preprocessing established in Lemma 7.5. The preprocessing complexity is stated in Lemma 7.7. The time complexity for the single-source query comes from the observation that ConcurQuery spends quadratic time in each encountered bag, and the result follows from the recurrence analysis of Eq. (7.12) in Lemma 7.6. The time complexity for the pair query follows from the $O(1)$ time to access the LCA bag $\overline{B}$ of $\overline{B}_u$ and $\overline{B}_v$, and the $O(|\overline{B}|) = O(n^{k-1})$ time

required to iterate over all nodes $x \in \overline{B} \cap V$. Finally, the time complexity for the partial pair query follows from the $O(1)$ time lookup in the constructed maps FWD, F and T. □

Note that a single-source query from a strictly partial node $\overline{u}$ can be answered in $O(n^k)$ time by breaking it down to n^k partial pair queries. The most common case in analysis of concurrent programs is that of two threads, for which we obtain the following corollary.

Corollary 7.1. *Let $G = (V, E)$ be a concurrent graph of two constant-treewidth graphs G_1, G_2, and* $\mathsf{wt} : E \to \Sigma$ *a weight function of G. For any fixed $\epsilon > 0$, the data structure* ConcurSD *correctly answers single-source and pair queries and requires:*

1. *Preprocessing time $O(n^{3+\epsilon})$.*

2. *Preprocessing space $O(n^3)$.*

3. *Single-source query time $O(n^{2+\epsilon})$.*

4. *Pair query time $O(n)$.*

5. *Partial pair query time $O(1)$.*

Remark 7.2. In contrast to Corollary 7.1, the existing methods for handling even one pair query require hexic time and quartic space [Lehmann, 1977; Floyd, 1962; Warshall, 1962; Kleene, 1956] by computing the transitive closure. While our improvements are most significant for semiring distance queries, they imply improvements also for special cases like reachability (expressed in Boolean semirings). For reachability, the complete preprocessing requires quartic time, and without preprocessing every query requires quadratic time. In contrast, with almost cubic preprocessing we can answer pair (resp., partial pair) queries in linear (resp. constant) time.

Note that Item 4 of ConcurPreprocess is required for handling pair queries only. By skipping this step, we can handle every (partial) pair query $\overline{u}, \overline{v}$ similarly to the single source query from $\overline{u}$, but restricting the BFS to the path $P : \overline{B}_{\overline{u}} \rightsquigarrow \overline{B}_{\overline{v}}$, and spending $O(|\overline{B}|^2)$ time for each bag $\overline{B}$ of P. Recall (Theorem 7.1) that the size of each bag B in T (and thus the size of the corresponding bag $\overline{B}$ in $\overline{T}$) decreases geometrically every γ levels. Then, the time required for this operation is $O(|\overline{B}'|^2) = O(n^2)$, where $\overline{B}'$ is the bag of P with the smallest level. This leads to the following corollary.

Corollary 7.2. *Let $G = (V, E)$ be a concurrent graph of two constant-treewidth graphs G_1, G_2, and* $\mathsf{wt} : E \to \Sigma$ *a weight function of G. For any fixed ϵ, the data structure* ConcurSD *(by skipping Item 4 in* ConcurPreprocess*) correctly answers single-source and pair queries and requires:*

1. *Preprocessing time $O(n^3)$.*

2. *Preprocessing space $O(n^{2+\epsilon})$.*

3. *Single-source query time $O(n^{2+\epsilon})$.*

4. *Pair and partial pair query time $O(n^2)$.*

Finally, we can use ConcurSD to obtain the transitive closure of G by performing n^2 single-source queries. The preprocessing space is $O(n^{2+\epsilon})$ by Corollary 7.2, and the space of the output is $O(n^4)$, since there are n^4 pairs for the computed distances. Hence the total space requirement is $O(n^4)$. The time requirement is $O(n^{4+\epsilon})$, since by Corollary 7.2, every single-source query requires $O(n^{2+\epsilon})$ time. We obtain the following corollary.

Corollary 7.3. *Let $G = (V, E)$ be a concurrent graph of two constant-treewidth graphs G_1, G_2, and* $\mathsf{wt} : E \to \Sigma$ *a weight function of G. For any fixed $\epsilon > 0$, the transitive closure of G wrt* wt *can be computed in $O(n^{4+\epsilon})$ time and $O(n^4)$ space.*

7.6 Conditional Optimality for Two Graphs

In the current section we establish the optimality of Corollary 7.2 in handling semiring distance queries in a concurrent graph that consists of two constant-treewidth components. The key idea is to show that for any arbitrary graph (i.e., without the constant-treewidth restriction) G of n nodes, we can construct a concurrent graph G' as a 2-self-concurrent asynchronous composition of a constant-treewidth graph G'' of $2 \cdot n$ nodes, such that semiring queries in G coincide with semiring queries in G'.

Arbitrary graphs as composition of two constant-treewidth graphs. We fix an arbitrary graph $G = (V, E)$ of n nodes, and a weight function $\mathsf{wt} : E \to \Sigma$. Let x_i, $1 \leqslant i \leqslant n$ range over

the nodes V of G, and construct a graph $G''=(V'',E'')$ such that $V''=\{x_i,y_i:\ 1\leqslant i\leqslant n\}$ and $E''=\{(x_i,y_i),(y_i,x_i):\ 1\leqslant i\leqslant n\}\cup\{(y_i,y_{i+1}),(y_{i+1},y_i):\ 1\leqslant i<n\}$.

Claim 7.2. *The treewidth of G'' is* 1.

Proof. Observe that if we (i) ignore the direction of the edges and (ii) remove multiple appearances of the same edge, we obtain a tree. It is known that trees have treewidth 1. □

Given G'', we construct a graph G' as a 2-self-concurrent asynchronous composition of G''. Informally, a node x_i of G corresponds to the node $\langle x_i,x_i\rangle$ of G'. An edge (x_i,x_j) in G is simulated by two paths in G'.

1. The first path has the form $P_1:\langle x_i,x_i\rangle\rightsquigarrow\langle x_i,x_j\rangle$, and is used to witness the weight of the edge in G, i.e., $\mathsf{wt}(x_i,x_j)=\otimes(P_1)$. It traverses a sequence of nodes, where the first constituent is fixed to x_i, and the second constituent forms the path $x_i\rightarrow y_i\rightarrow y_{i'}\rightarrow\cdots\rightarrow y_j\rightarrow x_j$. The last transition will have weight equal to $\mathsf{wt}(x_i,x_j)$, and the other transitions have weight $\overline{\mathbf{1}}$. Any path that has the above form can be taken as P_1.

2. The second path has the form $P_2:\langle x_i,x_j\rangle\rightsquigarrow\langle x_j,x_j\rangle$, it has no weight (i.e., $\otimes(P_2)=\overline{\mathbf{1}}$), and is used to reach the node $\langle x_j,x_j\rangle$. It traverses a sequence of nodes, where the second constituent is fixed to x_j, and the first constituent forms the path $x_i\rightarrow y_i\rightarrow y_{i'}\rightarrow\cdots\rightarrow y_j\rightarrow x_j$. Any path that has the above form can be taken as P_2.

Then the concatenation of P_1 and P_2 creates a path $P:\langle x_i,x_i\rangle\rightsquigarrow\langle x_j,x_j\rangle$ with $\otimes(P)=\otimes(P_1)\otimes\otimes(P_2)=\mathsf{wt}(x_i,x_j)\otimes\overline{\mathbf{1}}=\mathsf{wt}(x_i,x_j)$.

Formal construction. We construct a graph $G'=(V',E')$ as a 2-self-concurrent asynchronous composition of G'', by including the following edges.

1. *Black edges.* For all $1\leqslant i\leqslant n$ and $1\leqslant j<n$ we have $(\langle x_i,y_j\rangle,\langle x_i,y_{j+1}\rangle),(\langle x_i,y_{j+1}\rangle,\langle x_i,y_j\rangle)\in E'$, and for all $1\leqslant i<n$ and $1\leqslant j\leqslant n$ we have $(\langle y_i,x_j\rangle,\langle y_{i+1},x_j\rangle),(\langle y_{i+1},x_j\rangle,\langle y_i,x_j\rangle)\in E'$.

2. *Blue edges.* For all $1\leqslant i\leqslant n$ we have $(\langle x_i,x_i\rangle,\langle x_i,y_i\rangle),(\langle y_i,x_i\rangle,\langle x_i,x_i\rangle)\in E'$.

3. *Red edges.* For all $(x_i,x_j)\in E$ we have $(\langle x_i,y_j\rangle,\langle x_i,x_j\rangle)\in E'$.

4. *Green edges.* For all $1\leqslant i,j\leqslant n$ with $i\neq j$ we have $(\langle x_i,x_j\rangle,\langle y_i,x_j\rangle)\in E'$.

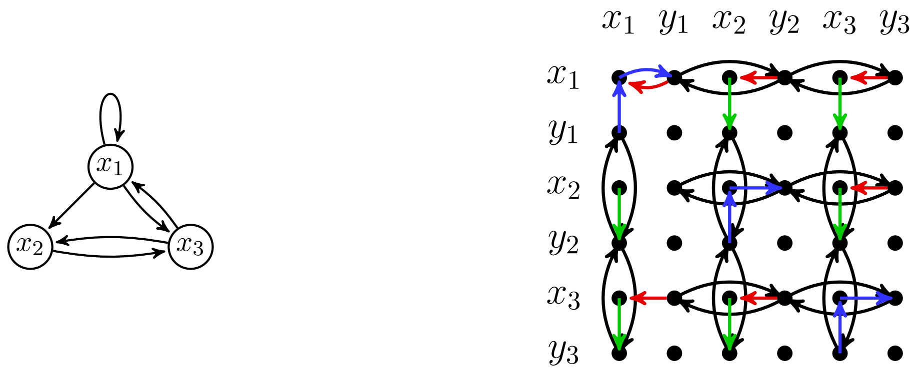


Figure 7.4: A graph G (left), and G' that is a 2-self-product of a graph G'' of treewidth 1 (right). The weighted edges of G correspond to weighted red edges on G'. The distance $d(x_i, x_j)$ in G equals the distance $d(\langle x_i, x_i\rangle, \langle x_j, x_j\rangle) = d(\langle \bot, x_i\rangle, \langle \bot, x_j\rangle)$ in G'.

Additionally, we construct a weight function such that $\mathsf{wt}'(\langle x_i, y_j\rangle, \langle x_i, x_j\rangle) = \mathsf{wt}(x_i, x_j)$ for every red edge $(\langle x_i, y_j\rangle, \langle x_i, x_j\rangle)$, and $\mathsf{wt}'(u, v) = \overline{\mathbf{1}}$ for every other edge (u, v). Fig. 7.4 provides an illustration of the construction.

Lemma 7.8. *For every $x_i, x_j \in V$, there exists a path $P : x_i \rightsquigarrow x_j$ with $\otimes(P) = z$ in G iff there exists a path $P' : \langle x_i, x_i\rangle \rightsquigarrow \langle x_j, x_j\rangle$ with $\otimes(P') = z$ in G'.*

Proof. Recall that only red edges contribute to the weights of paths in G'. We argue that there is path $\overline{P} : \langle x_i, x_i\rangle \rightsquigarrow \langle x_j, x_j\rangle$ in G' that traverses a single red edge iff there is an edge (x_i, x_j) in G with $\otimes(\overline{P}) = \mathsf{wt}(x_i, x_j)$.

1. Given the edge (x_i, x_j), the path $\overline{P}$ is formed by traversing the red edge $(\langle x_i, y_j\rangle, \langle x_i, x_j\rangle)$ as

$$\langle x_i, x_i\rangle \to \langle x_i, y_i\rangle \rightsquigarrow \langle x_i, y_j\rangle \to \langle x_i, x_j\rangle \to \langle y_i, x_j\rangle \rightsquigarrow \langle y_i, x_j\rangle \to \langle x_j, y_j\rangle$$

   Since $\mathsf{wt}((\langle x_i, y_j\rangle, \langle x_i, x_j\rangle)) = \mathsf{wt}(x_i, x_j)$ and all other edges of $\overline{P}$ have weight $\overline{\mathbf{1}}$, we have that $\otimes(\overline{P}) = \mathsf{wt}(x_i, x_j)$.

2. Every path $\overline{P}$ that traverses a red edge $\langle x_{i'}, y_{j'}\rangle \to \langle x_{i'}, x_{j'}\rangle$ has to traverse a blue edge to $\langle x_{j'}, x_{j'}\rangle$. Then $x_{j'}$ must be x_j, otherwise $\overline{P}$ will traverse a second red edge before reaching $\langle x_j, x_j\rangle$.

The result follows easily from the above. □

Lemma 7.8 implies that for every $x_i, x_j \in V$, we have $d(x_i, x_j) = d(\langle x_i, x_i \rangle, \langle x_j, x_j \rangle)$, i.e., pair queries in G for nodes x_i, x_j coincide with pair queries $(\langle x_i, x_i \rangle, \langle x_j, x_j \rangle)$ in G'. Observe that in G' we have $d(\langle x_i, x_i \rangle, \langle x_j, x_j \rangle) = d(\langle \bot, x_i \rangle, \langle \bot, x_j \rangle)$, and hence pair queries in G also coincide with partial pair queries in G'.

Theorem 7.3. *For every graph $G = (V, E)$ and weight function $\mathsf{wt} : E \to \Sigma$ there exists a graph $G' = (V \times V, E')$ that is a 2-self-concurrent asynchronous composition of a constant-treewidth graph, together with a weight function $\mathsf{wt}' : E' \to \Sigma$, such that for all $u, v \in V$, and $\langle u, u \rangle, \langle v, v \rangle \in V'$ we have $d(u, v) = d(\langle u, u \rangle, \langle v, v \rangle) = d(\langle \bot, u \rangle, \langle \bot, v \rangle)$. Moreover, the graph G' can be constructed in quadratic time in the size of G.*

This leads to the following corollary.

Corollary 7.4. *Let $\mathcal{T}_S(n) = \Omega(n^2)$ be a lower bound on the time required to answer a single semiring distance query wrt to a semiring S on arbitrary graphs of n nodes. Consider any concurrent graph G which is an asynchronous self-composition of two constant-treewidth graphs of n nodes each. For any data structure DS, let $\mathcal{T}_{\mathsf{DS}}(G, r)$ be the time required by DS to preprocess G and answer r pair queries. We have $\mathcal{T}_{\mathsf{DS}}(G, 1) = \Omega(\mathcal{T}_S(n))$.*

Conditional optimality of Corollary 7.2. Note that for $r = O(n)$ pair queries, Corollary 7.2 yields that the time spent by our data structure ConcurSD for preprocessing G and answering r queries is $\mathcal{T}_{\mathsf{ConcurSD}}(G, r) = O(n^3)$. The long-standing (over five decades) upper bound for answering even one pair query for semiring distances in arbitrary graphs of n nodes is $O(n^3)$. Theorem 7.3 implies that any improvement upon our results would yield the same improvement for the long-standing upper bound, which would be a major breakthrough.

Almost-optimality of Theorem 7.2 and Corollary 7.3. Finally, we highlight some almost-optimality results obtained by variants of ConcurSD for the case of two graphs. By almost-optimality we mean that the obtained bounds are $O(n^{\epsilon})$ factor worse that optimal, for any fixed $\epsilon > 0$ arbitrarily close to 0.

1. According to Theorem 7.2, after $O(n^{3+\epsilon})$ preprocessing time, single-source queries are handled in $O(n^{2+\epsilon})$ time, and partial pair queries in $O(1)$ time. The former (resp. later) query time is almost linear (resp. exactly linear) in the size of the output. Hence the former queries are handled almost-optimally, and the latter indeed optimally. Moreover, this is achieved using $O(n^{3+\epsilon})$ preprocessing time, which is far less than the $\Omega(n^4)$ time required

for the transitive closure computation (which computes the distance between all n^4 pairs of nodes).

2. According to Corollary 7.3, the transitive closure can be computed in $O(n^{4+\epsilon})$ time, for any fixed $\epsilon > 0$, and $O(n^4)$ space. Since the size of the output is $\Theta(n^4)$, the transitive closure is computed in almost-optimal time and optimal space.

7.7 Experimental Results

In the current section we report on experimental evaluation of our algorithms, in particular of the algorithms of Corollary 7.3. We have tested their performance for obtaining the transitive closure on various concurrent graphs. We have focused on the transitive closure for a fair comparison with the existing algorithmic methods, which compute the transitive closure even for a single query. Since the contributions of this chapter are algorithmic improvements for semiring distance queries, we considered the most fundamental representative of this framework, namely, the shortest path problem. Our comparison is done against the standard Bellman-Ford algorithm, which (i) has the best worst-case complexity for the problem, and (ii) allows for practical improvements, such as early termination.

Basic setup. We outline the basic setup used in all experiments. We have used two different sets of benchmarks, and obtain the control flow graphs of Java programs using Soot [Vallée-Rai *et al.*, 1999], and use LibTW [van Dijk *et al.*, 2006b] to obtain the tree decompositions of the corresponding graphs. For every obtained graph G', we have constructed a concurrent graph G as a 2-self asynchronous composition of G', and then assign random integer weights in the range $[-10^3, 10^3]$, without negative cycles. Although this last restriction does not affect the running time of our algorithms, it allows for early termination of the Bellman-Ford algorithm (and thus only benefits the latter). The 2-self composition is a natural situation arising in practice, e.g. in concurrent data structures where two threads of the same method access the data structure. We note that the 2-self composition is no simpler than the composition of any two constant-treewidth graphs, (recall that the lower-bound of Section 7.6 is established on a 2-self composition).

DaCapo benchmarks. In our first setup, we extracted control flow graphs of methods from the DaCapo suit [Blackburn, 2006]. The average treewidth of the input graphs was around 6. This

λ	2	3	4	5	6	7	8
%	6	7	16	22	25	57	17

Table 7.2: Percentage of cases for which the transitive closure of the graph G for the given value of λ is at most 5% slower than the time required to obtain the transitive closure of G for the best λ.

supplied a large pool of 120 concurrent graphs, for which we used Corollary 7.3 to compute the transitive closure. This allowed us to test the scalability of our algorithms, as well as their practical dependence on input parameters. Recall that our transitive closure time complexity is $O(n^{4+\epsilon})$, for any fixed $\epsilon > 0$, which is achieved by choosing a sufficiently large $\lambda \in \mathbb{N}$ and a sufficiently small $\delta \in \mathbb{R}$ when running the algorithm of Theorem Theorem 2.2. We computed the transitive closure for various λ. In practice, δ has effects only for very large input graphs. For this, we fixed it to a large value ($\delta = \frac{1}{3}$) which can be proved to have no effect on the obtained running times. Table 7.2 shows for each value of λ, the percentage of cases for which that value is at most 5% slower than the smallest time (among all tested λ) for each examined case. We find that $\lambda = 7$ works best most of the time.

Fig. 7.5 shows the time required to compute the transitive closure on each concurrent graph G by our algorithm (for $\lambda = 7$) and the baseline Bellman-Ford algorithm. We see that our algorithm significantly outperforms the baseline method. Note that our algorithm seems to scale much better than its theoretical worst-case bound of $O(n^{4+\epsilon})$ of Corollary 7.3.

Concurrency with locks. Our second set of experiments is on methods from containers of the java.util.concurrent library that use locks as their synchronization mechanism. The average treewidth of the input graphs was around 8. In this case, we expanded the node set of the concurrent graph G with the lock set $[3]^{\ell}$, where ℓ is the number of locks used by G'. Intuitively, the i-th value of the lock set denotes which of the two components owns the i-th lock (the value is 3 if the lock is free). Transitions to nodes that perform lock operations are only allowed wrt the lock semantics. That is, a transition to a node of G where the value of the i-th lock is

1. *(Lock aquire):* $j \in [2]$, is only allowed from nodes where the value of that lock is 3, and the respective graph G_j is performing a lock operation on that edge.

2. *(Lock release):* 3, is only allowed from nodes where the value of that lock is $j \in [2]$, and

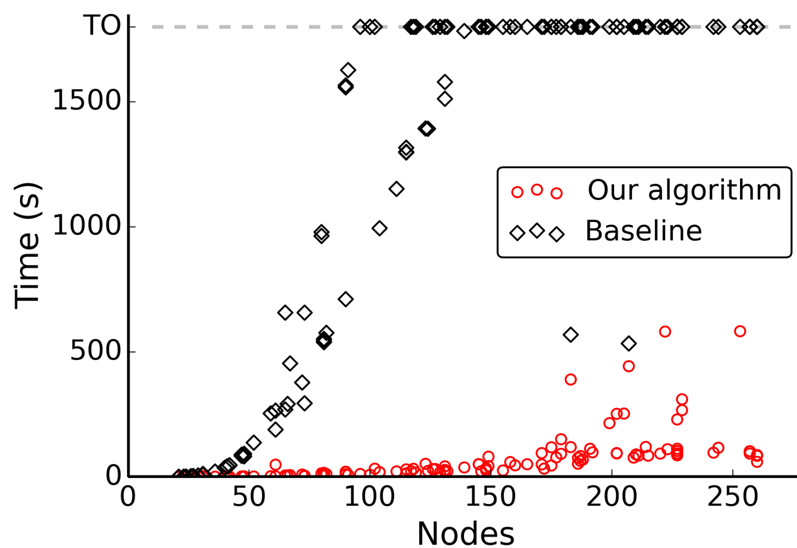


Figure 7.5: Time required to compute the transitive closure on concurrent graphs of various sizes. Our algorithm is run for $\lambda = 7$. TO denotes that the computation timed out after 30 minutes.

the respective graph G_j is performing an unlock operation on that edge.

Similarly as before, we compared our transitive closure time with the standard Bellman-Ford algorithm. Table 7.3 shows a time comparison between our algorithms and the baseline method. We observe that our transitive closure algorithm is significantly faster, and also scales better.

Java method	$\mathbf{n}$	$\mathbf{T_o(s)}$	$\mathbf{T_b(s)}$
ArrayBlockingQueue: poll	19	19	60
ArrayBlockingQueue: peek	20	20	81
LinkedBlockingDeque: advance	25	29	195
PriorityBlockingQueue: removeEQ	25	32	176
ArrayBlockingQueue: init	26	47	249
LinkedBlockingDeque: remove	26	49	290
ArrayBlockingQueue: offer	26	56	304
ArrayBlockingQueue: clear	28	33	389
ArrayBlockingQueue: contains	32	205	881
DelayQueue: remove	42	267	3792
ConcurrentHashMap: scanAndLockForPut	46	375	2176
ArrayBlockingQueue: next	46	407	3915
ConcurrentHashMap: put	72	1895	$>$ 8 h

Table 7.3: Time required for the transitive closure on 2-self concurrent graphs extracted from methods of the java.util.concurrent library. Each constituent graph has $\mathbf{n}$ nodes. $\mathbf{T_o(s)}$ and $\mathbf{T_b(s)}$ correspond to our method and the baseline method respectively.

8 Quantitative Verification on Constant-treewidth Graphs

8.1 Introduction

In this chapter we study three classic quantitative verification properties, namely, the minimum mean-payoff, minimum ratio, and minimum initial credit for energy properties. We provide several algorithms for exact and approximate solutions to the minimum mean-payoff and minimum ratio problems on graphs of constant treewidth. We also study the minimum initial credit for energy problem on arbitrary graphs, and obtain a significant improvement over the existing approach. Finally, we present a significant algorithmic improvement for the problem restricted on constant-treewidth graphs.

The mean-payoff, ratio and minimum initial credit for energy problems. The three quantitative properties that have been studied for their relevance in analysis of reactive systems are as follows. First, the *mean-payoff* property consists of a weight function that assigns to every transition an integer-valued weight and assigns to each trace the long-run average of the weights of the transitions of the trace. Second, the *ratio* property consists of two weight functions (one of which is a positive weight function) and assigns to each trace the ratio of the two mean-payoff properties (the denominator is wrt the positive function). The *minimum initial credit for energy* property consists of a weight function (as in the mean-payoff property) and assigns to each trace the minimum number to be added such that the partial sum of the weights for every prefix of the trace is non-negative. For example, the mean-payoff property is used for average waiting time, worst-case execution time analysis [Chatterjee *et al.*, 2010a; Cerný *et al.*, 2013; Chatterjee *et al.*, 2015a]; the ratio property is used in robustness analysis of

Minimum mean-cycle value			Minimum ratio-cycle value		
[Orlin and Ahuja, 1992]	[Karp, 1978]	**Our result [Theorem 8.3] (ϵ-approximate)**	[Burns, 1991]	[Lawler, 1976]	**Our result [Corollary 8.2]**
$O(n^{1.5} \cdot \log(n \cdot W))$	$O(n^2)$	$O(n \cdot \log(n/\epsilon))$	$O(n^3)$	$O(n^2 \cdot \log(n \cdot W))$	$O(n \cdot \log(\lvert a \cdot b \rvert))$

Table 8.1: Time complexity of existing and our solutions for the minimum mean-cycle value and ratio-cycle value problem in constant treewidth weighted graphs with n nodes and largest absolute weight W, when the output is the (irreducible) fraction $\frac{a}{b} \neq 0$.

	[Bouyer *et al.*, 2008]	**Our result [Theorem 8.4, Corollary 8.3]**	**Our result [Theorem 8.6] (constant treewidth)**
Time (decision)	$O(n^2 \cdot m)$	$\mathbf{O(n \cdot m)}$	$O(n \cdot \log n)$
Time	$O(n^3 \cdot m \cdot \log(n \cdot W))$	$O(n^2 \cdot m)$	$O(n \cdot \log n)$
Space	$O(n)$	$O(n)$	$O(n)$

Table 8.2: Complexity of the existing and our solution for the minimum initial credit problem on weighted graphs of n nodes, m edges, and largest absolute weight W.

systems [Bloem *et al.*, 2009b]; and the minimum initial credit for energy property for measuring resource consumptions [Bouyer *et al.*, 2008].

Algorithmic problems. Given a graph and a quantitative property, the value of a node is the infimum value of all traces that start at the respective node. The algorithmic problem (namely, the *value* problem) in the analysis of quantitative properties consists of a graph and a quantitative property, and asks to compute either the exact value or an approximation of the value for every node in the graph. The algorithmic problems are at the heart of many applications, such as automata emptiness, model measuring, quantitative abstraction refinement, etc.

Previous results and our contributions. In this section we consider general graphs and graphs with constant treewidth, and the algorithmic problems to compute the exact value or an approximation of the value for every node wrt to quantitative properties given as the mean-payoff, the ratio, or the minimum initial credit for energy. We first present the relevant previous results, and

then our contributions.

Previous results. We consider graphs with n nodes, m edges, and let W denote the largest absolute value of the weights. The running time of the algorithms is characterized by the number of arithmetic operations (i.e., each operation takes constant time); and the space usage is characterized by the maximum number of integers the algorithm stores. The classic algorithm for graphs with mean-payoff properties is the minimum mean-cycle problem of Karp [Karp, 1978], and the algorithm requires $O(n \cdot m)$ running time and $O(n^2)$ space. A different algorithm was proposed in [Madani, 2002] that requires $O(n \cdot m)$ running time and $O(n)$ space. Orlin and Ahuja [Orlin and Ahuja, 1992] gave an algorithm running in time $O(\sqrt{n} \cdot m \cdot \log(n \cdot W))$. For some special cases there exist faster approximation algorithms [Chatterjee *et al.*, 2014a]. There is a straightforward reduction of the ratio problem to the mean-payoff problem. For computing the exact minimum ratio, the fastest known strongly polynomial time algorithm is Burns' algorithm [Burns, 1991] running in time $O(n^2 \cdot m)$. Also, there is an algorithm by Lawler [Lawler, 1976] that uses $O(n \cdot m \cdot \log(n \cdot W))$ time. For the minimum initial credit for energy problem, the decision problem (i.e., is the energy required for node v at most c?) can be solved in $O(n^2 \cdot m)$ time, leading to an $O(n^3 \cdot m \cdot \log(n \cdot W))$ time algorithm for the minimum initial credit for energy problem [Bouyer *et al.*, 2008]. All the above algorithms are for general graphs (without the constant-treewidth restriction).

Our contributions. Our main contributions are as follows.

1. *Finding the mean-payoff and ratio values in constant-treewidth graphs.* We present two results for constant treewidth graphs. First, for the exact computation we present an algorithm that requires $O(n \cdot \log(|a \cdot b|))$ time and $O(n)$ space, where $\frac{a}{b} \neq 0$ is the (irreducible) ratio/mean-payoff of the output. If $\frac{a}{b} = 0$ then the algorithm uses $O(n)$ time. Note that $\log(|a \cdot b|) \leqslant 2\log(n \cdot W)$. We also present a space-efficient version of the algorithm that requires only $O(\log n)$ space. Second, we present an algorithm for finding an ϵ-factor approximation that requires $O(n \cdot \log(n/\epsilon))$ time and $O(n)$ space, as compared to the $O(n^{1.5} \cdot \log(n \cdot W))$ time solution of Orlin & Ahuja, and the $O(n^2)$ time solution of Karp (see Table 8.1).

2. *Finding the minimum initial credit in graphs.* We present two results. First, we consider the exact computation for general graphs, and present (i) an $O(n \cdot m)$ time algorithm

for the decision problem (improving the previous known $O(n^2 \cdot m)$ bound), and (ii) an $O(n^2 \cdot m)$ time algorithm to compute the value of all nodes (improving the previous known $O(n^3 \cdot m \cdot \log(n \cdot W))$ bound). Finally, we consider the computation of the exact value for graphs with constant treewidth and present an algorithm that requires $O(n \cdot \log n)$ time (improving the previous known $O(n^4 \cdot \log(n \cdot W))$ bound) (see Table 8.2).

3. *Experimental results.* We have implemented our algorithms for the minimum mean cycle and minimum initial credit problems and have run them on standard benchmarks (DaCapo suit [Blackburn, 2006] for the minimum mean cycle problem, and DIMACS challenges [dim] for the minimum initial credit problem). For the minimum mean cycle problem, our results show that our algorithm has lower running time than all the classic polynomial-time algorithms. For the minimum initial credit problem, our algorithm provides a significant speedup over the existing method. Both improvements are demonstrated even on graphs of small/medium size.

Technical contributions. The key technical contributions of our work are as follows:

1. *Mean-payoff and ratio values in constant-treewidth graphs.* Given a graph with constant treewidth, let c^* be the smallest weight of a simple cycle. First, we present a linear-time algorithm that computes c^* exactly (if $c^* \geqslant 0$) or approximately within a polynomial factor (if $c^* < 0$). Then, we show that if the minimum ratio value ν^* is the irreducible fraction $\frac{a}{b}$, then ν^* can be computed by evaluating $O(\log(|a \cdot b|))$ inequalities of the form $\nu^* \geqslant \nu$. Each such inequality is evaluated by computing the smallest weight of a simple cycle in a modified graph. Finally, for ϵ-approximating the value ν^*, we show that $O(\log(n/\epsilon))$ such inequalities suffice.

2. *Minimum initial credit problem.* We show that for general graphs, the decision problem can be solved with two applications of Bellman-Ford-type algorithms, and the value problem reduces to finding non-positive cycles in the graph, followed by one instance of the single-source shortest-path problem. We then show how the invariants of the algorithm for the value problem on general graphs can be maintained by a particular graph traversal of the tree-decomposition for constant-treewidth graphs.

Organization. The rest of this chapter is organized as follows.

1. In Section 8.2 we define the problems we study in this chapter, namely, the mean-payoff, the ratio, and the minimum initial credit for energy problem.
2. In Section 8.3 we present an algorithm for dealing with a related graph problem, namely the detection of a minimum-weight simple cycle of a graph with constant treewidth.
3. In Section 8.4 we present algorithms for exact solutions to the ratio problem, as well as exact solutions and approximations to the mean-payoff problem on graphs of constant treewidth.
4. In Section 8.5 we present algorithms for solving the minimum initial credit for energy problem on both arbitrary graphs and graphs of constant treewidth.
5. In Section 8.6 we present an experimental evaluation of our algorithms for the mean-payoff and minimum initial credit problems.

8.2 Definitions

We start with a few small modifications on the definitions regarding weighted graphs and paths from Section 2.3.1. In particular, here we consider weighted graphs with possibly two weight functions (instead of the usual one weight function), and possibly infinite paths (instead of only finite paths).

Weighted graphs. We consider *weighted directed graphs* $G = (V, E, \mathsf{wt}, \mathsf{wt}')$ where V is the set of n *nodes*, $E \subseteq V \times V$ is the edge relation of m *edges*, $\mathsf{wt} : E \to \mathbb{Z}$ is a *weight function* that assigns an integer weight $\mathsf{wt}(e)$ to each edge $e \in E$, and $\mathsf{wt}' : E \to \mathbb{N}^+$ is a weight function that assigns strictly positive integer weights. For technical simplicity, we assume that there exists at least one outgoing edge from every node. In certain cases where the function wt' is irrelevant, we will consider weighted graphs $G = (V, E, \mathsf{wt})$, i.e., without the function wt'.

Paths, weights and values. The functions wt and wt' naturally extend to paths, so that the weight of a finite path P with $|P| > 0$ wrt the weight functions wt and wt' is $\mathsf{wt}(P) = \sum_{1 \leqslant i < j} \mathsf{wt}(x_i, x_{i+1})$ and $\mathsf{wt}'(P) = \sum_{1 \leqslant i < j} \mathsf{wt}'(x_i, x_{i+1})$. The *value* of P is defined to be $\overline{\mathsf{wt}}(P) = \frac{\mathsf{wt}(P)}{\mathsf{wt}'(P)}$. For the case where $|P| = 0$, we define $\mathsf{wt}(P) = 0$, and $\overline{\mathsf{wt}}(P)$ is undefined. An

infinite path $\mathcal{P} = (x_1, x_2, \dots)$ of G is an infinite sequence of nodes such that every finite prefix P of $\mathcal{P}$ is a finite path of G. The functions wt and wt′ assign to $\mathcal{P}$ a value in $\mathbb{Z} \cup \{-\infty, \infty\}$: we have $\mathsf{wt}(\mathcal{P}) = \sum_i \mathsf{wt}(x_i, x_{i+1})$ and $\mathsf{wt}'(\mathcal{P}) = \infty$. For a (possibly infinite) path P, Given a finite path $P_1 = (x_1, \dots x_k)$ and a possibly infinite path $P_2 = (y_1, \dots)$ with $x_k = y_1$, we denote by $P_1 \circ P_2$ the path resulting from the concatenating P_2 on P_1.

Distances and witness paths. For nodes $u, v \in V$, we denote by $d(u, v) = \inf_{P: u \rightsquigarrow v} \mathsf{wt}(P)$ the *distance* from u to v. A finite path $P : u \rightsquigarrow v$ is a *witness* of the distance $d(u, v)$ if $\mathsf{wt}(P) = d(u, v)$. An infinite path $\mathcal{P}$ is a witness of the distance $d(u, v)$ if the following conditions hold:

1. $d(u, v) = \mathsf{wt}(\mathcal{P}) = -\infty$, and

2. $\mathcal{P}$ starts from u, and v is reachable from every node of $\mathcal{P}$.

Observe that $d(u, v) = -\infty$ is only witnessed by infinite paths, whereas $d(u, v) = \infty$ is not witnessed by any path. We note that this is consistent with our semiring treatment of semiring distances as introduced in Section 2.3.1. Indeed, using a variant of the tropical semiring $(\mathbb{Z} \cup \{-\infty, \infty\}, \inf, +, \infty, 0)$ (where we define $-\infty + \infty = \infty$ and $-\infty$ represents "arbitrarily small") it is easy to verify that $d(u, v) = \oplus_{P: u \rightsquigarrow v} \otimes (P) = \infty$ iff there exists an infinite path $\mathcal{P}$ that is a witness of $d(u, v)$.

Throughout the chapter, we follow the convention that the supremum (or maximum) and infimum (or minimum) of the empty set is $-\infty$ and ∞ respectively, i.e., $\sup(\varnothing) = \max(\varnothing) = -\infty$ and $\inf(\varnothing) = \min(\varnothing) = \infty$. In the sequel we consider only nicely rooted, balanced and binary tree-decompositions of constant width and $= O(n)$ bags (and hence of height $O(\log n)$). Such tree decompositions can be constructed using Theorem 2.1 and Lemma 2.6.

8.2.1 Problems Considered

In this chapter, our interest is on the following three quantitative problems.

The minimum mean cycle problem [Karp, 1978]. Given a weighted directed graph $G = (V, E, \mathsf{wt})$, the minimum mean cycle problem asks to determine for each node u the *mean value* $\mu^*(u) = \min_{C \in \mathcal{C}_u} \frac{\mathsf{wt}(C)}{|C|}$, where $\mathcal{C}_u$ is the set of simple cycles reachable from u in G. A cycle C

with $\frac{\mathsf{wt}(C)}{|C|} = \mu^*(u)$ is called a minimum mean cycle of u. For $0 < \epsilon < 1$, we say that a value μ is an ϵ-approximation of the mean value $\mu^*(u)$ if $|\mu - \mu^*(u)| \leqslant \epsilon \cdot |\mu^*(u)|$.

The minimum ratio cycle problem [Hartmann and Orlin, 1993]. Given a weighted directed graph $G = (V, E, \mathsf{wt}, \mathsf{wt}')$, the minimum ratio cycle problem asks to determine for each node u the *ratio value* $\nu^*(u) = \min_{C \in \mathcal{C}_u} \overline{\mathsf{wt}}(C)$, where $\overline{\mathsf{wt}}(C) = \frac{\mathsf{wt}(C)}{\mathsf{wt}'(C)}$ and $\mathcal{C}_u$ is the set of simple cycles reachable from u in G. A cycle C with $\overline{\mathsf{wt}}(C) = \nu_u^*$ is called a minimum ratio cycle of u. The minimum mean cycle problem follows as a special case of the minimum ratio cycle problem for $\mathsf{wt}'(e) = 1$ for each edge $e \in E$.

The minimum initial credit problem [Bouyer *et al.*, 2008]. Given a weighted directed graph $G = (V, E, \mathsf{wt})$, the minimum initial credit value problem asks to determine for each node u the smallest *energy value* $\mathsf{E}(u) \in \mathbb{N} \cup \{\infty\}$ with the following property: there exists an infinite path $\mathcal{P} = (u_1, u_2 \dots)$ with $u = u_1$, such that for every finite prefix P of $\mathcal{P}$ we have $\mathsf{E}(u) + \mathsf{wt}(P) \geqslant 0$. Conventionally, we let $\mathsf{E}(u) = \infty$ if no finite value exists. The associated decision problem asks given a node u and an initial credit $c \in \mathbb{N}$ whether $\mathsf{E}(u) \leqslant c$.

8.3 Minimum Cycle

In the current section we deal with a related graph problem, namely the detection of a minimum-weight simple cycle of a graph. In Section 8.4 we use solutions to the minimum cycle problem to obtain the minimum ratio and minimum mean values of a graph.

The minimum cycle problem. Given a weighted graph $G = (V, E, \mathsf{wt})$, the minimum cycle problem asks to determine the weight c^* of a minimum-weight simple cycle in G, i.e., $c^* = \min_{C \in \mathcal{C}} \mathsf{wt}(C)$, where $\mathcal{C}$ is the set of simple cycles in G.

Here we present an algorithm called MinCycle that operates on a tree-decomposition $\mathrm{Tree}(G)$ of an input graph G, and has the following properties.

1. If G has no negative cycles, then MinCycle returns the weight c^* of a minimum-weight cycle in G.

2. If G has negative cycles, then $\mathsf{MinCycle}$ returns a value that is at most a polynomial (in n) factor smaller than c^*.

U-shaped paths. Recall the definition of U-shaped paths from Section 3.2. Given a bag B and nodes $u, v \in B$, we say that a path $P : u \rightsquigarrow v$ is U-*shaped* in B, if one of the following conditions hold:

1. Either $|P| > 1$ and for all intermediate nodes $w \in P$, we have that B is an ancestor of B_w,
2. or $|P| \leqslant 1$ and B is B_u or B_v (i.e., B is the root bag of u or v).

The following remark follows from the definition of tree decompositions, and states that every simple cycle C can be seen as a U-shaped path P from the smallest-level node of C to itself. Consequently, we can determine the value c^* by only considering U-shaped paths in $\mathrm{Tree}(G)$.

Remark 8.1. Let $C = (u_1, \dots, u_k)$ be a simple cycle in G, and $u_j = \arg\min_{u_i \in C} \mathsf{Lv}(u_i)$. Then $P = (u_j, u_{j+1}, \dots u_k, u_1, \dots, u_j)$ is a U-shaped path in B_{u_j}, and $\mathsf{wt}(P) = \mathsf{wt}(C)$.

Informal description of $\mathsf{MinCycle}$**.** Note that integer-valued weights are a special case of the tropical semiring. Our algorithm $\mathsf{MinCycle}$ is similar to the algorithm $\mathsf{Preprocess}$ (Algorithm 2) from Section 3.2.1 phrased for complete semirings. It consists of a depth-first traversal of $\mathrm{Tree}(G)$, and for each examined bag B computes a *local* U-*shaped distance* map $\mathsf{LUD}_B : B \times B \to \mathbb{Z} \cup \{\infty\}$ such that for each $u, v \in B$, we have (i) $\mathsf{LUD}_B(u, v) = \mathsf{wt}(P)$ for some path $P : u \rightsquigarrow v$, and (ii) $\mathsf{LUD}_B(u, v) \leqslant \min_P \mathsf{wt}(P)$, where P are taken to be simple $u \rightsquigarrow v$ paths (or simple cycles) that are U-shaped in B. This is achieved by traversing $\mathrm{Tree}(G)$ in post-order, and for each root bag B_x of a node x, we update every $\mathsf{LUD}_{B_x}(u, v)$ with $\mathsf{LUD}_{B_x}(u, x) + \mathsf{LUD}_{B_x}(x, v)$ (i.e., we do path-shortening from node u to node v, by considering paths that go through x). See Fig. 3.1 for an illustration.

In the end, $\mathsf{MinCycle}$ returns $\min_x \mathsf{LUD}_{B_x}(x, x)$, i.e., the weight of the smallest-weight U-shaped (not necessarily simple) cycle it has discovered. Algorithm 24 gives $\mathsf{MinCycle}$ in pseudocode. For brevity, in Line 5 we consider that if $\{u, v\} \notin E$ or $\{u, v\} \not\subseteq B_i$ for some child B_i of B, then $\mathsf{LUD}_{B_i}(u, v) = \infty$.

In essence, $\mathsf{MinCycle}$ performs repeated summarizations of paths in G. The following lemma follows easily from Lemma 3.3, and states that $\mathsf{LUD}_B(u, v)$ is upper bounded by the smallest

Algorithm 24: MinCycle

Input: A weighted graph $G = (V, E, \mathsf{wt})$ and a balanced binary tree-decomposition $\mathrm{Tree}(G)$

Output: A value c

1 Assign $c \leftarrow \infty$
2 Apply a post-order traversal on $\mathrm{Tree}(G)$, and examine each bag B with children B_1, B_2
3 **begin**
4   **foreach** $u, v \in B$ **do**
5   Assign $\mathsf{LUD}_B(u, v) \leftarrow \min(\mathsf{LUD}_{B_1}(u, v), \mathsf{LUD}_{B_2}(u, v), \mathsf{wt}(u, v))$
6   **end**
7   Discard $\mathsf{LUD}_{B_1}, \mathsf{LUD}_{B_2}$
8   **if** *B is the root bag of a node x* **then**
9   **foreach** $u, v \in B$ **do**
10   Assign $\mathsf{LUD}'_B(u, v) \leftarrow \min(\mathsf{LUD}_B(u, v), \mathsf{LUD}_B(u, x) + \mathsf{LUD}_B(x, v))$
11   **end**
12   Assign $\mathsf{LUD}_B \leftarrow \mathsf{LUD}'_B$
13   Assign $c \leftarrow \min(c, \mathsf{LUD}_B(x, x))$
14 **end**
15 **return** c

weight of a U-shaped simple $u \rightsquigarrow v$ path in B.

Lemma 8.1. *For every examined bag B and nodes $u, v \in B$, we have*

1. $\mathsf{LUD}_B(u,v) = \mathsf{wt}(P)$ *for some path $P : u \rightsquigarrow v$ (and $\mathsf{LUD}_B(u,v) = \infty$ if no such P exists),*
2. $\mathsf{LUD}_B(u,v) \leqslant \min_{P:u\rightsquigarrow v} \mathsf{wt}(P)$ *where P ranges over* U*-shaped simple paths and simple cycles in B.*

At the end of the computation, the returned value c is the weight of a (generally non-simple) cycle C, captured as a U-shaped path on its smallest-level node. The cycle C can be recovered by tracing backwards the updates of Line 10 performed by the algorithm, starting from the node x that performed the last update in Line 13. Hence, if C traverses k distinct edges, we can write

$$c = \mathsf{wt}(C) = \sum_{i=1}^{k} k_i \cdot \mathsf{wt}(e_i) \tag{8.1}$$

where each e_i is a distinct edge, and k_i is the number of times it appears in C.

Lemma 8.2. *Let h be the height of* $\mathrm{Tree}(G)$. *For every k_i in Eq.* (8.1)*, we have $k_i \leqslant 2^h$.*

Proof. Note that the edge $e_i = (u_i, v_i)$ is first considered by MinCycle in the root bag B_i of node x_i, where $x_i = \arg\max_{y_i \in \{u_i, v_i\}} \mathsf{Lv}(y_i)$ (Line 10). As MinCycle backtracks from B_i to the root of $\mathrm{Tree}(G)$, the edge e_i can be traversed at most twice as many times in each step (because of Line 10, once for each term of the sum $\mathsf{LUD}_B(u,x) + \mathsf{LUD}_B(x,v)$). Hence, this doubling will occur at most h times, and thus $k_i \leqslant 2^h$. □

Lemma 8.3. *Let c be the value returned by* MinCycle*, h be the height of* $\mathrm{Tree}(G)$*, and $c^* = \min_C \mathsf{wt}(C)$ over all simple cycles C in G. The following assertions hold:*

1. *If G has no negative cycles, then $c = c^*$.*
2. *If G has a negative cycle, then*
   - *(a)* $c \leqslant c^*$.
   - *(b)* $|c| = O\left(|c^*| \cdot n \cdot 2^h\right)$.

Proof. By Remark 8.1, we have that $c^* = \mathsf{wt}(P)$ for a U-shaped path $P : x \rightsquigarrow x$. By Lemma 8.1, after $\mathsf{MinCycle}$ examines B_x, it will be $c \leqslant \mathsf{LUD}_{B_x}(x, x) \leqslant c^*$, with the equalities holding if there are no negative cycles in G (by the definition of c^*, as then $\mathsf{LUD}_{B_x}(x, x)$ is witnessed by a simple cycle). By Line 10, c can only decrease afterwards, and again by the definition of c^* this can only happen if there are negative cycles in G. This proves Items 1 and 2a, and the remaining of the proof focuses on showing that $|c| = O\left(|c^*| \cdot n \cdot 2^h\right)$.

By rearranging the sum of Eq. (8.1), we can decompose the obtained cycle C into a set of k'^+ non-negative simple cycles C_i^+, and a set of k'^- negative simple cycles C_i^-, and each cycle C_i^+ and C_i^- appears with multiplicity k_i^+ and k_i^- respectively. Then we have

$$\begin{aligned} |c| = |\mathsf{wt}(C)| = \left|\sum_{i=1}^{k'^+} k_i^+ \cdot \mathsf{wt}(C_i^+) + \sum_{i=1}^{k'^-} k_i^- \cdot \mathsf{wt}(C_i^-)\right| \leqslant \left|\sum_{i=1}^{k'^-} k_i^- \cdot \mathsf{wt}(C_i^-)\right| \\ \leqslant \sum_{i=1}^{k^-} k_i^- \cdot |\mathsf{wt}(C_i^-)| \leqslant |c^*| \cdot \sum_{i=1}^{k'^-} k_i^- \leqslant |c^*| \cdot \sum_{i=1}^{k} k_i = O\left(|c^*| \cdot n \cdot 2^h\right) \end{aligned} \tag{8.2}$$

The first inequality follows from $c < 0$, the third inequality holds by the definition of c^*, and the last inequality holds since the total number of (non-positive) simple cycle traversals of C cannot be more than the total number of the edge traversals. Finally, we have $\sum_{i=1}^{k} k_i = O\left(n \cdot 2^h\right)$, since $k = O(n)$, and by Lemma 8.2 we have $k_i \leqslant 2^h$. □

Next we discuss the time and space complexity of $\mathsf{MinCycle}$.

Lemma 8.4. *Let h be the height of* $\mathrm{Tree}(G)$*.* $\mathsf{MinCycle}$ *accesses each bag of* $\mathrm{Tree}(G)$ *a constant number of times, and uses $O(h)$ additional space.*

Proof. $\mathsf{MinCycle}$ accesses each bag a constant number of times, as it performs a post-order traversal on $\mathrm{Tree}(G)$ (Line 2). Because it computes the local distances in a postorder manner, the number of local distance maps LUD_B it remembers is bounded by the height h of $\mathrm{Tree}(G)$. Since $\mathrm{Tree}(G)$ has constant width, LUD_B requires a constant number of words for storing a constant number of nodes and weights in each B. Hence the total space usage is $O(h)$, and the result follows. □

The following theorem summarizes the results of this section.

Theorem 8.1. *Let $G = (V, E, \mathsf{wt})$ be a weighted graph of n nodes with constant treewidth, and a balanced, binary tree-decomposition $\mathrm{Tree}(G)$ of G be given. Let c^*, be the smallest weight of a simple cycle in G. Algorithm* MinCycle *uses $O(n)$ time and $O(\log n)$ additional space, and returns a value c such that:*

1. *If G has no negative cycles, then $c = c^*$.*
2. *If G has a negative cycle, then*
   (a) $c \leqslant c^*$.
   (b) $|c| = |c^*| \cdot n^{O(1)}$.

8.4 The Minimum Ratio and Mean Cycle Problems

In the current section we present algorithms for solving the minimum ratio and mean cycle problems for weighted graphs $G = (V, E, \mathsf{wt}, \mathsf{wt}')$ of constant treewidth.

Remark 8.2. If G is not strongly connected, we can compute its maximal strongly connected components (SCCs) in linear time [Tarjan, 1972], and use the algorithms of this section to compute the minimum cycle ratio ν_i^* in every component $\mathcal{G}_i$. Afterwards, we assign the ratio values $\nu^*(u)$ for all nodes u as follows. First, mark every SCC $\mathcal{G}_i$ with $M(\mathcal{G}_i) = \nu_i^*$. Then, for every bottom SCC $\mathcal{G}_i$, (i) for every u in $\mathcal{G}_i$ assign $\nu^*(u) = M(\mathcal{G}_i)$, (ii) for every neighbor SCC $\mathcal{G}_j$ of $\mathcal{G}_i$, mark $\mathcal{G}_j$ with $M(\mathcal{G}_j) = \min(M(\mathcal{G}_j), M(\mathcal{G}_i))$, (iii) remove $\mathcal{G}_i$ and repeat. Since these operations require linear time in total, they do not impact the time complexity. We also argue that we can focus on SCCs while using $O(\log n)$ space. Using [Elberfeld *et al.*, 2010], we can solve directed s-t connectivity in logspace for constant-treewidth graphs. For any node v, let $\mathsf{SCC}(v)$ denote the strongly connected component of v. For each v we can find the value of the minimum ratio cycle when the graph is restricted to $\mathsf{SCC}(v)$, using any algorithm that can solve the problem if the graph is strongly connected, by simply ignoring all nodes w such that v cannot reach w or w cannot reach v. Then, for any node u, the value $\nu^*(u)$ is computed by solving the minimum ratio cycle problem restricted in $\mathsf{SCC}(v)$, for every node v reachable from u, and returning the minimum of all these values. Therefore, we consider graphs G that are strongly connected, and we will speak about the minimum ratio ν^* and mean μ^* values of G.

In light of Remark 8.2, we consider graphs that are strongly connected, and hence it follows that $\nu^*(u)$ is the same for every node u, and thus we will speak about the minimum ratio ν^* and mean μ^* values of G.

Lemma 8.5. *Let ν^* be the ratio value of G. Then $\nu^* \geqslant \nu$ iff for every cycle C of G we have $\mathsf{wt}_\nu(C) \geqslant 0$, where $\mathsf{wt}_\nu(e) = \mathsf{wt}(e) - \mathsf{wt}'(e) \cdot \nu$ for each edge $e \in E$.*

Proof. Indeed, for any cycle C we have $\overline{\mathsf{wt}}(C) \geqslant \nu^* \geqslant \nu$. Then

$$\overline{\mathsf{wt}}(C) \geqslant \nu \iff \overline{\mathsf{wt}}(C) - \nu \geqslant 0 \iff \frac{\mathsf{wt}(C) - \nu \cdot \mathsf{wt}'(C)}{\mathsf{wt}'(C)} \geqslant 0$$

$$\iff \mathsf{wt}(C) - \nu \cdot \mathsf{wt}'(C) \geqslant 0 \iff \sum_{e \in C}(\mathsf{wt}(e) - \mathsf{wt}'(e) \cdot \nu) \geqslant 0 \iff \mathsf{wt}_\nu(C) \geqslant 0$$

with the equality holding iff $\overline{\mathsf{wt}}(C) = \nu$. □

Hence, given a tree-decomposition $\mathrm{Tree}(G)$, for any guess ν of the ratio value ν^*, we can evaluate whether $\nu^* \geqslant \nu$ by constructing the weight function $\mathsf{wt}_\nu = \mathsf{wt} - \nu$ and executing algorithm $\mathsf{MinCycle}$ on input $G_\nu = (V, E, \mathsf{wt}_\nu)$. By Item 2a of Theorem 8.1 and Lemma 8.5 we have that the returned value c of $\mathsf{MinCycle}$ is $c \geqslant 0$ iff $\mathsf{wt}_\nu(C) \geqslant 0$ for all cycles C, iff $\nu^* \geqslant \nu$ (and in fact $c = 0$ iff $\nu^* = \nu$).

Lemma 8.6. *Let $G = (V, E, \mathsf{wt}, \mathsf{wt}')$ be a weighted graph of n nodes with constant treewidth and minimum ratio value ν^*. Let $\mathrm{Tree}(G)$ be a given balanced, binary tree-decomposition of G of constant width. For any rational ν, the decision problem of whether $\nu^* \geqslant \nu$ (or $\nu^* = \nu$) can be solved in $O(n)$ time and $O(\log n)$ extra space.*

Proof. By Lemma 8.5, we can test whether $\nu^* \geqslant \nu$ by testing whether $G_\nu = (V, E, \mathsf{wt}_\nu)$ has a negative cycle. By Theorem 8.1, a negative cycle in G_ν can be detected in $O(n)$ time and using $O(\log n)$ space. □

8.4.1 Exact Solution

We now describe the method for determining the value ν^* of G exactly. This is done by making various guesses ν such that $\nu^* \geqslant \nu$ and testing for negative cycles in the graph $G_\nu = (V, E, \mathsf{wt}_\nu)$. We first determine whether $\nu^* = 0$, using Lemma 8.6. In the remaining of this section we assume that $\nu^* \neq 0$.

Solution overview. Consider that $\nu^* > 0$. First, we either find that $\nu^* \in (0,1)$ (hence $\lfloor \nu^* \rfloor = 0$), or perform an *exponential search* of $O(\log \nu^*)$ iterations to determine $j \in \mathbb{N}^+$ such that $\nu^* \in [2^{j-1}, 2^j]$. In the latter case, we perform a binary search of $O(\log \nu^*)$ iterations in the interval $[2^{j-1}, 2^j]$ to determine $\lfloor \nu^* \rfloor$ (see Fig. 8.1). Then, we can write $\nu^* = \lfloor \nu^* \rfloor + x$, where $x < 1$ is an irreducible fraction $\frac{a'}{b}$. It has been shown [Papadimitriou, 1979] that such x can be determined by evaluating $O(\log b)$ inequalities of the form $x \geqslant \nu$. The case for $\nu^* < 0$ is handled similarly.

Lemma 8.7. *Let $\nu^* \neq 0$ be the ratio value of G. The value $\lfloor \nu^* \rfloor$ can be obtained by evaluating $O(\log |\nu^*|)$ inequalities of the form $\nu^* \geqslant \nu$.*

Proof. First determine whether $\nu^* > 0$, and assume w.l.o.g. that this is the case (the process is similar if $\nu^* < 0$). Perform an exponential search on the interval $(0, 2 \cdot \lfloor \nu^* \rfloor)$ by a sequence of evaluations of the inequality $\nu^* \geqslant \nu_i = 2^i$. After $\log \lfloor \nu^* \rfloor + 1$ steps we either have $\lfloor \nu^* \rfloor \in (0,1)$, or have determined a $j > 0$ such that $\nu^* \in [\nu_{j-1}, \nu_j]$. Then, perform a binary search in the interval $[\nu_{j-1}, \nu_j]$, until the running interval $[\ell, r]$ has length at most 1. Since $\nu_j - \nu_{j-1} = \nu_{j-1} \leqslant \nu^*$, this will happen after at most $\log \lceil \nu^* \rceil$ steps. Then either $\lfloor \nu^* \rfloor = \lfloor \ell \rfloor$ or $\lfloor \nu^* \rfloor = \lfloor r \rfloor$, which can be determined by evaluating the inequality $\nu^* \geqslant \lfloor r \rfloor$. A similar process can be carried out when $\nu^* < 0$. Fig. 8.1 shows an illustration of the search. □

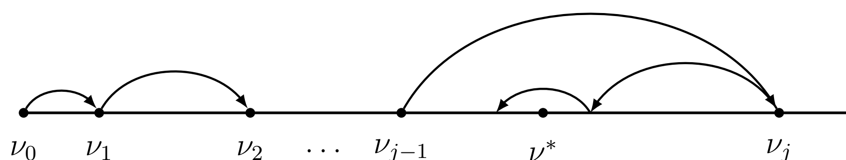


Figure 8.1: Exponential search followed by a binary search to determine $\lfloor \nu^* \rfloor$

Let $T_{\max} = \max_e \mathsf{wt}'(e)$ be the largest weight of an edge wrt wt'. Since ν^* is a number with denominator at most $(n-1) \cdot T_{\max}$, it can be determined exactly by carrying the binary search of Lemma 8.7 until the length of the running interval becomes at most $\frac{1}{((n-1) \cdot T_{\max})^2}$ (thus containing a unique rational with denominator at most $(n-1) \cdot T_{\max}$). Then ν^* can be obtained by using continued fractions, e.g. as in [Kwek and Mehlhorn, 2003]. We rely in the work of [Papadimitriou, 1979] to obtain a tighter bound.

Lemma 8.8. *Let $\nu^* \neq 0$ be the ratio value of G, such that ν^* is the irreducible fraction $\frac{a}{b} \in (-1,1)$. Then ν^* can be determined by evaluating $O(\log b)$ inequalities of the form $\nu^* \geqslant \nu$.*

Proof. Consider that $\nu^* > 0$ (the proof is similar when $\nu^* < 0$). It is shown in [Papadimitriou, 1979] that a rational with denominator at most b can be determined by evaluating $O(\log b)$ inequalities of the form $\nu^* \geqslant \nu$. We remark that b is not required to be known, although the work of [Papadimitriou, 1979] assumes that a bound on the denominator of ν^* is known in advance. $\square$

Sketch. For completeness, we outline the method of [Papadimitriou, 1979]. We only consider the case where $\nu^* \geqslant 0$, as the complementary case can be handled analogously. A Farey sequence [Graham *et al.*, 1989] F_k of order k is the sequence of all irreducible fractions $\frac{x_i^k}{y_i^k}$ in increasing order for which $0 \leqslant x_i^k \leqslant y_i^k \leqslant k$. Since $\overline{\mathsf{wt}}(C) = \nu^*$, it follows that ν^* is a rational with denominator at most $\mathsf{wt}'(C)$, therefore ν^* is a fraction in each $F_{\overline{k}}$ for $\overline{k} \geqslant \mathsf{wt}'(C)$. We can determine ν^* by considering a sequence of Farey sequences of exponential orders $(F_{2^k})_{1 \leqslant k \leqslant \lceil \log \mathsf{wt}'(C) \rceil}$, and for each such sequence determine the successive fractions such that

$$\frac{x_i^{2^k}}{y_i^{2^k}} \leqslant \nu^* \leqslant \frac{x_{i+1}^{2^k}}{y_{i+1}^{2^k}}$$

For $k = 1$, this is trivial. As we transition from k to $k + 1$, we need to determine successive fractions $\frac{x_i^{2^{k+1}}}{y_i^{2^{k+1}}}$ and $\frac{x_{i+1}^{2^{k+1}}}{y_{i+1}^{2^{k+1}}}$ such that

$$\frac{x_i^{2^{k+1}}}{y_i^{2^{k+1}}} \leqslant \nu^* \leqslant \frac{x_{i+1}^{2^{k+1}}}{y_{i+1}^{2^{k+1}}}$$

It is shown in [Papadimitriou, 1979] that there exist $\alpha_1, \beta_1, \alpha_2, \beta_2 \geqslant 0$ such that

$$\frac{x_i^{2^{k+1}}}{y_i^{2^{k+1}}} = \frac{\alpha_1 \cdot x_i^{2^k} + \beta_1 \cdot x_{i+1}^{2^k}}{\alpha_1 \cdot y_i^{2^k} + \beta_1 \cdot y_{i+1}^{2^k}} \qquad \text{and} \qquad \frac{x_{i+1}^{2^{k+1}}}{y_{i+1}^{2^{k+1}}} = \frac{\alpha_2 \cdot x_i^{2^k} + \beta_2 \cdot x_{i+1}^{2^k}}{\alpha_2 \cdot y_i^{2^k} + \beta_2 \cdot y_{i+1}^{2^k}}$$

Additionally, the number of inequalities that we need to evaluate for determining all such $\alpha_1, \beta_1, \alpha_2, \beta_2$ for every transition up to stage k is $O(k)$. Thus, after $O(\log \mathsf{wt}'(C))$ evaluations we have determined the fractions of $F_{\lceil \log \mathsf{wt}'(C) \rceil}$ for which

$$\frac{x_i^{2^{\lceil \log \mathsf{wt}'(C) \rceil}}}{y_i^{2^{\lceil \log \mathsf{wt}'(C) \rceil}}} \leqslant \nu^* \leqslant \frac{x_{i+1}^{2^{\lceil \log \mathsf{wt}'(C) \rceil}}}{y_{i+1}^{2^{\lceil \log \mathsf{wt}'(C) \rceil}}}$$

and one of the inequalities is an equality. The desired result follows.

Note that $\mathsf{wt}'(C)$ is not required to be known, although the work of [Papadimitriou, 1979] assumes that a bound on the denominator of ν^* is known in advance. $\square$

Theorem 8.2. *Let $G = (V, E, \mathsf{wt}, \mathsf{wt}')$ be a weighted graph of n nodes with constant treewidth, and $\lambda = \max_u |a_u \cdot b_u|$ such that $\nu^*(u)$ is the irreducible fraction $\frac{a_u}{b_u}$. Let $\mathcal{T}(G)$ and $\mathcal{S}(G)$ denote the required time and space for constructing a balanced binary tree-decomposition $\mathrm{Tree}(G)$ of G with constant width. The minimum ratio cycle problem for G can be computed in*

1. *$O(\mathcal{T}(G) + n \cdot \log(\lambda))$ time and $O(\mathcal{S}(G) + n)$ space; and*

2. *$O(\mathcal{S}(G) + \log n)$ space.*

Proof. In view of Remark 8.2 the graph G is strongly connected and has a minimum ratio value ν^*. Let $\nu^* = \lfloor \nu^* \rfloor + \frac{a'}{b}$ with $|\frac{a'}{b}| < 1$. By Lemma 8.7, $\lfloor \nu^* \rfloor$ can be determined by evaluating $O(\log |\nu^*|) = O(\log |a|)$ inequalities of the form $\nu^* \geqslant \nu$, and by Lemma 8.8, $\frac{a'}{b}$ can be determined by evaluating $O(b)$ such inequalities. A balanced binary tree-decomposition $\mathrm{Tree}(G)$ can be constructed once in $\mathcal{T}(G)$ time and $\mathcal{S}(G)$ space, and stored in $O(n)$ space. $\mathrm{Tree}(G)$ is also a tree-decomposition of every G_ν required by Lemma 8.5. By Theorem 8.1 a negative cycle in G_ν can be detected in $O(n)$ time and using $O(\log n)$ space. This concludes Item 1. Item 2 is obtained by the same process, but with re-computing $\mathrm{Tree}(G)$ every time $\mathsf{MinCycle}$ traverses from a bag to a neighbor (thus not storing $\mathrm{Tree}(G)$ explicitly). □

Using Theorem 2.1 we obtain from Theorem 8.2 the following corollary.

Corollary 8.1. *Let $G = (V, E, \mathsf{wt}, \mathsf{wt}')$ be a weighted graph of n nodes with constant treewidth, and $\lambda = \max_u |a_u \cdot b_u|$ such that $\nu^*(u)$ is the irreducible fraction $\frac{a_u}{b_u}$. The minimum ratio value problem for G can be computed in*

1. *$O(n \cdot \log(\lambda))$ time and $O(n)$ space; and*

2. *$O(\log n)$ space.*

By setting $\mathsf{wt}'(e) = 1$ for each $e \in E$ in Corollary 8.1 we obtain the following corollary for the minimum mean cycle.

Corollary 8.2. *Let $G = (V, E, \mathsf{wt})$ be a weighted graph of n nodes with constant treewidth, and $\lambda = \max_u |\mu^*(u)|$. The minimum mean value problem for G can be computed in*

1. *$O(n \cdot \log(\lambda))$ time and $O(n)$ space; and*

2. *$O(\log n)$ space.*

8.4.2 Approximating the Minimum Mean Cycle

We now focus on the minimum mean cycle problem, and present algorithms for ϵ-approximating the mean value μ^* of G for any $0 < \epsilon < 1$ in $O(n \cdot \log(\frac{n}{\epsilon}))$ time, i.e., independent of μ^*.

Approximate solution in the absence of negative cycles. We first consider graphs G that do not have negative cycles. Let C be a minimum mean value cycle, and C' a minimum weight simple cycle in G, and note that $\mu^* \in [0, \mathsf{wt}(C')]$. Additionally, we have

$$\mathsf{wt}(C') \leqslant \mathsf{wt}(C) \implies \mathsf{wt}(C') \leqslant \frac{n}{|C|} \cdot \mathsf{wt}(C) \implies \mathsf{wt}(C') \leqslant (n) \cdot \mu^*$$

Consider a binary search in the interval $[0, \mathsf{wt}(C')]$, which in step i approximates μ^* by the right endpoint μ_i of its current interval. The error is bounded by the length of the interval, hence $\mu_i - \mu^* \leqslant \mathsf{wt}(C') \cdot 2^{-i} \leqslant (n-1) \cdot \mu^* \cdot 2^{-i}$. To approximate within a factor ϵ we require

$$2^{-i} \cdot (n-1) \leqslant \epsilon \implies i \geqslant \log(n) + \log(\frac{1}{\epsilon}) \tag{8.3}$$

steps.

Remark 8.3. Note that for the minimum ratio value we have $\mathsf{wt}(C') \leqslant W' \cdot n \cdot \nu^*$, where $W' = \max_{e \in E} \mathsf{wt}'(e)$. For ϵ-approximating ν^* we would need $i \geqslant \log(\frac{n \cdot W'}{\epsilon})$ steps.

Approximate solution in the presence of negative cycles. We now turn our attention to ϵ-approximating μ^* in the presence of negative cycles in G. Note that uniformly increasing the weight of each edge so that no negative edges exist does not suffice, as the error can be of order $\epsilon \cdot |W^-|$ rather than $\epsilon \cdot \mu^*$, where W^- is the minimum edge weight.

Instead, let c be the value returned by $\mathsf{MinCycle}$ on input G. Item 2a of Theorem 8.1 guarantees that for the weight function $\mathsf{wt}_{-|c|}(e) = \mathsf{wt}(e) + |c|$, the graph $G_{-|c|} = (V, E, \mathsf{wt}_{-|c|})$ has no negative cycles (although it might still have negative edges). The following lemma states that μ^* can be ϵ-approximated by ϵ'-approximating the value μ'^* of $G_{-|c|}$, for some ϵ' polynomially (in n) smaller than ϵ.

Lemma 8.9. *Let μ^* and μ'^* be the value of G and $G_{-|c|}$ respectively, and ϵ some desired approximation factor of μ^*, with $0 < \epsilon < 1$. There exists an $\epsilon' = \frac{\epsilon}{n^{O(1)}}$ such that if μ' is an ϵ'-approximation of μ'^* in $G_{-|c|}$, then $\mu = \mu' - |c|$ is an ϵ-approximation of μ^* in G.*

Proof. By construction, we have $\mu'^* = \mu^* + |c|$, where c defined above is the value returned by MinCycle on G. Let c^* be the weight of a minimum-weight simple cycle in G. By Theorem 8.1 Item 2b, we have that $|c| = |c^*| \cdot n^{O(1)}$. Note that $|c^*| \leqslant (n-1) \cdot |\mu^*|$, hence $\mu'^* = \mu^* + |c^*| \cdot n^{O(1)} \leqslant |\mu^*| \cdot \alpha$ for $\alpha = n^{O(1)}$. Let $\epsilon' = \frac{\epsilon}{\alpha}$. By ϵ'-approximating μ'^* by μ' we have

$$
\begin{aligned}
|\mu' - \mu'^*| \leqslant \epsilon' \cdot |\mu'^*| &\implies |(\mu' - |c|) - (\mu'^* - |c|)| \leqslant \epsilon' \cdot |\mu'^*| \\
&\implies |\mu - \mu^*| \leqslant \epsilon' \cdot |\mu^*| \cdot \alpha \leqslant \epsilon \cdot |\mu^*|
\end{aligned}
$$

The desired result follows. □

Theorem 8.3. *Let $G = (V, E, \mathsf{wt})$ be a weighted graph of n nodes with constant treewidth. For any $0 < \epsilon < 1$, the minimum mean value problem can be ϵ-approximated in $O(n \cdot \log(\frac{n}{\epsilon}))$ time and $O(n)$ space.*

Proof. In view of Remark 8.2 the graph G is strongly connected and has a minimum mean value μ^*. First, we construct a balanced binary tree-decomposition $\mathrm{Tree}(G)$ of G in $O(n)$ time and space using Theorem 2.1. Let c be the value returned by MinCycle on the input graph G. If $c \geqslant 0$, by Lemma 8.3 we have $\mu^* \geqslant 0$, and by Eq. (8.3) μ^* can be ϵ-approximated in $O(\log(\frac{n}{\epsilon}))$ steps. If $c < 0$, we construct the graph $G_{-|c|} = (V, E, \mathsf{wt}_{-|c|})$. By Lemma 8.9, μ^* can be ϵ-approximated by ϵ' approximating the mean value μ'^* of $G_{-|c|}$, where $\epsilon' = \frac{\epsilon}{n^{O(1)}}$. By construction, $G_{-|c|}$ does not contain negative cycles, thus $\mu'^* \geqslant 0$, and by Eq. (8.3) μ'^* can be approximated in $O(\log(\frac{n}{\epsilon'})) = O(\log(\frac{n}{\epsilon}))$ steps. By Lemma 8.4, each step requires $O(n)$ time. The statement follows. □

8.5 The Minimum Initial Credit Problem

In the current section we present algorithms for solving the minimum initial credit problem on weighted graphs $G = (V, E, \mathsf{wt})$. We first deal with arbitrary graphs, and provide (i) an $O(n \cdot m)$

algorithm for the decision problem, and (ii) an $O(n^2 \cdot m)$ for the value problem, improving the previously best upper bounds. Afterwards we adapt our approach on graphs of constant treewidth to obtain an $O(n \cdot \log n)$ algorithm for the value problem.

Non-positive minimum initial credit. For technical convenience we focus on a variant of the minimum initial credit problem, where energies are non-positive, and the goal is to keep partial sums of path prefixes non-positive. Formally, given a weighted graph $G = (V, E, \mathsf{wt})$, the non-positive minimum initial credit value problem asks to determine for each node $u \in V$ the largest energy value $\mathsf{E}(u) \in \mathbb{Z}_{\leqslant 0} \cup \{-\infty\}$ with the following property: there exists an infinite path $\mathcal{P} = (u_1, u_2 \dots)$ with $u = u_1$, such that for every finite prefix P of $\mathcal{P}$ we have $\mathsf{E}(u) + \mathsf{wt}(P) \leqslant 0$. Conventionally, we let $\mathsf{E}(u) = -\infty$ if no finite such value exists. The associated decision problem asks given a node u and an initial credit $c \in \mathbb{Z}_{\leqslant 0}$ whether $\mathsf{E}(u) \geqslant c$. Hence, here minimality is wrt the absolute value of the energy. A solution to the standard minimum initial credit problem can be obtained by inverting the sign of each edge weight and solving the non-positive minimum initial credit problem in the resulting graph.

We start with some definitions and lemmas that will give the intuition for the algorithms to follow. First, we define the minimum initial credit of a pair of nodes u, v, which is the energy to reach v from u (i.e., the energy is wrt a finite path).

Finite minimum initial credit. For nodes $u, v \in V$, we denote by $\mathsf{E}_v(u) \in \mathbb{Z}_{\leqslant 0} \cup \{-\infty\}$ the largest value with the following property: there exists a path $P : u \rightsquigarrow v$ such that for every prefix P' of P we have $\mathsf{E}_v(u) + \mathsf{wt}(P') \leqslant 0$. Note that for every pair of nodes $u, v \in V$, we have $\mathsf{E}(u) \geqslant \mathsf{E}_v(u) + \mathsf{E}(v)$. Conventionally, we let $\mathsf{E}_v(u) = -\infty$ if no such value exists (i.e., there is no path $u \rightsquigarrow v$).

Remark 8.4. For any $u \in V$, let $P : u \rightsquigarrow v$ be a witness path for $\mathsf{E}_v(u) > -\infty$. Then

$$\mathsf{E}_v(u) + \mathsf{wt}(P) \leqslant 0 \implies \mathsf{E}_v(u) \leqslant -\mathsf{wt}(P) \leqslant -d(u, v)$$

i.e., the energy to reach v from u is upper bounded by minus the distance from u to v.

Highest-energy nodes. Given a (possibly infinite) path P with $\mathsf{wt}(P) < \infty$, we say that a node $x \in P$ is a *highest-energy node* of P if there exists a *highest-energy prefix* P_1 of P ending in x such that for any prefix P_2 of P we have $\mathsf{wt}(P_1) \geqslant \mathsf{wt}(P_2)$. Note that since the weights are integers, for every pair of paths P_1', P_2', it is either $|\mathsf{wt}(P_1') - \mathsf{wt}(P_2')| = 0$ or

$|\mathsf{wt}(P_1') - \mathsf{wt}(P_2')| \geqslant 1$. Therefore the set $\{\mathsf{wt}(P_i)\}_i$ of weights of prefixes of P has a maximum, and thus a highest-energy node always exists when $\mathsf{wt}(P) < \infty$. The following properties are easy to verify:

1. If x is a highest-energy node in a path $P : u \rightsquigarrow v$, then $\mathsf{E}_v(x) = 0$.

2. If x is a highest-energy node in an infinite path $\mathcal{P}$, then $\mathsf{E}(x) = 0$.

The following lemma states that the energy $\mathsf{E}(u)$ of a node u is the maximum energy $\mathsf{E}_v(u)$ to reach a 0-energy node v.

Lemma 8.10. *For every $u \in V$, we have $\mathsf{E}(u) = \max_{v:\mathsf{E}(v)=0} \mathsf{E}_v(u)$.*

Proof. The direction $\mathsf{E}(u) \geqslant \max_{v:\mathsf{E}(v)=0} \mathsf{E}_v(u)$ is straightforward. For the other direction, consider that $\mathsf{E}(u) > -\infty$ (trivially, $-\infty \leqslant \max_{v:\mathsf{E}(v)=0} \mathsf{E}_v(u)$) and let $\mathcal{P}$ be a witness path for $\mathsf{E}(u)$. Since $\mathsf{E}(u) > -\infty$, we have $\mathsf{wt}(\mathcal{P}) < \infty$, and $\mathcal{P}$ has some highest-energy node x, thus $\mathsf{E}(x) = 0$. Since x is on the witness $\mathcal{P}$ of $\mathsf{E}(u)$, we have $\mathsf{E}(u) \leqslant \mathsf{E}_x(u) \leqslant \max_{v:\mathsf{E}(v)=0} \mathsf{E}_v(u)$. The result follows. □

8.5.1 The Decision Problem for General Graphs

Here we address the decision problem, namely, given some node $u \in V$ and an initial credit $c \in \mathbb{Z}_{\leqslant 0}$, determine whether $\mathsf{E}(u) \geqslant c$. The following lemma states that if $\mathsf{E}(u) \geqslant c$, then a non-positive cycle can be reached from u with initial credit c, by paths of length less than n.

Lemma 8.11. *For every $u \in V$ and $c \in \mathbb{Z}_{\leqslant 0}$, we have that $\mathsf{E}(u) \geqslant c$ iff there exists a simple cycle C such that (i) $\mathsf{wt}(C) \leqslant 0$ and (ii) for every $v \in C$ we have that $\mathsf{E}_v(u) \geqslant c$, which is witnessed by a path $P_v : u \rightsquigarrow v$ with $|P_v| < n$.*

Proof. For the one direction, if $\mathsf{wt}(C) \leqslant 0$ we have $\mathsf{wt}(C^\omega) < \infty$, thus C contains a 0-energy node w. By Lemma 8.10, $\mathsf{E}(u) = \max_{v:\mathsf{E}(v)=0} \mathsf{E}_v(u) \geqslant \mathsf{E}_w(u) \geqslant c$. For the other direction, let $\mathcal{P}$ be a witness path for $\mathsf{E}(u)$, and we can assume w.l.o.g. that $\mathcal{P}$ does not contain positive cycles. Then for every prefix $P_v : u \rightsquigarrow v$ of $\mathcal{P}$ we have $\mathsf{E}(u) + \mathsf{wt}(P_v) \leqslant 0$, thus $\mathsf{E}_v(u) \geqslant \mathsf{E}(u) \geqslant c$, and the n-th such prefix contains a non-positive cycle C. The result follows. □

Algorithm $\mathsf{DecisionEnergy}$**.** Lemma 8.11 suggests a way to decide whether $\mathsf{E}(u) \geqslant c$. First, we start with energy c from u, and perform a sequence of $n-1$ relaxation steps, similar to the Bellman-Ford algorithm, to discover the set V_u^c of nodes that can be reached from u with initial credit c by a path of length at most $n-1$. Afterwards, we perform a Bellman-Ford computation on the subgraph $G[V]_u^c$ induced by the set V_u^c. By Lemma 8.11, we have that $\mathsf{E}(u) \geqslant c$ iff $G[V]_u^c$ contains a non-positive cycle. Algorithm 25 (DecisionEnergy) gives a formal description. The *for* loop in Line 6-Line 12 is similar to the procedure ROUND from the algorithm of [Bouyer *et al.*, 2008].

Detecting non-positive cycles. It is known that the Bellman-Ford algorithm can detect negative cycles. To detect non-positive cycles in a graph G with n nodes and weight function wt, we execute Bellman-Ford on G with a slightly modified weight function wt' for which $\mathsf{wt}'(e) = \mathsf{wt}(e) - \frac{1}{n}$. Then for any simple cycle C in G we have $\mathsf{wt}(C) \leqslant 0$ iff $\mathsf{wt}'(C) < 0$. Indeed,

$$\mathsf{wt}'(C) < 0 \iff \sum_{e \in C} \mathsf{wt}(e) - \sum_{e \in C} \frac{1}{n} < 0 \iff \mathsf{wt}(C) < \frac{|C|}{n} \iff \mathsf{wt}(C) \leqslant 0$$

since $|C| \leqslant n$ and $\mathsf{wt}(C) \in \mathbb{Z}$.

The correctness of DecisionEnergy follows directly from Lemma 8.11. The time complexity is $O(n \cdot m)$ time spent in the *for* loop of Line 6-Line 12, plus $O(n \cdot m)$ time for the Bellman-Ford. We thus obtain the following theorem.

Theorem 8.4. *Let $G = (V, E, \mathsf{wt})$ be a weighted graph of n nodes and m edges. Let $u \in V$ be an initial node, and $c \in \mathbb{Z}_{\leqslant 0}$ be an initial credit. The decision problem of whether $\mathsf{E}(u) \geqslant c$ can be solved in $O(n \cdot m)$ time and $O(n)$ space.*

8.5.2 The Value Problem for General Graphs

We now turn our attention to the value version of the minimum initial credit problem, where the task is to determine $\mathsf{E}(u)$ for every node u. The following lemma establishes that if for all energies to reach some node v we have $\mathsf{E}_v(w) < 0$, then $\mathsf{E}_v(u) = -d(u, v)$, i.e., the energy to reach v from every node u is minus the distance from u to v.

Lemma 8.12. *If for all $w \in V \setminus \{v\}$ we have $\mathsf{E}_v(w) < 0$, then for each $u \in V \setminus \{v\}$ we have $\mathsf{E}_v(u) = -d(u, v)$.*

Algorithm 25: DecisionEnergy

Input: A weighted graph $G = (V, E, \mathsf{wt})$, a node $u \in V$, an initial energy $c \in \mathbb{Z}_{\leqslant 0}$

Output: True iff $\mathsf{E}(u) \geqslant c$

```
   // Initialization
1  foreach v ∈ V do
2  |  Assign D(s) ← ∞
3  end
4  Assign D(u) ← c
5  Assign V_u^c ← {u}
   // n − 1 relaxation steps to discover V_u^c
6  for i ← 1 to n − 1 do
7  |  foreach (v, w) ∈ E do
8  |  |  if D(w) ⩾ D(v) + wt(v, w) and D(v) + wt(v, w) ⩽ 0 then
9  |  |  |  Assign D(w) ← D(v) + wt(v, w)
10 |  |  |  Assign V_u^c ← V_u^c ∪ {w}
11 |  end
12 end
13 Execute Bellman-Ford on G[V]_u^c
14 return True iff a non-positive cycle is discovered
```

Proof. Let $P : u \rightsquigarrow v$ be a witness path to the distance, i.e., $\mathsf{wt}(P) = d(u,v) < \infty$ (if $d(u,v) = \infty$ the statement is trivially true). Since every highest-energy node x of P has $\mathsf{E}_v(x) = 0$, we have that $x = v$. Hence, P is a highest-energy prefix of itself, and for each prefix P' of P we have $-\mathsf{wt}(P) + \mathsf{wt}(P') \leqslant 0$ and thus $\mathsf{E}_v(u) \geqslant -\mathsf{wt}(P) = -d(u,v)$. By Remark 8.4, it is $\mathsf{E}_v(u) \leqslant -d(u,v)$. The result follows. ☐

An $O(n^2 \cdot m)$ time solution to the value problem. Lemma 8.12 together with Theorem 8.4 lead to an $O(n^2 \cdot m)$ method for solving the minimum initial credit value problem. First, we compute the set $X = \{v \in V : \mathsf{E}(v) = 0\}$ in $O(n^2 \cdot m)$ time, by testing whether $\mathsf{E}(u) \geqslant 0$ for each node u. Afterwards, we contract the set X to a new node z, and by Lemma 8.10 for every remaining node u we have $\mathsf{E}(u) = \max_{v \in X} \mathsf{E}_v(u) = \mathsf{E}_z(u)$. Since $u \notin X$, the energy of u is strictly negative, and thus $\mathsf{E}_z(u) < 0$. Finally, by Lemma 8.12, we have $\mathsf{E}_z(u) = -d(u,z)$. Hence it suffices to compute the distance of each node u to z, which can be obtained in $O(n \cdot m)$ time.

In the remaining of this subsection we provide a refined solution of $O(k \cdot n \cdot m)$ time, where $k = |X| + 1$ is the number of 0-energy nodes (plus one). Hence this solution is faster in graphs where $k = o(n)$. This is achieved by algorithm ZeroEnergyNodes for computing the set X faster.

Determining the 0-energy nodes. The first step for solving the minimum initial credit problem is determining the set X of all 0-energy nodes of G. To achieve this, we construct the graph $G_2 = (V_2, E_2, \mathsf{wt}_2)$ with a fresh node $z \notin V$ as follows:

1. The node set is $V_2 = V \cup \{z\}$,
2. The edge set is $E_2 = E \cup (\{z\} \times V)$,
3. The weight function $\mathsf{wt}_2 : E_2 \to \mathbb{Z}$ is

$$\mathsf{wt}_2(u,v) = \begin{cases} 0 & \text{if } u = z \\ \mathsf{wt}(u,v) & \text{otherwise} \end{cases}$$

Remark 8.5. Since for every outgoing edge (z,x) of z we have $\mathsf{wt}_2(z,x) = 0$, if z is a highest-energy node in a path of G_2, so is x. Hence every non-positive cycle in G_2 has a highest-energy node other than z.

Note that for every $u \in V$, the energy $\mathsf{E}(u)$ is the same in G and G_2.

Algorithm ZeroEnergyNodes**.** Algorithm 26 describes ZeroEnergyNodes for obtaining the set of all 0-energy nodes in G_2. Informally, the algorithm performs a sequence of modifications on a graph $\mathcal{G}$, initially identical to G_2. In each step, the algorithm executes a Bellman-Ford computation on the current graph $\mathcal{G}$ with z as the source node, as long as a non-positive cycle C is discovered. For every such C, it determines a highest-energy node w of C, inserts w to a set of discovered nodes $\mathcal{X}$, and modifies $\mathcal{G}$ by replacing every incoming edge (x, w) with an edge (x, z) of the same weight, and then removing w. Finally, the algorithm returns the set $\mathcal{X}$. See Fig. 8.2 for an illustration.

As 0-energy nodes are discovered, ZeroEnergyNodes performs a sequence of modifications to the graph $\mathcal{G}$. We denote by $\mathcal{G}^k$ the graph $\mathcal{G}$ after the k-th node has been added to $\mathcal{X}$ (and $\mathcal{G}^0 = G_2$). We also use the superscript-k in our graph notation to make it specific to $\mathcal{G}^k$ (e.g. $d^k(u, z)$ and $\mathsf{E}^k_z(u)$ denote respectively the distance from u to z, and the energy to reach z from u in $\mathcal{G}^k$). The following two lemmas establish the correctness of ZeroEnergyNodes.

Lemma 8.13. *For every $w \in \mathcal{X}$ we have $\mathsf{E}(w) = 0$.*

Proof. The proof is by induction on the size of $\mathcal{X}$. It is trivially true when $|\mathcal{X}| = 0$. For the inductive step, let w be the $k+1$-th node added in $\mathcal{X}$. By Line 7, w is a highest-energy node in a non-positive cycle C of $\mathcal{G}^k$. We split into two cases.

1. If $z \notin C$, then C is also a cycle of G, hence w is a highest-energy node in the infinite path $\mathcal{P} = C^\omega$ of G, and $\mathsf{E}(w) = 0$.

2. If $z \in C$, let x be the node before z in C. By the modifications of Line 11 and Line 14, it is $wt^k(x, z) = \mathsf{wt}_2(x, w')$, where w' is a node that has been added to $\mathcal{X}$ when the algorithm run on $\mathcal{G}^i$ for some $i < k$. It follows that w is a highest-energy node in a path $P : z \rightsquigarrow w'$ in G_2, and thus a highest-energy node in a suffix $P' : w \rightsquigarrow w'$ of P, where P' is a path in G. Hence $\mathsf{E}_{w'}(w) = 0$. By the induction hypothesis, w' is a 0-energy node, i.e., $\mathsf{E}(w') = 0$, thus by Lemma 8.10 we have $\mathsf{E}(w) \geqslant \mathsf{E}_{w'}(w) = 0$.

The result follows. □

Lemma 8.14. *For every $w \in V : \mathsf{E}(w) = 0$ we have $w \in \mathcal{X}$.*

Algorithm 26: ZeroEnergyNodes

Input: A weighted graph $G_2 = (V_2, E_2, \mathsf{wt}_2)$

Output: The set $\{v \in V_2 \setminus \{z\} : \mathsf{E}(v) = 0\}$

```
1  Initialize sets 𝒱 ← V_2, ℰ ← E_2 and map wt ← wt_2
2  Let 𝒢 = (𝒱, ℰ, wt)
3  Initialize set 𝒳 ← ∅
4  while True do
5      Execute Bellman-Ford from source node z in 𝒢
6      if exists non-positive cycle C then
7          Determine a highest-energy node w ≠ z in C
8          Assign 𝒳 ← 𝒳 ∪ {w}
9          foreach edge (x, w) ∈ ℰ do
10             if (x, z) ∉ ℰ then
11                 Assign ℰ ← ℰ ∪ {(x, z)}
12                 Assign wt(x, z) ← wt_2(x, w)
13             else
14                 Assign wt(x, z) ← min(wt_2(x, w), wt(x, z))
15             end
16         end
17         Assign 𝒱 ← 𝒱 \ {w}
18     else
19         return 𝒳
20     end
21 end
```

Proof. Consider any $w \in V : \mathsf{E}(w) = 0$. For some $i \in \mathbb{N}$, we say that $\mathcal{G}^i$ "is aware of w" if either $\mathcal{G}^i$ has a non-positive cycle $C : w \rightsquigarrow w$, or $w \in \mathcal{X}$ when $|\mathcal{X}| = i$. Note that when ZeroEnergyNodes terminates there are no non-positive cycles in $\mathcal{G}^{|\mathcal{X}|}$. Hence, it suffices to argue that there exists a $k \in \mathbb{N}$ such that for each $i \geqslant k$, $\mathcal{G}^i$ is aware of w. We first argue that there exists a k for which $\mathcal{G}^k$ is aware of w.

Let $\mathcal{P}$ be a witness for $\mathsf{E}(w) = 0$, hence $\mathcal{P}$ traverses a non-positive cycle C_1 in G, thus C_1 exists in $\mathcal{G}^0$. Then there exists a smallest $j \in \mathbb{N}$ such that some node w' of $\mathcal{P}$ is identified as a highest-energy node in a non-positive cycle C_2 of $\mathcal{G}^j$ (possibly $C_1 = C_2$), and inserted to X. If $w = w'$, we have that $\mathcal{G}^j$ is aware of w. Otherwise, since $\mathsf{E}(w) = 0$ and w' is a node in the witness $\mathcal{P}$, we have $\mathsf{E}_{w'}(w) = 0$. By the choice of w', the path $\mathcal{P}$ exists in $\mathcal{G}^j$, therefore $\mathsf{E}^j_{w'}(w) = \mathsf{E}_{w'}(w) = 0$, and by Remark 8.4, we have $d^j(w, w') \leqslant 0$. It is straightforward that after the modifications in Lines 11 and 14, we have that $d^{j+1}(w, z) \leqslant d^j(w, w') \leqslant 0$, and since $wt^j(z, w) = \mathsf{wt}_2(z, w) = 0$, we have a non-positive cycle $C : w \rightsquigarrow w$ in $\mathcal{G}^{j+1}$ through z. Hence either $\mathcal{G}^j$ or $\mathcal{G}^{j+1}$ is aware of w, thus there exists a $k \in \mathbb{N}$ for which $\mathcal{G}^k$ is aware of w.

Finally, observe that the distance $d^i(w, z)$ does not increase in any $\mathcal{G}^i$ for $i \geqslant k$ until w is inserted to $\mathcal{X}$, hence for each $i \geqslant k$, the graph $\mathcal{G}^i$ is aware of w. The desired result follows. □

Lemmas 8.13 and 8.14 establish that $\mathcal{X} = X$, i.e. the set $\mathcal{X}$ returned by the algorithm is the set X of zero-energy nodes of G.

Determining the negative-energy nodes. Having computed the set X of all the 0-energy nodes of G, the second step for solving the minimum initial value credit problem is to determine the energy of every other node $u \in V \setminus X$. Recall the graph $\mathcal{G}^{|X|} = (\mathcal{V}^{|X|}, \mathcal{E}^{|X|}, wt^{|X|})$ after the end of ZeroEnergyNodes.

Lemma 8.15. *For every $u \in V \setminus X$ we have $\mathsf{E}(u) = -d^{|X|}(u, z)$.*

Proof. Consider any node $u \in V \setminus X = \mathcal{V}^{|X|} \setminus \{z\}$. By Lemma 8.12, in the graph G we have $\mathsf{E}(u) = \max_{v:\mathsf{E}(v)=0} \mathsf{E}_v(u)$, and by the correctness of ZeroEnergyNodes from Lemmas 8.13 and 8.14 we have $X = \{v : \mathsf{E}(v) = 0\}$, thus $\mathsf{E}(u) = \max_{v\in X} \mathsf{E}_v(u)$. It is straightforward to verify that at the end of ZeroEnergyNodes, we have $\max_{v\in X} \mathsf{E}_v(u) = \mathsf{E}^{|X|}_z(u)$, i.e., the maximum energy to reach the set X in G is the energy to reach z in $\mathcal{G}^{|X|}$. For all $v \in \mathcal{V}^{|X|} \setminus \{z\}$ it is

$\mathsf{E}_z^{|X|}(v) < 0$, otherwise we would have $\mathsf{E}(v) = 0$ and thus $v \in X$ and $v \notin \mathcal{V}^{|X|}$. Then by Lemma 8.12, $\mathsf{E}_z^{|X|}(u) = -d^{|X|}(u, z)$. We conclude that $\mathsf{E}(u) = -d^{|X|}(u, z)$. □

Hence, to compute the energy $\mathsf{E}(u)$ of every node $u \in V \setminus X$, it suffices to compute its distance to z in $\mathcal{G}^{|X|}$. This is straightforward by reversing the edges of $\mathcal{G}^{|X|}$ and performing a Bellman-Ford computation with z as the source node. Fig. 8.2 illustrates the algorithms on a small example. We obtain the following theorem.

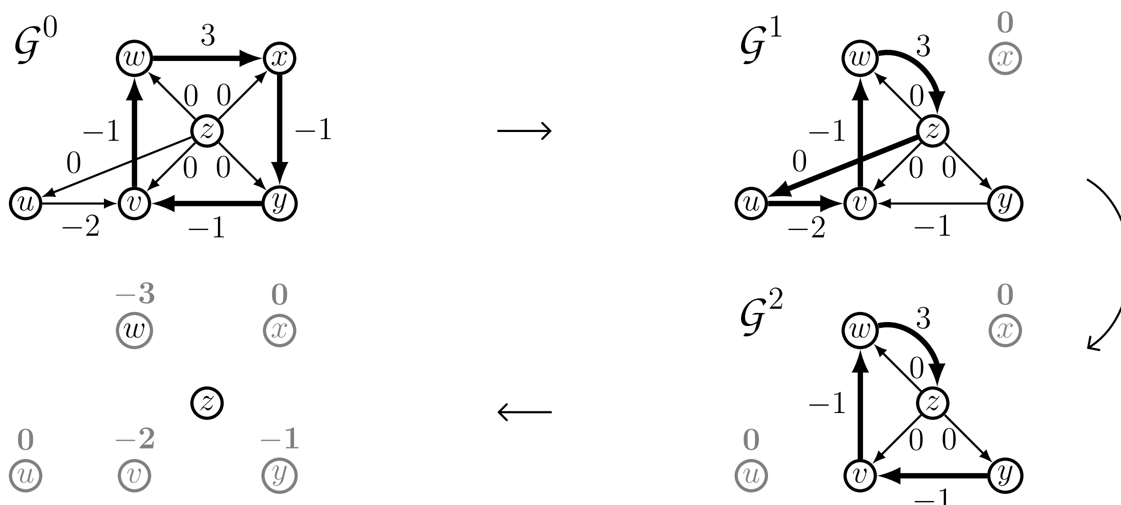


Figure 8.2: Solving the value problem using operations on the graph $\mathcal{G}$. Initially we examine $\mathcal{G}^0$, and a non-positive cycle is found (boldface edges) with highest-energy node x. Thus $\mathsf{E}(x) = 0$, and we proceed with $\mathcal{G}^1$, to discover $\mathsf{E}(u) = 0$. In $\mathcal{G}^2$ all cycles are positive, and the energy of each remaining node is minus its distance to z.

Theorem 8.5. *Let $G = (V, E, \mathsf{wt})$ be a weighted graph of n nodes and m edges, and $k = |\{v \in V : \mathsf{E}(v) = 0\}| + 1$. The minimum initial credit value problem for G can be solved in $O(k \cdot n \cdot m)$ time and $O(n)$ space.*

Proof. Lemmas 8.13 to 8.15 establish the correctness, so it remains to argue about the complexity. The *while* block of Line 4 is executed at most once for each 0-energy node, hence at most k times. Inside the block, the execution of Bellman-Ford in Line 5 requires $O(n \cdot m)$ time and $O(m)$ space. Since the Bellman-Ford algorithm uses backpointers to remember predecessors of nodes in distances, a highest-energy node w of a non-positive cycle C in Line 7 can be determined in $O(n)$. Finally, the *for* loop of Line 9 will consider each edge (x, w) at most once, hence it requires $O(m)$ for all iterations of the *while* loop. Thus $\mathsf{ZeroEnergyNodes}$ uses $O(k \cdot n \cdot m)$

time and $O(n)$ space in total. The last execution of Bellman-Ford to determine the energy of negative-energy nodes does not affect the complexity. The result follows. □

Corollary 8.3. *Let $G = (V, E, \mathsf{wt})$ be a weighted graph of n nodes and m edges. The minimum initial credit value problem for G can be solved in $O(n^2 \cdot m)$ time and $O(n)$ space.*

8.5.3 The Value Problem for Constant-treewidth Graphs

We now turn our attention to the minimum initial credit value problem for constant-treewidth graphs $G = (V, E, \mathsf{wt})$. Note that in such graphs $m = O(n)$, thus Theorem 8.5 gives an $O(n^3)$ time solution as compared to the existing $O(n^4 \cdot \log(n \cdot W))$ time solution. This section shows that we can do significantly better, namely reduce the time complexity to $O(n \cdot \log n)$. This is mainly achieved by algorithm ZeroEnergyNodesTW for computing the set X of 0-energy nodes fast in constant-treewidth graphs.

Extended $+$ and $\min$ operators. Recall the graph $G_2 = (V_2, E_2, \mathsf{wt}_2)$ from the last section. Given $\mathrm{Tree}(G)$, a nicely rooted, balanced and binary tree-decomposition $\mathrm{Tree}(G_2)$ of G_2 with width increased by 1 can be easily constructed by (i) inserting z to every bag of $\mathrm{Tree}(G)$, and (ii) adding a new root bag that contains only z. Let $\mathcal{I} = \mathbb{Z} \times V \times \mathbb{Z}$. For a map $f : V_2 \times V_2 \to \mathbb{Z}$, define the map $g_f : V_2 \times V_2 \to \mathcal{I}$ as

$$g_f(u, v) = \begin{cases} (f(u, v), u, 0) & \text{if } f(u, v) < 0 \text{ or } v = z \\ (f(u, v), v, f(u, v)) & \text{otherwise} \end{cases}$$

and for triplets of elements $\alpha_1 = (a_1, b_1, c_1), \alpha_2 = (a_2, b_2, c_2) \in \mathcal{I}$, define the operations

1. $\mathbf{min}(\alpha_1, \alpha_2) = \alpha_i$ with $i = \arg\min_{j \in \{1,2\}} a_j$
2. $\alpha_1 + \alpha_2 = (a_1 + a_2, b, c)$, where $c = \max(c_1, a_1 + c_2)$ and $b = b_1$ if $c = c_1$ else $b = b_2$.

In words, if f is a weight function, then $g_f(u, v)$ selects the weight of the edge (u, v), and its highest-energy node (i.e., u if $f(u, v) < 0$, and v otherwise, except when $v = z$), together with the weight to reach that highest energy node node from u. Recall that algorithm MinCycle from Section 8.3 traverses a tree-decomposition bottom-up, and for each encountered bag B stores a map LD_B such that $\mathsf{LD}_B(u, v)$ is upper bounded by the weight of the shortest U-shaped simple

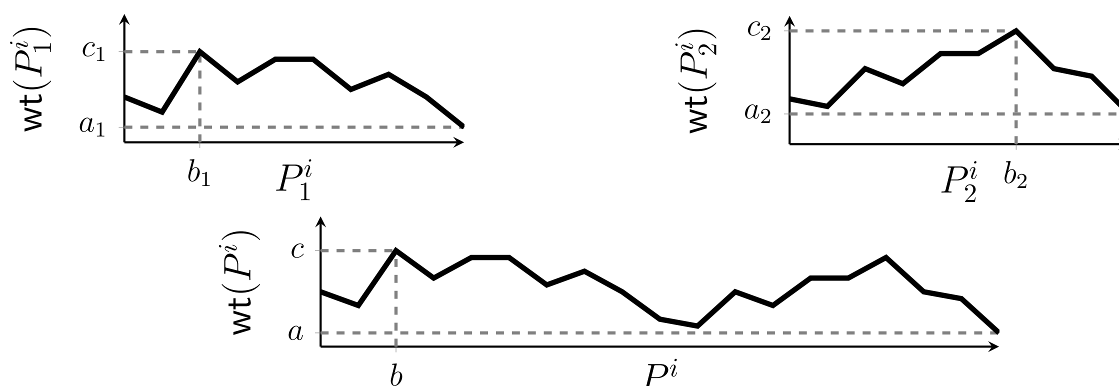


Figure 8.3: Illustration of the $\alpha_1 + \alpha_2$ operation, corresponding to concatenating paths P_1 and P_2. The path P_j^i denotes the i-th prefix of P_j. We have $P = P_1 \circ P_2$, and the corresponding tripplet $\alpha = (a, b, c)$ denotes the weight a of P, its highest-energy node b, and the weight c of a highest-energy prefix.

path $u \rightsquigarrow v$ (or simple cycle, if $u = v$). Our algorithm ZeroEnergyNodesTW for determining all 0-energy nodes is similar, only that now LD_B stores triplets (a, b, c) where a is the weight of a U-shaped path P, b is a highest-energy node of P, and c the weight of a highest-energy prefix of P. For two triplets $\alpha_1 = (a_1, b_1, c_1), \alpha_2 = (a_2, b_2, c_2) \in \mathcal{I}$ corresponding to U-shaped paths P_1 and P_2, $\mathbf{min}(\alpha_1, \alpha_2)$ selects the path with the smallest weight, and $\alpha_1 + \alpha_2$ determines the weight, a highest-energy node, and the weight of a highest-energy prefix of the path $P_1 \circ P_2$ (see Fig. 8.3).

Algorithm ZeroEnergyNodesTW**.** The algorithm ZeroEnergyNodesTW for computing the set of 0-energy nodes in constant-treewidth graphs follows the same principle as ZeroEnergyNodes for general graphs. It stores a map of edge weights $wt : E_2 \rightarrow \mathbb{Z} \cup \{\infty\}$, and initially $wt(u, v) = \mathsf{wt}_2(u, v)$ for each $(u, v) \in E_2$. The algorithm performs a bottom-up pass, and computes in each bag the local distance map $\mathsf{LD}_B : B \times B \rightarrow \mathcal{I}$ that captures U-shaped $u \rightsquigarrow v$ paths, together with their highest-energy nodes. When a non-positive cycle C is found in some bag B, the method KillCycle is called to modify the edges of a highest-energy node w of C and its incoming neighbors by updating the map wt. These updates generally affect the distances between the rest of the nodes in the graph, hence some local distance maps LD_B need to be corrected. However, each such edge modification only affects the local distance map of bags that appear in a path from a bag B' to some ancestor B'' of B'. Instead of restarting the computation as in ZeroEnergyNodes, the method Update is called to correct those local distance maps along the path $B' \rightsquigarrow B''$.

Algorithm 27: ZeroEnergyNodesTW

Input: A weighted graph $G_2 = (V_2, E_2, \mathsf{wt}_2)$ and a nicely rooted, binary tree-decomposition $\mathrm{Tree}(G_2)$

Output: The set $\{v \in V_2 \setminus \{z\} : \mathsf{E}(v) = 0\}$

```
   // Initialization
1  Assign X ← ∅
2  foreach u, v ∈ V_2 do
3  |  if (u, v) ∈ E_2 then
4  |  |  Assign wt(u, v) ← wt_2(u, v)
5  |  else
6  |  |  Assign wt(u, v) ← ∞
7  |  end
8  end
   // Computation
9  Apply a post-order traversal on Tree(G), and examine each bag B with children B_1, B_2
10 begin
11 |  foreach u, v ∈ B do
12 |  |  Assign LD_B(u, v) ← min(LD_{B_1}(u, v), LD_{B_2}(u, v), g_wt(u, v))
13 |  end
14 |  if B is the root bag of a node x then
15 |  |  foreach u, v ∈ B do
16 |  |  |  Assign LD'_B(u, v) ← min(LD_B(u, v), LD_B(u, x) + LD_B(x, v))
17 |  |  end
18 |  |  Assign LD_B ← LD'_B
19 |  |  if ∃u ∈ B with LD_B(u, u) = (a, b, c) where a ⩽ 0 then
20 |  |  |  Assign X ← X ∪ {b}
21 |  |  |  Execute KillCycle on b and B
22 end
23 return X
```

Algorithm 28: KillCycle

Input: A 0-energy node w and a bag B of $\mathrm{Tree}(G_2)$

Output: Updates the local distance function LD_B

1 **foreach** *edge* $(x, w) \in E_2$ **do**
2 Assign $wt(x, z) \leftarrow \min(\mathsf{wt}_2(x, w), wt(x, z))$
3 Assign $wt(x, w) \leftarrow \infty$
4 Assign $y \leftarrow \arg\max_{u \in \{x,w\}} \mathsf{Lv}(u)$
5 Let B' be the smallest-level ancestor of B_y examined by ZeroEnergyNodesTW so far
6 Execute Update on B_y and its ancestor B'
7 **end**
8 **return** LD_B

Algorithm 29: Update

Input: A bag B' and an ancestor B''

Output: The local distances LD_B along the path $B' \rightsquigarrow B''$

1 Traverse the path $B' \rightsquigarrow B''$ bottom-up, and examine each bag B with children B_1, B_2
2 **begin**
3 **foreach** $u, v \in B$ **do**
4 Assign $\mathsf{LD}_B(u, v) \leftarrow \mathbf{min}(\mathsf{LD}_{B_1}(u, v), \mathsf{LD}_{B_2}(u, v), g_{wt}(u, v))$
5 **end**
6 **if** B *is the root bag of a node* x **then**
7 **foreach** $u, v \in B$ **do**
8 Assign $\mathsf{LD}'_B(u, v) \leftarrow \mathbf{min}(\mathsf{LD}_B(u, v), \mathsf{LD}_B(u, x) + \mathsf{LD}_B(x, v))$
9 **end**
10 Assign $\mathsf{LD}_B \leftarrow \mathsf{LD}'_B$
11 **if** $\exists u \in B$ *with* $\mathsf{LD}_B(u, u) = (a, b, c)$ *where* $a \leqslant 0$ **then**
12 Assign $X \leftarrow X \cup \{b\}$
13 Execute KillCycle on b and B
14 **end**

The following lemma establishes the correctness of ZeroEnergyNodesTW. Similarly as for Lemmas 8.13 and 8.14 we denote by $\mathcal{G}^k$ the graph obtained by considering the edges (u, v) for which $wt(u, v) < \infty$ when $|X| = k$.

Lemma 8.16. *For every $v \in V \setminus \{z\}$ we have $v \in X$ iff $\mathsf{E}(v) = 0$.*

Proof. We only need to argue that ZeroEnergyNodesTW correctly computes the non-positive cycles in every $\mathcal{G}^k$, as then the correctness follows from the correctness Lemmas 8.13 and 8.14 of ZeroEnergyNodes. Since by Remark 8.1 every cycle is a U-shaped path in some bag, it suffices to argue that whenever ZeroEnergyNodesTW examines a bag B (either directly, or through Update), every U-shaped simple cycle in B has been considered by the algorithm. This is true if no calls to KillCycle are made (*if* block in Line 19), as then ZeroEnergyNodesTW is the same as MinCycle, and hence it follows from Lemma 8.1.

Now consider that KillCycle is called and B' is the smallest-level bag examined by ZeroEnergyNodesTW so far. Let w be the 0-energy node, x an incoming neighbor of w, and $y = \arg\max_{u \in \{x,w\}} \mathsf{Lv}(u)$ (as in Line 4 of KillCycle). By the definition of U-shaped paths, the edge (x, w) appears only in paths that are U-shaped in bags along the path $B_y \rightsquigarrow B'$. Hence, after setting $wt(x, w) = \infty$ (Line 3 of KillCycle), it suffices to update the local distance maps of these bags. Similarly, after setting $wt(x, z) \leftarrow \min(\mathsf{wt}_2(x, w), wt(x, z))$ (Line 2 of KillCycle), since B_z is the root of $\mathrm{Tree}(G_2)$, it suffices to update the local distance maps in the bags along the path $B_x \rightsquigarrow B'$. Either $x = y$, or, by the properties of tree-decompositions, B_x is an ancestor of B_y. Hence in either case $B_x \rightsquigarrow B'$ is a subpath of $B_y \rightsquigarrow B'$, and both edge modifications in Lines 2 and 3 are handled correctly by calling Update on B_y and its ancestor B'. The result follows. $\square$

Lemma 8.17. *Algorithm* ZeroEnergyNodesTW *runs in $O(n \cdot \log n)$ time and $O(n)$ space.*

Proof. Let $h = O(\log n)$ be the height of $\mathrm{Tree}(G_2)$.

1. The method Update performs a constant number of operations to each bag in the path $B' \rightsquigarrow B''$ where B'' is ancestor of B', hence each call to Update requires $O(h)$ time.

2. The method KillCycle performs a constant number of operations locally and one call to Update for each incoming edge of w. Hence if w has k_w incoming edges, KillCycle requires

$O(h \cdot k_w)$ time. Since $\mathsf{KillCycle}$ sets $wt(x, w) = \infty$ for all incoming edges of w, the node w will not appear in non-positive cycles thereafter.

3. The algorithm $\mathsf{ZeroEnergyNodesTW}$ is similar to $\mathsf{MinCycle}$ which runs in $O(n)$ time and space (Lemma 8.4). The difference is in the additional *if* block in Line 19. Since $\mathsf{KillCycle}$ is called when a non-positive cycle is detected, it will be called at most once for each node $u \in V_2 \setminus \{z\}$ (from either $\mathsf{ZeroEnergyNodesTW}$ or Update). It follows that the total time of $\mathsf{ZeroEnergyNodesTW}$ is

$$O\left(n + \sum_u (h \cdot k_u)\right) = O(n + h \cdot |E_2|) = O(n \cdot \log n)$$

where k_u is the number of incoming edges of node u. Since $\mathsf{KillCycle}$ stores constant size of information in each bag of $\mathrm{Tree}(G_2)$, the $O(n)$ space bound follows.

□

After the set X of 0-energy nodes has been computed, it remains to execute one instance of the single-source shortest path problem on the graph $\mathcal{G}^{|X|}$ (similarly as for our solution on general graphs). Since single-source distances on tree-decompositions of constant width can be computed in $O(n)$ time [Chaudhuri and Zaroliagis, 1995], we obtain the following theorem.

Theorem 8.6. *Let $G = (V, E, \mathsf{wt})$ be a weighted graph of n nodes with constant treewidth. The minimum initial credit value problem for G can be solved in $O(n \cdot \log n)$ time and $O(n)$ space.*

8.6 Experimental Results

In the current section we report on preliminary experimental evaluation of our algorithms, and compare them to existing methods. Our algorithm for the minimum mean cycle problem provides improvement for constant-treewidth graphs, and has thus been evaluated on low-treewidth graphs obtained from the control-flow graphs of programs. For the minimum initial credit problem, we have implemented our algorithm for arbitrary graphs, thus the benchmarks used in this case are general graphs (i.e., not constant-treewidth graphs).

8.6.1 Minimum Mean Cycle

We have implemented our approximation algorithm for the minimum mean cycle problem, and we let the algorithm run for as many iterations until a minimum mean cycle was discovered, instead of terminating after $O(\log(\frac{n}{\epsilon}))$ iterations required by Theorem 8.3. We have tested its performance in running time and space against six other minimum mean cycle algorithms from Table 8.3 in control-flow graphs of programs. The algorithms of Burns and Lawler solve the more general ratio cycle problem, and have been adapted to the mean cycle problem as in [Dasdan *et al.*, 1998].

	[Madani, 2002]	[Burns, 1991]	[Lawler, 1976]	[Dasdan and Gupta, 1998]	[Hartmann and Orlin, 1993]	[Karp, 1978]
Time	$O(n^2)$	$O(n^3)$	$O(n^2 \cdot \log(n \cdot W))$	$O(n^2)$	$O(n^2)$	$O(n^2)$
Space	$O(n)$	$O(n)$	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n^2)$

Table 8.3: Asymptotic complexity of compared minimum mean cycle algorithms.

Setup. The algorithms were executed on control-flow graphs of methods of programs from the DaCapo benchmark suit [Blackburn, 2006], obtained using the Soot framework [Vallée-Rai *et al.*, 1999]. For each benchmark we focused on graphs of at least 500 nodes. This supplied a set of medium sized graphs (between 500 and 1300 nodes), in which integer weights were assigned uniformly at random in the range $\{-10^3, \dots, 10^3\}$. Memory usage was measured with [Brosius].

Results. Fig. 8.4 shows the average time and space performance of the examined algorithms (bars that exceeded the maximum value in the y-axis have been truncated). Our algorithm has much smaller running time than each other algorithm, in almost all cases. In terms of space, our algorithm significantly outperforms all others, except for the algorithms of Lawler, Burns, and Madani. Both ours and these three algorithms have linear space complexity, but ours also suffers some constant factor overhead from the tree-decomposition (i.e., the same node generally appears in multiple bags). Note that the strong performance of these three algorithms in space is followed by poor performance in running time.

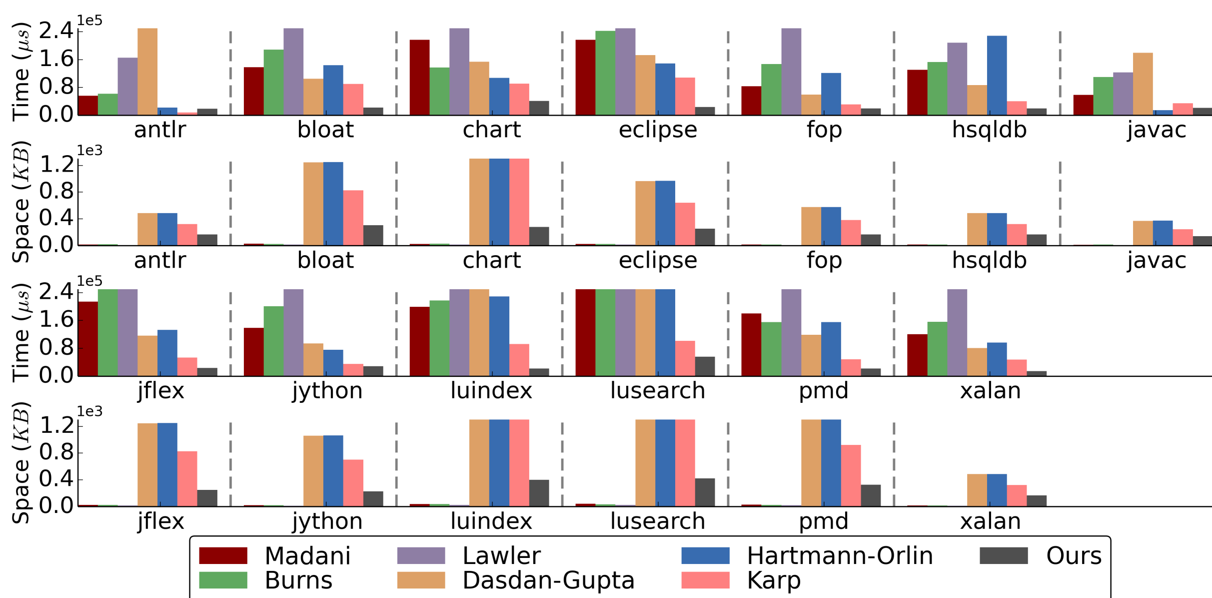


Figure 8.4: Average performance of minimum mean cycle algorithms.

8.6.2 Minimum Initial Credit

We have implemented our algorithm for the minimum initial credit problem on general graphs and experimentally evaluated its performance on a subset of benchmark weighted graphs from the DIMACS implementation challenges [dim]. Our algorithm was tested against the existing method of [Bouyer *et al.*, 2008]. The direct implementation of the algorithm of [Bouyer *et al.*, 2008] performed poorly, and for this we also implemented an optimized version (using techniques such as caching of intermediate results and early loop termination). Note that we compare algorithms for general graphs, without the low-treewidth restriction.

Setup. For each input graph we first computed its minimum mean value μ^* using Karp's algorithm, and then subtracted μ^* from the weight of each edge to ensure that at least one non-positive cycle exists (thus the energies are finite).

Results. Fig. 8.5 depicts the running time of the algorithm of [Bouyer *et al.*, 2008] (with and without optimizations) vs our algorithm. A timeout was forced at $10^{10}\mu s$. Our algorithm is orders of magnitude faster, and scales better than the existing method.

	Madani	Burns	Lawler	Dasdan-Gupta	Hartmann-Orlin	Karp	**Ours**
antlr	55814	61571	165789	284996	21893	7824	18402
bloat	138416	188356	350302	105145	144171	89949	22391
chart	216962	137112	573767	154062	107229	90717	40890
eclipse	216859	242323	667869	172792	148523	107864	23486
fop	83080	147384	406371	59176	121742	31557	19306
hsqldb	131041	153232	208328	86840	228632	40486	19957
javac	58443	110149	122996	179647	14719	34188	20874
jflex	214297	524822	554093	116820	133323	53329	23860
jython	139106	200922	503766	94052	75569	34864	28760
luindex	199650	217980	1240411	274319	228856	92379	22142
lusearch	433211	447280	1180051	263467	333297	101584	55652
pmd	180551	155118	585315	118578	155682	48326	21978
xalan	120897	156111	394458	81103	96873	47996	14493

Table 8.4: The time performance of Fig. 8.4 (in μs).

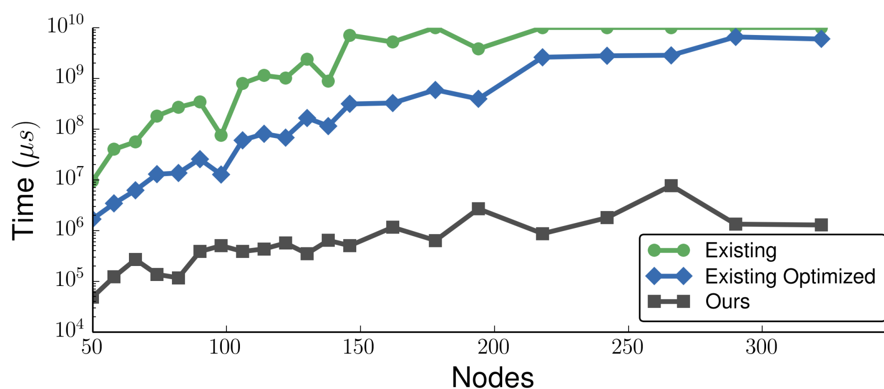


Figure 8.5: Comparison of running times for the minimum initial credit problem.

	Madani	Burns	Lawler	Dasdan-Gupta	Hartmann-Orlin	Karp	**Ours**
antlr	16805	21018	11144	486435	489176	322384	168648
bloat	29723	24500	19458	1245272	1249444	826645	306026
chart	27130	30567	18172	2025448	2029294	1347048	278586
eclipse	24215	26488	16293	965063	968595	640720	254393
fop	16845	17975	11052	576174	578646	382338	169738
hsqldb	16798	19309	11144	486435	489096	322384	168648
javac	14681	17047	9664	372697	375453	247019	144721
jflex	24561	26946	16322	1244495	1248036	826743	251549
jython	22518	23337	14899	1059291	1062570	703581	228207
luindex	39309	40223	25604	3521607	3526792	2342833	399076
lusearch	41488	33350	26991	3387914	3393343	2253403	422679
pmd	32204	24481	21021	1391551	1395786	923975	326137
xalan	16798	17763	11144	486435	489102	322384	168648

Table 8.5: The space performance of Fig. 8.4 (in KB).

n	Existing	Existing Optimized	**Ours**
50	9453565	1680924	48635
58	39744129	3394193	121774
66	55766874	6201044	267825
74	180080064	12833610	136239
82	267993314	13563936	116518
90	342779026	25453589	383292
98	74622910	12648395	501365
106	791441986	60294150	385799
114	1133055323	80584700	432290
122	1004898322	67982455	564838
130	2354354250	165193753	348112
138	881117317	114743182	636481
146	7050113907	311146051	501314
162	5179877563	324877384	1154447
178	Timeout	589873640	635155
194	3799301931	391240954	2672127
218	Timeout	2596083382	866213
242	Timeout	2774469734	1779512
266	Timeout	2839496222	7676638
290	Timeout	6526762301	1332403
322	Timeout	5929433611	1282258

Table 8.6: The time performance of Fig. 8.5 in μs.

9 Data-centric Dynamic Partial Order Reduction

9.1 Introduction

In this chapter we focus on the stateless model checking of concurrent programs. The standard partial-order reduction enumerative methods rely on the Mazurkiewicz equivalence [Mazurkiewicz, 1987] between traces, and attempt to explore as few representative traces as possible from each Mazurkiewicz class. There exists a rich body of dynamic partial-order reduction (DPOR) techniques for the enumerative exploration of the trace space based on the Mazurkiewicz equivalence [Flanagan and Godefroid, 2005; Godefroid, 2005; Musuvathi *et al.*, 2008; Abdulla *et al.*, 2014; Abdulla *et al.*, 2015; Zhang *et al.*, 2015]. A basic and fundamental question is whether coarser equivalence classes than the Mazurkiewicz equivalence can be applied to the stateless model checking and whether some DPOR-like approach can be developed based on such coarser equivalences. Here we give a positive answer to this question. We present a new, data-centric dynamic partial-order reduction (DC-DPOR) which explores a coarser partitioning of the trace space than the Mazurkiewicz partitioning, and does so by spending polynomial time per class.

9.1.1 Our contributions

In this chapter our contributions are as follows.

Observation equivalence. We introduce the new notion of *observation equivalence* (Sec-

tion 9.3.1), which is intuitively as follows: An observation function of a trace maps every read event to the write event it observes under sequentially consistent semantics. In contrast to every possible ordering of dependent control locations of the Mazurkiewicz equivalence, in the observation equivalence two traces are equivalent if they have the same observation function. The observation equivalence has the following properties.

1. *Soundness.* The observation equivalence is sufficient for exploring all local states of each process, and is thus sufficient for model checking wrt to local properties (similar to the Mazurkiewicz equivalence).

2. *Coarser.* Second, the observation equivalence is coarser than Mazurkiewicz equivalence, i.e., if two traces are Mazurkiewicz equivalent, then they are also observation equivalent (Section 9.3.1).

3. *Exponentially coarser.* Third, the observation equivalence can be exponentially more succinct than Mazurkiewicz equivalence, i.e., we present examples where the ratio of the number of equivalence classes between observation and Mazurkiewicz equivalence is exponentially small (Section 9.3.2).

In summary, the observation equivalence allows for a sound exploration method of the trace space which is always coarser, and in cases, strictly coarser than the fundamental Mazurkiewicz equivalence.

Principal difference. The principal difference between the Mazurkiewicz and our new observation equivalence is that while the Mazurkiewicz equivalence is *control-centric*, observation equivalence is *data-centric*. The data-centric approach takes into account read-write and memory consistency restrictions as opposed to the event-dependency relation of the Mazurkiewicz equivalence.

Data-centric DPOR. We devise a DPOR exploration of the trace space, called *data-centric* DPOR, based on the observation equivalence. Our DPOR algorithm is based on a notion of *annotations*, which are intended observation functions (see Section 9.4). The basic computational problem is, given an annotation, decide whether there exists a trace which realizes the annotation. We show that the computational problem is NP-complete in general, but for the important special case of *acyclic* architectures we present a polynomial-time (cubic-time) algorithm based on

reduction to 2-SAT (details in Section 9.4). Based our polynomial-time solution of the annotation problem, we present an algorithm for the stateless exploration of the trace space, which has the following properties.

1. For acyclic architectures, our algorithm is guaranteed to explore *exactly one* representative trace from each observation equivalence class, while spending *polynomial time* per class. Hence, our algorithm is *optimal* wrt the observation equivalence, and in several cases explores exponentially fewer traces than *any* enumerative method based on the Mazurkiewicz equivalence (details in Section 9.5).

2. For cyclic architectures, we consider an equivalence between traces which is finer than the observation equivalence; but coarser than the Mazurkiewicz equivalence, and in many cases is exponentially coarser. For this equivalence on traces, we again present an algorithm for DPOR that explore *exactly one* representative trace from each observation class, while spending *polynomial time* per class. Thus again our data-centric DPOR algorithm remains optimal under this trace equivalence for cyclic architectures (details in Section 9.6).

Experimental results. Finally, we perform a basic experimental comparison between the existing Mazurkiewicz-based DPOR and our data-centric DPOR on a set of academic benchmarks. Our results show a significant reduction in both running time and the number of explored traces.

Comparison with most relevant existing work. In the past, there have been attempts to devise enumerative explorations of the trace space. with respect to trace equivalences which are coarser than the Mazurkiewicz equivalence. We list here some of these works.

1. In [Wang *et al.*, 2008] the authors introduce the Peephole Partial-order Reduction, in which various transitions can be considered independent in some, but not all traces of the system. However, their approach utilizes a SAT/SMT solver to prune away redundant interleavings.

2. The work of [Huang, 2015] builds upon the theoretical model of maximal causality of [Şerbănuţă *et al.*, 2013] to provide an enumerative exploration where every explored trace corresponds to a distinct maximal causal model which captures the largest possible set of causally equivalent executions. The construction of each such trace queries an SMT solver with Integer Difference Logic (IDL) constraints, which requires exponential time in the worst case. In contrast, our algorithm has polynomial worst-case complexity per

trace. Even though SMT solvers scale well in practice, the important theoretical question of coarsening the trace space with polynomial worst-case guarantees had remained open. The experimental section of [Huang, 2015] also reports that the running time on larger instances is dominated by the time spent in the SMT procedure.

3. In [Rodríguez *et al.*, 2015] the authors introduce the unfolding-based Partial-order Reduction, which relies on state caching and cutoff events to further prune the trace space, and is thus stateful.

Organization The rest of this chapter is organized as follows.

1. In Section 9.2 we define the concurrent model, instances of which are used as inputs to our data-centric partial order reduction algorithm. We also introduce several definitions that help with the exposition of the ideas in this chapter.

2. In Section 9.3 we introduce our new, observation equivalence, and compare it with the standard Mazurkiewicz equivalence. In high level, two traces are observation equivalent if they contain the same events, and every read event observes the same write event in both traces.

3. In Section 9.4 we introduce the concept of annotations, which are used in guiding our enumerative exploration. Intuitively, annotations specify the intended relationship between read and write events of a trace. We also establish some complexity results regarding the problem of generating system traces that agree with a given annotation.

4. In Section 9.5 we present our DC-DPOR algorithm for the enumerative exploration of the classes of the observation equivalence of a concurrent system that comprises an acyclic architecture topology. Our algorithm is optimal, in the sense that it explore each observation class once, and spends only polynomial time for each class.

5. In Section 9.6 we extend DC-DPOR to cyclic architectures. In this case, the algorithm explores a partitioning of the state space that is finer that the observation equivalence, but remains coarser than the Mazurkiewicz equivalence, and can be even exponentially coarser.

6. In Section 9.7 we present a preliminary experimental evaluation of our DC-DPOR algo-

rithm on some academic benchmarks.

9.2 Preliminaries

In this section we introduce a simple model for concurrent programs that will be used for stating rigorously the key ideas of our data-centric DPOR. Similar (but syntactically richer) models have been used in [Flanagan and Godefroid, 2005; Abdulla *et al.*, 2014]. In Section 9.2.3 we discuss our various modeling choices and possible extensions.

Informal model. We consider a *concurrent system* of k processes under sequential consistency semantics. For the ease of presentation, we do not allow dynamic thread creation, i.e., k is fixed during any execution of the system. Each process is defined over a set of *local variables* specific to the process, and a set of *global variables*, which is common for all processes. Each process is represented as an acyclic *control-flow graph*, which results from unrolling the body of the process. A process consists of statements over the local and global variables, which we call *events*. The precise kind of such events is immaterial to our model, as we are only interested in the variables involved. In particular, in any such event we identify the local and global variables it involves, and distinguish between the variables that the event *reads* from and at most one variable that the event *writes* to. Such an event is *visible* if it involves global variables, and *invisible* otherwise. We consider that processes are *deterministic*, meaning that at any given time there is at most one event that each process can execute. Given the current state of the system, a scheduler chooses one process to execute a sequence of events that is invisibly maximal, that is, the sequence does not end while an invisible event from that process can be taken. The processes communicate by writing to and reading from the global variables. The system can exhibit nondeterministic behavior which is solely attributed to the scheduler, by choosing nondeterministically the next process to take an invisibly maximal sequence of events from any given state. We consider *locks* as the only synchronization primitive, with the available operations being acquiring a lock and releasing a lock. Since richer synchronization primitives are typically built using locks, this consideration is not restrictive, and helps with keeping the exposition of the key ideas simple.

9.2.1 Concurrent Computation Model

Here we present our model formally. Relevant notation is summarized in Table 9.1.

Relations and equivalence classes. A binary relation $\sim$ on a set X is an equivalence relation iff $\sim$ is reflexive, symmetric and transitive. Given an equivalence $\sim_R$ and some $x \in X$, we denote by $[x]_R$ the equivalence class of x under $\sim_R$, i.e.,

$$[x]_R = \{y \in X : \; x \sim_R y\}$$

The *quotient set* $X/ \sim_R := \{[x]_R \mid x \in X\}$ of X under $\sim_R$ is the set of all equivalence classes of X under $\sim_R$.

Notation on functions. We write $f : X \nrightarrow Y$ to denote that f is a partial function from X to Y. Given a (partial) function f, we denote by $\mathsf{dom}(f)$ and $\mathsf{img}(f)$ the domain and image set of f, respectively. For technical convenience, we think of a (partial) function f as a set of pairs $\{(x_i, y_i)\}_i$, meaning that $f(x_i) = y_i$ for all i, and use the shorthand notation $(x, y) \in f$ to indicate that $x \in \mathsf{dom}(f)$ and $f(x) = y$. Given (partial) functions f and g, we write $f \subseteq g$ if $\mathsf{dom}(f) \subseteq \mathsf{dom}(g)$ and for all $x \in \mathsf{dom}(f)$ we have $f(x) = g(x)$, and $f = g$ if $f \subseteq g$ and $g \subseteq f$. Finally, we write $f \subset g$ if $f \subseteq g$ and $f \neq g$.

Model syntax. We consider a *concurrent architecture* $\mathcal{P}$ that consists of a fixed number of *processes* $p_1, \dots, p_k$, i.e., there is no dynamic thread creation. Each process p_i is defined over a set of n_i *local variables* $\mathcal{V}_i$, and a set of *global variables* $\mathcal{G}$, which is common for all processes. We distinguish a set of *lock variables* $\mathcal{L} \subseteq \mathcal{G}$ which are used for process synchronization. All variables are assumed to range over a finite domain $\mathcal{D}$. Every process p_i is represented as an acyclic control-flow graph CFG_i which results from unrolling all loops in the body of p_i. Every edge of CFG_i is labeled, and called an *event*. In particular, the architecture $\mathcal{P}$ is associated with a set of *events* $\mathcal{E}$, a set of *read events* (or *reads*) $\mathcal{R} \subseteq \mathcal{E}$, a set of *write events* (or *writes*) $\mathcal{W} \subseteq \mathcal{E}$. Furthermore, locks are manipulated by a set of *lock-acquire* events $\mathcal{L}^A \subseteq \mathcal{R}$ and a set of *lock-release* events $\mathcal{L}^R \subseteq \mathcal{W}$, which are considered read events and write events respectively. The control-flow graph CFG_i of process p_i consists of events of the following types (where $\mathcal{V}_i = \{v_1, \dots, v_{n_i}\}$, $g \in \mathcal{G}$, $l \in \mathcal{L}$, $f_i : \mathcal{D}^{n_i} \to \mathcal{D}$ is a function on n_i arguments, and $b : \mathcal{V}_i^{n_i} \to \{\mathsf{True}, \mathsf{False}\}$ is a boolean function on n_i arguments).

1. $e : v \leftarrow \mathsf{read}\ g$, in which case $e \in \mathcal{R}$,

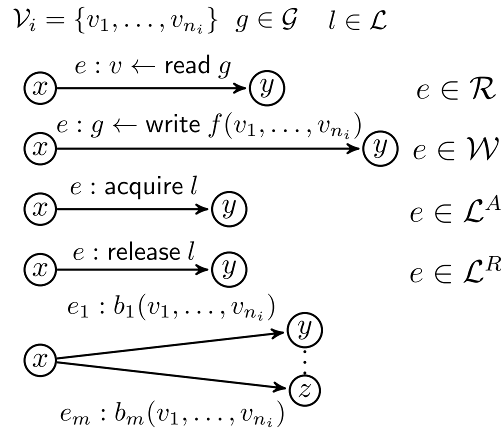


Figure 9.1: The control-flow graph CFG_i is a sequential composition of these five atomic graphs.

2. $e : g \leftarrow$ write $f(v_1, \dots, v_{n_i})$, in which case $e \in \mathcal{W}$,

3. e : acquire l, in which case $e \in \mathcal{R}$,

4. e : release l, in which case $e \in \mathcal{W}$,

5. $e_1 : b(v_1, \dots, v_{n_i})$.

Each CFG_i is a directed acyclic graph with a distinguished *root* node r_i, such that there is a path $r_i \rightsquigarrow x$ to every other node x of CFG_i. Each node x of CFG_i has either

1. zero outgoing edges, or

2. one outgoing edge (x, y) labeled with an event of a type listed in Item 1-Item 4, or

3. $m \geqslant 2$ outgoing edges $(x, y_1), \dots, (x, y_m)$ labeled with events $e_j : b_j(v_1, \dots, v_{n_i})$ of Item 5, and such that for all values of $v_1, \dots, v_{n_i}$, we have $b_j(v_1, \dots, v_n) \implies \neg b_l(v_1, \dots, v_{n_i})$ for all $j \neq l$.In this case, we call x a *branching* node.

For simplicity, we require that if x is a branching node, then for each edge (x, y) in CFG_i, the node y is not branching. Indeed, such edges can be easily contracted in a preprocessing phase. Fig. 9.1 provides a summary of the model syntax. We let $\mathcal{E}_i \subseteq \mathcal{E}$ be the set of events that appear in CFG_i of process p_i, and similarly $\mathcal{R}_i \subseteq \mathcal{R}$ and $\mathcal{W}_i \subseteq \mathcal{W}$ the sets of read and write events of p_i. Additionally, we require that $\mathcal{E}_i \cap \mathcal{E}_j = \varnothing$ for all $i \neq j$ i.e., all $\mathcal{E}_i$ are pairwise disjoint, and denote by $\mathsf{proc}(e)$ the process of event e. The *location* of an event $\mathsf{loc}(e)$ is the unique global variable it involves. Given two events $e, e' \in \mathcal{E}_i$ for some p_i, we write $\mathsf{PS}(e, e')$ if there is a path $e \rightsquigarrow e'$ in

CFG_i (i.e., we write $\mathsf{PS}(e, e')$ to denote that e is ordered before e' in the *program structure*).

We distinguish a set of *initialization events* $\mathcal{W}^{\mathcal{I}} \subseteq \mathcal{W}$ with $|\mathcal{W}^{\mathcal{I}}| = |\mathcal{G}|$ which are attributed to process p_1, and are used to initialize all the global variables to some fixed values. For every initialization write event $w^{\mathcal{I}}$ and for any event $e \in \mathcal{E}_i$ of process p_i, we define that $\mathsf{PS}(w^{\mathcal{I}}, e)$ (i.e., the initialization events occur before any event of each process). Fig. 9.2 illustrates the above definitions on the typical bank account example.

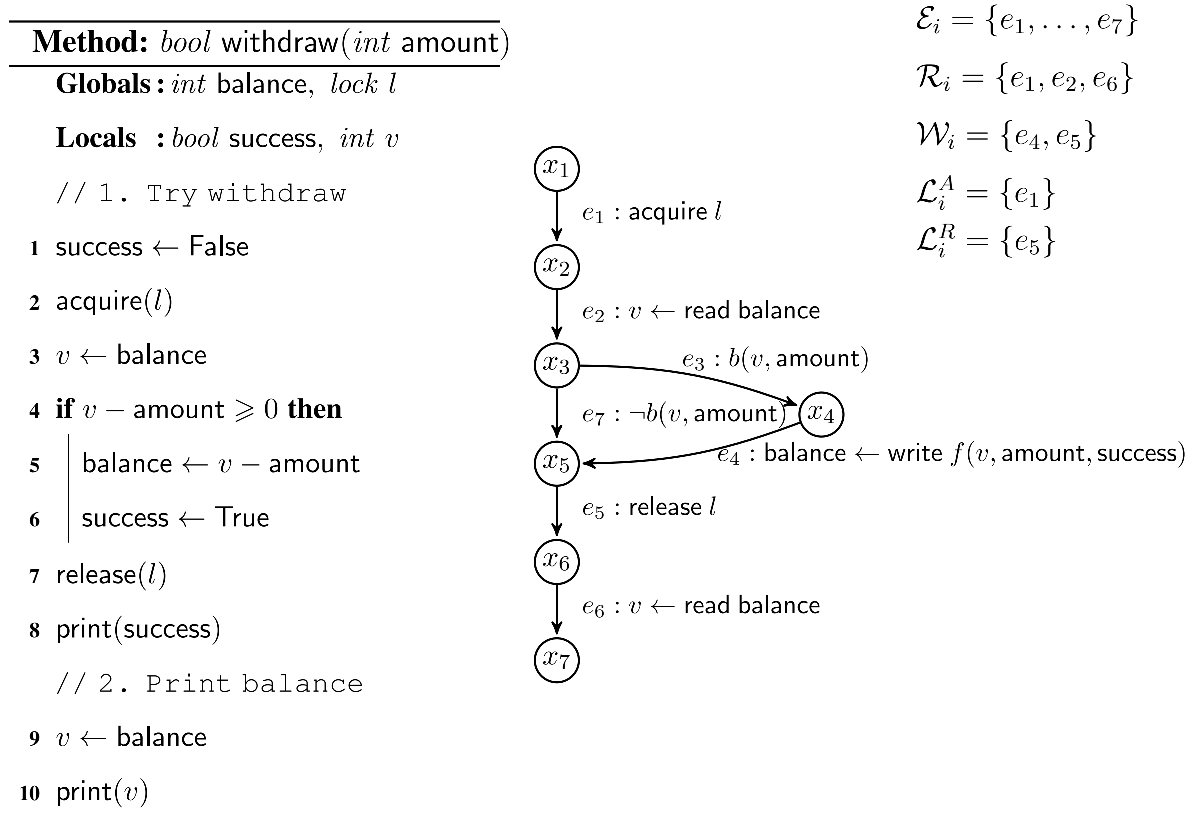


Figure 9.2: *(Left)*: A method withdraw executed whenever some amount is to be extracted from the balance of a bank account. *(Right)*: Representation of withdraw in our concurrent model. The root node is x_1. The program structure orders $\mathsf{PS}(e_2, e_4)$. We have $\mathsf{loc}(e_1) = \mathsf{loc}(e_5)$ and $\mathsf{loc}(e_2) = \mathsf{loc}(e_4) = \mathsf{loc}(e_6)$.

Model semantics. A *local state* of a process p_i is a pair $s_i = (x_i, \mathsf{val}_i)$ where x_i is a node of CFG_i (i.e., the program counter) and val_i is a valuation on the local variables $\mathcal{V}_i$. A *global state* of $\mathcal{P}$ is a tuple $s = (\mathsf{val}, s_1, \dots, s_k)$, where val is a valuation on the global variables $\mathcal{G}$ and s_i is a local state of process p_i. An event e along an edge (x, y) of a process p_i is *enabled* in s if $s_i = (x, \mathsf{val}_i)$ (i.e., the program counter is on node x) and additionally,

1. if e : $\mathsf{acquire}\ l$, then $\mathsf{val}(l) = \mathsf{False}$, and

2. if $e : b_j(v_1, \dots, v_{n_i})$, then $b_j(\mathsf{val}_i(v_1), \dots, \mathsf{val}_i(v_{n_i})) = \mathsf{True}$.

In words, if e acquires a lock l, then e is enabled iff l is free in s, and if x is a branching node, then e is enabled iff it respects the condition of the branch in s. Given a state s, we denote by $\mathsf{enabled}(s) \subseteq \mathcal{E}$ the set of enabled events in s, and observe that there is at most one enabled event in each state s from each process. The execution of an enabled event e along an edge (x, y) of p_i in state $s = (\mathsf{val}, s_1, \dots, s_k)$ results in a state $s' = (\mathsf{val}', s_1, \dots, s'_i, \dots, s_k)$, where $s'_i = (y, \mathsf{val}'_i)$. That is, the program counter of p_i has progressed to y, and the valuation functions val' and val'_i have been modified according to standard semantics, as follows:

1. $e : v \leftarrow \mathsf{read}\ g$ then $\mathsf{val}'_i(v) = \mathsf{val}(g)$,

2. $e : g \leftarrow \mathsf{write}\ f(v_1, \dots, v_{n_i})$ then $\mathsf{val}'(g) = f(\mathsf{val}_i(v_1), \dots, \mathsf{val}_i(v_{n_i}))$,

3. e : $\mathsf{acquire}\ l$ then $\mathsf{val}'(l) = \mathsf{True}$,

4. e : $\mathsf{release}\ l$ then $\mathsf{val}'(l) = \mathsf{False}$.

Moreover, val agrees with val' and val_i agrees with val'_i on all other variables. We write $s \xrightarrow{e} s'$ to denote that the execution of event e in s results in state s'. Let $\mathcal{S}_\mathcal{P}$ be the finite set (since variables range over a finite domain) of states of $\mathcal{P}$. The semantics of $\mathcal{P}$ are defined in terms of a transition system $\mathcal{A}_\mathcal{P} = (\mathcal{S}_\mathcal{P}, \Delta, s^0)$, where s^0 is the initial state, and $\Delta \subseteq \mathcal{S}_\mathcal{P} \times \mathcal{S}_\mathcal{P}$ is the transition relation such that

$$(s, s') \in \mathcal{A}_\mathcal{P} \text{ iff } \exists e \in \mathsf{enabled}(s) : \ s \xrightarrow{e} s'$$

and either e is an initialization event, or the program counter of p_1 has passed all initialization edges of p_1. We write $s \xrightarrow{e_1, \dots e_n} s'$ if there exists a sequence of states $\{s^i\}_{1 \leqslant i < n}$ such that

$$s \xrightarrow{e_1} s^1 \xrightarrow{e_2} \dots s^{n-1} \xrightarrow{e_n} s'$$

The initial state $s^0 = (\mathsf{val}, s^0_1, \dots, s^0_k)$ is such that the value $\mathsf{val}(g)$ of each global variable g comes from the unique initialization write event w with $\mathsf{loc}(w) = g$, and for each $s^0_i = (x_i, \mathsf{val}_i)$ we have that $x_i = r_i$ (i.e., the program counter of process p_i points to the root node of CFG_i). For simplicity we restrict $\mathcal{S}_\mathcal{P}$ to states s that are reachable from the initial state s^0 by a sequence of events $s^0 \xrightarrow{e_1, \dots, e_n} s$. We focus our attention on state spaces $\mathcal{S}_\mathcal{P}$ that are acyclic.

Architecture topologies. The architecture $\mathcal{P}$ induces a labeled undirected communication graph $G_{\mathcal{P}} = (V_{\mathcal{P}}, E_{\mathcal{P}}, \mathsf{wt}_{\mathcal{P}})$ where $V_{\mathcal{P}} = \{p_i\}_i$. There is an edge (p_i, p_j) if processes p_i, p_j access a common global variable or a common lock. The label $\mathsf{wt}(p_i, p_j)$ is the set of all such global variables and locks. We call $\mathcal{P}$ *acyclic* if $G_{\mathcal{P}}$ does not contain cycles. The class of acyclic architectures includes, among others, all architectures with two processes, star architectures, pipelines, tree-like and hierarchical architectures.

Notation	Interpretation
$\mathcal{P} = (p_i)_{i=1}^k$	the concurrent architecture of k processes
$\mathcal{G}, \mathcal{V}, \mathcal{L}$	the global, local and lock variables
$\mathcal{E}, \mathcal{W}, \mathcal{R}, \mathcal{L}^A$, $\mathcal{L}^R, \mathcal{W}^{\mathcal{I}}$	the set of events, write, read, lock-acquire lock-release and initialization events
$\mathsf{val}_i, \mathsf{val}$	valuations of local, global variables
$\mathsf{enabled}(s) \subseteq \mathcal{E}$	the set of enabled events in s
$s \xrightarrow{e_1,\dots,e_n} s'$	sequence of events from s to s'
$\mathsf{proc}(e), \mathsf{loc}(e)$	the process, the global variable of event e
CFG_i, $\mathsf{PS} \subseteq \mathcal{E} \times \mathcal{E}$	the control-flow graph of process p_i, and the program structure relation
$G_{\mathcal{P}} = (V_{\mathcal{P}}, E_{\mathcal{P}}, \mathsf{wt}_{\mathcal{P}})$	the communication graph of $\mathcal{P}$

Table 9.1: Notation on the concurrent architecture.

9.2.2 Traces

In this section we develop various helpful definitions on traces. Relevant notation is summarized in Table 9.2.

Notation on traces. A (concrete, concurrent) *trace* is a sequence of events $t = e_1, \dots, e_j$ such that for all $1 \leqslant i < j$, we have $s^{i-1} \xrightarrow{e_i} s^i$, where $s^i \in \mathcal{S}_{\mathcal{P}}$ and s^0 is the initial state of $\mathcal{P}$. In such a case, we write succinctly $s^0 \xrightarrow{t} s^j$. We fix the first $|\mathcal{G}|$ events $e_1, \dots, e_{|\mathcal{G}|}$ of each trace t to be initialization events that write the initial values to the global variables. That is, for all $1 \leqslant i \leqslant |\mathcal{G}|$ we have $e_i \in \mathcal{W}$, and hence every trace t starts with an *initialization trace* $t^{\mathcal{I}}$ as a

prefix. Given a trace t, we denote by $\mathcal{E}(t)$ the set of events that appear in t, with $\mathcal{R}(t) = \mathcal{E}(t) \cap \mathcal{R}$ the read events in t, and with $\mathcal{W}(t) = \mathcal{E}(t) \cap \mathcal{W}$ the write events in t, and let $|t| = |\mathcal{E}(t)|$ be the *length* of t. For an event $e \in \mathcal{E}(t)$, we write $\mathcal{I}_t(e) \in \mathbb{N}^+$ to denote the index of e in t. Given some $\ell \in \mathbb{N}$, we denote by $t[\ell]$ the prefix of t up to position ℓ, and we say that t is an *extension* of $t[\ell]$. We let $\mathsf{enabled}(t)$ denote the set of enabled events in the state at the end of t, and call t *maximal* if $\mathsf{enabled}(t) = \varnothing$. We write $\mathcal{T}_\mathcal{P}$ for the set of all maximal traces of $\mathcal{P}$. We denote by $s(t)$ the unique state of $\mathcal{P}$ such that $s^0 \xrightarrow{t} s(t)$, and given an event $e \in \mathcal{R}(t) \cup \mathcal{W}(t)$, denote by $\mathsf{val}_t(e) \in \mathcal{D}$ the *value* that the unique global variable of e has in $s(t[\mathcal{I}_t(e)])$. We call a maximal trace t *lock-free* if the value of every lock variable in $s(t)$ is False (i.e., all locks have been released at the end of t). An event e is *inevitable* in a trace t if every every lock-free maximal extension of t contains e. Given a set of events A, we denote by $t|A$ the *projection* of t on A, which is the unique subsequence of t that contains all events of $A \cap \mathcal{E}(t)$, and only those. A sequence of events t' is called the *global projection* of another sequence t if $t' = t|(\mathcal{R} \cup \mathcal{W})$.

Sequential traces. Given a process p_i, a *sequential trace* τ_i is a sequence of events that correspond to a path in CFG_i, starting from the root node r_i. Note that a sequential trace is only wrt CFG_i, and is not necessarily a trace of the system. The notation on traces is extended naturally to sequential traces (e.g., $\mathcal{E}(\tau_i)$ and $\mathcal{R}(\tau_i)$ denote the events and read events of the sequential trace τ_i, respectively). Given k sequential traces $\tau_1, \tau_2, \dots, \tau_k$, so that each τ_i is wrt p_i, we denote by $\tau_1 * \tau_2 * \dots * \tau_k$ the (possibly empty) set of all traces t such that $\mathcal{E}(t) = \bigcup_{1 \leqslant i \leqslant k} \mathcal{E}(\tau_i)$.

Conflicting events, dependent events and happens-before relations. Two events $e_1, e_2 \in \mathcal{R} \cup \mathcal{W}$ are said to *conflict*, written $\mathsf{Confl}(e_1, e_2)$ if $\mathsf{loc}(e_1) = \mathsf{loc}(e_2)$ and at least one is a write event. The events are said to be in *read-write conflict* if $e_1 \in \mathcal{R}$, $e_2 \in \mathcal{W}$ and $\mathsf{Confl}(e_1, e_2)$. Two events e_1, e_2 are said to be *independent* [Godefroid, 1996; Flanagan and Godefroid, 2005] if $p(e_1) \neq p(e_2)$ and

1. for each $i \in \{1, 2\}$ and pair of states s_1, s_2 such that $s_1 \xrightarrow{e_i} s_2$, we have that $e_{3-i} \in \mathsf{enabled}(s_1)$ iff $e_{3-i} \in \mathsf{enabled}(s_2)$, and
2. for any pair of states s_1, s_2 such that $e_1, e_2 \in \mathsf{enabled}(s_1)$, we have that $s_1 \xrightarrow{e_1, e_2} s_2$ iff $s_1 \xrightarrow{e_2, e_1} s_2$,

and *dependent* otherwise. Following the standard approach in the literature, we will consider two conflicting events to be always dependent [Godefroid, 1997, Chapter 3] (e.g., two conflicting

write events are dependent, even if they write the same value). A sequence of events t induces a *happens-before* relation $\rightarrow_t \subseteq \mathcal{E}(t) \times \mathcal{E}(t)$, which is the smallest transitive relation on $\mathcal{E}(t)$ such that

$$e_1 \rightarrow_t e_2 \quad \text{if} \quad \mathcal{I}_t(e_1) \leqslant \mathcal{I}_t(e_2) \text{ and } e_1 \text{ and } e_2 \text{ are dependent.}$$

Observe that $\rightarrow_t$ orders all pairwise conflicting events, as well as all the events of any process.

Notation	**Interpretation**
t, τ_i	a trace and a sequential trace
$\mathsf{Confl}(e_1, e_2)$	conflicting events
$t[\ell]$, $\|t\|$	the prefix up to index ℓ, and length of t
$\mathcal{E}(t), \mathcal{W}(t), \mathcal{R}(t)$	the events, write and read events of trace t
$\mathcal{I}_t(e)$, $\mathsf{val}_t(e)$	the index and value of event e in trace t
$t\|X$	projection of trace t on event set X
$\mathsf{enabled}(t)$	the enabled events in the state reached by t
$\rightarrow_t$	the happens-before relation on t
O_t	the observation function of t

Table 9.2: Notation on traces.

9.2.3 Discussion and Remarks

The concurrent model we consider here is minimalistic, to allow for a clear exposition of the ideas used in our data-centric DPOR. Here we discuss some of the simplifications we have adopted to keep the presentation simple.

Global variables. First, note that the location $\mathsf{loc}(e)$ of every event $e \in \mathcal{R} \cup \mathcal{W}$ is taken to be fixed in each CFG_i. The dynamic access of a static, global data structure g based on the value of a local variable v (e.g., accessing the element $g[v]$ of a global array g) can be modeled by using a different global variable g_i to encode the i-th location of g, and a sequence of branching nodes that determine which g_i should be accessed based on the value of v. Our framework can

be strengthened to allow use of global arrays directly, and our algorithms apply straightforwardly to this richer framework. However, this would complicate the presentation, and is thus omitted in the theoretical exposition of the paper. A brief discussion on how arrays are handled directly is provided in Section 9.7.1 where we discuss how static arrays are handled in the benchmarks of the experiments.

Invisible computations. Each process p_i is deterministic, and the only source of nondeterminism in the executions of the system comes from a nondeterministic scheduler that chooses an enabled event to be executed from a given state. The model uses the functions f and b on events $e : g \leftarrow \mathsf{write}\ f(v_1, \dots, v_j)$ and $e : b(v_1, \dots, b_n)$ respectively to collapse deterministic invisible computations of each process, and only consider the value that f writes on a global variable (in addition to the side-effects that f has on local the variables of process p_i). This is a standard approach in modeling concurrent systems, as interleaving invisible events does not change the set of reachable local states of the processes.

Locks and synchronization mechanisms. We treat lock-acquire events as reads and lock-release events as writes. In a trace t, a lock-acquire event e is considered to read the value of the last lock-release event e' on the same lock l (or some initialization event Init if e is the first lock event on l in t). Our approach can be extended to richer communication (e.g., message passing) and synchronization primitives (e.g. semaphores, wait-notify), which are often implemented using some low-level locking mechanism.

Maximal lock-free traces. We also assume that in every maximal trace of the system, every lock-acquire is followed by a corresponding lock-release. Traces without this property are typically considered erroneous, and some modern programming languages even force this restriction syntactically.

9.3 Observation Trace Equivalence

In this section we introduce the observation equivalence $\sim_{\mathsf{O}}$ on traces, upon which in the later sections we develop our data-centric DPOR. We explore the relationship between the control-centric Mazurkiewicz equivalence $\sim_M$ and the observation equivalence. In particular, we show that $\sim_{\mathsf{O}}$ refines $\sim_M$, that is, every two traces that are equivalent under observations are also

equivalent under reordering of independent events. We conclude by showing that $\sim_{\mathsf{O}}$ can be exponentially more succinct, both in the number of processes, and the size of each process.

9.3.1 Mazurkiewicz and Observation Equivalence

In this section we introduce our notion of observation equivalence. We start with the classic definition of Mazurkiewicz equivalence and then the notion of observation functions.

Mazurkiewicz trace equivalence. Two traces $t_1, t_2 \in \mathcal{T}_{\mathcal{P}}$ are called *Mazurkiewicz equivalent* if one can be obtained from the other by swapping adjacent, independent events. Formally, we write $\sim_M$ for the Mazurkiewicz equivalence on $\mathcal{T}_{\mathcal{P}}$, and we have $t_1 \sim_M t_2$ iff

1. $\mathcal{E}(t_1) = \mathcal{E}(t_2)$, and
2. for every pair of events $e_1, e_2 \in \mathcal{E}(t_1)$ we have that $e_1 \rightarrow_{t_1} e_2$ iff $e_1 \rightarrow_{t_2} e_2$.

Observation functions. The concurrent model introduced in Section 9.2.1 follows *sequential consistency* [Lamport, 1979], i.e., all processes observe the same order of events, and a read event of some variable will observe the value written by the last write event to that variable in this order. Throughout the paper, an *observation function* is going to be a partial function $\mathsf{O} : \mathcal{R} \nrightarrow \mathcal{W}$. A trace t induces a total observation function $\mathsf{O}_t : \mathcal{R}(t) \to \mathcal{W}(t)$ following the sequential consistency axioms. That is, $\mathsf{O}_t(r) = w$ iff

1. $\mathcal{I}_t(w) < \mathcal{I}_t(r)$, and
2. for all $w' \in \mathcal{W}(t)$ such that $\mathsf{Confl}(r, w')$ we have that $\mathcal{I}_t(w') < \mathcal{I}_t(w)$ or $\mathcal{I}_t(w') > \mathcal{I}_t(r)$.

We say that t is *compatible* with an observation function O if $\mathsf{O} \subseteq \mathsf{O}_t$, and that t *realizes* O if $\mathsf{O} = \mathsf{O}_t$.

Observation equivalence. We define the *observation equivalence* $\sim_{\mathsf{O}}$ on the trace space $\mathcal{T}_{\mathcal{P}}$ as follows. For $t_1, t_2 \in \mathcal{T}_{\mathcal{P}}$ we have $t_1 \sim_{\mathsf{O}} t_2$ iff $\mathcal{E}(t_1) = \mathcal{E}(t_2)$ and $\mathsf{O}_{t_1} = \mathsf{O}_{t_2}$, i.e., the two observation functions coincide.

We start with the following crucial lemma. In words, it states that if two traces agree on their observation functions, then they also agree on the values seen by their common read events.

Lemma 9.1. *Consider two traces t_1, t_2 such that $\mathsf{O}_{t_1} \subseteq \mathsf{O}_{t_2}$. Then*

- *for all read events $r \in \mathcal{R}(t_1)$ we have that $\mathsf{val}_{t_1}(r) = \mathsf{val}_{t_2}(r)$, and*
- *for all write events $w \in \mathcal{W}(t_1) \cap \mathcal{W}(t_2)$ we have that $\mathsf{val}_{t_1}(w) = \mathsf{val}_{t_2}(w)$.*

Proof. The proof is by induction on the prefixes of t_1. We show inductively that for every $0 \leqslant \ell \leqslant |t_1|$, for all events $e \in \mathcal{E}(t_1[\ell])$ we have that if $e \in \mathcal{E}(t_2)$ then $\mathsf{val}_{t_1}(r) = \mathsf{val}_{t_2}(r)$. Note that in the case where $e = r$ is a read event, then $r \in \mathcal{E}(t_2)$ follows directly from $\mathsf{O}_{t_1} \subseteq \mathsf{O}_{t_2}$. The claim is true for $\ell = 0$, since in that case $t_1[\ell] = \varepsilon$ and no event appears in $t_1[\ell]$. Now assume that the claim holds for all prefixes $t_1[j]$ for $0 \leqslant j \leqslant \ell$, and let

$$e = \arg \min_{e' \in \mathcal{E}(t_1) \setminus \mathcal{E}(t_1[\ell])} \mathcal{I}_{t_1}(e')$$

be the next global event in t_1. We distinguish two cases based on the type of e.

- $e = w \in \mathcal{W}$ is a write event of the form $g \leftarrow \mathsf{write}\ f(v_1, \dots, v_{n_i})$, such that $p_i = \mathsf{proc}(w)$. Then the value of each local variable v_j equals some $\alpha_j = \mathsf{val}_{t_1}(r_j)$, with $\mathcal{I}_{t_1}(r_j) \leqslant \ell$, and by the induction hypothesis we have $r_j \in \mathcal{E}(t_2)$ and $\mathsf{val}_{t_1}(r_j) = \mathsf{val}_{t_2}(r_j) = \alpha_j$. Since p_i is deterministic, it easily follows that if $w \in \mathcal{E}(t_2)$ then

$$\begin{aligned} \mathsf{val}_{t_2}(w) &= f(\mathsf{val}_{t_1}(r_1), \dots, \mathsf{val}_{t_1}(r_{n_i})) \\ &= f(\alpha_1, \dots, \alpha_{n_i}) = f(\mathsf{val}_{t_2}(r_1), \dots, \mathsf{val}_{t_2}(r_{n_i})) \\ &= \mathsf{val}_{t_1}(w) \end{aligned}$$

- $e = r \in \mathcal{R}$ is a read event, and let $w = \mathsf{O}_{t_1}(r)$. Let $p_i = \mathsf{proc}(w)$, and since p_i is deterministic, it easily follows from the induction hypothesis that $r \in \mathcal{E}(t_2)$, and moreover that $\mathsf{val}_{t_1}(w) = \mathsf{val}_{t_2}(w)$. Since $\mathsf{O}_{t_2}(r) = w$, we have that

$$\mathsf{val}_{t_2}(r) = \mathsf{val}_{t_2}(w) = \mathsf{val}_{t_1}(w) = \mathsf{val}_{t_1}(r)$$

  The desired result follows.

□

The following is an easy consequence of Lemma 9.1.

Lemma 9.2. *Consider two traces t_1, t_2 such that $\mathsf{O}_{t_1} \subseteq \mathsf{O}_{t_2}$ and t_2 is maximal. Then (i) $\mathcal{E}(t_1) \subseteq \mathcal{E}(t_2)$, and (ii) for all events $e \in \mathcal{R}(t_1) \cup \mathcal{W}(t_1)$ we have that $\mathsf{val}_{t_1}(e) = \mathsf{val}_{t_2}(e)$.*

Soundness. Lemma 9.2 implies that two maximal traces which agree on their observation function have the same observable behavior, i.e., each global event has the same value in the two traces. Since all local states of each process can be explored by exploring maximal traces, it suffices to explore all the (maximal) observation functions of $\mathcal{P}$.

The Mazurkiewicz trace equivalence is *control-centric*, i.e., equivalent traces share the same order between the dependent control locations of the program. In contrast, the observation trace equivalence is *data-centric*, as it is based on which write events are observed by the read events of each trace. Note that two conflicting events are dependent, and thus must be ordered in the same way by two Mazurkiewicz-equivalent traces.

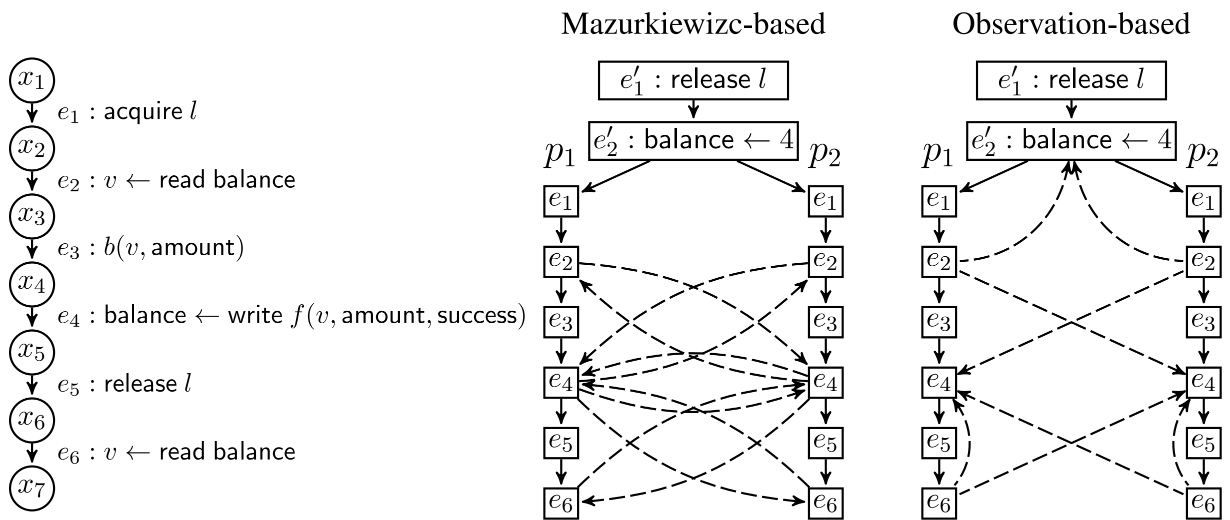


Figure 9.3: Trace exploration on the system of Fig. 9.2 with two processes, where initially balance $\leftarrow 4$ and both withdrawals succeed.

Example 9.1 (Observation vs Mazurkiewicz equivalence). Fig. 9.3 illustrates the difference between the Mazurkiewicz and observation trace equivalence on the example of Fig. 9.2. Every execution of the system starts with an initialization trace $t^{\mathcal{I}}$ that initializes the lock l to False, and the initial value desposit $= 4$. Consider that p_1 is executed with parameter amount $= 1$ and p_2 is executed with parameter amount $= 2$, (hence both withdrawals succeed). The primed events e'_1, e'_2 represent the system initialization.

- *(Left)*: The sequential trace of p_1, p_2.

- *(Center)*: Trace exploration using the Mazurkiewicz equivalence $\sim_M$. Solid lines represent the happens-before relation enforced by the program structure. Dashed lines represent potential happens-before relations between dependent events. A control-centric DPOR based on $\sim_M$ will resolve scheduling choices by exploring all possible realizable sets of the happens-before edges.

- *(Right)*: Trace exploration using the observation equivalence $\sim_{\mathsf{O}}$. Solid lines represent the happens-before relation enforced by the program structure. This time, dashed lines represent potential observation functions. Our data-centric DPOR based on $\sim_{\mathsf{O}}$ will resolve scheduling choices by exploring all possible realizable sets of the observation edges.

Both methods are guaranteed to visit all local states of each process. However, the data-centric DPOR achieves this by exploring potentially fewer scheduling choices.

The formal relationship between the two equivalences is established in the following theorem.

Theorem 9.1. *For any two traces* $t_1, t_2 \in \mathcal{T}_{\mathcal{P}}$*, if* $t_1 \sim_M t_2$ *then* $t_1 \sim_{\mathsf{O}} t_2$*.*

Proof. Consider any read event $r \in \mathcal{R}(t_1)$ and assume towards contradiction that $\mathsf{O}_{t_1}(r) \neq \mathsf{O}_{t_2}(r)$. Let $w_1 = \mathsf{O}_{t_1}(r)$ and $w_2 = \mathsf{O}_{t_2}(r)$. Since $t_1 \sim_M t_2$, we have that $w_1 \in \mathcal{E}(t_2)$ and $w_2 \in \mathcal{E}(t_1)$. Then $w_1 \rightarrow_{t_1} r$ and $w_2 \rightarrow_{t_2} r$, and one of the following holds.

1. $r \rightarrow_{t_1} w_2$, and since $w_2 \rightarrow_{t_2} r$ then $t_1 \not\sim_M t_2$, a contradiction.

2. $w_2 \rightarrow_{t_1} w_1$, and since $t_1 \sim_M t_2$ we have that $w_2 \rightarrow_{t_2} w_1$, and thus $r \rightarrow_{t_2} w_1$. Since $w_1 \rightarrow_{t_1} r$, we have $t_1 \not\sim_M t_2$, a contradiction.

The desired result follows. $\square$

9.3.2 Exponential succinctness

As we have already seen in the example of Fig. 1.3, Theorem 9.1 does not hold in the other direction, i.e., $\sim_{\mathsf{O}}$ can be strictly coarser than $\sim_M$. Here we provide two simple examples in which $\sim_{\mathsf{O}}$ is exponentially more succinct than $\sim_M$. Traditional enumerative model checking methods of concurrent systems are based on exploring *at least* one trace from every partition of the Mazurkiewicz equivalence using POR techniques that prune away equivalent traces (e.g.

Process p_1 :	Process p_2 :
1. $\mathsf{write}\ x$	1. $\mathsf{write}\ x$
2. $\mathsf{write}\ x$	2. $\mathsf{write}\ x$
...	...
$n+1$. $\mathsf{read}\ x$	$n+1$. $\mathsf{read}\ x$

Figure 9.4: An architecture of two processes with $n+1$ events each.

sleep sets [Godefroid, 1996], persistent sets [Flanagan and Godefroid, 2005], source sets and wakeup trees [Abdulla *et al.*, 2014]). Such a search is *optimal* if it explores at most one trace from each class. Any optimal enumerative exploration based on the observation equivalence is guaranteed by Theorem 9.1 to examine no more traces than any enumerative exploration based on the Mazurkiewicz equivalence. The two examples show $\sim_{\mathsf{O}}$ can offer exponential improvements wrt two parameters: (i) the number of processes, and (ii) the size of each process.

Example 9.2 (Two processes of large size). Consider the system $\mathcal{P}$ of $k=2$ processes of Fig. 9.4, and for $i \in \{1,\dots,n\}, j \in \{1,2\}$, denote by w_i^j (resp. r^j) the i-th write event (resp. the read event) of p_j. In any maximal trace, there are two ways to order the read events r^1, r^2, i.e., r^j occurs before r^{3-j} for the two choices of $j \in \{1,2\}$. In any such ordering, r^{3-j} can only observe either w_{n-1}^{3-j} or w_{n-1}^j, whereas there are at most $n+1$ possible write events for r^j to observe (either w_n^j or one of the w_i^{3-j}). Hence $\mathcal{T}_{\mathcal{P}}/\sim_{\mathsf{O}}$ has size $O(n)$. In contrast, $\mathcal{T}_{\mathcal{P}}/\sim_M$ has size $\Omega(\binom{2\cdot n}{n}) = \Omega(2^n)$, as there are $(2\cdot n)!$ ways to order the $2\cdot n$ write events of the two processes, but $n!\cdot n!$ orderings are invalid as they violate the program structure. Hence, even for only two processes, the observation equivalence reduces the number of partitions from exponential to linear.

Example 9.3 (Many processes of small size). We now turn our attention to a system $\mathcal{P}$ of k identical processes $p_1,\dots,p_k$ with two events each, in Fig. 9.5. There is only one global variable x, and each process performs a read and then a write to x. There are $O(k^k)$ realizable observation functions, by choosing for each one among k read events, one among k write events it can observe. Hence $\mathcal{T}_{\mathcal{P}}/\sim_{\mathsf{O}}$ has size $O(k^k)$. In contrast, the size of $\mathcal{T}_{\mathcal{P}}/\sim_M$ is $\Omega((k!)^2)$. This holds as there are $k!$ ways to order the k write events, and for each such permutation there are $k!$ ways to assign each of the k read events to the write event that it observes. To see this second part,

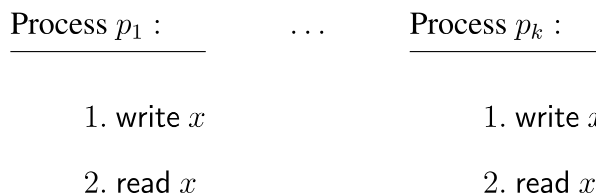


Figure 9.5: An architecture of k processes with two events each.

let $w_1, \dots, w_k$ be any permutation of the write events, and let r_i be the read event in the same process as w_i. Then r_i can be placed right after any w_j with $i \leqslant j$. Observe that $\mathcal{T}_\mathcal{P}/\sim_\mathsf{O}$ is exponentially more succinct than $\mathcal{T}_\mathcal{P}/\sim_M$, as

$$\frac{\Omega((k!)^2)}{O(k^k)} = \Omega\left(\frac{\prod_{i=1}^{k} i \cdot \lceil \frac{k}{i} \rceil}{k^k} \cdot \prod_{i=\lceil \frac{k}{2} \rceil + 1}^{k-1} i\right) = \Omega(2^k).$$

9.3.3 Solution Overview

Traditional DPOR algorithms exploit the Mazurkiewicz equivalence, and use various techniques such as persistent sets and sleep sets to explore each Mazurkiewicz class by few representative traces. Our goal is to develop an analogous DPOR that utilizes the observation equivalence, which by Theorem 9.1 is more succinct. In high level, our approach consists of the following steps.

1. In Section 9.4 we introduce the concept of annotations. An annotation is a function from read to write events, and serves as an intended observation function. Given an annotation, the goal is to obtain a trace whose observation function coincides with the annotation. We restrict our attention to a certain class of *well-formed* annotations, and show that although the problem is NP-complete in general, it admits a polynomial time (in fact, cubic in the size of the trace) solution in acyclic architectures.

2. In Section 9.5 we present our data-centric DPOR. Section 9.5.1 introduces the notion of causal past cones in a trace. The concept is similar to Lamport's *happens-before* relation [Lamport, 1978], and is used to identify past events that may causally affect a current event in a trace. We note that this concept is different from the happens-before relation used in the Mazurkiewicz equivalence. We use the notions of annotations

and causal cones to develop our algorithm, and prove its correctness and optimality (in Section 9.5.2).

3. In Section 9.6 we extend our algorithm to cyclic architectures.

Table 9.1 and Table 9.2 summarize relevant notation in the proofs.

9.4 Annotations

In this section we introduce the notion of *annotations*, which are intended constraints on the observation functions that traces discovered by our data-centric DPOR (DC-DPOR) are required to meet.

Annotations. An *annotation pair* $\mathsf{A} = (\mathsf{A}^+, \mathsf{A}^-)$ is a pair of

1. a *positive annotation* $\mathsf{A}^+ : \mathcal{R} \nrightarrow \mathcal{W}$, and
2. a *negative annotation* $\mathsf{A}^- : \mathcal{R} \nrightarrow 2^{\mathcal{W}}$

such that for all read events r, if $\mathsf{A}^+(r) = w$, then we have $\mathsf{Confl}(r, w)$ and it is not the case that $\mathsf{PS}(r, w)$. We will use annotations to guide the recursive calls of DC-DPOR towards traces that belong to different equivalence classes than the ones explored already, or will be explored by other branches of the algorithm. A positive annotation A^+ forces DC-DPOR to explore traces that are compatible with A^+ (or abort the search if no such trace can be generated). Since a positive annotation is an "intended" observation function, we say that a trace t *realizes* A^+ if $\mathsf{O}_t = \mathsf{A}^+$, in which case A^+ is called *realizable*. A negative annotation A^- prevents DC-DPOR from exploring traces t in which a read event observes a write event that belongs to its negative annotation set (i.e., $\mathsf{O}_t(r) \in \mathsf{A}^-(r)$). In the remaining section we focus on positive annotations, and the problem of deciding whether a positive annotation is realizable.

The value function $\mathsf{val}_{\mathsf{A}^+}$**.** Given a positive annotation A^+, we define the relation $<_{\mathsf{A}^+} \subseteq \mathsf{img}(\mathsf{A}^+) \times \mathsf{dom}(\mathsf{A}^+)$ such that $w <_{\mathsf{A}^+} r$ iff $(r, w) \in \mathsf{A}^+$. The positive annotation A^+ is *acyclic* if the relation $\mathsf{PS} \cup <_{\mathsf{A}^+}$ is a strict partial order (i.e., it contains no cycles). The *value function* $\mathsf{val}_{\mathsf{A}^+} : \mathsf{dom}(\mathsf{A}^+) \cup \mathsf{img}(\mathsf{A}^+) \to \mathcal{D}$ of an acyclic positive annotation A^+ is the unique function defined inductively, as follows.

1. For each $w \in \mathsf{img}(\mathsf{A}^+)$ of the form $w : g \leftarrow \mathsf{write}\ f(v_1, \dots, v_{n_i})$, we have $\mathsf{val}_{\mathsf{A}^+}(w) = f(\alpha_1, \dots, \alpha_{n_i})$, where for each α_j we have
   (a) $\alpha_j = \mathsf{val}_{\mathsf{A}^+}(r)$ if there exists a read event $r \in \mathsf{dom}(\mathsf{A}^+)$ such that (i) r is of the form $r : v_j \leftarrow \mathsf{read}\ g'$ and (ii) $\mathsf{PS}(r, w)$ and (iii) there exists no other $r' \in \mathsf{dom}(\mathsf{A}^+)$ with $\mathsf{PS}(r, r')$ and which satisfies conditions (i) and (ii).
   (b) α_j equals the initial value of v_i otherwise.
2. For each $r \in \mathsf{dom}(\mathsf{A}^+)$ we have $\mathsf{val}_{\mathsf{A}^+}(r) = \mathsf{val}_{\mathsf{A}^+}(\mathsf{A}^+(r))$.

Note that $\mathsf{val}_{\mathsf{A}^+}$ is well-defined, as for any read event r that is used to define the value of a write event w we have $\mathsf{PS}(r, w)$, and thus by the acyclicity of A^+, $\mathsf{val}_{\mathsf{A}^+}(r)$ does not depend on $\mathsf{val}_{\mathsf{A}^+}(w)$.

Remark 9.1. If A^+ is realizable then it is acyclic, and for any trace t that realizes A^+ we have that $\mathsf{val}_t = \mathsf{val}_{\mathsf{A}^+}$.

Well-formed annotations and basis of annotations. A positive annotation A^+ is called *well-formed* if the following conditions hold:

1. A^+ is acyclic.
2. For every lock-release event $e_a \in \mathsf{img}(\mathsf{A}^+) \cap \mathcal{L}^A$ there is at most one lock-acquire event $e_r \cap \mathcal{L}^R$ such that $\mathsf{A}^+(e_a) = e_r$.
3. There exist sequential traces $(\tau_i)_i$, one for each process p_i, such that each τ_i ends in a global event, and the following conditions hold.
   (a) for every pair of lock-acquire events $e_a^1, e_a^2 \in \mathcal{E}(\tau_i) \cap \mathcal{L}^A$ such that $\mathcal{I}_{\tau_i}(e_a^1) < \mathcal{I}_{\tau_i}(e_a^2)$ and $\mathsf{loc}(e_a^1) = \mathsf{loc}(e_a^2)$ there exists a lock release event $e_r \in \mathcal{E}(\tau_i) \cap \mathcal{L}^R$ such that $\mathcal{I}_{\tau_i}(e_a^1) < \mathcal{I}\tau_i(e_r) < \mathcal{I}\tau_i(e_a^2)$ and $\mathsf{loc}(e_r) = \mathsf{loc}(e_a^1) = \mathsf{loc}(e_a^2)$.
   (b) $\bigcup_i \mathcal{R}(\tau_i) = \mathsf{dom}(\mathsf{A}^+)$ and $\mathsf{img}(\mathsf{A}^+) \subseteq \bigcup_i \mathcal{W}(\tau_i)$, i.e., $(\tau_i)_i$ contains precisely the read events of A^+ and a superset of the write events.
   (c) Each τ_i corresponds to a deterministic computation of process p_i, where the value of every global event e during the computation is taken to be $\mathsf{val}_{\mathsf{A}^+}(e)$.

The sequential traces $(\tau_i)_i$ are called a *basis* of A^+ if every τ_i is minimal in length. The following lemma establishes properties of well-formedness and basis.

Lemma 9.3. *Let* $X = \mathsf{dom}(\mathsf{A}^+) \cup \mathsf{img}(\mathsf{A}^+)$ *be the set of events that appear in a positive annotation* A^+*, and* $X_i = X \cap \mathcal{E}_i$ *the subset of events of* X *from process* p_i*. The following assertions hold:*

1. *If* A^+ *is well-formed, then it has a unique basis* $(\tau_i)_i$*.*

2. *Computing the basis of* A^+ *(or concluding that* A^+ *is not well-formed) can be done in* $O(n)$ *time, where* $n = \sum_i(|\tau_i|)$ *if* A^+ *is well-formed, otherwise* $n = \sum_i \ell_i$*, where* ℓ_i *is the length of the longest path from the root* r_i *of* CFG_i *to an event* $e \in X_i$*.*

3. *For every trace* t *that realizes* A^+ *we have that* A^+ *is well-formed and* $t \in \tau_1 * \ldots * \tau_k$*.*

Proof. 1. Assume that A^+ has two distinct bases $(\tau_i)_i$ and $(\tau'_i)_i$. Since each sequential trace corresponds to a deterministic computation of the corresponding process using $\mathsf{val}_{\mathsf{A}^+}$ as the value function of global events, we have that each τ_i and τ'_i share a prefix relationship (i.e., one is prefix of the other). Since the two basis are distinct, for some j we have that one of τ_j and τ'_j is a proper prefix of the other. Assume w.l.o.g. that τ'_j is a strict prefix of τ_j. Then replacing τ_i with τ'_i in $(\tau_i)_i$ yields another basis, thus τ_i is not minimal, a contradiction.

2. First, testing the conditions of Item 1 and Item 2 of well-formedness can be done in $O(|\mathsf{A}^+|)$ time. We now outline the process of constructing the basis $(\tau_i)_i$. As a preprocessing step, we compute for each process p_i the unique event $e_i \in X_i$ which is maximal wrt the program structure PS. Note that if e_i is not unique for each process, then A^+ is not well-formed. This requires $O(|\mathsf{A}^+|)$ time for all processes p_i, simply by iterating over all events in X. Then, the unique basis $(\tau_i)_i$ can be constructed by executing each process locally, and using the value function $\mathsf{val}_{\mathsf{A}^+}$ for assigning values to global events. The execution stops when e_i is reached, and the constructed sequential trace τ_i is returned. Finally, $(\tau_i)_i$ constitute a basis of A^+ if the conditions (a) and (b) of Item 3 of well-formedness are met, which can be done in $O(n)$ time.

3. It follows easily from Remark 9.1 and the above construction that if t realizes A^+, then A^+ must be well-formed and $t \in \tau_1 * \ldots * \tau_k$.

□

9.4.1 The Hardness of Realizing Positive Annotations

A core step in our data-centric DPOR algorithm is constructing a trace that realizes a positive annotation. That is, given a positive annotation A^+, the goal is to obtain a trace t (if one exists) such that $\mathsf{O}_t = \mathsf{A}^+$, i.e., t contains precisely the read events of A^+, and every read event in t observes the write event specified by A^+. Here, we show that the problem is NP-complete in the general case. Membership in NP is trivial, since, given a trace t, it is straightforward to verify that $\mathsf{O}_t = \mathsf{A}^+$ in $O(|t|)$ time. Hence our focus will be on establishing NP-hardness. For doing so, we introduce a new graph problem, namely ACYCLIC EDGE ADDITION, which is closely related to the problem of realizing a positive annotation under sequential consistency semantics. We first show that ACYCLIC EDGE ADDITION is NP-hard, and afterwards that the problem is polynomial-time reducible to realizing a positive annotation.

The problem ACYCLIC EDGE ADDITION**.** The input to the problem is a pair (G, H) where $G = (V, E)$ is a directed acyclic graph, and $H = \{(x_i, y_i, z_i)\}_i$ is a set of triplets of distinct nodes such that

1. $x_i, y_i, z_i \in V$, $(y_i, z_i) \in E$ and $(x_i, y_i), (z_i, x_i) \notin E$, and
2. each node x_i and y_i appears only once in H.

An *edge addition set* $X = \{e_i\}_{i=1}^{|H|}$ for (G, H) is a set of edges $e_i \in E$ such that for each e_i we have either $e_i = (x_i, y_i)$ or $e_i = (z_i, x_i)$. The problem ACYCLIC EDGE ADDITION asks whether there exists an edge addition set X for (G, H) such that the graph $G_X = (V, E \cup X)$ remains acyclic.

Lemma 9.4. ACYCLIC EDGE ADDITION *is NP-hard.*

Proof. The proof is by reduction from MONOTONTE ONE-IN-THREE SAT [Garey and Johnson, 1979, LO4]. In MONOTONTE ONE-IN-THREE SAT, the input is a propositional 3CNF formula ϕ in which every literal is positive, and the goal is to decide whether there exists a satisfying assignment for ϕ that assigns exactly one literal per clause to True.

The reduction proceeds as follows. In the following, we let C and D range over the clauses and x_i over the variables of ϕ. We assume w.l.o.g. that no variable repeats in the same clause. For every variable x_i, we introduce a node $w'_i \in V$. For every clause $C = (x_{C_1} \vee x_{C_2} \vee x_{C_3})$, we introduce a pair of nodes $w^C_{C_j}, r^C_{C_j} \in V$ and an edge $(w^C_{C_j}, r^C_{C_j}) \in E$, where $j \in \{1, 2, 3\}$. Additionally, we introduce an edge $(w^C_{C_j}, w'_{C_l}) \in E$ for every pair $j, l \in \{1, 2, 3\}$ such that $j \neq l$, and an edge $(w'_{C_j}, r^C_{C_l})$ for each $j \in \{1, 2, 3\}$, where $l = (j + 1) \mod 3 + 1$. Finally, for every pair of clauses C, D and $l_1, l_2 \in \{1, 2, 3\}$ such that $C_{l_1} = D_{l_2} = \ell$ (i.e., C and D share the same variable x_ℓ in positions l_1 and l_2), we add edges $(w^C_\ell, r^D_\ell), (w^D_\ell, r^C_\ell) \in E$. The set H consists of triplets of nodes $(w'_{C_j}, w^C_{C_j}, r^C_{C_j})$ for every clause C and $j \in \{1, 2, 3\}$. Fig. 9.6 illustrates the above construction.

Let X be an edge addition set that solves ACYCLIC EDGE ADDITION on input (G, H) and note that for every pair of triplets $(w'_i, w^C_i, r^C_i), (w'_i, w^D_i, r^D_i) \in H$, we have that $(w'_i, w^C_i) \in X$ iff $(w'_i, w^D_i) \in X$, i.e., for every node w'_i, the set X contains either only all incoming, or only all outgoing edges of w'_i specified by H. To see this, observe that if there exists such a pair of triplets with $(w'_i, w^C_i), (r^D_i, w'_i) \in H$, then $G_X = (V, E \cup X)$ would contain the cycle

$$w'_i \to w^C_i \to r^D_i \to w'_i$$

which contradicts that X is a solution to the problem. Given such an edge addition set X, we obtain an assignment on the variables of ϕ by setting $x_i = \text{True}$ iff X contains an edge $(w'_{C_j}, w^C_{C_j})$ for some clause C and $j \in \{1, 2, 3\}$. By the previous remark, the assignment of values to variables of ϕ is well-defined.

It is easy to verify that the construction takes polynomial time in the size of ϕ. In the following, we argue that the answer to MONOTONTE ONE-IN-THREE SAT is true iff the answer to ACYCLIC EDGE ADDITION is also true.

($\Rightarrow$). Let X be a solution to ACYCLIC EDGE ADDITION on (G, H), and we argue that every clause C of ϕ contains exactly one variable set to True. Indeed, C contains at least one such variable, otherwise G_X would contain a cycle

$$r^C_{C_1} \to w'_{C_1} \to r^C_{C_2} \to w'_{C_2} \to r^C_{C_3} \to w'_{C_3} \to r^C_{C_1}$$

Similarly, C contains at most one variable of C set to True, as two such variables x_i, x_j would imply that G_X contains a cycle

$$w'_i \to w^C_i \to w'_j \to w^C_j \to w'_i$$

$(\Leftarrow)$. Consider any satisfying assignment of ϕ, and construct an edge addition set $X = \{e_i\}_i$ such that for each triplet $(w'_i, w^C_i, r^C_i) \in H$ we have

$$e_i = \begin{cases} (w'_i, w^C_i), & \text{if } x_i = \mathsf{True} \\ (r^C_i, w'_i), & \text{if } x_i = \mathsf{False} \end{cases}$$

Given a clause C, we denote by $V_C = \bigcup_{i=1}^{3}\{w'_{C_i}, w^C_{C_i}, r^C_{C_i}\}$, and by $G_X[V_C]$ the subgraph of G_X restricted to nodes in V_C. We now argue that G_X does not contain a cycle. Assume towards contradiction otherwise, and let $\mathcal{C}$ be such a cycle.

1. If $\mathcal{C}$ contains a node of the form $w^C_{C_j}$ for some $j \in \{1,2,3\}$ then $\mathcal{C}$ traverses the edge $(w'_{C_j}, w^C_{C_j})$, and thus x_ℓ is assigned True, where $\ell = C_j$ Since G_X contains no edge of the form (r^D_ℓ, w'_ℓ) for any clause D, $\mathcal{C}$ must traverse an edge $(w^D_{D_i}, w'_\ell)$ for some clause D. Hence there is a first node $w^D_{D_i}$ traversed by $\mathcal{C}$ after w^C_ℓ, for some clause D and $i \in \{1,2,3\}$ and thus G_X contains an edge $(w'_{D_i}, w^D_{D_i})$, and hence x_l is assigned True, where $l = D_i$. By our choice of $w^D_{D_i}$, the node w'_{D_i} can only be reached via the edge $(r^D_{D_i}, w'_{D_i})$, which requires that x_l is assigned False, a contradiction.

2. If $\mathcal{C}$ contains no node of the form $w^C_{C_j}$, then it must be a cycle in $G[V_C]$, for some clause C. The only such cycle that traverses no $w^C_{C_j}$ can be

$$r^C_{C_1} \to w'_{C_1} \to r^C_{C_2} \to w'_{C_2} \to r^C_{C_3} \to w'_{C_3} \to r^C_{C_1}$$

which requires that all x_{C_i} are assigned False, for each $i \in \{1,2,3\}$, which contradicts that C has a variable assigned True.

The desired result follows. □

From ACYCLIC EDGE ADDITION **to annotations.** Finally, we argue that ACYCLIC EDGE ADDITION is polynomial-time reducible to realizing a positive annotation. Given an instance (G, H) of ACYCLIC EDGE ADDITION, with $G = (V, E)$, we construct an architecture $\mathcal{P}$ of $k = 2 \cdot |H|$ processes $(p_i)_i$, and a positive annotation A^+. We assume, w.l.o.g., that the set of nodes V is precisely the set of nodes that appear in the triplets of H. Indeed, any other nodes can be removed while maintaining the connectivity between the nodes in the triplets of H, and any edge addition set X solves the problem in the original graph iff it does so in the reduced graph. The construction proceeds in two steps.

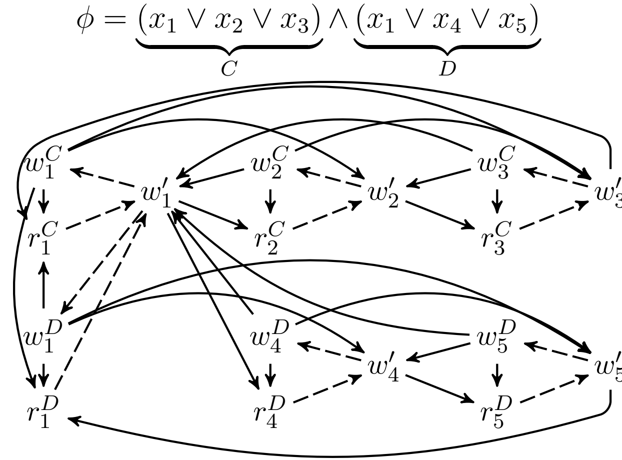


Figure 9.6: The reduction of 3SAT over ϕ to ACYCLIC EDGE ADDITION over (G, H). The nodes and solid edges represent the graph G. The dashed edges represent the triplets in H.

1. For every triplet $(x_i, y_i, z_i) \in H$, we create two events $w_i \in \mathcal{W}$, $r_i \in \mathcal{R}$ in p_i such that $\mathsf{PS}(w_i, r_i)$, and an event w_i' in process $p_{|H|+i}$. For all three events we set $\mathsf{loc}(r_i) = \mathsf{loc}(w_i) = \mathsf{loc}(w_i') = g_i$, where $g_i \in \mathcal{G}$ is some fresh global variable of $\mathcal{P}$. Finally, we introduce $(r_i, w_i) \in \mathsf{A}^+$. Given any node $u \in V$, let $e(u)$ denote the event associated with u.

2. For every edge (u, v) we introduce a new global variable $g \in \mathcal{G}$, and two events $w_{u,v} \in \mathcal{W}$, $r_{u,v} \in \mathcal{R}$ such that $\mathsf{loc}(w_{u,v}) = \mathsf{loc}(r_{u,v}) = g$. We make $w_{u,v}$ an event of the same process as $e(u)$, and $r_{u,v}$ an event of the same process as $e(v)$, and additionally $\mathsf{PS}(e(u), w_{u,v})$ and $\mathsf{PS}(r_{u,v}, e(v))$. Finally, we introduce $(r_{u,v}, w_{u,v}) \in \mathsf{A}^+$.

Observe that the above construction is linear in the size of (G, H). The following lemma states the correctness of the reduction.

Lemma 9.5. *The decision problem of* ACYCLIC EDGE ADDITION *on input* $(G = (V, E), H)$ *admits a positive answer iff the positive annotation* A^+ *is realizable in* $\mathcal{P}$.

Proof. We present both directions of the proof.

- $(\Rightarrow)$. Let t be any trace that realizes A^+. Observe that for any edge $(u, v) \in E$ we have $\mathcal{I}_t(e(u)) < \mathcal{I}_t(e(v))$, since $\mathsf{PS}(e(u), w_{u,v})$ and $\mathsf{PS}(r_{u,v}, e(v))$ and $\mathsf{A}^+(r_{u,v}) = w_{u,v}$. Additionally, t satisfies that either $\mathcal{I}_t(w_i') < w_t(w_i)$, or $\mathcal{I}_t(r_i) < w_t(w_i')$. In the former case we an edge (x_i, y_i), and in the latter case we add (z_i, x_i) in an edge set X. Since t

induces a total order on the vertices of G, G_X is acyclic, and thus X is an edge addition set for (G, H).

- $(\Leftarrow)$. If X is an edge addition set for (G, H) then G_X is acyclic and any topological order of the vertices of $G_X = (V, E \cup X)$ induces a trace that realizes A^+.

The desired result follows. □

9.4.2 Realizing Positive Annotations in Acyclic Architectures

We now turn our attention to a tractable fragment of the positive annotation problem. Here we show that if $\mathcal{P}$ is an acyclic architecture, then the problem admits a polynomial-time solution (in fact, cubic in the size of the constructed trace).

Procedure Realize**.** Let $\mathcal{P}$ be an acyclic architecture, and A^+ a positive annotation over $\mathcal{P}$. We describe a procedure $\mathsf{Realize}(\mathsf{A}^+)$ which returns a trace t that realizes A^+, or $\perp$ if A^+ is not realizable. The procedure works in two phases. In the first phase, $\mathsf{Realize}(\mathsf{A}^+)$ uses Lemma 9.3 to extract a basis $(\tau_i)_i$ of A^+. In the second phase, $\mathsf{Realize}(\mathsf{A}^+)$ determines whether the events of $\bigcup_i \mathcal{E}(\tau_i)$ can be linearized in a trace t such that $\mathsf{O}_t = \mathsf{A}^+$. Informally, the second phase consists of constructing a 2SAT instance over variables x_{e_1,e_2}, where $e_1, e_2 \in \bigcup_i \mathcal{E}(\tau_i)$. Setting x_{e_1,e_2} to True corresponds to making e_1 happen before e_2 in the witness trace t. The clauses of the 2SAT instance capture four properties that each such ordering needs to meet, namely that

1. the resulting assignment produces a total order (totality, antisymmetry and transitivity) between all of the events that appear in adjacent processes in the communication graph $G_{\mathcal{P}}$,

2. the produced total order respects the positive annotation, i.e., every write event w' that conflicts with an annotated read/write pair $(r, w) \in \mathsf{A}^+$ must either happen before w or after r, and

3. the produced total order respects the partial order induced by the program structure PS and the positive annotation A^+.

The formal description of the second phase is given in Algorithm 31.

Algorithm 31: Realize(A^+)

Input: A positive annotation A^+ with basis $(\tau_i)_i$

Output: A trace t that realizes A^+ or $\perp$ if A^+ is not realizable

```
1  Construct a directed graph G = (V, E) where
2  V = ⋃_i 𝓔(τ_i)  and  E = {(e1, e2) : (e2, e1) ∈ A+ or PS(e1, e2)}
3  G* = (V, E*) ← the transitive closure of G
   // A set 𝒞 of 2SAT clauses over variables V_𝒞
4  𝒞 ← ∅
5  V_𝒞 ← {x_{e1,e2} : e1, e2 ∈ V, and e1 ≠ e2, and proc(e1) = proc(e2) or (proc(e1), proc(e2)) ∈ E_𝒫}
   // 1. Antisymmetry clauses
6  foreach x_{e1,e2} ∈ V_𝒞 do
7  |  𝒞 ← 𝒞 ∪ {(x_{e1,e2} ⇒ ¬x_{e2,e1}), (¬x_{e2,e1} ⇒ x_{e1,e2})}
8  end
   // 2. Transitivity clauses
9  foreach x_{e1,e2} ∈ V_𝒞 do
10 |  foreach (e2, e3) ∈ E* do
11 |  |  𝒞 ← 𝒞 ∪ {(x_{e1,e2} ⇒ x_{e1,e3})}
12 |  end
13 |  foreach (e3, e1) ∈ E* do
14 |  |  𝒞 ← 𝒞 ∪ {(x_{e1,e2} ⇒ x_{e3,e2})}
15 |  end
16 end
   // 3. Annotation clauses
17 foreach (r, w) ∈ A+ and w' ∈ V ∩ 𝒲 s.t. Confl(r, w') do
18 |  𝒞 ← 𝒞 ∪ {(x_{w',r} ⇒ x_{w',w})}
19 end
   // 4. Fact clauses
20 foreach (e1, e2) ∈ E* with e1 ≠ e2 do
21 |  𝒞 ← 𝒞 ∪ {(x_{e1,e2})}
22 end
23 Compute a satisfying assignment f : V_𝒞 → {False, True}^{|V_𝒞|} of the 2SAT over 𝒞, or return ⊥ if 𝒞
     is unsatisfiable
24 Let G' = (V, E') where E' ← E ∪ {(e1, e2) : f(x_{e1,e2}) = True}
25 return a trace t by topologically sorting the nodes of G'
```

Lemma 9.6. *Given a well-formed positive annotation* A^+ *over a basis* $(\tau_i)_i$, $\mathsf{Realize}$ *constructs a trace* t *that realizes* A^+ *(or concludes that* A^+ *is not realizable) and requires* $O(n^3)$ *time, where* $n = \sum_i |\tau_i|$.

Proof. We present the correctness proof and complexity analysis.

Correctness. We first argue about correctness.

1. If $\mathsf{Realize}$ returns a sequence of events t (Line 25) then clearly t is a trace since t respects the program structure PS (Line 20), and by the definition of the well-formed annotation, lock semantics are respected (i.e., critical regions protected by locks do not overlap). Additionally, t realizes A^+, as the sequential consistency axioms are satisfied because of Line 17 and Line 20.

2. If A^+ is realizable by a trace t, then t is a linearization of E^*, thus for every pair of distinct conflicting events $(e_1, e_2) \in E^*$ we have $\mathcal{I}_t(e_1) < \mathcal{I}_t(e_2)$ (Line 20). By the sequential consistency axioms, for every pair $(r, w) \in \mathsf{A}^+$ and $w' \neq w$ with $\mathsf{Confl}(r, w')$ we have either $\mathcal{I}_t(w') < \mathcal{I}_t(w)$ or $\mathcal{I}_t(r) < \mathcal{I}_t(w')$ (Line 17). Finally, since t induces a total order on V, it is clearly transitive (Line 9) and antisymmetric (Line 6). Hence the set of 2SAT clauses $\mathcal{C}$ is satisfiable. It suffices to argue that $G' = (V', E')$ (Line 24) is acyclic, as then any topological order of G' will satisfy the sequential consistency axioms (Line 17). Assume towards contradiction otherwise. If G^* has a cycle, then A^+ is not realizable, as t must linearize E^*. Hence G^* must be acyclic, Thus, any cycle C in G' traverses an edge $(e_1, e_2) \in E' \setminus E^*$, hence $f(x_{e_1,e_2}) = \mathsf{True}$. We distinguish the following cases.

   (a) If there exists a cycle C which traverses a single edge $(e_1, e_2) \in E' \setminus E^*$, then there is a path $e_2 \rightsquigarrow e_1$ traversing only edges of E^*. Since E^* is transitively closed, we have that $(e_2, e_1) \in E^*$ and hence $f(x_{e_2,e_1}) = \mathsf{True}$ (Line 20). Thus, by antisymmetry, $f(x_{e_1,e_2}) = \mathsf{False}$ (Line 6), a contradiction.

   (b) Otherwise, let C be a simple cycle in G^* that traverses the fewest number of edges in $E' \setminus E^*$, and C must traverse at least two edges $(e_1, e_2), (e_3, e_4) \in E' \setminus E^*$. Observe that $\mathsf{proc}(e_1) \neq \mathsf{proc}(e_2)$ and $\mathsf{proc}(e_3) \neq \mathsf{proc}(e_4)$ as otherwise we would have $(e_2, e_1) \in E^*$ or $(e_4, e_3) \in E^*$, and there would exist a cycle that traverses a single edge from $E' \setminus E^*$ (namely, $e_1 \to e_2 \to e_1$ or $e_3 \to e_4 \to e_3$). Since $\mathcal{P}$ is acyclic, by

construction (Line 5) $|\{\mathsf{proc}(e_1), \mathsf{proc}(e_2), \mathsf{proc}(e_3), \mathsf{proc}(e_4)\}| = 2$, i.e., there exist two processes p_i, p_j such that

$$e_1 \in \mathcal{E}_i \text{ and } e_2 \in \mathcal{E}_j \qquad \text{and} \qquad e_3, e_4 \in \mathcal{E}_i \cup \mathcal{E}_j \text{ and } e_3, e_4 \not\in \mathcal{E}_i \cap \mathcal{E}_j$$

Then C traverses an edge (e'_3, e'_4) such that either $e'_3 = e_1$ or $\mathsf{PS}(e'_3, e_1)$, and either $e_2 = e'_4$ or $\mathsf{PS}(e_2, e'_4)$. In all cases, we have $(e'_3, e_1), (e_2, e'_4) \in E^*$. Since $(e'_3, e'_4) \in E'$ we have $f(x_{e'_3,e'_4}) = \mathsf{True}$, and by transitivity (Line 9), we have that $f(x_{e_2,e_1}) = \mathsf{True}$, a contradiction.

Complexity. The transitive closure requires $O(n^3)$ time, since $|V| = n$. The set $V_\mathcal{C}$ (Line 5) has $O(n^2)$ variables and each of the loops for constructing clauses iterates over triplets of nodes, hence the 2SAT instance is constructed in $O(n^3)$ time. Computing a satisfying assignment for $\mathcal{C}$ (or concluding that none exists) requires linear time in $|\mathcal{C}|$ [Aspvall *et al.*, 1979], hence this step costs $O(n^3)$. Finally, constructing G' and computing a topological sorting of its vertices requires $O(n^2)$ time in total. The desired result follows. □

The following theorem summarizes the results of this section.

Theorem 9.2. *Consider any architecture $\mathcal{P} = (p)_i$ and let A^+ be any well-formed positive annotation over a basis $(\tau)_i$. Deciding whether A^+ is realizable is NP-complete. If $\mathcal{P}$ is acyclic, the problem can be solved in $O(n^3)$ time, where $n = \sum_i |\tau_i|$.*

9.5 Data-centric Dynamic Partial Order Reduction

In this section we develop our data-centric DPOR algorithm called DC-DPOR and prove its correctness and compactness, namely that the algorithm explores each observation equivalence class of $\mathcal{T}_\mathcal{P}$ *once*. We start with the notion of causal past cones, which will help in proving the properties of our algorithm.

9.5.1 Causal Cones

Intuitively, the causal past cone of an event e appearing in a trace t is the set of events that precede e in t and may be responsible for enabling e in t.

Causal cones. Given a trace t and some event $e \in \mathcal{E}(t)$, the *causal past cone* $\mathsf{Past}_t(e)$ of e in t is the smallest set that contains the following events:

1. if there is an event $e' \in \mathcal{E}(t)$ with $\mathsf{PS}(e', e)$, then $e' \in \mathsf{Past}_t(e)$,
2. if $e_1 \in \mathsf{Past}_t(e)$, for every event $e_2 \in \mathcal{E}(t)$ such that $\mathsf{PS}(e_2, e_1)$, we have that $e_2 \in \mathsf{Past}_t(e)$, and
3. if there exists a read $r \in \mathsf{Past}_t(e) \cap \mathcal{R}$, we have that $\mathsf{O}_t(r) \in \mathsf{Past}_t(e)$.

In words, the causal past cone of e in t is the set of events e' that precede e in t and may causally affect the enabling of e in t. Note that for every event $e' \in \mathsf{Past}_t(e)$ we have that $e' \to_t e$, i.e., every event in the causal past cone of e also happens before e in t. However, the inverse is not true in general, as e.g. for some read r we have $\mathsf{O}_t(r) \to_t r$ but possibly $\mathsf{O}_t(r) \not\in \mathsf{Past}_t(r)$.

Remark 9.2. If $e' \in \mathsf{Past}_t(e)$, then $e' \to_t e$ and $\mathsf{Past}_t(e') \subseteq \mathsf{Past}_t(e)$.

Remark 9.3. For every trace t and event $e \in \mathcal{E}(t)$ we have that $t|(\mathsf{Past}_t(e) \cup e)$ is a valid trace.

The following lemma states the main property of causal past cones used throughout the paper. Intuitively, if the causal past of an event e in some trace t_1 also appears in another trace t_2, and the read events in the causal past observe the same write events in both traces, then e is inevitable in t_2, i.e., every maximal extension of t_2 will contain e.

Lemma 9.7. *Consider two traces t_1, t_2 and an event $e \in \mathcal{E}(t_1)$ such that for every read $r \in \mathsf{Past}_{t_1}(e)$ we have $r \in \mathcal{E}(t_2)$ and $\mathsf{O}_{t_1}(r) = \mathsf{O}_{t_2}(r)$. Then e is inevitable in t_2.*

Proof. We argue that every event $e' \in \mathsf{Past}_{t_1}(e) \cup \{e\}$ is inevitable in t_2. Let t_2' be any lock-free maximal extension of t_2. Assume towards contradiction that $(\mathsf{Past}_{t_1}(e) \cup \{e\}) \setminus \mathcal{E}(t_2') \neq \emptyset$, and let

$$e_m = \arg \min_{e' \in (\mathsf{Past}_{t_1}(e) \cup \{e\}) \setminus \mathcal{E}(t_2')} \mathcal{I}_{t_1}(e')$$

be the first such event in t_1, and let $p_i = \mathsf{proc}(e_m)$ be the process of e_m. By Remark 9.2, for every event $e' \in \mathsf{Past}_{t_1}(e_m)$ we have $e' \in \mathsf{Past}_{t_1}(e)$, and since $\mathcal{I}_{t_1}(e') < \mathcal{I}_{t_1}(e_m)$, we have $e' \in \mathcal{E}(t_2')$. Let (x, y) be the edge of CFG_i labeled with e_m. Since $e' \in \mathcal{E}(t_2')$, the program counter of p_i becomes x at some point in t_2'. We examine the number of outgoing edges from node x.

1. If x has one outgoing edge, we distinguish whether e_m is a lock-acquire event or not.

(a) If e_m is not a lock-acquire event, then e_m is always enabled after e' in t'_2, hence t'_2 is not maximal, a contradiction.

(b) If $e_m : \mathsf{acquire}\ l$, since t'_2 is a lock-free trace, l is released in $s(t'_2)$, hence $e_m \in \mathsf{enabled}(t'_2)$ and t'_2 is not maximal, a contradiction.

2. If x has at least two outgoing edges then e_m is of the form $e_m : b_j(v_1, \dots, v_{n_i})$, where $v_i \in \mathcal{V}_i$ are local variables of p_i. Let $t_{e_m} = t'_2|\mathsf{Past}_{t'_1}(e_m)$. Since for every read $r \in \mathsf{Past}_{t_1}(e)$ we have $r \in \mathcal{E}(t_2)$ and $\mathsf{O}_{t_1}(r) = \mathsf{O}_{t_2}(r)$, by Remark 9.2, the same holds for reads $r \in \mathsf{Past}_{t_1}(e_m)$, i.e., for every read $r \in \mathsf{Past}_{t_1}(e_m)$ we have $r \in \mathcal{E}(t_2)$ and $\mathsf{O}_{t_1}(r) = \mathsf{O}_{t_2}(r)$. It is easy to see that additionally $r \in \mathcal{E}(t_{e_m})$ and $\mathsf{O}_{t_1}(r) = \mathsf{O}_{t_{e_m}}(r)$. Thus, we have $\mathsf{O}_{t_{e_m}} \subseteq \mathsf{O}_{t_1}$. By Lemma 9.1, we have $\mathsf{val}_{t_{e_m}}(r) = \mathsf{val}_{t_1}(r)$ for every read r, and since p_i is deterministic, the value of v on x is a function of those reads, and thus each v_i has the same value when the program counter of p_i reaches node x in t_1 and t'_2. Since e_m appears in t_1, e_m is always enabled after e' in t'_2, hence t'_2 is not maximal, a contradiction.

The desired result follows. □

9.5.2 Data-centric Dynamic Partial Order Reduction

Algorithm $\mathsf{DC\text{-}DPOR}$**.** We now present our data-centric DPOR algorithm. The algorithm receives as input a maximal trace t and annotation pair $\mathsf{A} = (\mathsf{A}^+, \mathsf{A}^-)$, where t is compatible with A^+. The algorithm scans t to detect conflicting read-write pairs of events that are not annotated, i.e, a read event $r \in \mathcal{R}(t)$ and a write event $w \in \mathcal{W}(t)$ such that $r \not\in \mathsf{dom}(\mathsf{A}^+)$ and $\mathsf{ConflRW}(r, w)$. If $w \not\in \mathsf{A}^-(r)$, then DC-DPOR will try to *mutate* r to w, i.e., the algorithm will push (r, w) in the positive annotation A^+ and call Realize to obtain a trace that realizes the new positive annotation. If the recursive call succeeds, then the algorithm will push w to the negative annotation of r, i.e., will insert w to $\mathsf{A}^-(r)$. This will prevent recursive calls from pushing (r, w) into their positive annotation. Algorithm 32 provides a formal description of DC-DPOR. Initially DC-DPOR is executed on input (t, A) where t is some arbitrary maximal trace, and $\mathsf{A} = (\varnothing, \varnothing)$ is a pair of empty annotations.

We say that DC-DPOR *explores* a class of $\mathcal{T}_{\mathcal{P}}/\sim_{\mathsf{O}}$ when it is called on some annotation input $\mathsf{A} = (\mathsf{A}^+, \mathsf{A}^-)$, where A^+ is realized by some (and hence, every) trace in that class. The

Algorithm 32: $\mathsf{DC\text{-}DPOR}(t, \mathsf{A})$

```
Input: A maximal trace t, an annotation pair A = (A+, A-)
   // Iterate over reads not yet mutated
1  foreach r ∈ E(t) \ dom(A+) in increasing index I_t(r) do
      // Find conflicting writes allowed by A-
2     foreach w ∈ E(t) s.t. Confl(r, w) and w ∉ A-(r) do
3        A+_{r,w} ← A+ ∪ {(r, w)}
         // Attempt mutation and update A-
4        Let t' ← Realize(A+_{r,w})
5        if t' ≠ ⊥ then
6           t'' ← a maximal extension of t'
7           A-(r) ← A-(r) ∪ {w}
8           A_{r,w} ← (A+_{r,w}, A-)
9           Call DC-DPOR(t'', A_{r,w})
10    end
11 end
```

representative trace is then the trace t' returned by Realize. The following two lemmas show the optimality of DC-DPOR, namely that the algorithm explores every such class at most once (*compactness*) and at least once (*completeness*). They both rely on the use of annotations, and the correctness of the procedure Realize (Lemma 9.6). We first state the compactness property, which follows by the use of negative annotations.

Lemma 9.8 (Compactness). *Consider any two executions of* DC-DPOR *on inputs* (t_1, A_1) *and* (t_2, A_2). *Then* $\mathsf{A}_1^+ \neq \mathsf{A}_2^+$.

Proof. Examine the recursion tree T generated by DC-DPOR, where every node u is labeled with the trace t_u and annotation input $\mathsf{A}_u = (\mathsf{A}_u^+, \mathsf{A}_u^-)$ given to DC-DPOR. Let x and y be the nodes that correspond to inputs (t_1, A_1) and (t_1, A_2) respectively, and we argue that $\mathsf{A}_x \neq \mathsf{A}_y$. If x is an ancestor of y, then when DC-DPOR was executed on input (t_x, A_x), a read r created $\mathsf{A}_{r,w}^+$ in Line 3 on the branch from x to the direction of y in T, with the property that $r \notin \mathsf{dom}(\mathsf{A}_x^+)$. Since the algorithm never removes pairs (r, w) from the positive annotations, we have that $(r, w) \in \mathsf{A}_y$, hence $\mathsf{A}_x \neq \mathsf{A}_y$. A similar argument holds if y is an ancestor of x. Now consider the case that x and y do not have an ancestor-descendant relationship. Let z be the lowest common

ancestor of x and y in T, and z_x (resp. z_y) the child of z in the direction of x (resp. y). Let r_x (resp. r_y) be the read on Line 3 that generated $\mathsf{A}^+_{z_x}$ (resp. $\mathsf{A}^+_{z_y}$), i.e.,

$$\mathsf{A}^+_{z_x} = \mathsf{A}^+_z \cup \{(r_x, w_x)\} \quad \text{and} \quad \mathsf{A}^+_{z_y} = \mathsf{A}^+_z \cup \{(r_y, w_y)\}$$

If $r_x = r_y = r$ then $w_x \neq w_y$, thus $\mathsf{A}^+_{z_x}(r) \neq \mathsf{A}^+_{z_y}$, and since the algorithm never removes pairs (r, w) from positive annotations we have that $\mathsf{A}_x \neq \mathsf{A}_y$. Now assume that $r_x \neq r_y$, and w.l.o.g. that $\mathcal{I}_{t_j}(r_x) < \mathcal{I}_{t_j}(r_y)$. Then, before DC-DPOR($\mathsf{A}_{z_y}$) is executed, Line 7 adds $w_x \in \mathsf{A}^-(r_x)$. Since the algorithm never removes entries from the negative annotation, by Line 3, we have that $(r_x, w_x) \notin \mathsf{A}^+_y$. In all cases we have $\mathsf{A}_x \neq \mathsf{A}_y$, as desired. □

We now turn our attention to completeness, namely that every realizable observation function is realized by a trace explored by DC-DPOR. The proof shows inductively that if t is a trace that realizes an observation function O, then DC-DPOR will explore a trace t_i that agrees with t on the first few read events. Then, Lemma 9.7 guarantees that the first read event r on which the two traces disagree appears in t_i, and so does the write event w that r observes in O. Hence DC-DPOR either will mutate $r \to w$ (if $w \notin \mathsf{A}^-(r)$), or it has already done so in some earlier steps of the recursion (if $w \in \mathsf{A}^-(r)$).

Lemma 9.9 (Completeness). *For every realizable observation function* O*,* DC-DPOR *generates a trace t that realizes* O*.*

Proof. Let T be the recursion tree of DC-DPOR. Given a node u of T, we denote by t_u and $\mathsf{A}_u = (\mathsf{A}^+_u, \mathsf{A}^-_u)$ the input of DC-DPOR on u. Since t_u is always a maximal extension of a trace returned by procedure Realize on input A^+_u, by Lemma 9.6 we have that t_u is compatible with A^+_u, thus it suffices to show that DC-DPOR is called with a positive annotation being equal to O. We define a traversal on T with the following properties:

1. If u is the current node of the traversal, then $\mathsf{A}^+_u \subseteq \mathsf{O}$.
2. If $\mathsf{A}^+_u \subset \mathsf{O}$, then the traversal proceeds either
   (a) to a node v of T with $\mathsf{A}^+_u \subset \mathsf{A}^+_v$,
   (b) to some other node of T,

   and Item 2b happens a finite number of times.

Observe that every time the traversal executes Item 2a, O agrees with A_v^+ on more reads than A_u^+. Since Item 2b is executed a finite number of times, any such traversal is guaranteed to reach a node w with $\mathsf{A}_w = \mathsf{O}$.

The traversal starts from u being the root of T, and Item 1 holds as then $\mathsf{A}_u^+ = \varnothing \subseteq \mathsf{O}$. Now consider that the traversal is on any node u that satisfies Item 1. Since t_u is a maximal extension of a trace returned by procedure Realize on input A_u^+, Lemma 9.6 guarantees that t_u is a maximal trace that realizes A_u^+. Let t^* be a trace that realizes O, and r be the first read of t^* not in A_u^+, i.e.,

$$r = \arg \min_{r' \in \mathcal{E}(t^*) \setminus \mathsf{dom}(\mathsf{A}_u^+)} \mathcal{I}_{t^*}(r')$$

and $w = \mathsf{O}_{t^*}(r)$. Then for every $r' \in \mathsf{Past}_{t^*}(r)$, we have $\mathcal{I}_{t^*}(r') < \mathcal{I}_{t^*}(r')$ and thus $r' \in \mathcal{E}(t_u)$ and $\mathsf{O}_{t^*}(r') = \mathsf{O}_{t_u}(r')$. By Lemma 9.7, we have $r \in \mathcal{E}(t_u)$. Since $\mathcal{I}_{t^*}(w) < \mathcal{I}_{t^*}(r)$, a similar argument yields that $w \in \mathcal{E}(t_u)$. We now distinguish two cases, based on whether $w \in \mathsf{A}_u^-(r)$ or not.

1. If $w \notin \mathsf{A}_u^-(r)$, then in Line 4 the algorithm will generate a trace t' that is compatible with the strengthened annotation $\mathsf{A}_u^+ \cup (r, w)$, and call itself recursively on some child v of u with $\mathsf{A}_v^+ = \mathsf{A}_u^+ \cup (r, w)$. The traversal proceeds to v and Item 1 holds, as desired.

2. If $w \in \mathsf{A}_u^-(r)$, then there exists a highest ancestor x of u in T where DC-DPOR was called with $(r, w) \in \mathsf{A}_x^-$. Following Line 7, this can only have happened if x has a sibling v in T with $(r, w) \in \mathsf{A}_v^+$. Let z be the parent of x, v, and we have $\mathsf{A}_z^+ \subset \mathsf{A}_u^+ \subset \mathsf{O}$, and $\mathsf{A}_v^+ = \mathsf{A}_z^+ \cup (r, w) \subseteq \mathsf{O}$. Thus, the traversal proceeds to v and Item 1 holds, as desired. In this case, we say that the traversal *backtracks to x through z*.

Finally, we argue that Item 2b will only occur a finite number of times in the traversal. Since T is finite, it suffices to argue that the traversal backtracks through any node z a finite number of times. Indeed, fix such a node z and let $x_1, x_2, \dots$ be the sequence of (not necessarily distinct) children of z that the traversal backtracks to, through z. Let r_i be the unique read in $\mathsf{dom}(\mathsf{A}_{x_i}^+) \setminus \mathsf{dom}(\mathsf{A}_z^+)$. By Line 1, we have that $\mathcal{I}_{t_z}(r_{i+1}) < \mathcal{I}_{t_z}(r_i)$, hence no node repeats in the sequence $(x_i)_i$, and thus the traversal backtracks through z a finite number of times. The desired result follows. $\square$

We thus arrive to the following theorem.

Theorem 9.3. *Consider a concurrent acyclic architecture $\mathcal{P}$ of processes on an acyclic state space, and $n = \max_{t \in \mathcal{T}_{\mathcal{P}}} |t|$ the maximum length of a trace of $\mathcal{P}$. The algorithm* DC-DPOR *explores each class of $\mathcal{T}_{\mathcal{P}} / \sim_{\mathsf{O}}$ exactly once, and requires $O\left(|\mathcal{T}_{\mathcal{P}} / \sim_{\mathsf{O}}| \cdot n^5\right)$ time.*

Proof. Lemma 9.8 and Lemma 9.9 guarantee the optimality of DC-DPOR, i.e., that DC-DPOR explores each class of $\mathcal{T}_{\mathcal{P}} / \sim_{\mathsf{O}}$ exactly once. The time spent in each class is the time for attempting all possible mutations on the witness trace t which is the trace used by the algorithm to explore the corresponding observation function. There are at most n^2 such mutations, and according to Theorem 9.2, each such mutation requires $O(n^3)$ time to be applied (or conclude that t cannot be mutated in the attempted way). The desired result follows. □

We note that our main goal is to explore the exponentially large $\mathcal{T}_{\mathcal{P}} / \sim_{\mathsf{O}}$ by spending polynomial time in each class. The n^5 factor in the bound comes from a crude complexity analysis.

9.6 Beyond Acyclic Architectures

In the current section we turn our attention to cyclic architectures. Recall that according to Theorem 9.2, procedure Realize is guaranteed to find a trace that realizes a positive annotation A^+, provided that the underlying architecture is acyclic.

Architecture acyclic reduction. Consider a cyclic architecture $\mathcal{P}$, and the corresponding communication graph $G_{\mathcal{P}} = (V_{\mathcal{P}}, E_{\mathcal{P}}, \mathsf{wt}_{\mathcal{P}})$. We call a set of edges $X \subseteq E_{\mathcal{P}}$ an *all-but-two cycle set* of $G_{\mathcal{P}}$ if every cycle of $G_{\mathcal{P}}$ contains at most two edges outside of X. Given an all-but-two cycle set, $X \subseteq E_{\mathcal{P}}$ we construct a second architecture $\mathcal{P}^X$, called the *acyclic reduction* of $\mathcal{P}$ over X, by means of the following process.

1. Let $Y = \bigcup_{(p_i, p_j) \in X} \mathsf{wt}_{\mathcal{P}}(p_i, p_j)$ be the set of variables that appear in edges of the set X. We introduce a set of new locks $\mathcal{L}^{\mathcal{O}}$ in $\mathcal{P}^X$ such that we have exactly one new lock $l_g \in \mathcal{L}^{\mathcal{O}}$ for each variable $g \in Y$.

2. For every process p_i, every write event $w \in \mathcal{W}_i$ with $\mathsf{loc}(w) \in Y$ is surrounded by an acquire/release pair on the new lock variable $l_{\mathsf{loc}(w)}$.

Observation equivalence refined by an edge set. Consider a cyclic architecture $\mathcal{P}$ and X an edge set of the underlying communication graph $G_{\mathcal{P}}$. We define a new equivalence on the trace space $\mathcal{T}_{\mathcal{P}}$ as follows. Two traces $t_1, t_2 \in \mathcal{T}_{\mathcal{P}}$ are *observationally equivalent refined by* X, denoted by $\sim_{\mathsf{O}}^{X}$, if the following hold:

1. $t_1 \sim_{\mathsf{O}} t_2$, and
2. for every edge $(p_i, p_j) \in X$, for every pair of distinct write events $w_1, w_2 \in \mathcal{W}(t_1) \cap (\mathcal{W}_i \cup \mathcal{W}_j)$ with $\mathsf{loc}(w_1) = \mathsf{loc}(w_2) = g$ and $g \in \mathsf{wt}_{\mathcal{P}}(p_i, p_j)$, we have that $\mathcal{I}_{t_1}(w_1) < \mathcal{I}_{t_1}(w_2)$ iff $\mathcal{I}_{t_2}(w_1) < \mathcal{I}_{t_2}(w_2)$

Clearly, $\sim_{\mathsf{O}}^{X}$ refines the observation equivalence $\sim_{\mathsf{O}}$. The following lemma captures that the Mazurkiewicz equivalence refines the observation equivalence refined by an edge set X.

Lemma 9.10. *For any two traces $t_1, t_2 \in \mathcal{T}_{\mathcal{P}}$, if $t_1 \sim_M t_2$ then $t_1 \sim_{\mathsf{O}}^{X} t_2$.*

Proof. By Theorem 9.2, we have $t_1 \sim_{\mathsf{O}} t_2$. Consider any pair of distinct write events $w_1, w_2 \in \mathcal{W}(t_1) \cap (\mathcal{W}_i \cup \mathcal{W}_j)$ with $\mathsf{loc}(w_1) = \mathsf{loc}(w_2) = g$ and $g \in \mathsf{wt}_{\mathcal{P}}(p_i, p_j)$, and observe that w_1 and w_2 are dependent. Hence, we have $w_1 \rightarrow_{t_1} w_2$ iff $w_1 \rightarrow_{t_2} w_2$, and thus $\mathcal{I}_{t_1}(w_1) < \mathcal{I}_{t_1}(w_2)$ iff $\mathcal{I}_{t_2}(w_1) < \mathcal{I}_{t_2}(w_2)$, as desired. □

Exponential succinctness. Similar to $\sim_{\mathsf{O}}$, we present an instance of an cyclic architectures where the equivalence $\sim_{\mathsf{O}}^{X}$ is exponentially more succinct than $\sim_M$, since in general it considers fewer reorderings of events that access variables of the edges of $E_{\mathcal{P}} \setminus X$, than the Mazurkiewicz reorderings on those events. Consider the architecture $\mathcal{P}$ in Fig. 9.7, which consists of three processes and two single global variables x and y. We choose an edge set as $X = \{(p_1, p_2)\}$, and X is an all-but-two cycle set of $G_{\mathcal{P}}$. We argue that $\sim_{\mathsf{O}}^{X}$ is exponentially more succinct than $\sim_M$ by showing exponentially many traces which are pairwise equivalence under $\sim_{\mathsf{O}}^{X}$ but not under $\sim_M$. Indeed, consider the set T which consists of all traces such that the following hold

1. All traces start with p_1 executing to completion, then p_2 executing its first statement, and p_3 executing its first statement.
2. All traces end with the last three events of p_2 followed by the last two events of p_3.

Note that $|T| = \binom{2 \cdot n}{n}$ as there are $(2 \cdot n)!$ ways to order the $2 \cdot n$ $\mathsf{write}\ y$ events of the two processes, but $n! \cdot n!$ orderings are invalid as they violate the program structure. All traces in T

Process p_1 :	Process p_2 :	Process p_3 :
1. write x	1. write x	1. write x
2. read x	2. write y	2. write y
	$\dots$	$\dots$
	$n+2$. write y	$n+2$. write y
	$n+3$. read y	$n+3$. read y
	$n+4$. read x	$n+4$. read x

Figure 9.7: A cyclic architecture of three processes.

have the same observation function, yet they are inequivalent under $\sim_M$ since every pair of them orders two write y events differently. Finally, $\mathcal{T}_{\mathcal{P}}/\sim_{\mathsf{O}}^{X}$ is only exponentially large, and since

$$|(\mathcal{T}_{\mathcal{P}}/\sim_M) \setminus (\mathcal{T}_{\mathcal{P}}/\sim_{\mathsf{O}}^{x})| \geqslant |T| - 1$$

we have that $\sim_{\mathsf{O}}^{X}$ is exponentially more succinct than $\sim_M$.

Data-centric DPOR on a cyclic architecture. We are now ready to outline the steps of the data-centric DPOR algorithm on a cyclic architecture $\mathcal{P}$, called DC-DPOR-Cyclic. First, we determine an all-but-two cycle set X of the underlying communication graph $G_{\mathcal{P}} = (V_{\mathcal{P}}, E_{\mathcal{P}}, \mathsf{wt}_{\mathcal{P}})$, and construct the acyclic reduction $\mathcal{P}^X$ of $\mathcal{P}$ over X. The set X can be chosen arbitrarily, e.g. by letting $|X| = |E_{\mathcal{P}}| - 2$ (i.e., adding in X all the edges of $G_{\mathcal{P}}$ except for two). Then, we execute DC-DPOR on $\mathcal{P}^X$, with the following two modifications on the procedure Realize.

1. Consider the graph $G = (V, E)$ constructed in Line 1 of Realize (Algorithm 31). For every pair of write events w, w' protected by some of the new locks $\ell \in \mathcal{L}^{\mathcal{O}}$, for every read event r such that $\mathsf{A}^{+}(r) = w$, if $(w, w') \in E$ then we add an edge (r, w) in E, and if $(w', r) \in E$, then we add an edge (w', w) in E.

2. If at the end of Item 1 G has a cycle, Realize returns $\perp$.

3. In Line 5 we use the edge set $E_{\mathcal{P}^X} \setminus X$. Hence for every variable x_{e_1,e_2} used in the 2SAT reduction, we have either $\mathsf{proc}(e_1) = \mathsf{proc}(e_2)$ or $(\mathsf{proc}(e_1), \mathsf{proc}(e_2)) \in E_{\mathcal{P}^X} \setminus X$.

Lemma 9.11. *Given a well-formed positive annotation* A^+ *over a basis* $(\tau_i)_i$*, and the modified communication graph* $G'_{\mathcal{P}^X} = (V_{\mathcal{P}^X}, E_{\mathcal{P}^X} \setminus X, \mathsf{wt}_{\mathcal{P}^X})$*,* Realize *constructs a trace* t *that realizes* A^+ *(or concludes that* A^+ *is not realizable) and requires* $O(n^3)$ *time, where* $n = \sum_i |\tau_i|$.

(Sketch). First, note that due to the new locks $\mathcal{L}^{\mathcal{O}}$, A^+ already induces a total order on the lock-acquire and lock-release events that access the same lock $l_g \in \mathcal{L}^{\mathcal{O}}$. Hence A^+ already induces a total order between the write events that are protected by the same lock $l_g \in \mathcal{L}^{\mathcal{O}}$. Thus, given the basis $(\tau_i)_i$, A^+ is realizable iff that total order respects A^+, and there is a way to order the remaining pairwise dependent events between pairs of processes $(p_i, p_j) \not\in X$, such that the ordering respects the sequential consistency axioms. The crucial property is that since X is an all-but-two cycle set of $G_{\mathcal{P}}$, the transitivity (Line 9) and antisymmetry (Line 6) clauses ensure that a satisfying assignment preserves acyclicity of the graph G' constructed in Line 24. The complexity analysis is similar to Lemma 9.6. □

We arrive at the following theorem.

Theorem 9.4. *Consider a concurrent architecture* $\mathcal{P}$ *of processes on an acyclic state space, and* $n = \max_{t \in \mathcal{T}_{\mathcal{P}}} |t|$ *the maximum length of a trace of* $\mathcal{P}$*. Let* X *be an all-but-two cycle set of the communication graph* $G_{\mathcal{P}}$*. The algorithm* DC-DPOR-Cyclic *explores each class of* $\mathcal{T}_{\mathcal{P}}/\sim_{\mathsf{O}}^X$ *exactly once, and requires* $O\left(|\mathcal{T}_{\mathcal{P}}/\sim_{\mathsf{O}}^X| \cdot n^5\right)$ *time.*

Proof. We argue that DC-DPOR-Cyclic is then optimal for the cyclic architecture $\mathcal{P}$ wrt the equivalence $\sim_{\mathsf{O}}^X$.

1. (*Compactness*). For any two distinct positive annotations A_1^+, A_2^+ examined by DC-DPOR when exploring the trace space of $\mathcal{P}^X$, Lemma 9.8 guarantees that $\mathsf{A}_1^+ \neq \mathsf{A}_2^+$. Let t_1 and t_2 be the traces returned by Realize on inputs A_1^+ and A_2^+ respectively. Assume that t_1 is not a prefix of t_2 (the argument is similar if t_2 is not a prefix of t_1, and since $\mathsf{A}_1^+ \neq \mathsf{A}_2^+$, it is not the case that each is a prefix of the other). This implies that at least one of the following holds.

   (a) There is a read event $r \in \mathcal{E}(t_1)$ such that $(r, \mathsf{O}_{t_1}(r)) \not\in \mathsf{O}_{t_2}$, in which case two different classes of $\sim_{\mathsf{O}}$ are explored. Since $\sim_{\mathsf{O}}^X$ refines $\sim_{\mathsf{O}}$ it follows that two different classes of $\sim \mathsf{O}^X$ are explored.

(b) There is a lock-acquire event $e_a \in \mathcal{E}(t_1)$ such that $(e_a, \mathsf{O}_{t_1}(e_a)) \notin \mathsf{O}_{t_2}$. In this case, the write event w protected by the lock-acquire event e_a either does not appear t_2 or there exists a conflicting write event w' in t_1 such that t_1 and t_2 are ordered differently in t_1 and t_2. Hence the two annotations A_1^+ and A_2^+ are used to explore different classes of $\sim_{\mathsf{O}}^X$.

2. (*Completeness*). Lemma 9.11 together with the completeness statement of Lemma 9.9 guarantees that for every observation function O of the trace space of $\mathcal{P}^X$, there is an annotation function A^+ used by $\mathsf{DC\text{-}DPOR}$ such that $\mathsf{O} = \mathsf{A}^+$. Since the lock-acquire and lock-release events are read and write events respectively, any two traces which have a different order on a pair of write events w, w' such that w and w' are protected by observable locks, will also be explored.

Finally, the maximum size of a trace in $\mathcal{P}$ is asymptotically equal to the maximum size of a trace in $\mathcal{P}^X$, from which the complexity bound follows. $\square$

9.7 Experiments

Here we report on the implementation and experimental evaluation of our data-centric DPOR algorithm.

9.7.1 Implementation Details

Implementation. We have implemented our data-centric DPOR in C++, by extending the tool Nidhugg[1]. Nidhugg is a powerful tool that utilizes the LLVM compiler infrastructure, and hence our treatment of programs is in the level of LLVM's intermediate representation (IR). Concurrent architectures are supported via POSIX threads.

Handling static arrays. The challenge in handling arrays (and other data structures) lies in the difficulty of determining whether two global events access the same location of the array (and thus are in conflict) or not. Indeed, this is not evident from the CFG of each process, but

[1] `https://github.com/nidhugg/nidhugg`

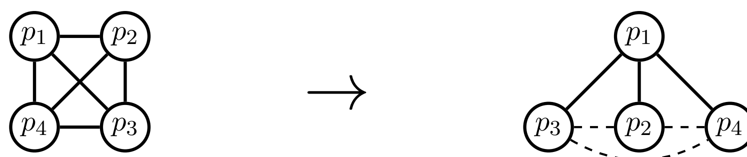


Figure 9.8: Converting a cyclic architecture to a star architecture which is acyclic. On the star, solid edges correspond to observation equivalence interleavings, and dashed edges correspond to Mazurkiewicz equivalence interleavings.

depends on the values of indexing variables (e.g. the value of local variable i in an access to $\mathsf{table}[i]$). DPOR methods offer increased precision, as during the exploration of the trace space, backtracking points are computed dynamically, given a trace, where the value of indexing variables is known. In our case, the value of indexing variables is also needed when procedure Realize is invoked to construct a trace which realizes a positive annotation A^+. Observe that the values of all such variables are determined by the value function $\mathsf{val}_{\mathsf{A}^+}$, and thus in every sequential trace τ_i of the basis $(\tau_i)_i$ of A^+ these values are also known. Hence, arrays are handled naturally by the dynamic flavor of the exploration.

Handling cyclic architectures. In order to effectively handle cyclic architectures, we followed the following process. Wlog, we considered that the input architecture always has the most difficult topology, namely it is a clique. First, the cyclic architecture is converted to a star architecture, by choosing some distinguished process p_1 as the root of the star, and the remaining processes $p_2, \dots p_k$ are the leaves. Recall that a a positive annotation yields a sequential trace for each process. We use the Mazurkiewicz equivalence to generate all possible Mazurkiewicz-based interleavings between traces of the leaf processes, and our observation equivalence to generate all possible observation-based interleavings between the root and every leaf process. Hence the observation equivalence is wrt the star sub-architecture, which is acyclic, and thus our techniques from Theorem 9.3 are applicable. We note that since the Mazurkiewicz interleavings always have to be generated among sequential traces (i.e., straight-line programs), we are generating them optimally (i.e., obtaining exactly one trace per Mazurkiewicz class) easily, using vector clocks [Mattern, 1989]. See Figure 9.8 for an illustration.

Optimizations. Since our focus is on demonstrating a new, data-centric principle of DPOR, we focused on a basic implementation and avoided engineering optimizations. We outline two straightforward algorithmic optimizations which were simple and useful.

1. (*Burst mutations*). Instead of performing one mutation at a time, the algorithm performs a sequence of several mutations at once. In particular, given a trace t, any time we want to add a pair (r, w) to the positive annotation, we also add $(r', \mathsf{O}_t(r'))$, where $r' \in \mathsf{Past}_t(r) \cup \mathsf{Past}_t(w)$ ranges over all read events in the causal past of r and w in t. This makes the recursion tree shallower, as now we do not need to apply any mutation (r, w), where $w = \mathsf{O}_t(r)$, individually.

2. (*Cycle detection*). As a preprocessing step, before executing procedure Realize on some positive annotation A^+ input, we test whether the graph G (in Line 1) already contains a cycle. The existence of a cycle is a proof that A^+ is not realizable, and requires linear instead of cubic time, as the graph is sparse.

9.7.2 Experimental Results

We now turn our attention to the experimental results. Our comparison is with the Source-DPOR algorithm from [Abdulla *et al.*, 2014] and the tool Nidhugg that implements it [Abdulla *et al.*, 2015]. To our knowledge, Source-DPOR is the latest and state-of-the-art DPOR which has implemented for C programs.

Experimental setup. In our experiments, we have compared our data-centric DPOR, with the Mazurkiewicz-based Source-DPOR introduced recently in [Abdulla *et al.*, 2014] as an important improvement over the traditional DPOR [Flanagan and Godefroid, 2005]. Our benchmark set consists of synthetic benchmarks, as well as benchmarks obtained from the TACAS Competition on Software Verification (SV-COMP). Most of the benchmarks have tunable size, by specifying a loop-unroll bound, or the number of threads running in parallel. In all cases, we compared th running time and number of traces explored by DC-DPOR and Source-DPOR. We have set a timeout of 1 hour. All benchmarks were executed on an Ubuntu-based virtual machine, given 4GB of memory and one 2GHz CPU.

Two synthetic benchmarks. First we analyze the two synthetic benchmarks *lastzero* and *opt_lock* found in Table 9.3a and Table 9.3b, respectively. The benchmark *lastzero* was introduced in [Abdulla *et al.*, 2014] to demonstrate the superiority of Source-DPOR over the traditional DPOR from [Flanagan and Godefroid, 2005]. It consists of n threads writing to an array, and 1

Benchmark	Traces		Time (s)	
	DC-DPOR	S-DPOR	DC-DPOR	S-DPOR
lastzero(4)	38	2,118	0.21	0.84
lastzero(5)	113	53,172	0.34	19.29
lastzero(6)	316	1,765,876	0.63	856
lastzero(7)	937	-	1.8	-
lastzero(8)	3,151	-	9.32	-
lastzero(9)	12,190	-	47.97	-
lastzero(10)	52,841	-	383.12	-

(a) Experiments on lastzero(n), for $n + 1$ threads. '-' indicates a timeout after 1 hour.

Benchmark	Traces		Time (s)	
	DC-DPOR	S-DPOR	DC-DPOR	S-DPOR
opt_lock(12)	141	785,674	0.35	252.64
opt_lock(13)	153	2,056,918	0.36	703.90
opt_lock(14)	165	5,385,078	0.43	1,880.12
opt_lock(15)	177	-	0.46	-
opt_lock(50)	597	-	5.91	-
opt_lock(100)	1,197	-	43.82	-
opt_lock(200)	2,397	-	450.99	-

(b) Experiments on opt_lock(n), where n is the number of attempts to optimistically lock. '-' indicates a timeout after 1 hour.

Table 9.3: Experimental results on two synthetic benchmarks.

thread reading from it (pseudocode in Algorithm 33). We observe that our DC-DPOR explores exponentially fewer traces than Source-DPOR. In fact, the number of traces explored by our data-centric approach scales polynomially, whereas the number explored by the Mazurkiewicz-based approach grows exponentially with the number of threads. Consequently, our DC-DPOR runs much faster, and manages to scale on larger input sizes. We note that the number of traces explored from Source-DPOR differs from the number reported in [Abdulla *et al.*, 2014]. This is natural as the implementation of [Abdulla *et al.*, 2014] handles programs written in Erlang, a functional language with concurrency mechanisms much different from C.

The benchmark opt_lock mimics an optimistic locking scheme of 2 threads (pseudocode in Algorithm 34). Each thread tries to update some variable, and afterwards checks if it was interrupted. If not, it terminates, otherwise it tries again, up to a total n number of attempts. Again, we see that the number of explored traces by DC-DPOR grows polynomially, whereas the number explored by Source-DPOR grows exponentially. Hence, our algorithm manages to handle much larger input sizes than the Mazurkiewicz-based Source-DPOR. Recall that, as Theorem 9.3 states, this exponential reduction in the explored traces comes with polynomial-time guarantees per trace.

Benchmarks from SV-COMP. We now turn our attention to benchmarks from SV-COMP, namely *fib_bench*, *pthread_demo*, *sigma_false* and *parker*, which are found in Table 9.4a, Table 9.4b, Table 9.4c and Table 9.4d, respectively. Similarly to our findings on the synthetic benchmarks, the data-centric DC-DPOR manages to explore fewer traces than the Mazurkiewicz-

Benchmark	Traces		Time (s)	
	DC-DPOR	S-DPOR	DC-DPOR	S-DPOR
fib_bench(4)	1,233	19,605	0.93	3.03
fib_bench(5)	8,897	218,243	7.41	37.82
fib_bench(6)	70,765	2,364,418	85.71	463.52

(a) Experiments on fib_bench(n), where n is the loop-unroll bound.

Benchmark	Traces		Time (s)	
	DC-DPOR	S-DPOR	DC-DPOR	S-DPOR
pthread_demo(8)	256	12,870	0.37	3.17
pthread_demo(10)	1,024	184,756	1.23	49.51
pthread_demo(12)	4,096	2,704,156	5.30	884.99

(b) Experiments on pthread_demo(n), where n is the loop-unroll bound.

Benchmark	Traces		Time (s)	
	DC-DPOR	S-DPOR	DC-DPOR	S-DPOR
sigma_false(6)	16	10,395	0.22	2.57
sigma_false(7)	22	135,135	0.26	38.41
sigma_false(8)	29	2,027,025	0.28	658.27
sigma_false(9)	37	-	0.38	-
sigma_false(10)	46	-	0.44	-

(c) Experiments on sigma_false(n), where n is the loop-unroll bound. ’-’ indicates a timeout after 1 hour.

Benchmark	Traces		Time (s)	
	DC-DPOR	S-DPOR	DC-DPOR	S-DPOR
parker(8)	1,254	3,343	1.52	1.33
parker(10)	2,411	6,212	5.03	3.96
parker(12)	4,132	10,361	8.09	5.62
parker(14)	6,529	16,022	11.96	6.86
parker(16)	9,714	23,427	19.89	10.85

(d) Experiments on parker(n), where n is the loop-unroll bound.

Table 9.4: Experimental results on four benchmarks from SV-COMP.

based Source-DPOR. In almost all cases, our algorithm run much faster, offering exponential gains in terms of time. One exception is the benchmark *parker*, where our DC-DPOR is slower. Although the number of traces explored is less than that of Source-DPOR, the latter method managed to spend less time in discovering each trace, which led to a smaller overall time. We note, however, that the improvement of Source-DPOR over DC-DPOR appears to grow only as a small polynomial wrt the input size. Recall that new traces are discovered by DC-DPOR using the procedure Realize, which can take cubic time in the worst case (Theorem 9.2). Hence, we identify optimizations to Realize as an important challenge that will contribute further to the scalability of our approach.

Algorithm 33: *lastzero*(n), for $n+1$ processes

```
Globals : int array[n+1]
// ------ Process j = 0 ------
Locals  : int i
1 i ← n
2 while array[i] ≠ 0 do
3 |  i ← i − 1
4 end

// ------ Process 1 < j ⩽ n ------
5 array[j] ← array[j − 1] + 1
```

Algorithm 34: *opt_lock*(n), for 2 processes and n attempts

```
Globals : int last_id, x
// ------ Process 0 < j < 2 ------
Locals  : int i
1 i ← 0
2 while i < n do
3 |  i ← i + 1
4 |  last_id ← j
5 |  x ← get_message(j)
6 |  if last_id = j then
7 |  |  return x
8 end
9 return -1
```

10 Automated Competitive Analysis in Real-time Scheduling with Graphs and Games

10.1 Introduction

In this chapter we study the well-known problem of scheduling a sequence of dynamically arriving real-time task instances with firm deadlines on a single processor by using graph games. In firm-deadline scheduling, a task instance (a *job*) that is completed by its deadline contributes a positive utility value to the system; a job that does not meet its deadline does not harm, but does not add any utility. The goal of the scheduling algorithm is to maximize the cumulated utility.

Problems considered. Since the taskset arising in a particular application is usually known, the present work focuses on the competitive analysis problem for *given* tasksets: Rather than from *all* possible tasksets as in [Baruah *et al.*, 1992], the task releases used for determining the *competitive ratio* are chosen from a taskset given as an input. We study the two relevant problems for the *automated* competitive analysis for given tasksets:

(1) The *competitive analysis* question asks to compute the competitive ratio of a given on-line algorithm.

(2) The *competitive synthesis* question asks to construct an on-line algorithm with optimal competitive ratio.

Both question are relevant in online-scheduling settings where the taskset is known in advance.

The competitive analysis problem can determine the performance of existing schedulers, and help with choosing the one that is best in the given setting. The competitive synthesis problem can provide a scheduler which is optimal by construction in the given setting.

Organization. The rest of this chapter is organized as follows.

Competitive Analysis. Given a taskset $\mathcal{T}$ and an on-line scheduling algorithm $\mathcal{A}$, the competitive analysis question asks to determine the competitive ratio of $\mathcal{A}$ when the arriving jobs are instances of tasks from $\mathcal{T}$. Our respective results are provided in the following sections:

1. In Section 10.2, we formally define our real-time scheduling problem.

2. In Section 10.3, we provide a formalism for on-line and clairvoyant scheduling algorithms as labeled transitions systems. We also show how automata on infinite words can be used to express natural constraints on the set of released job sequences (such as sporadicity and workload constraints).

3. In Sections 2.3.1 and 10.4.2, we define graph objectives on weighted multi-graphs and provide algorithms for solving those objectives.

4. In Section 10.4.3, we present a formal reduction of the competitive analysis problem to solving a multi-objective graph problem. Section 10.4.4 describes both general and implementation-specific optimizations for the above reduction, which considerably reduce the size of the obtained graph and thus make our approach feasible in practice.

5. In Section 10.4.5, we present a comparative study of the competitive ratio of several existing firm-deadline real-time scheduling algorithms. Our results show that the competitive ratio of any algorithm varies highly when varying tasksets, which highlights the usefulness of an automated competitive analysis framework: Our framework allows to replace human ingenuity (required for finding worst-case scenarios) by computing power, as the application designer can analyze different scheduling algorithms for the specific taskset arising in some application and compare their competitive ratio.

Competitive Synthesis. Given a taskset $\mathcal{T}$, the competitive synthesis question asks to construct an on-line scheduling algorithm $\mathcal{A}$ with optimal competitive ratio for $\mathcal{T}$. The competitive ratio

of $\mathcal{A}$ for $\mathcal{T}$ is at least as large as the competitive ratio of any other on-line scheduling algorithm for $\mathcal{T}$. Our respective results are presented in Section 10.5:

1. In Section 10.5.1, we consider a game model (a partial-observation game with memoryless strategies for Player 1 with mean-payoff and ratio objectives) that is suitable for the competitive synthesis of real-time scheduling algorithms. The mean-payoff (resp. ratio) objective allows to compute the cumulated utility (resp. competitive ratio) of the best on-line algorithm under the worst-case task sequence.

2. In Section 10.5.2, we establish that the relevant decision problems for the underlying game are NP-complete in the size of the game graph.

3. In Section 10.5.3, we use the game of Section 10.5.1 to tackle two relevant synthesis problems for a given taskset $\mathcal{T}$. First, we show that constructing an on-line scheduling algorithm with optimal worst-case average utility for $\mathcal{T}$ is in NP $\cap$ CONP in general, and polynomial in the size of the underlying game graph for reasonable choices of task utility values. Second, we show that constructing an on-line scheduling algorithm with optimal competitive ratio for $\mathcal{T}$ is in NP. These complexities are wrt the size of the constructed algorithm, represented explicitly as a labeled transition system. As a function of the input taskset $\mathcal{T}$ given in binary, all polynomial upper bounds become exponential upper bounds in the worst case.

10.2 Problem Definition

We consider a finite set of tasks $\mathcal{T} = \{\tau_1, \ldots, \tau_N\}$, to be executed on a single processor. We assume a discrete notion of real-time $t = k\varepsilon$, $k \geqslant 1$, where $\varepsilon > 0$ is both the unit time and the smallest unit of preemption (called a *slot*). Since both task releases and scheduling activities occur at slot boundaries only, all timing values are specified as positive integers. Every task τ_i releases countably many task instances (called *jobs*) $J_{i,j} := (\tau_i, j) \in \mathcal{T} \times \mathbb{N}^+$ (where $\mathbb{N}^+$ is the set of positive integers) over time (i.e., $J_{i,j}$ denotes that a job of task i is released at time j). All jobs, of all tasks, are independent of each other and can be preempted and resumed during execution without any overhead. Every task τ_i, for $1 \leqslant i \leqslant N$, is characterized by a 3-tuple $\tau_i = (C_i, D_i, V_i)$ consisting of its non-zero *worst-case execution time* $C_i \in \mathbb{N}^+$

(slots), its non-zero *relative deadline* $D_i \in \mathbb{N}^+$ (slots) and its non-zero *utility value* $V_i \in \mathbb{N}^+$ (rational utility values $V_1, \ldots, V_N$ can be mapped to integers by proper scaling). We denote by $D_{\max} = \max_{1 \leqslant i \leqslant N} D_i$ the maximum relative deadline in $\mathcal{T}$. Every job $J_{i,j}$ needs the processor for C_i (not necessarily consecutive) slots exclusively to execute to completion. All tasks have firm deadlines: only a job $J_{i,j}$ that completes within D_i slots, as measured from its release time, provides utility V_i to the system. A job that misses its deadline does not harm but provides zero utility. The goal of a real-time scheduling algorithm in this model is to maximize the *cumulated utility*, which is the sum of V_i times the number of jobs $J_{i,j}$ that can be completed by their deadlines, in a sequence of job releases generated by the *adversary*.

Notation on sequences. Let X be a finite set. For an infinite sequence $x = (x^\ell)_{\ell \geqslant 1} = (x^1, x^2, \ldots)$ of elements in X, we denote by x^ℓ the element in the ℓ-th position of x, and denote by $x(\ell) = (x^1, x^2, \ldots, x^\ell)$ the finite prefix of x up to position ℓ. We denote by X^∞ the set of all infinite sequences of elements from X. Given a function $f : X \to \mathbb{Z}$ (where $\mathbb{Z}$ is the set of integers) and a sequence $x \in X^\infty$, we denote by $f(x, k) = \sum_{\ell=1}^{k} f(x^\ell)$ the sum of the images of the first k sequence elements under f.

Job sequences. The released jobs form a discrete sequence, where at each time point the adversary releases at most one new job from every task. Formally, the adversary generates an infinite *job sequence* $\sigma = (\sigma^\ell)_{\ell \geqslant 1} \in \Sigma^\infty$, where $\Sigma = 2^{\mathcal{T}}$. The release of one job of task τ_i in time ℓ, for some $\ell \in \mathbb{N}^+$, is denoted by having $\tau_i \in \sigma^\ell$,. Then, a (single) new job $J_{i,j}$ of task τ_i is released at the beginning of slot ℓ: $j = \ell$ denotes the *release time* of $J_{i,j}$, which is also the earliest time that the job $J_{i,j}$ can be executed, and $d_{i,j} = j + D_i$ denotes its absolute *deadline*.

Admissible job sequences. We present a flexible framework, where the set of admissible job sequences that the adversary can generate may be restricted. The set $\mathcal{J}$ of *admissible* job sequences from Σ^∞ can be obtained by imposing one or more of the following (optional) admissibility restrictions:

($\mathcal{S}$) Safety constraints, which are restrictions that have to hold in every finite prefix of an admissible job sequence; e.g., they can be used to enforce job release constraints such as periodicity or sporadicity, and to impose temporal workload restrictions.

($\mathcal{L}$) Liveness restrictions, which assert infinite repetition of certain job releases in a job

sequence; e.g., they can be used to force the adversary to release a certain task infinitely often.

($\mathcal{W}$) Limit-average constraints, which restrict the long run average behavior of a job sequence; e.g., they can be used to enforce that the average load in the job sequences does not exceed a threshold.

These three types of constraints will be made precise in the next section where we formally state the problem definition.

Schedule. Given an admissible job sequence $\sigma \in \mathcal{J}$, the *schedule* $\pi = (\pi^\ell)_{\ell \geqslant 1} \in \Pi^\infty$, where $\Pi = ((\mathcal{T} \times \{0, \dots, D_{\max} - 1\}) \cup \varnothing)$, computed by a real-time scheduling algorithm for σ, is a function that assigns at most one job for execution to every slot $\ell \geqslant 1$: π^ℓ is either $\varnothing$ (i.e., no job is executed) or else (τ_i, j) (i.e., the job $J_{i,\ell-j}$ of task τ_i released j slots ago is executed). The latter must satisfy the following constraints:

1. $\tau_i \in \sigma^{\ell-j}$ (the job has been released),
2. $j < D_i$ (the job's deadline has not passed),
3. $|\{k : k > 0 \text{ and } \pi^{\ell-k} = (\tau_i, j') \text{ and } k + j' = j\}| < C_i$ (the job released in slot $\ell - j$ has not been completed).

Note that our definition of schedules uses relative indexing in the scheduling algorithms: At time point ℓ, the algorithm for schedule π^ℓ uses index j to refer to slot $\ell - j$. Recall that $\pi(k)$ denotes the prefix of length $k \geqslant 1$ of π. We define $\gamma_i(\pi, k)$ to be the number of jobs of task τ_i that are completed by their deadlines in $\pi(k)$. The cumulated utility $V(\pi, k)$ (also called utility for brevity) achieved in $\pi(k)$ is defined as $V(\pi, k) = \sum_{i=1}^{N} \gamma_i(\pi, k) \cdot V_i$.

Competitive ratio. We are interested in evaluating the performance of deterministic *on-line* scheduling algorithms $\mathcal{A}$, which, at time ℓ, do not know any of the σ^k for $k > \ell$ when running on $\sigma \in \mathcal{J}$. In order to assess the performance of $\mathcal{A}$, we will compare the cumulated utility achieved in the schedule $\pi_\mathcal{A}$ to the cumulated utility achieved in the schedule $\pi_\mathcal{C}$ provided by an optimal *off-line* scheduling algorithm, called a *clairvoyant* algorithm $\mathcal{C}$, working on the same job sequence. Formally, given a taskset $\mathcal{T}$, let $\mathcal{J} \subseteq \Sigma^\infty$ be the set of all admissible job sequences of $\mathcal{T}$ that satisfy given (optional) safety, liveness, and limit-average constraints. For every $\sigma \in \mathcal{J}$,

we denote by $\pi^\sigma_{\mathcal{A}}$ (resp. $\pi^\sigma_{\mathcal{C}}$) the schedule produced by $\mathcal{A}$ (resp. $\mathcal{C}$) under σ. The *competitive ratio* of the on-line algorithm $\mathcal{A}$ for the taskset $\mathcal{T}$ under the admissible job sequence set $\mathcal{J}$ is defined as

$$\mathcal{CR}_{\mathcal{J}}(\mathcal{A}) = \inf_{\sigma \in \mathcal{J}} \liminf_{k \to \infty} \frac{1 + V(\pi^\sigma_{\mathcal{A}}, k)}{1 + V(\pi^\sigma_{\mathcal{C}}, k)} \tag{10.1}$$

that is, the worst-case ratio of the cumulated utility of the on-line algorithm versus the clairvoyant algorithm, under all admissible job sequences. Note that adding 1 in numerator and denominator simply avoids division by zero issues.

Remark 10.1. Since, according to the definition of the competitive ratio $\mathcal{CR}_{\mathcal{J}}$ in Eq. (10.1), we focus on worst-case analysis, we do not consider randomized algorithms (such as Locke's best-effort policy [Locke, 1986]). Generally, for worst-case analysis, randomization can be handled by additional choices for the adversary. For the same reason, we do not consider scheduling algorithms that can use the unbounded history of job releases to predict the future (e.g., to capture correlations).

10.3 Modeling Formalisms in Our Framework

In this section, we present the definitions of several types of labeled transition systems (LTSs). We use LTSs as the modeling formalism for on-line and clairvoyant scheduling algorithms, as well as for modeling optional constraints on the released job sequences.

10.3.1 Labeled Transition Systems

We will consider both on-line and off-line scheduling algorithms that are formally modeled as *labeled transition systems (LTSs)*: Every deterministic finite-state on-line scheduling algorithm can be represented as a deterministic LTS, such that every input job sequence generates a unique run that determines the corresponding schedule. On the other hand, an off-line algorithm can be represented as a non-deterministic LTS, which uses the non-determinism to guess the appropriate job to schedule.

Labeled transitions systems (LTSs). Formally, a *labeled transition system* (LTS) is a tuple $L = (S, s_1, \Sigma, \Pi, \Delta)$, where S is a finite set of states, $s_1 \in S$ is the initial state, Σ is a finite set

of input actions, Π is a finite set of output actions, and $\Delta \subseteq S \times \Sigma \times S \times \Pi$ is the transition relation. Intuitively, $(s, x, s', y) \in \Delta$ if, given the current state s and input x, the LTS outputs y and makes a transition to state s'. If the LTS is deterministic, then there is always a unique output and next state, i.e., Δ is a function $\Delta : S \times \Sigma \to S \times \Pi$. Given an input sequence $\sigma \in \Sigma^\infty$, a *run* of L on σ is a sequence $\rho_\mathcal{A} = (p_\ell, \sigma_\ell, q_\ell, \pi_\ell)_{\ell \geqslant 1} \in \Delta^\infty$ such that $p_1 = s_1$ and for all $\ell \geqslant 2$, we have $p_\ell = q_{\ell-1}$. For a deterministic LTS, for each input sequence, there is a unique run.

Deterministic LTS for an on-line algorithm. For our analysis, on-line scheduling algorithms are represented as deterministic LTSs. Recall the definition of the sets $\Sigma = 2^\mathcal{T}$, and $\Pi = ((\mathcal{T} \times \{0, \ldots, D_{\max} - 1\}) \cup \varnothing)$. Every deterministic on-line algorithm $\mathcal{A}$ that uses finite state space (for all job sequences) can be represented as a deterministic LTS $L_\mathcal{A} = (S_\mathcal{A}, s_\mathcal{A}, \Sigma, \Pi, \Delta_\mathcal{A})$, where the states $S_\mathcal{A}$ correspond to the state space of $\mathcal{A}$, and $\Delta_\mathcal{A}$ correspond to the execution of $\mathcal{A}$ for one slot. Note that, due to relative indexing, for every current slot ℓ, the schedule π^ℓ of $\mathcal{A}$ contains elements from the set Π, and $(\tau_i, j) \in \pi^\ell$ uniquely determines the job $J_{i,\ell-j}$. Finally, we associate with $L_\mathcal{A}$ a reward function $r_\mathcal{A} : \Delta_\mathcal{A} \to \mathbb{N}$ such that $r_\mathcal{A}(\delta) = V_i$ if the transition δ completes a job of task τ_i, and $r_\mathcal{A}(\delta) = 0$ otherwise. Given the unique run $\rho^\sigma_\mathcal{A} = (\delta^\ell)_{\ell \geqslant 1}$ of $L_\mathcal{A}$ for the job sequence σ, where δ^ℓ denotes the transition taken at the beginning of slot ℓ, the cumulated utility in the prefix of the first k transitions in $\rho^\sigma_\mathcal{A}$ is $V(\rho^\sigma_\mathcal{A}, k) = \sum_{\ell=1}^{k} r_\mathcal{A}(\delta^\ell)$.

Most scheduling algorithms (such as EDF, FIFO, DOVER [Koren and Shasha, 1995], TD1 [Baruah *et al.*, 1992]) can be represented as a deterministic LTS. An illustration for EDF is given in the following example.

Example 10.1 (EDF as an LTS). Consider the taskset $\mathcal{T} = \{\tau_1, \tau_2\}$, with $D_1 = 3$, $D_2 = 2$ and $C_1 = C_2 = 2$. Fig. 10.1 represents the EDF (Earliest Deadline First) scheduling policy as a deterministic LTS for $\mathcal{T}$. Each state is represented by a matrix M, such that $M[i, j]$, $1 \leqslant i \leqslant N$, $1 \leqslant j \leqslant D_{\max} - 1$, denotes the remaining execution time of the job of task τ_i released j slots ago. Every transition is labeled with a set $T \in \Sigma$ of released tasks as well as with $(\tau_i, j) \in \Pi$, which denotes the unique job $J_{i,\ell-j}$ to be scheduled in the current slot ℓ. Released jobs with no chance of being scheduled are not included in the state space. For example, while being in the topmost state, the release of τ_1 makes the LTS take the transition to the leftmost state, where 1 unit of work is scheduled for the released task, and 1 unit remains, encoded by writing 1 in position $(1, 1)$ of the matrix M. In the next round, a new release of τ_2 will take the LTS to the middle state, with 2 units of workload in position $(1, 1)$. This is because the 2nd workload of the

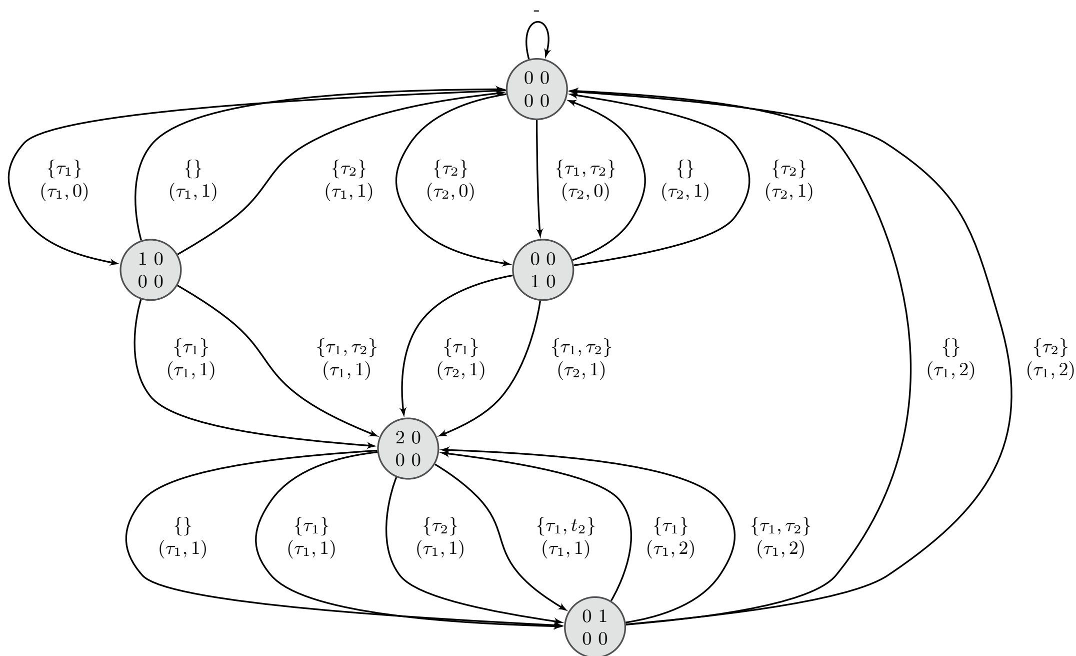


Figure 10.1: EDF for $\mathcal{T} = \{\tau_1, \tau_2\}$ with $D_1 = 3$, $D_2 = 2$ and $C_1 = C_2 = 2$, represented as a deterministic LTS.

first job is scheduled (thus the first job is scheduled to completion), and the newly released job is not scheduled in the current slot. Thus all 2 units of workload of the currently released job remain.

All scheduling algorithms considered here have been encoded similarly to EDF using the matrix M. Some more involved schedulers, such as DOVER, require some extra bits of information stored in the state.

The non-deterministic LTS. The clairvoyant algorithm $\mathcal{C}$ is formally a non-deterministic LTS $L_{\mathcal{C}} = (S_{\mathcal{C}}, s_{\mathcal{C}}, \Sigma, \Pi, \Delta_{\mathcal{C}})$, where each state in $S_{\mathcal{C}}$ is a $N \times (D_{\max} - 1)$ matrix M. For each time slot ℓ, the entry $M[i, j]$, $1 \leqslant i \leqslant N$, $1 \leqslant j \leqslant D_{\max} - 1$, denotes the remaining execution time of the job $J_{i,\ell-j}$ (i.e., the job of task i released j slots ago). For matrices M, M', subset $T \in \Sigma$ of newly released tasks, and scheduled job $P = (\tau_i, j) \in \Pi$, we have $(M, T, M', P) \in \Delta_{\mathcal{C}}$ iff $M[i, j] > 0$ and M' is obtained from M by

1. inserting all $\tau_i \in T$ into M,

2. decrementing the value at position $M[i, j]$, and

3. shifting the contents of M by one column to the right.

That is, M' corresponds to M after inserting all released tasks in the current state, executing a pending task for one unit of time, and reducing the relative deadlines of all tasks currently in the system. The initial state $s_{\mathcal{C}}$ is represented by the zero $N \times (D_{\max} - 1)$ matrix, and $S_{\mathcal{C}}$ is the smallest $\Delta_{\mathcal{C}}$-closed set of states that contains $s_{\mathcal{C}}$ (i.e., if $M \in S_{\mathcal{C}}$ and $(M, T, M', P) \in \Delta_{\mathcal{C}}$ for some T, M' and P, we have $M' \in S_{\mathcal{C}}$). Finally, we associate with $L_{\mathcal{C}}$ a reward function $r_{\mathcal{C}} : \Delta_{\mathcal{C}} \to \mathbb{N}$ such that $r_{\mathcal{C}}(\delta) = V_i$ if the transition δ completes a task τ_i, and $r_{\mathcal{C}}(\delta) = 0$ otherwise.

Remark 10.2. Note that the size of the above LTSs is the size of the state space of the corresponding scheduling algorithm. If the input consists of a succinct description of these algorithms (e.g., as a circuit [Galperin and Wigderson, 1983]), then the size of the corresponding LTS is, in general, exponential in the size of the input. This state-space explosion is generally unavoidable [Clarke *et al.*, 1999a]. In the complexity analysis of our algorithms, we consider the input schedulers to be in the explicit form of LTSs. When appropriate, we will state what the obtained results imply for the case where the input is succinct.

10.3.2 Admissible Job Sequences

Our framework allows to restrict the adversary to generate admissible job sequences $\mathcal{J} \subseteq \Sigma^{\infty}$, which can be specified via different constraints. Since a constraint on job sequences can be interpreted as a language (which is a subset of infinite words Σ^{∞} here), we will use automata as acceptors of such languages. Since an automaton is a deterministic LTS with no output, all our constraints will be described as LTSs with an empty set of output actions. We allow the following types of constraints:

($\mathcal{S}$) Safety constraints are defined by a deterministic LTS $L_{\mathcal{S}} = (S_{\mathcal{S}}, s_{\mathcal{S}}, \Sigma, \varnothing, \Delta_{\mathcal{S}})$, with a distinguished *absorbing* reject state $s_r \in S_{\mathcal{S}}$. An absorbing state is a state that has outgoing transitions only to itself. Every job sequence σ defines a unique run $\rho^{\sigma}_{\mathcal{S}}$ in $L_{\mathcal{S}}$, such that either no transition to s_r appears in $\rho^{\sigma}_{\mathcal{S}}$, or every such transition is followed solely by self-transitions to s_r. A job sequence σ is *admissible* to $L_{\mathcal{S}}$, if $\rho^{\sigma}_{\mathcal{S}}$ does not contain a transition to s_r. To obtain a safety LTS that does not restrict $\mathcal{J}$ at all, we simply use a trivial deterministic $L_{\mathcal{S}}$ with no transition to s_r.

Safety constraints restrict the adversary to release job sequences, where every finite prefix

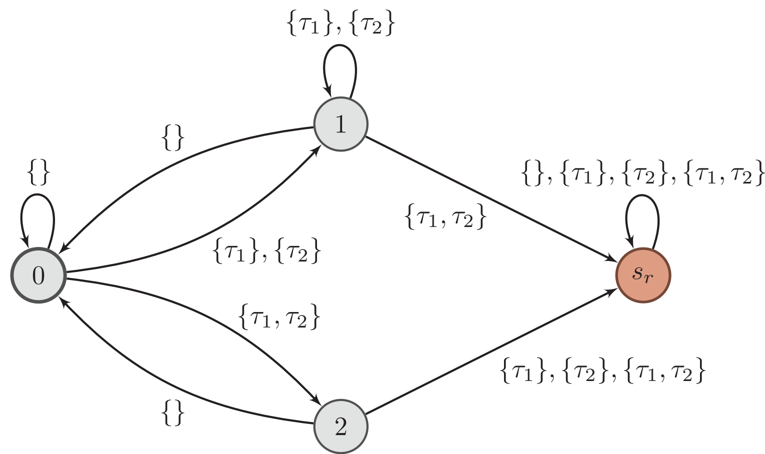


Figure 10.2: Example of a safety LTS $L_{\mathcal{S}}$ that restricts the adversary to release at most 2 units of workload in the last 2 slots. In state 0 no workload has been released in the last 2 slots, and thus all task releases are allowed for the next time slot. In state 1 there has been 1 unit of workload released in the last 2 slots, and thus in the next round only one task can be released. If no task is released in the next slot, then we transition back to state 0, to indicate that in the next time slot any combination of task releases is allowed. In state 2, there have been 2 units of workload released in the last 2 slots, and thus no task release is allowed in the next round. If no tasks are released, then the LTS transitions back to state 0, as in the next time slot any combination of task releases is allowed. If any of the above rules is violated, the safety LTS transitions to the absorbing state s_r, and remains there forever to indicate that the workload restriction has been violated.

satisfies some property (as they lead to the absorbing reject state s_r of $L_{\mathcal{S}}$ otherwise). Some well-known examples of safety constraints are (i) periodicity and/or sporadicity constraints, where there are fixed and/or a minimum time between the release of any two consecutive jobs of a given task, and (ii) absolute workload constraints [Golestani, 1991; Cruz, 1991], where the total workload released in the last k slots, for some fixed k, is not allowed to exceed a threshold λ. For example, in the case of absolute workload constraints, $L_{\mathcal{S}}$ simply encodes the workload in the last k slots in its state, and makes a transition to s_r whenever the workload exceeds λ. Fig. 10.2 shows an example of a constraint LTS for the taskset $\mathcal{T} = \{\tau_1, \tau_2\}$ with $C_1 = C_2 = 1$ that restricts the adversary to release at most 2 units of workload in the last 2 slots.

($\mathcal{L}$) Liveness constraints are modeled as a deterministic LTS $L_{\mathcal{L}} = (S_{\mathcal{L}}, s_{\mathcal{L}}, \Sigma, \varnothing, \Delta_{\mathcal{L}})$ with a distinguished *accept* state $s_a \in S_{\mathcal{L}}$. A job sequence σ is *admissible* to the liveness LTS $L_{\mathcal{L}}$

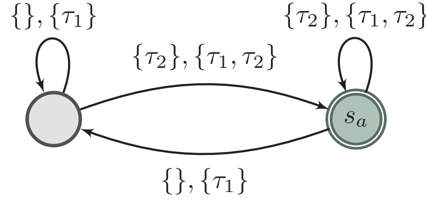


Figure 10.3: Example of a liveness LTS $L_{\mathcal{L}}$ that forces τ_2 to be released infinitely often. Each time the τ_2 is released, the LTS transitions to the accepting state s_a to indicate the release of the desired task. Recall that the accepting condition of $L_{\mathcal{L}}$ is that s_a needs to be appear infinitely often in an accepting path, meaning that the task τ_2 appears infinitely often.

if $\rho^{\sigma}_{\mathcal{L}}$ contains infinitely many transitions to s_a. For the case where there are no liveness constraint in $\mathcal{J}$, we use a LTS $L_{\mathcal{L}}$ consisting of state s_a only.

Liveness constraints force the adversary to release job sequences that satisfy some property infinitely often. For example, they could be used to guarantee that the release of some particular task τ_i does not eventually stall; the constraint is specified by a two-state LTS $L_{\mathcal{L}}$ that visits s_a whenever the current job set includes τ_i. A liveness constraint can also be used to prohibit infinitely long periods of overload [Baruah *et al.*, 1992], by choosing s_a as the idle state. Fig. 10.3 shows an example of a constraint LTS for the taskset $\mathcal{T} = \{\tau_1, \tau_2\}$ that forces the adversary to release τ_2 infinitely often.

($\mathcal{W}$) Limit-average constraints are defined by a deterministic weighted LTS $L_{\mathcal{W}} = (S_{\mathcal{W}}, s_{\mathcal{W}}, \Sigma, \varnothing, \Delta_{\mathcal{W}})$ equipped with a weight function $\mathsf{wt} : \Delta_{\mathcal{W}} \to \mathbb{Z}^d$ that assigns a vector of weights to every transition $\delta_{\mathcal{W}} \in \Delta_{\mathcal{W}}$. Given a threshold vector $\vec{\lambda} \in \mathbb{Q}^d$, where $\mathbb{Q}$ denotes the set of all rational numbers, a job sequence σ and the corresponding run $\rho^{\sigma}_{\mathcal{W}} = (\delta^{\ell}_{\mathcal{W}})_{\ell \geqslant 1}$ of $L_{\mathcal{W}}$, the job sequence is *admissible* to $L_{\mathcal{W}}$ if $\liminf_{k\to\infty} \frac{1}{k} \cdot \mathsf{wt}(\rho^{\sigma}_{\mathcal{W}}, k) \leqslant \vec{\lambda}$ with $\mathsf{wt}(\rho^{\sigma}_{\mathcal{W}}, k) = \sum_{i=1}^{k} \mathsf{wt}(\delta^{\ell}_{\mathcal{W}})$.

Consider a relaxed notion of workload constraints, where the adversary is restricted to generate job sequences whose *average* workload does not exceed a threshold λ. Since this constraint still allows "busy" intervals where the workload temporarily exceeds λ, it cannot be expressed as a safety constraint. To support such interesting average constraints of admissible job sequences, where the adversary is more relaxed than under absolute constraints, our framework explicitly supports limit-average constraints. Therefore, it is possible to express the average workload assumptions commonly used in the analysis of

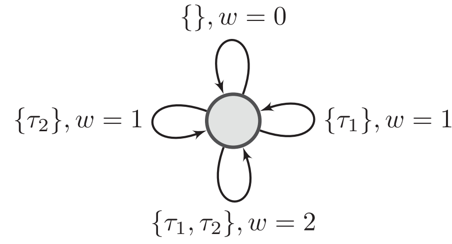


Figure 10.4: Example of a limit-average LTS $L_{\mathcal{W}}$ that tracks the average workload of jobs released by the adversary. This is achieved by having the weight function indicate the total workload released in each time slot. In this case we have $C_1 = C_2 = 1$, and the total workload equals the number of released tasks.

aperiodic task scheduling in soft-real-time systems [Abeni and Buttazzo, 1998; Haritsa *et al.*, 1990]. Other interesting cases of limit-average constraints include restricting the average sporadicity, and, in particular, average energy: ensuring that the limit-average of the energy consumption is below a certain threshold is an important concern in modern real-time systems [Aydin *et al.*, 2004]. Fig. 10.4 shows an example of a constraint LTS for the taskset $\mathcal{T} = \{\tau_1, \tau_2\}$ with $C_1 = C_2 = 1$, which can be used to restrict the average workload the adversary is allowed to release in the long run.

Remark 10.3. While, in general, such constraints are encoded as independent automata, it is often possible to encode certain constraints directly in the non-deterministic LTS of the clairvoyant scheduler instead. In particular, this is possible for restricting the limit-average workload, generating finite intervals of overload, and releasing a particular job infinitely often.

Synchronous product of LTSs. The *synchronous product* of two LTSs $L_1 = (S_1, s_1, \Sigma, \Pi, \Delta_1)$ and $L_2 = (S_2, s_2, \Sigma, \Pi, \Delta_2)$ is an LTS $L = (S, s, \Sigma, \Pi', \Delta)$ such that:

1. $S \subseteq S_1 \times S_2$,

2. $s = (s_1, s_2)$,

3. $\Pi' = \Pi \times \Pi$, and

4. $\Delta \subseteq S \times \Sigma \times S \times \Pi'$ such that $((q_1, q_2), T, (q_1', q_2'), (P_1, P_2)) \in \Delta$ iff $(q_1, T, q_1', P_1) \in \Delta_1$ and $(q_2, T, q_2', P_2) \in \Delta_2$.

The set of states S is the smallest Δ-closed subset of $S_1 \times S_2$ that contains s (i.e., $s \in S$, and for each $q \in S$, if there exist $q' \in S_1 \times S_2$, $T \in \Sigma$ and $P \in \Pi'$ such that $(q, T, q', P) \in \Delta$, then

$q' \in S$). That is, the synchronous product of L_1 with L_2 captures the joint behavior of L_1 and L_2 in every input sequence $\sigma \in \Sigma^\infty$ (L_1 and L_2 synchronize on input actions). Note that if both L_1 and L_2 are deterministic, so is their synchronous product. The synchronous product of $k > 2$ LTSs $L_1, \ldots, L_k$ is defined iteratively as the synchronous product of L_1 with the synchronous product of $L_2, \ldots, L_k$.

Overall approach for computing $\mathcal{CR}$. Our goal is to determine the worst-case competitive ratio $\mathcal{CR}_\mathcal{J}(\mathcal{A})$ for a given on-line algorithm $\mathcal{A}$. The inputs to the problem are the given taskset $\mathcal{T}$, an on-line algorithm $\mathcal{A}$ specified as a deterministic LTS $L_\mathcal{A}$, and the safety, liveness, and limit-average constraints specified as deterministic LTSs $L_\mathcal{S}, L_\mathcal{L}$ and $L_\mathcal{W}$, respectively, which constrain the admissible job sequences $\mathcal{J}$. Our approach uses a reduction to a multi-objective graph problem, which consists of the following steps:

1. Construct a non-deterministic LTS $L_\mathcal{C}$ corresponding to the clairvoyant off-line algorithm $\mathcal{C}$. Note that since $L_\mathcal{C}$ is non-deterministic, for every admissible job sequence σ, there are many possible runs in $L_\mathcal{C}$, of course also including the runs with maximum cumulative utility.

2. Take the synchronous product LTS $L_\mathcal{A} \times L_\mathcal{C} \times L_\mathcal{S} \times L_\mathcal{L} \times L_\mathcal{W}$. By doing so, a path in the product graph corresponds to *identically* labeled paths in the LTSs, and thus ensures that they agree on the same job sequence σ. This product can be represented by a multi-objective graph (as introduced in Section 2.3.1).

3. Determine $\mathcal{CR}_\mathcal{J}(\mathcal{A})$ by reducing the computation of the ratio given in Eq. (10.1) to solving a multi-objective problem on the product graph.

4. Employ several optimizations in order to reduce the size of product graph (see Sections 10.4.3 and 10.4.4.

10.4 Competitive Analysis of On-line Scheduling Algorithms

In this section, we address the competitive analysis problem: Given a taskset, a LTS $L_\mathcal{A}$ for the on-line scheduling algorithm, and optional constraint automata $L_\mathcal{S}$, $L_\mathcal{L}$, $L_\mathcal{W}$ for the set of admissible job sequences $\mathcal{J}$, our algorithms compute the competitive ratio $\mathcal{CR}_\mathcal{J}(\mathcal{A})$ of $\mathcal{A}$ in $\mathcal{J}$.

Our presentation is organized as follows: In Section 10.4.1, we define qualitative and quantitative objectives on multi-graphs. In Section 10.4.2, we provide algorithms for solving these graph objectives. In Section 10.4.3, we establish a formal reduction of computing the competitive ratio $\mathcal{CR}_{\mathcal{J}}(\mathcal{A})$ of an on-line scheduling algorithm $\mathcal{A}$ to solving for graph objectives on a suitable multi-graph. In Section 10.4.4, we describe several generic optimizations for this reduction that make the reduction practical. In Section 10.4.5, we provide the results of an automatic competitive analysis of a wide range of classic on-line scheduling algorithms, using a prototype implementation of our framework.

10.4.1 Graphs with Multiple Objectives

In this subsection, we define various objectives on graphs and outline the algorithms for solving them. We later show how the competitive analysis of on-line schedulers reduces to the solution algorithms of this section.

Multi-graphs. A *multi-graph* $G = (V, E)$, hereinafter called simply a *graph*, consists of a finite set V of n *nodes*, and a finite set of m *directed multiple edges* $E \subset V \times V \times \mathbb{N}^+$. For brevity, we will refer to an edge (u, v, i) as (u, v), when i is not relevant. We consider graphs in which for all $u \in V$, we have $(u, v) \in E$ for some $v \in V$, i.e., every node has at least one outgoing edge. An *infinite path* ρ of G is an infinite sequence of edges $e^1, e^2, \dots$ such that, for all $i \geqslant 1$ with $e^i = (u^i, v^i)$, we have $v^i = u^{i+1}$. Every such path ρ induces a sequence of nodes $(u^i)_{i \geqslant 1}$, which we will also call a path when the distinction is clear from the context, where ρ^i refers to u^i instead of e^i. Finally, we denote by Ω the set of all paths of G.

Objectives. Given a graph G, an objective Φ is a subset of Ω that defines the desired set of paths. We will consider safety, liveness, mean-payoff (limit-average), and ratio objectives, and their conjunction for multiple objectives.

Safety and liveness objectives: We consider safety and liveness objectives, both defined with respect to some subset of nodes $X, Y \subseteq V$. Given $X \subseteq V$, the *safety* objective, defined as $\mathsf{Safe}(X) = \{\rho \in \Omega : \ \forall i \geqslant 1, \rho^i \notin X\}$, represents the set of all paths that never visit the set X. The *liveness* objective defined as $\mathsf{Live}(Y) = \{\rho \in \Omega : \ \forall j \exists i > j \text{ s.t. } \rho^i \in Y\}$ represents the set of all paths that visit Y infinitely often.

Mean-payoff and ratio objectives: We consider the mean-payoff and ratio objectives, defined with respect to a weight function and a threshold. A *weight function* $\mathsf{wt} : E \to \mathbb{Z}^d$ assigns to each edge of G a vector of d integers. A weight function naturally extends to paths, with $\mathsf{wt}(\rho, k) = \sum_{i=1}^{k} \mathsf{wt}(\rho^i)$. The *mean-payoff* (or limit-average) of a path ρ is defined as:

$$\mathsf{MP}(\mathsf{wt}, \rho) = \liminf_{k \to \infty} \frac{1}{k} \cdot \mathsf{wt}(\rho, k);$$

i.e., it is the long-run average of the weights of the path. Given a weight function wt and a threshold vector $\vec{\nu} \in \mathbb{Q}^d$, the corresponding objective is given as:

$$\mathsf{MP}(\mathsf{wt}, \vec{\nu}) = \{\rho \in \Omega : \ \mathsf{MP}(\mathsf{wt}, \rho) \leqslant \vec{\nu}\};$$

that is, the set of all paths such that the mean-payoff of their weights is at most $\vec{\nu}$ (where we consider pointwise comparison for vectors). For weight functions $\mathsf{wt}_1, \mathsf{wt}_2 : E \to \mathbb{N}^d$, the *ratio* of a path ρ is defined as:

$$\mathsf{Ratio}(\mathsf{wt}_1, \mathsf{wt}_2, \rho) = \liminf_{k \to \infty} \frac{\vec{\mathbf{1}} + \mathsf{wt}_1(\rho, k)}{\vec{\mathbf{1}} + \mathsf{wt}_2(\rho, k)},$$

which denotes the limit infimum of the coordinate-wise ratio of the sum of weights of the two functions; $\vec{\mathbf{1}}$ denotes the d-dimensional all-1 vector. Given weight functions wt_1, wt_2 and a threshold vector $\vec{\nu} \in \mathbb{Q}^d$, the ratio objective is given as:

$$\mathsf{Ratio}(\mathsf{wt}_1, \mathsf{wt}_2, \vec{\nu}) = \{\rho \in \Omega : \ \mathsf{Ratio}(\mathsf{wt}_1, \mathsf{wt}_2, \rho) \leqslant \vec{\nu}\}$$

that is, the set of all paths such that the ratio of cumulative rewards w.r.t wt_1 and wt_2 is at most $\vec{\nu}$.

Example 10.2 (Multi-graph). Consider the multi-graph shown in Fig. 10.5, with a weight function of dimension $d = 2$. Note that there are two edges from node 3 to node 5 (represented as edges $(3, 5, 1)$ and $(3, 5, 2)$). In the graph we have a weight function with dimension 2. Note that the two edges from node 3 to node 5 have incomparable weight vectors.

Decision problem. The decision problem we consider is as follows: Given the graph G, an initial node $s \in V$, and an objective Φ (which can be a conjunction of several objectives), determine whether there exists a path ρ that starts from s and belongs to Φ, i.e., $\rho \in \Phi$. For simplicity of presentation, we assume that every $u \in V$ is reachable from s (unreachable nodes can be discarded by preprocessing G in $O(m)$ time). We first present algorithms for each of safety, liveness, mean-payoff, and ratio objectives separately, and then for their conjunction.

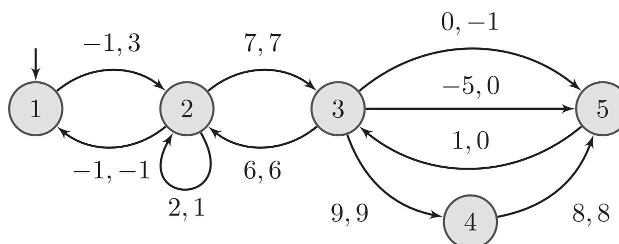


Figure 10.5: An example of a multi-graph G.

10.4.2 Algorithms for Solving Graphs with Multiple Objectives

We now describe the algorithms for solving the graph objectives introduced in the last subsection.

Algorithms for safety and liveness objectives. The algorithm for the objective $\mathsf{Safe}(X)$ is straightforward. We first remove the set X of nodes and then perform an SCC (maximal strongly connected component) decomposition of G. Then, we perform a single graph traversal to identify the set of nodes V_X which can reach an SCC that contains at least one edge (i.e., it contains either a single node with a self-loop, or more than one nodes). In the end, we obtain a graph $G = (V_X, E_X)$ such that $E_X = E \cap V_X \times V_X$. Thus, the objective $\mathsf{Safe}(X)$ is satisfied in the resulting graph, and the algorithm answers yes iff $s \in V_X$. Using the algorithm of [Tarjan, 1972] for performing the SCC decomposition, this algorithm requires $O(m)$ time.

To solve for the objective $\mathsf{Live}(Y)$, initially perform an SCC decomposition of G. We call an SCC V_{SCC} *live*, if (i) either $|V_{\text{SCC}}| > 1$, or $V_{\text{SCC}} = \{u\}$ and $(u, u) \in E$; and (ii) $V_{\text{SCC}} \cap Y \neq \emptyset$. Then $\mathsf{Live}(Y)$ is satisfied in G iff there exists a live SCC V_{SCC} that is reachable from s. This is because every node u in V_{SCC} can reach every node in V_{SCC}, and thus there is a path $u \rightsquigarrow u$ in V_{SCC}. Since V_{SCC} is a live SCC, the same holds for nodes $u \in V_{\text{SCC}} \cap Y$. Then a witness path can be constructed which first reaches some node $u \in V_{\text{SCC}} \cap Y$, and then keeps repeating the path $u \rightsquigarrow u$. Using the algorithm of [Tarjan, 1972] for performing the SCC decomposition, this algorithm also requires $O(m)$ time.

Algorithms for mean-payoff objectives. We distinguish between the case when the weight function has a single dimension ($d = 1$) versus the case when the weight function has multiple dimensions ($d > 1$).

Single dimension: In the case of a single-dimensional weight function, a single weight is assigned to every edge, and the decision problem of the mean-payoff objective reduces to

determining the mean weight of a minimum-weight simple cycle in G, as the latter also determines the mean-weight by infinite repetition. Using the algorithms of [Karp, 1978; Madani, 2002], this process requires $O(n \cdot m)$ time. When the objective is satisfied, the process also returns a simple cycle C, as a witness to the objective. From C, a path $\rho \in \mathsf{MP}(\mathsf{wt}, \vec{\nu})$ is constructed by infinite repetitions of C.

Multiple dimensions: When $d > 1$, the mean-payoff objective reduces to determining the feasibility of a linear program (LP). For $u \in V$, let $\mathsf{IN}(u)$ be the set of incoming, and $\mathsf{OUT}(u)$ the set of outgoing edges of u. As shown in [Velner *et al.*, 2015a], G satisfies $\mathsf{MP}(\mathsf{wt}, \vec{\nu})$ iff the following set of constraints on $\vec{x} = (x_e)_{e \in E_{\text{SCC}}}$ with $x_e \in \mathbb{Q}$ is satisfied simultaneously on some SCC V_{SCC} of G with induced edges $E_{\text{SCC}} \subseteq E$.

$$\begin{aligned} & x_e \geqslant 0 && e \in E_{\text{SCC}} \\ & \sum_{e \in \mathsf{IN}(u)} x_e = \sum_{e \in \mathsf{OUT}(u)} x_e && u \in V_{\text{SCC}} \\ & \sum_{e \in E_{\text{SCC}}} x_e \cdot \mathsf{wt}(e) \leqslant \vec{\nu} \\ & \sum_{e \in E_{\text{SCC}}} x_e \geqslant 1 \end{aligned} \tag{10.2}$$

The quantities x_e are intuitively interpreted as “flows”. The first constraint specifies that the flow of each edge is non-negative. The second constraint is a flow-conservation constraint. The third constraint specifies that the objective is satisfied if we consider the relative contribution of the weight of each edge, according to the flow of the edge. The last constraint asks that the preceding constraints are satisfied by a non-trivial (positive) flow. Hence, when $d > 1$, the decision problem reduces to solving a LP, and the time complexity is polynomial [Khachiyan, 1979].

The witness path construction from a feasible solution consists of two steps:

1. Construction of a multi-cycle from the feasible solution; and

2. Construction of an infinite witness path from the multi-cycle.

We describe the two steps in detail. Formally, a *multi-cycle* is a finite set of cycles with multiplicity $\mathcal{MC} = \{(C_1, m_1), (C_2, m_2), \ldots, (C_k, m_k)\}$, such that every C_i is a simple cycle and m_i is its multiplicity. The construction of a multi-cycle from a feasible solution

$\vec{x}$ is as follows. Let $\mathcal{E} = \{e : x_e > 0\}$. By scaling each edge flow x_e by a common factor z, we construct the set $\mathcal{X} = \{(e, z \cdot x_e) : e \in \mathcal{E}\}$, with $\mathcal{X} \subset E_{\text{SCC}} \times \mathbb{N}^+$. Then, we start with $\mathcal{MC} = \varnothing$ and apply iteratively the following procedure until $\mathcal{X} = \varnothing$:

(i) find a pair $(e_i, m_i) = \arg\min_{(e_j, m_j) \in \mathcal{X}} m_j$,

(ii) form a cycle C_i that contains e_i and only edges that appear in $\mathcal{X}$ (because of Eq. (10.2), this is always possible),

(iii) add the pair (C_i, m_i) in the multi-cycle $\mathcal{MC}$,

(iv) subtract m_i from all elements (e_j, m_j) of $\mathcal{X}$ such that the edge e_j appears in C_i,

(v) remove from $\mathcal{X}$ all $(e_j, 0)$ pairs, and repeat.

Since V_{SCC} is an SCC, there is a path $C_i \rightsquigarrow C_j$ for all C_i, C_j in $\mathcal{MC}$. Given the multi-cycle $\mathcal{MC}$, the infinite path that achieves the weight at most $\vec{\nu}$ is not periodic, but generated by Algorithm 35. Note that Line 9 is used so that the effects of the intermediate paths $C_1 \rightsquigarrow C_2$ and $C_2 \rightsquigarrow C_1$ diminish by staying in each C_1, C_2 for increasingly large ℓ.

Algorithm 35: Multi-objective witness

Input: A graph $G = (V, E)$, and a multi-cycle $\mathcal{MC} = \{(C_1, m_1), (C_2, m_2), \dots, (C_k, m_k)\}$

Output: An infinite path $\rho \in \mathsf{MP}(\mathsf{wt}, \vec{\nu})$

```
1  ℓ ← 1
2  while True do
3      Repeat C_1 for ℓ · m_1 times
4      C_1 ⇝ C_2
5      Repeat C_2 for ℓ · m_2 times
6      ...
7      Repeat C_k for ℓ · m_k times
8      C_k ⇝ C_1
9      ℓ ← ℓ + 1
10 end
```

Algorithm for ratio objectives. We now consider ratio objectives, and present a reduction to mean-payoff objectives. Consider the weight functions wt_1, wt_2 and the threshold vector $\vec{\nu} = \frac{\vec{p}}{\vec{q}}$ as the component-wise division of vectors $\vec{p}, \vec{q} \in \mathbb{N}^d$. We define a new weight function

$\mathsf{wt} : E \to \mathbb{Z}^d$ such that, for all $e \in E$, we have $\mathsf{wt}(e) = \vec{q} \cdot \mathsf{wt}_1(e) - \vec{p} \cdot \mathsf{wt}_2(e)$ (where $\cdot$ denotes component-wise multiplication). It is easy to verify that $\mathsf{Ratio}(\mathsf{wt}_1, \mathsf{wt}_2, \vec{\nu}) = \mathsf{MP}(\mathsf{wt}, \vec{0})$, and thus we solve the ratio objective by solving the new mean-payoff objective, as described above.

Algorithms for conjunctions of objectives. Finally, we consider the conjunction of a safety, a liveness, and a mean-payoff objective (note that we have already described a reduction of ratio objectives to mean-payoff objectives). More specifically, given a weight function wt, a threshold vector $\vec{\nu} \in \mathbb{Q}$, and sets $X, Y \subseteq V$, we consider the decision problem for the objective $\Phi = \mathsf{Safe}(X) \cap \mathsf{Live}(Y) \cap \mathsf{MP}(\mathsf{wt}, \vec{\nu})$. The procedure is as follows:

1. Initially compute G_X from G as in the case of a single safety objective.

2. Then, perform an SCC decomposition on G_X.

3. For every live SCC V_{SCC} that is reachable from s, solve for the mean-payoff objective in V_{SCC}. Return yes, if $\mathsf{MP}(\mathsf{wt}, \vec{\nu})$ is satisfied in any such V_{SCC}.

If the answer to the decision problem is yes, then the witness consists of a live SCC V_{SCC}, along with a multi-cycle (resp. a cycle for $d = 1$). The witness infinite path is constructed as in Algorithm 35, with the only difference that at the end of each while loop a live node from Y in the SCC V_{SCC} is additionally visited. The time required for the conjunction of objectives is dominated by the time required to solve for the mean-payoff objective. Theorem 10.1 summarizes the results of this section.

Theorem 10.1. *Let $G = (V, E)$ be a graph, $s \in V$, $X, Y \subseteq V$, $\mathsf{wt} : E \to \mathbb{Z}^d$, wt_1, wt_2: $E \to \mathbb{N}^d$ weight functions, and $\vec{\nu} \in \mathbb{Q}^d$. Let $\Phi_1 = \mathsf{Safe}(X) \cap \mathsf{Live}(Y) \cap \mathsf{MP}(\mathsf{wt}, \vec{\nu})$ and $\Phi_2 = \mathsf{Safe}(X) \cap \mathsf{Live}(Y) \cap \mathsf{Ratio}(\mathsf{wt}_1, \mathsf{wt}_2, \vec{\nu})$. The decision problem of whether G satisfies the objective Φ_1 (resp. Φ_2) from s requires*

1. *$O(n \cdot m)$ time, if $d = 1$.*

2. *Polynomial time, if $d > 1$.*

If the objective Φ_1 (resp. Φ_2) is satisfied in G from s, then a finite witness (an SCC and a cycle for single dimension, and an SCC and a multi-cycle for multiple dimensions) exists and can be constructed in polynomial time.

Example 10.3 (Mean-payoff objective with one dimensional weight functions). Consider the graph in Fig. 10.5. Starting from node 1, the mean-payoff-objective $\mathsf{MP}(\mathsf{wt}, \vec{0})$ is satisfied by the multi-cycle $\mathcal{MC} = \{(C_1, 1), (C_2, 2)\}$, with $C_1 = ((1,2),(2,1))$ and $C_2 = ((3,5),(5,3))$. A solution to the corresponding LP is $x_{(1,2)} = x_{(2,1)} = \frac{1}{3}$ and $x_{(3,5)} = x_{(5,3)} = \frac{2}{3}$, and $x_e = 0$ for all other $e \in E$. Algorithm 35 then generates a witness path for the objective. The objective is also satisfied in conjunction with $\mathsf{Safe}(\{4\})$ or $\mathsf{Live}(\{4\})$. In the latter case, a witness path additionally traverses the edges $(3,4)$ and $(4,5)$ before transitioning from C_1 to C_2.

Example 10.4 (Mean-payoff objective with two dimensional weight function). Consider the same graph of Fig. 10.5, where now instead of a single weight function of two dimensions, we have two weight functions $\mathsf{wt}_1, \mathsf{wt}_2 : E \to \mathbb{Z}$, of a single dimension each. The first (resp. second) weight of each edge is wrt to the weight function wt_1 (resp. wt_2). The ratio objective $\mathsf{Ratio}(\mathsf{wt}_1, \mathsf{wt}_2, -4)$ is satisfied by traversing the cycle $C = ((3,5),(5,3))$ repeatedly.

10.4.3 Reduction of Competitive Analysis to Graphs with Multiple Objectives

We present a formal reduction of the computation of the competitive ratio of an on-line scheduling algorithm with constraints on job sequences to the multi-objective graph problem. The input consists of the taskset, a deterministic LTS for the on-line algorithm, a non-deterministic LTS for the clairvoyant algorithm, and optional deterministic LTSs for the constraints. We first describe the process of computing the competitive ratio $\mathcal{CR}_{\mathcal{J}}(\mathcal{A})$, where $\mathcal{J}$ is a set of job sequences only subject to safety and liveness constraints. We later show how to handle limit-average constraints.

Reduction for Safety and Liveness Constraints. Given the deterministic and non-deterministic LTS $L_{\mathcal{A}}$ and $L_{\mathcal{C}}$ with reward functions $r_{\mathcal{A}}$ and $r_{\mathcal{C}}$, respectively, and optionally safety and liveness LTS $L_{\mathcal{S}}$ and $L_{\mathcal{L}}$, let $L = L_{\mathcal{A}} \times L_{\mathcal{C}} \times L_{\mathcal{S}} \times L_{\mathcal{L}}$ be their synchronous product. Hence, L is a non-deterministic LTS $(S, s_1, \Sigma, \Pi, \Delta)$, and every job sequence σ yields a set of runs R in L, such that each $\rho \in R$ captures the joint behavior of $\mathcal{A}$ and $\mathcal{C}$ under σ. Note that for each such ρ the behavior of $\mathcal{A}$ is unchanged, but the behavior of $\mathcal{C}$ generally varies due to its non-determinism. Let $G = (V, E)$ be the multi-graph induced by L, that is, $V = S$ and $(M, M', j) \in E$ for all $1 \leqslant j \leqslant i$ iff there are i transitions $(M, T, M', P) \in \Delta$. Let $\mathsf{wt}_{\mathcal{A}}$ and $\mathsf{wt}_{\mathcal{C}}$ be the weight functions that assign to each edge of G the reward that the respective algorithm obtains from the

corresponding transition in L. Let X be the set of states in G whose $L_{\mathcal{S}}$ component is s_r, and Y the set of states in G whose $L_{\mathcal{L}}$ component is s_a. It follows that for all $\nu \in \mathbb{Q}$, we have that $\mathcal{CR}_{\mathcal{J}}(\mathcal{A}) \leqslant \nu$ iff the objective $\Phi_\nu = \mathsf{Safe}(X) \cap \mathsf{Live}(Y) \cap \mathsf{Ratio}(\mathsf{wt}_{\mathcal{A}}, \mathsf{wt}_{\mathcal{C}}, \nu)$ is satisfied in G from the state s_1. As the dimension in the ratio objective is one, Case 1 of Theorem 10.1 applies, and we obtain the following:

Lemma 10.1. *Given the product graph $G = (V, E)$ of n nodes and m edges, a rational $\nu \in \mathbb{Q}$, and a set of job sequences $\mathcal{J}$ admissible for safety and liveness LTSs, determining whether $\mathcal{CR}_{\mathcal{J}}(\mathcal{A}) \leqslant \nu$ requires $O(n \cdot m)$ time.*

Since $0 \leqslant \mathcal{CR}_{\mathcal{J}}(\mathcal{A}) \leqslant 1$, the problem of determining the competitive ratio reduces to finding $v = \sup\{\nu \in \mathbb{Q} : \Phi_\nu \text{ is satisfied in } G\}$. Because this value corresponds to the ratio of the corresponding rewards obtained in a simple cycle in G, it follows that v is the maximum of a finite set, and can be determined exactly by an *adaptive binary search*.

Reduction for Limit-Average Constraints. Finally, we turn our attention to limit-average constraints and the LTS $L_{\mathcal{W}}$. We follow a similar approach as above, but this time including $L_{\mathcal{W}}$ in the synchronous product, i.e., $L = L_{\mathcal{A}} \times L_{\mathcal{C}} \times L_{\mathcal{S}} \times L_{\mathcal{L}} \times L_{\mathcal{W}}$. Let $\mathsf{wt}_{\mathcal{A}}$ and $\mathsf{wt}_{\mathcal{C}}$ be weight functions that assign to each edge $e \in E$ in the corresponding multi-graph a vector of $d+1$ weights as follows. In the first dimension, $\mathsf{wt}_{\mathcal{A}}$ and $\mathsf{wt}_{\mathcal{C}}$ are defined as before, assigning to each edge of G the corresponding rewards of $\mathcal{A}$ and $\mathcal{C}$. In the remaining d dimensions, $\mathsf{wt}_{\mathcal{C}}$ is always 1, whereas $\mathsf{wt}_{\mathcal{A}}$ equals the value of the weight function wt of $L_{\mathcal{W}}$ on the corresponding transition. Let $\vec{\lambda}$ be the threshold vector of $L_{\mathcal{W}}$. It follows that for all $\nu \in \mathbb{Q}$, we have that $\mathcal{CR}_{\mathcal{J}}(\mathcal{A}) \leqslant \nu$ iff the objective $\Phi_\nu = \mathsf{Safe}(X) \cap \mathsf{Live}(Y) \cap \mathsf{Ratio}(\mathsf{wt}_{\mathcal{A}}, \mathsf{wt}_{\mathcal{C}}, (\nu, \vec{\lambda}))$ is satisfied in G from the state s that corresponds to the initial state of each LTS, where $(\nu, \vec{\lambda})$ is a $d+1$-dimension vector, with ν in the first dimension, followed by the d-dimension vector $\vec{\lambda}$. As the dimension in the ratio objective is greater than one, Case 2 of Theorem 10.1 applies, and we obtain the following:

Lemma 10.2. *Given the product graph $G = (V, E)$ of n nodes and m edges, a rational $\nu \in \mathbb{Q}$, and a set of job sequences $\mathcal{J}$ admissible for safety, liveness, and limit average LTSs, determining whether $\mathcal{CR}_{\mathcal{J}}(\mathcal{A}) \leqslant \nu$ requires polynomial time.*

Again, since $0 \leqslant \mathcal{CR}_{\mathcal{J}}(\mathcal{A}) \leqslant 1$, the competitive ratio is determined by an adaptive binary search. However, this time $\mathcal{CR}_{\mathcal{J}}(\mathcal{A})$ is not guaranteed to be realized by a simple cycle (the witness path in G is not necessarily periodic, see Algorithm 35), and is only approximated within some

desired error threshold $\epsilon > 0$.

Adaptive Binary Search. We employ an *adaptive* binary search for the competitive ratio in the interval $[0, 1]$, as follows. The algorithm maintains an interval $[\ell, r]$ such that $\ell \leqslant \mathcal{CR}_{\mathcal{J}}(\mathcal{A}) \leqslant r$ at all times, and exploits the nature of the problem for refining the interval as follows: First, if the current objective $\nu \in [\ell, r]$ (typically, $\nu = (\ell + r)/2$) is satisfied in G, i.e., Lemma 10.1 answers "yes" and provides the current minimum cycle C as a witness, the value r is updated to the ratio ν' of the on-line and off-line rewards in C, which is typically less than ν. This allows to reduce the current interval for the next iteration from $[\ell, r]$ to $[\ell, \nu']$, with $\nu' \leqslant \nu$, rather than $[\ell, \nu]$ (as a simple binary search would do). Second, since $\mathcal{CR}_{\mathcal{J}}(\mathcal{A})$ corresponds to the ratio of rewards on a simple cycle in G, if the current objective $\nu \in [\ell, r]$ is not satisfied in G, the algorithm assumes that $\mathcal{CR}_{\mathcal{J}}(\mathcal{A}) = r$ (i.e, the competitive ratio equals the right endpoint of the current interval), and tries $\nu = r$ in the next iteration. Hence, as opposed to a naive binary search, the adaptive version has the advantages of (i) returning the exact value of $\mathcal{CR}_{\mathcal{J}}(\mathcal{A})$ (rather than an approximation), and (ii) being faster in practice.

Algorithm 36: AdaptiveBinarySearch

Input: Graph $G = (V, E)$ and weight functions $\mathsf{wt}_{\mathcal{A}},\ \mathsf{wt}_{\mathcal{C}}$

Output: $\min_{C \in G} \frac{\mathsf{wt}_{\mathcal{A}}(C)}{\mathsf{wt}_{\mathcal{C}}(C)}$

```
1  ℓ ← 0, r ← 1, ν ← (ℓ+r)/2
2  while True do
3      Solve G for objective Φ_ν and find min simple cycle C
4      ν1 ← wt_A(C), ν2 ← wt_C(C)
5      if ν = v1/v2 then
6          return ν
7      else
8          if ν > v1/v2 then
9              r ← ν1/ν2, ν ← (ℓ+r)/2
10         else
11             ℓ ← ν, r ← min(ν1/ν2, r), ν ← r
12         end
13     end
14 end
```

Remark 10.4. Lemmas 10.1 and 10.2 give polynomial upper bounds for the complexity of determining the competitive ratio of an online scheduling algorithm $\mathcal{A}$ given as a LTS $L_{\mathcal{A}}$. If, instead, $\mathcal{A}$ is given in some succinct form using a description which is polylogarithmic in the number of states (e.g., as a circuit [Galperin and Wigderson, 1983]), then the corresponding upper bounds become exponential in the size of the description of $\mathcal{A}$.

10.4.4 Optimized Reduction

In Section 10.4.3, we established a formal reduction from determining the competitive ratio of an on-line scheduling algorithm in a constrained adversarial environment to solving multiple objectives on graphs. In this section, we present several optimizations for this reduction that significantly reduce the size of the generated LTSs.

Clairvoyant LTS reduction. Recall the clairvoyant LTS $L_{\mathcal{C}}$ with reward function $r_{\mathcal{C}}$ from Section 10.3, which non-deterministically models a scheduler. For our optimization, we encode the off-line algorithm as a non-deterministic LTS $L'_{\mathcal{C}} = (S'_{\mathcal{C}}, s'_{\mathcal{C}}, \Sigma, \varnothing, \Delta'_{\mathcal{C}})$ with reward function $r'_{\mathcal{C}}$ that lacks the property of being a scheduler, as information about released and scheduled jobs is lost. However, it preserves the property that, given a job sequence σ, there exists a run $\rho^{\sigma}_{\mathcal{C}}$ in $L_{\mathcal{C}}$ iff there exists a run $\widehat{\rho}^{\sigma}_{\mathcal{C}}$ in $L'_{\mathcal{C}}$ with $V(\rho^{\sigma}_{\mathcal{A}}, k) = V(\widehat{\rho}^{\sigma}_{\mathcal{A}}, k)$ for all $k \in \mathbb{N}^{+}$. That is, there is a bisimulation between $L_{\mathcal{C}}$ and $L'_{\mathcal{C}}$ that preserves rewards.

Intuitively, the clairvoyant algorithm need not partially schedule a job, i.e., it will either discard it immediately, or schedule it to completion. Hence, in every release of a set of tasks T, $L'_{\mathcal{C}}$ non-deterministically chooses a subset $T' \subseteq T$ to be scheduled, as well as allocates the future slots for their execution. Once these slots are allocated, $L'_{\mathcal{C}}$ is not allowed to preempt those in favor of a subsequent job. For the reward, we use $r'_{\mathcal{C}} = \sum_{\tau_i \in T'} V_i$.

The state space $S'_{\mathcal{C}}$ of $L'_{\mathcal{C}}$ consists of binary strings of length $D_{\max}$. For a binary string $B \in S'_{\mathcal{C}}$, we have $B[i] = 1$ iff the i-th slot in the future is allocated to some released job, and $s'_{\mathcal{C}} = \vec{\mathbf{0}}$. Informally, the transition relation $\Delta'_{\mathcal{C}}$ is such that, given a current subset $T \subseteq \Sigma$ of newly released jobs, there exists a transition δ from B to B' only if B' can be obtained from B by non-deterministically choosing a subset $T' \subseteq T$, and for each task $\tau_i \in T'$ allocating non-deterministically C_i free slots in B.

By definition, $|S'_{\mathcal{C}}| \leqslant 2^{D_{\max}}$. In laxity-restricted tasksets, however, we can obtain an even tighter bound: Let $L_{\max} = \max_{\tau_i \in \mathcal{T}} (D_i - C_i)$ be the maximum laxity in $\mathcal{T}$, and $I : S'_{\mathcal{C}} \to \{\bot, 1, \dots, D_{\max} - 1\}^{L_{\max}+1}$ denote a function such that $I(B) = (i_1, \dots, i_{L_{\max}+1})$ are the indexes of the first $L_{\max} + 1$ zeros in B. That is, $i_j = k$ iff $B[k]$ is the j-th zero location in B, and $i_j = \bot$ if there are less than j free slots in B.

Claim 10.1. *The function I is bijective.*

Proof. Fix a tuple $(i_1, \dots, i_{L_{\max}+1})$ with $i_j \in \{\bot, 1, \dots, D_{\max} - 1\}$, and let $B \in S'_{\mathcal{C}}$ be any state such that $I(B) = (i_1, \dots, i_{L_{\max}+1})$. We consider two cases.

1. If $i_{L_{\max}+1} = \bot$, there are less than $L_{\max} + 1$ empty slots in B, all uniquely determined by $(i_1, \dots, i_k)$, for some $k \leqslant L_{\max}$.

2. If $i_{L_{\max}+1} \neq \bot$, then all $i_j \neq \bot$, and thus any job to the right of $i_{L_{\max}+1}$ would have been stalled for more than $L_{\max}$ positions. Hence, all slots to the right of $i_{L_{\max}+1}$ are free in B, and B is also unique.

Hence, $I(B)$ always uniquely determines B, as desired. □

For $x, k \in \mathbb{N}^+$, denote by $\mathsf{Perm}(x, k) = x \cdot (x-1) \dots (x-k+1)$ the number of k-permutations on a set of size x. Claim 10.1 immediately implies the following Lemma 10.3:

Lemma 10.3. *Let $\mathcal{T}$ be a taskset with maximum deadline $D_{\max}$, and $L_{\max} = \max_{\tau_i \in \mathcal{T}} (D_i - C_i)$ be the maximum laxity. Then, $|S'_{\mathcal{C}}| \leqslant \min(2^{D_{\max}}, \mathsf{Perm}(D_{\max}, L_{\max} + 1))$.*

Hence, for zero and small laxity environments [Baruah *et al.*, 1992], as they typically arise in high-speed network switches [Englert and Westermann, 2007] and in NoCs [Lu and Jantsch, 2007], $S'_{\mathcal{C}}$ has polynomial size in $D_{\max}$. This affect the parameter n in Lemmas 10.1 and 10.2.

Clairvoyant LTS generation. We now turn our attention to efficiently generating the clairvoyant LTS $L'_{\mathcal{C}}$ as described in the previous paragraph. There is non-determinism in two steps: Both in choosing the subset $T' \subseteq T$ of the currently released tasks for execution, and in allocating slots for executing all tasks in T'. Given a current state B and T, this non-determinism leads to several identical transitions δ to a state B'. We have developed a recursive algorithm called $\mathsf{ClairvoyantSuccessor}$ (Algorithm 37) that generates each such transition δ exactly once.

Algorithm 37: ClairvoyantSuccessor

Input: A set $T \subseteq \mathcal{T}$, state B, index $1 \leqslant k \leqslant D_{\max}$

Output: A set $\mathcal{B}$ of successor states of B

```
1  if T = ∅ then return {B};
2  τ ← arg min_{τ_i ∈ T} D_i, C ← execution time of τ
3  T' ← T \ {τ}
   // Case 1:  τ is not scheduled
4  𝓑 ← ClairvoyantSuccessor(T', B, k)
   // Case 2:  τ is scheduled
5  𝓕 ← set of free slots in B greater than k
6  foreach F ⊆ 𝓕 with |F| = C do
7  |  B' ← Allocate F in B
8  |  k' ← rightmost slot in F
9  |  𝓑' ← ClairvoyantSuccessor(T', B', k')
   |  // Keep only non-redundant states
10 |  foreach B'' ∈ 𝓑' do
11 |  |  if B''[1] = 1 and knapsack(B'', 𝒯) then
12 |  |  |  𝓑 ← 𝓑 ∪ {B''}
13 |  |  end
14 |  end
15 end
16 return 𝓑
```

The intuition behind ClairvoyantSuccessor is as follows: It is well-known that the earliest deadline first (EDF) policy is optimal for scheduling job sequences where every released task can be completed [Dertouzos, 1974]. By construction, given a job sequence σ_1, L'_C non-deterministically chooses a job sequence σ_2, such that for all ℓ, we have $\sigma_2^\ell \subseteq \sigma_1^\ell$, and all jobs in σ_2 are scheduled to completion by L'_C. Therefore, it suffices to consider a transition relation Δ'_C that allows at least all possible choices that admit a feasible EDF schedule on every possible σ_2, for any generated job sequence σ_1.

In more detail, ClairvoyantSuccessor is called with a current state B, a subset of released tasks T and an index k, and returns the set $\mathcal{B}$ of all possible successors of B that schedule a subset $T' \subseteq T$, where every job of T' is executed later than k slots in the future. This is done by

extracting from T the task τ with the earliest deadline, and proceeding as follows: The set $\mathcal{B}$ is obtained by constructing a state B' that considers all the possible ways to schedule τ to the right of k (including the possibility of not scheduling τ at all), and recursively finding all the ways to schedule $T \setminus \{\tau\}$ in B', to the right of the rightmost slot allocated for task τ.

Finally, we exploit the following two observations to further reduce the state space of L'_C. First, we note that as long as there are some unfinished jobs in the state of L'_C (i.e., at least one bit of B is one), the clairvoyant algorithm gains no benefit by not executing any job in the current slot. Hence, besides the zero state $\vec{\mathbf{0}}$, every state B must have $B[1] = 1$. In most cases, this restriction reduces the state space by at least 50%.

Second, observe that for every two scheduled jobs J and J', the clairvoyant scheduler will never have to preempt J for J' and vice versa. Given a state B, we call a contiguous segment of zeros in B which is surrounded by ones a *gap*. We call a gap between positions $[i_1, i_2]$ of B *admissible* if there exists a multiset X of tasks from $\mathcal{T}$ such that $\sum_{\tau_i \in X} C_i = i_2 - i_1 + 1$. Observe that if state B contains a gap which is not admissible, then the clairvoyant scheduler produces a schedule in which either

1. no job is scheduled in some round, while there is some already released job J which will be scheduled in the future, or
2. two jobs J and J' are such that each one preempts the other.

It is straightforward that in both cases, the clairvoyant scheduler can obtain the same utility by producing another schedule in which none of the above cases occur. Hence, a state can be safely discarded if it contains a non-admissible gap. This reduces to solving a knapsack problem [Karp, 1972], where the size of the knapsack is the length of the gap, and the set of items is the whole taskset $\mathcal{T}$ (with multiplicities). We note that the problem has to be solved on identical inputs a large number of times, and techniques such as *memoization* are employed to avoid multiple evaluations of the same input.

These two improvements were found to reduce the state space by a factor up to 90% in all examined cases (see Section 10.4.5 and Table 10.5). In fact, despite the non-determinism, in all reported cases the generation of L_C was done in less than a second.

On-line LTS reduction. Typically, simple on-line scheduling algorithms do "lazy dropping"

of unsuccessful jobs, where such a job is dropped only when its deadline passes. An obvious improvement for reducing the size of the state space of the LTS is to implement some early dropping: Store only those jobs that could be scheduled, at least partially, under *some* sequence of future task releases. We do so by first creating the LTS naively, and then iterating through its states. For each state s and job $J_{i,j}$ in s with relative deadline D_i, we perform a *depth-limited search* originating in s for D_i steps, looking for a state s' reached by a transition that schedules $J_{i,j}$. If no such state is found, we merge state s to s'', where s'' is identical to s without job $J_{i,j}$.

10.4.5 Experimental Results

We have implemented a prototype of our automated competitive ratio analysis framework, and applied it in a comparative case study.

Our implementation has been done in Python 2.7 and C, and uses the lp_solve [Berkelaar *et al.*] package for linear programming solutions. All experiments were run on a standard desktop computer with a 3.2GHz CPU and 4GB of RAM running Debian Linux.

In our case study, five well-known scheduling policies, namely, EDF (Earliest Deadline First), LLF (Least Laxity First), SRT (Shortest Remaining Time), SP (Static Priorities), and FIFO (First-in First-out), as well as some more elaborate algorithms that provide non-trivial performance guarantees, in particular, DSTAR [Baruah *et al.*, 1991], TD1 [Baruah *et al.*, 1992], and DOVER [Koren and Shasha, 1995], were analyzed under a variety of tasksets (with and without additional constraints on the adversary). In addition, for TD1, we constructed a series of task sets according to the recurrence relation given in [Baruah *et al.*, 1992], which confirms its worst-case competitive ratio of $1/4$. All our on-line scheduler implementations use the same tie-breaking rules, namely, (i) favor lower-indexed tasks (in $\mathcal{T}$) over higher-indexed ones, and (ii) favor smaller deadlines over larger ones (and (i) has higher precedence over (ii)).

Varying tasksets without constraints. The algorithm DOVER was proved in [Koren and Shasha, 1995] to have optimal competitive factor, i.e., optimal competitive ratio under the worst-case taskset. However, our experiments reveal that this performance guarantee is not universal, in the sense that DOVER is outperformed by other schedulers for *specific* tasksets. Interestingly, this observation applies to all on-line algorithms examined: As shown in Fig. 10.6,

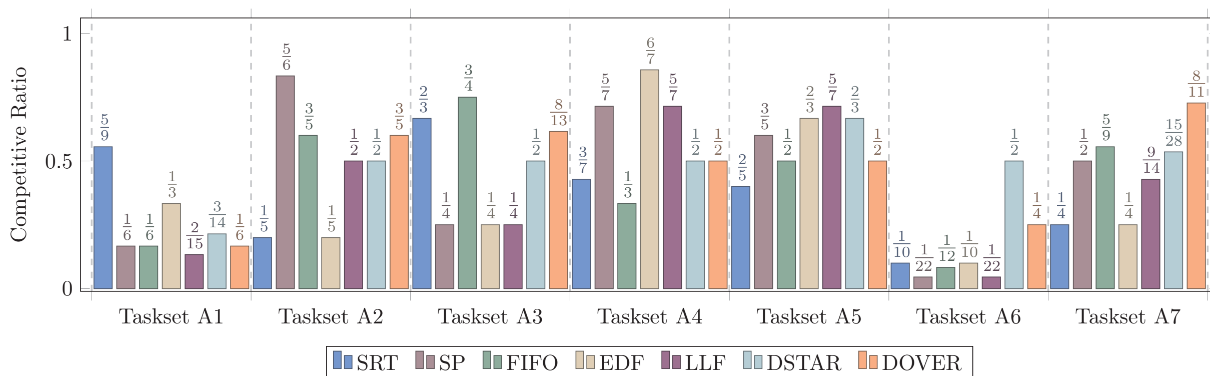


Figure 10.6: The competitive ratio of the examined algorithms in various tasksets under no constraints. Every examined algorithm is optimal in some taskset, among all others.

	A1 ($k = 6$)				A2 ($k = 5$)		A3 ($k = 4$)			A4 ($k = 3$)			A5 ($k = 2$)			A6 ($k = 4$)			A7 ($k = 5$)		
	τ_1	τ_2	τ_3	τ_4	τ_1	τ_2	τ_1	τ_2	τ_3	τ_1	τ_2	τ_3	τ_1	τ_2	τ_3	τ_1	τ_2	τ_3	τ_1	τ_2	τ_3
C_i	1	4	1	3	2	2	2	1	1	1	2	1	2	1	1	2	6	1	1	2	1
D_i	2	6	3	4	3	2	2	5	5	2	3	6	3	1	3	2	6	1	5	2	1
V_i	3	2	3	3	5	1	1	2	2	3	2	1	9	6	3	1	10	2	5	4	1

Table 10.1: The tasksets used to generate Fig. 10.6.

even without constraints on the adversary, there are tasksets in which every chosen scheduling algorithm outperforms all others, by achieving the highest competitive ratio for the particular taskset. This sensitivity of the optimally performing on-line algorithm on the given taskset makes our automated analysis framework a very interesting tool for the application designer.

Table 10.1 lists the tasksets A1-A7 used for Fig. 10.6. The task indices, hence their order in Table 10.1, reflect their static priorities (with τ_1 having highest priority); they are used by the SP scheduler, as well as for tie breaking by other schedulers. Along with each taskset, its importance ratio $k = \frac{\max_{\tau_i \in \mathcal{T}}\{V_i/C_i\}}{\min_{\tau_i \in \mathcal{T}}\{V_i/C_i\}}$ is shown [Baruah *et al.*, 1992].

Fixed taskset with varying constraints. We also analyzed fixed tasksets under various constraints (such as sporadicity or workload restrictions) for admissible job sequences. Fig. 10.7 shows some experimental results for workload safety constraints, which again reveal that, depending on particular workload constraints, we can have different optimal schedulers. The same was observed for limit-average constraints: As Table 10.2 shows, the optimal scheduler can vary

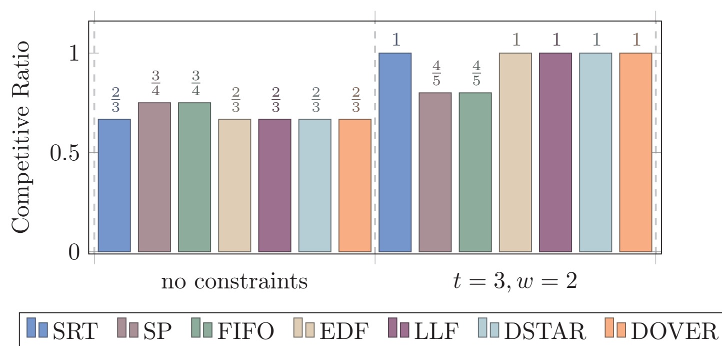


Figure 10.7: Restricting the absolute workload generated by the adversary typically increases the competitive ratio, and can vary the optimal scheduler. On the left, the performance of each scheduler is evaluated without restrictions: FIFO, SP behave best. When restricting the adversary to at most 2 units of workload in the last 3 rounds, FIFO and SP become suboptimal, and are outperformed by other schedulers.

	1.5	1	0.8	0.6	0.4	0.3	0.1	0.078	0.05
FIFO	✓	✓	✓	✓	✓				✓
SP	✓						✓		✓
SRT	✓				✓	✓	✓	✓	✓

Table 10.2: Columns show the mean workload restriction. The check-marks indicate that the corresponding scheduler is optimal for that mean workload restriction, among the six schedulers we examined. We see that the optimal scheduler can vary as the restrictions are tighter, and in a non-monotonic way. LLF, EDF, DSTAR and DOVER were not optimal in any case and hence not mentioned.

highly and non-monotonically with stronger limit-average workload restrictions. The tasksets for both experiments are shown in Table 10.3.

Competitive Ratio of TD1. We also analyzed the performance of the on-line scheduler TD1 for zero laxity tasksets with uniform value-density $k = 1$ (i.e., $C_i = D_i = V_i$ for each task τ_i). Following [Baruah *et al.*, 1992], we constructed a series of tasksets parametrized by some positive real $\eta < 4$, which guarantee that the competitive ratio of every on-line scheduler is upper bounded by $\frac{1}{\eta}$. Given η, each taskset consists of tasks τ_i such that C_i is given by the following recurrence, as long as $C_{i+1} > C_i$.

	τ_1	τ_2	τ_3
C_i	1	1	1
D_i	1	2	1
V_i	3	3	1

	τ_1	τ_2	τ_3
C_i	2	5	5
D_i	7	5	6
V_i	3	2	1

Table 10.3: Taskset of Fig. 10.7 (left) and Table 10.2 (right).

Taskset	η	Taskset	**Comp. Ratio**
C1	2	$\{1, 1\}$	1
C2	3	$\{1, 2, 3\}$	$\frac{1}{2}$
C3	3.1	$\{1, 3, 7, 13, 19\}$	$\frac{7}{25}$
C4	3.2	$\{1, 3, 7, 13, 20, 23\}$	$\frac{1}{4}$
C5	3.3	$\{1, 3, 7, 14, 24, 33\}$	$\frac{1}{4}$
C6	3.4	$\{1, 3, 7, 14, 24, 34\}$	$\frac{1}{4}$

Table 10.4: Competitive ratio of TD1.

$$(i)\ C_0 = 1 \qquad (ii)\ C_{i+1} = \eta \cdot C_i - \sum_{j=0}^{i} C_j$$

In [Baruah *et al.*, 1992], TD1 was shown to have competitive factor $\frac{1}{4}$, and hence a competitive ratio that approaches $\frac{1}{4}$ from above, as $\eta \to 4$ in the above series of tasksets. Table 10.4 shows the competitive ratio of TD1 in our constructed series of tasksets. Each taskset is represented as a set $\{C_i : 1 \leqslant i \leqslant n\}$, where each C_i is given by the above recurrence, rounded up to the next integer. We indeed see that the competitive ratio drops until it stabilizes to $\frac{1}{4}$. Note that, thanks to our optimizations, the zero-laxity restriction allowed us to process tasksets where $D_{\max}$ is much higher than for the tasksets reported in Table 10.5: The results of Table 10.4 were produced in less than a minute overall.

Running Times. Table 10.5 summarizes some key parameters of our various tasksets, and gives some statistical data on the observed running times in our respective experiments. Even though the considered tasksets are small, the very short running times of our prototype implementation reveal the principal feasibility of our approach. We believe that further application-specific

Taskset	N	$D_{\max}$	Size (nodes)		Time (s)	
			Clairv.	Product	Mean	Max
B01	2	7	19	823	0.04	0.05
B02	2	8	26	1997	0.39	0.58
B03	2	9	34	4918	10.02	15.21
B04	3	7	19	1064	0.14	0.40
B05	3	8	26	1653	0.66	2.05
B06	3	9	34	7705	51.04	136.62
B07	4	7	19	1711	2.13	6.34
B08	4	8	26	3707	13.88	34.12
B09	4	9	44	10040	131.83	311.94
B10	5	7	19	2195	5.73	16.42
B11	5	8	32	9105	142.55	364.92
B12	5	9	44	16817	558.04	1342.59

Table 10.5: Scalability of our approach for tasksets of various sizes N and $D_{\max}$. For each taskset, the size of the state space of the clairvoyant scheduler is shown, along with the mean size of the product LTS, and the mean and maximum time to solve one instance of the corresponding ratio objective.

optimizations, augmented by abstraction and symmetry reduction techniques, will allow to scale to larger applications.

10.5 Competitive Synthesis of On-line Scheduling Algorithms

In this section, we show how the powerful framework of *graph games* [Martin, 1975; Shapley, 1953] can be utilized for the *synthesis* of optimal real-time scheduling algorithms. As opposed to the the analysis problem considered in the previous sections (which can be viewed as a 1-player game of the adversary against a given scheduling algorithm), we now have to consider a two-player game between the (sought) optimal on-line algorithm (Player 1) and the adversary (Player 2). Our presentation is organized as follows:

- In Section 10.5.1, we introduce a suitable two-player partial-information game with mean-payoff and ratio objectives. Player 1 will represent the online algorithm, whereas Player 2 will represent both the adversary (which chooses the job sequence) and the clairvoyant algorithm (which knows the job sequence in advance). We use a partial-information setting to model that Player 1 is oblivious to the scheduling choices of Player 2, but Player 2 knows the scheduling choices of Player 1 for deciding which future jobs to release. The mean-payoff and ratio objectives model directly the worst-case utility and competitive ratio problems, respectively.

- In Section 10.5.2, we establish that the relevant decision problems for our game are NP-complete in the size of the game graph.

- In Section 10.5.3, we study the decision problems relevant for two particular synthesis questions: In *synthesis for worst-case average utility*, the goal is to automatically construct an on-line scheduling algorithm with the largest possible worst-case average utility for a given taskset. In *competitive synthesis*, we construct an on-line scheduling algorithm with the largest possible competitive ratio for the given taskset. The complexity results for our graph game reveal that the former problem is in NP∩coNP, whereas the latter is in NP. These complexities are wrt the size of the constructed algorithm, represented explicitly as a labeled transition system. As a function of the input taskset $\mathcal{T}$ given in binary, all polynomial upper bounds become exponential upper bounds in the worst case.

10.5.1 Partial-Information Mean-Payoff and Ratio Games

We first introduce a two-player partial-information game on graphs with mean-payoff and ratio objectives.

Notation on Graph Games. A *partial-observation game* (or simply a *game*) is a tuple $\mathscr{G} = \langle S, \Sigma_1, \Sigma_2, \delta, \mathsf{O}_S, \mathsf{O}_\Sigma \rangle$ with the following components:

State space: The set S is a finite set of states.

Actions: Σ_i $(i = 1, 2)$ is a finite set of actions for Player i.

Transition function: Given the current state $s \in S$, an action $\alpha_1 \in \Sigma_1$ for Player 1, and an action $\alpha_2 \in \Sigma_2$ for Player 2, the transition function $\delta : S \times \Sigma_1 \times \Sigma_2 \to S$ gives the next (or successor) state $s' = \delta(s, \alpha_1, \alpha_2)$. A shorter form to denote a transition is to write the tuple $(s, \alpha_2, \alpha_1, s')$; note that α_2 is listed before α_1 to stress that fact that Player 2 chooses its action before Player 1.

Observations: The set $\mathsf{O}_S \subseteq 2^S$ is a finite set of observations for Player 1 that partition the state space S. This partition uniquely defines a function $\mathsf{obs}_S : S \to \mathsf{O}_S$, which maps each state to its observation $\mathsf{obs}_S(s)$ in a way that ensures $s \in \mathsf{obs}_S(s)$ for all $s \in S$. In other words, the observation partitions the state space according to equivalence classes. Similarly, O_Σ is a finite set of observations for Player 1 that partitions the action set Σ_2, and analogously defines the function obs_Σ. Intuitively, Player 1 will have partial observation, and can only obtain the current observation of the state (not the precise state but only the equivalence class the state belongs to) and current observation of the action of Player 2 (but not the precise action of Player 2) to make her choice of action.

Plays. In a game, in each turn, first Player 2 chooses an action, then Player 1 chooses an action, and given the current state and the joint actions, we obtain the next state according to the transition function δ.

A *play* in $\mathscr{G}$ is an infinite sequence of states and actions $\mathscr{P} = s^1, \alpha_2^1, \alpha_1^1,\ s^2, \alpha_2^2, \alpha_1^2,\ s^3, \alpha_2^3, \alpha_1^3,\ s^4 \ldots$ such that, for all $j \geqslant 1$, we have $\delta(s^j, \alpha_1^j, \alpha_2^j) = s^{j+1}$. The *prefix up to* s^n of the play $\mathscr{P}$ is denoted by $\mathscr{P}(n)$ and corresponds to the starting state of the n-th turn. The set of plays in $\mathscr{G}$ is denoted by $\mathscr{P}^\infty$, and the set of corresponding finite prefixes is denoted by $\mathsf{Prefs}(\mathscr{P}^\infty)$.

Strategies. A *strategy* for a player is a recipe that specifies how to extend finite prefixes of plays. We will consider *memoryless* deterministic strategies for Player 1 (where its next action depends only on the current state, but not on the entire history) and general history-dependent deterministic strategies for Player 2. A strategy for Player 1 is a function $\pi : \mathsf{O}_S \times \mathsf{O}_\Sigma \to \Sigma_1$ that, given the current observation of the state and the current observation on the action of Player 2, selects the next action. A strategy for Player 2 is a function $\sigma : \mathsf{Prefs}(\mathscr{P}^\infty) \to \Sigma_2$ that, given the current prefix of the play, chooses an action. Observe that the strategies for Player 1 are both observation-based and memoryless; i.e., depend only on the current observations (rather than the whole history), whereas the strategies for Player 2 depend on the history. A memoryless

strategy for Player 2 only depends on the last state of a prefix. We denote by $\Pi_{\mathscr{G}}^{M}$, $\Sigma_{\mathscr{G}}$, $\Sigma_{\mathscr{G}}^{M}$ the set of all observation-based memoryless Player 1 strategies, the set of all Player 2 strategies, and the set of all memoryless Player 2 strategies, respectively. In sequel, when we write "strategy for Player 1", we consider only observation-based memoryless strategies. Given a strategy π and a strategy σ for Player 1 and Player 2, and an initial state s^1, we obtain a unique play $\mathscr{P}(s^1, \pi, \sigma) = s^1, \alpha_2^1, \alpha_1^1, s^2, \alpha_2^2, \alpha_1^2, s^3, \ldots$ such that, for all $n \geqslant 1$, we have $\sigma(\mathscr{P}(n)) = \alpha_2^n$ and $\pi(\mathsf{obs}_S(s^n), \mathsf{obs}_\Sigma(\alpha_2^n)) = \alpha_1^n$.

Objectives. Recall that, for the graphs with multiple objectives from Section 2.3.1, an objective is a set of paths. Here we extend this notion to games: An objective of a game $\mathscr{G}$ is a set of plays that satisfy some desired properties. For the sake of completeness, we present here the relevant definitions for mean payoff and ratio objectives with 1-dimensional weight functions.

For mean-payoff objectives, we will consider a reward function $\mathsf{wt} : S \times \Sigma_1 \times \Sigma_2 \times S \to \mathbb{Z}$ that maps every transition to an integer reward. The reward function naturally extends to plays: For $k \geqslant 1$, the sum of the rewards in the prefix $\mathscr{P}(k+1)$ is defined as $\mathsf{wt}(\mathscr{P}, k) = \sum_{i=1}^{k} \mathsf{wt}(s^i, \alpha_2^i, \alpha_1^i, s'^i)$. The mean-payoff of a play $\mathscr{P}$ is then

$$\mathsf{MP}(\mathsf{wt}, \mathscr{P}) = \liminf_{k \to \infty} \frac{1}{k} \cdot \mathsf{wt}(\mathscr{P}, k).$$

In the case of ratio objectives, we will consider two reward functions $\mathsf{wt}_1 : S \times \Sigma_1 \times \Sigma_2 \times S \to \mathbb{N}$ and $\mathsf{wt}_2 : S \times \Sigma_1 \times \Sigma_2 \times S \to \mathbb{N}$ that map every transition to a non-negative valued reward. Using the same extension of reward functions to plays as before, the ratio of a play $\mathscr{P}$ is defined as:

$$\mathsf{Ratio}(\mathsf{wt}_1, \mathsf{wt}_2, \mathscr{P}) = \liminf_{k \to \infty} \frac{\vec{\mathbf{1}} + \mathsf{wt}_1(\mathscr{P}, k)}{\vec{\mathbf{1}} + \mathsf{wt}_2(\mathscr{P}, k)}.$$

Decision problems. Analogous to Section 2.3.1, we define the relevant decision problems on games. Formally, given a game $\mathscr{G}$, a starting state s^1, reward functions $\mathsf{wt}, \mathsf{wt}_1, \mathsf{wt}_2$ and a threshold $\nu \in \mathbb{N}$, the decision problem for the mean payoff objective is to decide whether

$$\sup_{\pi \in \Pi_{\mathscr{G}}^M} \inf_{\sigma \in \Sigma_{\mathscr{G}}} \mathsf{MP}(\mathsf{wt}, \mathscr{P}(s^1, \pi, \sigma)) \geqslant \nu.$$

Similarly, the decision problem for the ratio objective is to decide whether

$$\sup_{\pi \in \Pi_{\mathscr{G}}^M} \inf_{\sigma \in \Sigma_{\mathscr{G}}} \mathsf{Ratio}(\mathsf{wt}_1, \mathsf{wt}_2, \mathscr{P}(s^1, \pi, \sigma)) \geqslant \nu.$$

Remark 10.5. Note that the decision problems of the graph game problem are defined over the $\sup_{\pi \in \Pi^M_{\mathscr{G}}}$, taking all possible memoryless strategies into account. This corresponds to all possible on-line scheduling strategies, whereas the multi-graph problem arising in the competitive analysis problem considered in the previous sections explicitly used the fixed deterministic strategy for the on-line scheduler only.

Perfect-information Games. *Games of complete-observation* (or perfect-information games) are a special case of partial-observation games where $\mathsf{O}_S = \{\{s\} \mid s \in S\}$ and $\mathsf{O}_\Sigma = \{\{\alpha_2\} \mid \alpha_2 \in \Sigma_2\}$, i.e., every individual state and action is fully visible to Player 1, and thus she has perfect information. For perfect-information games, for the sake of simplicity, we will omit the corresponding observation sets from the description of the game. The following theorem for perfect-information games with mean-payoff objectives follows from the results of [Ehrenfeucht and Mycielski, 1979; Zwick and Paterson, 1996; Brim *et al.*, 2011; Karp, 1978].

Theorem 10.2 (Complexity of perfect-information mean-payoff games [Ehrenfeucht and Mycielski, 1979; Zwick and Paterson, 1996; Brim *et al.*, 2011; Karp, 1978]). *The following assertions hold for perfect-information games with initial state s^1 and reward function $\mathsf{wt} : S \times \Sigma_1 \times \Sigma_2 \times S \to \mathbb{Z}$:*

1. (Determinacy). *We have*

$$\begin{aligned} &\sup_{\pi \in \Pi^M_{\mathscr{G}}} \inf_{\sigma \in \Sigma_{\mathscr{G}}} \mathsf{MP}(\mathsf{wt}, \mathscr{P}(s^1, \pi, \sigma)) \\ &= \inf_{\sigma \in \Sigma_{\mathscr{G}}} \sup_{\pi \in \Pi^M_{\mathscr{G}}} \mathsf{MP}(, \mathsf{wt}, \mathscr{P}(s^1, \pi, \sigma)) \\ &= \inf_{\sigma \in \Sigma^M_{\mathscr{G}}} \sup_{\pi \in \Pi^M_{\mathscr{G}}} \mathsf{MP}(\mathsf{wt}, \mathscr{P}(s^1, \pi, \sigma)). \end{aligned}$$

2. *Whether $\sup_{\pi \in \Pi^M_{\mathscr{G}}} \inf_{\sigma \in \Sigma_{\mathscr{G}}} \mathsf{MP}(\mathsf{wt}, \mathscr{P}(s^1, \pi, \sigma)) \geqslant \nu$ can be decided in* NP $\cap$ coNP*, for a rational threshold ν.*

3. *The computation of the optimal value $v^* = \sup_{\pi \in \Pi^M_{\mathscr{G}}} \inf_{\sigma \in \Sigma_{\mathscr{G}}} \mathsf{MP}(\mathsf{wt}, \mathscr{P}(s^1, \pi, \sigma))$ and an optimal memoryless strategy $\pi^* \in \Pi^M_{\mathscr{G}}$ such that $v^* = \inf_{\sigma \in \Sigma_{\mathscr{G}}} \mathsf{MP}(\mathsf{wt}, \mathscr{P}(s^1, \pi^*, \sigma))$ can be done in time $O(n \cdot m \cdot W)$, where n is the number of states, m is the number of transitions, and W is the maximum value of all the rewards (i.e., the algorithm runs in pseudo-polynomial time, and if the maximum value W of rewards is polynomial in the size of the game, then the algorithm is polynomial).*

Sketch of the Algorithm. The complexity of Item 3 of Theorem 10.2 is obtained in [Brim *et al.*, 2011]. Here we outline a simple algorithm for solving the same problem in time $O(n^4 \cdot m \cdot \log(n/m) \cdot W)$, as found in [Zwick and Paterson, 1996]. The algorithm operates in two steps. First, we compute for every node $u \in S$ the maximum mean payoff $v(u)$ that Player 1 can ensure in any play that starts from u. This is achieved by the standard value-iteration procedure executed for $\Theta(n^2 \cdot W)$ iterations. Hence, the time required for this step is $O(n^2 \cdot m \cdot W)$. Note that at this point $v(s^1)$ gives the mean payoff achieved by an optimal strategy $\pi^* \in \Pi_{\mathscr{G}}^M$, but not the strategy itself. Since an optimal *memoryless* strategy is guaranteed to exist, this strategy can be computed by a binary search on the actions of Player 1. Given a node $u \in S$, we denote by $\Sigma_1(u)$ the set of actions available to Player 1 on u. In the second step, we iteratively pick a node $u \in S$ which has more than one available actions for Player 1, and a set $X \subset \Sigma_1(u)$ which contains half of the actions of Player 1 on u. We let $\mathscr{G}'$ be the modified game where the actions for Player 1 on node u is the set X, and recompute the value $v'(u)$ in $\mathscr{G}'$. If $v'(u) = v(u)$, we repeat the process on $\mathscr{G}'$. Otherwise, we construct a new game $\mathscr{G}''$ which is identical to $\mathscr{G}$, but such that the available actions for Player 1 on node u is the set $\Sigma_1(u) \setminus X$. We repeat the process on $\mathscr{G}''$.

10.5.2 Complexity Results

In this section, we establish the complexity of the decision problems arising in partial-observation games with mean-payoff and ratio objectives. In particular, we will show that for partial-observation games with memoryless strategies for Player 1 all the decision problems are NP-complete.

Transformation. We start with a simple transformation that will allow us to technically simplify our proof. In our definition of games, every action was available for the players in every state for simplicity. We will now consider restricted games where, in certain states, some actions are not allowed for a player. The transformation of such restricted games to games where all actions are allowed is as follows: We add two absorbing dummy states (with only a self-loop), one for Player 1 and the other for Player 2, and assign rewards in a way such that the objectives are violated for the respective player. For example, for mean-payoff objectives with threshold $\nu > 0$, we assign reward 0 for the only out-going (self-loop) transition of the Player 1 dummy state, and a reward strictly greater than ν for the self-loop of the Player 2 dummy state; in the

case of ratio-objectives we assign the reward pairs similarly. Given a state s, if Player 1 plays an action that is not allowed at s, we go to the dummy Player 1 state; and if Player 2 plays an action that is not allowed, we go to the Player 2 dummy state. Obviously, this is a simple linear time transformation. Hence, for technical convenience, we can assume in the sequel that different states have different sets of available actions for the players. We first start with the hardness result.

Lemma 10.4. *The decision problems for partial-observation games with mean-payoff objectives and ratio objectives, i.e., whether* $\sup_{\pi\in\Pi_{\mathscr{G}}^{M}} \inf_{\sigma\in\Sigma_{\mathscr{G}}} \mathsf{MP}(\mathsf{wt}, \mathscr{P}(s^1,\pi,\sigma)) \geqslant \nu$ *(respectively* $\sup_{\pi\in\Pi_{\mathscr{G}}^{M}} \inf_{\sigma\in\Sigma_{\mathscr{G}}} \mathsf{Ratio}(\mathsf{wt}_1, \mathsf{wt}_2, \mathscr{P}(s^1,\pi,\sigma)) \geqslant \nu$*), are* NP*-hard in the strong sense.*

Proof. We present a reduction from the 3-SAT problem, which is NP-hard in the strong sense [Papadimitriou, 1993]. Let Ψ be a 3-SAT formula over n variables $x_1, x_2, \ldots, x_n$ in conjunctive normal form, with m clauses $\mathfrak{c}_1, \mathfrak{c}_2, \ldots, \mathfrak{c}_m$ consisting of a disjunction of 3 literals (a variable x_k or its negation $\overline{x}_k$) each. We will construct a game graph $\mathscr{G}_\Psi$ as follows:

State space: $S = \{s_{\mathsf{init}}\} \cup \{s_{i,j} \mid 1 \leqslant i \leqslant m, 1 \leqslant j \leqslant 3\} \cup \{\mathsf{dead}\}$; i.e., there is an initial state s_{init}, a dead state dead, and there is a state $s_{i,j}$ for every clause $\mathfrak{c}_i$ and literal j of $\mathfrak{c}_i$.

Actions: The set of actions applicable for Player 1 is $\{\mathsf{true}, \mathsf{false}, \bot\}$, the possible actions for Player 2 are $\{1, 2, \ldots, m\} \cup \{\bot\}$.

Transitions: In the initial state s_{init}, Player 1 has only one action $\bot$ available, Player 2 has actions $\{1, 2, \ldots, m\}$ available, and given action $1 \leqslant i \leqslant m$, the next state is $s_{i,1}$. In all other states, Player 2 has only one action $\bot$ available. In states $s_{i,j}$, Player 1 has two actions available, namely, true and false. The transitions are as follows:

- If the action of Player 1 is true in $s_{i,j}$, then (i) if the j-th literal in $\mathfrak{c}_i$ is x_k, then we have a transition back to the initial state; and (ii) if the j-th literal in $\mathfrak{c}_i$ is $\overline{x}_k$ (negation of x_k), then we have a transition to $s_{i,j+1}$ if $j \in \{1, 2\}$, and if $j = 3$, we have a transition to dead.

- If the action of Player 1 is false in $s_{i,j}$, then (i) if the j-th literal in $\mathfrak{c}_i$ is $\overline{x}_k$ (negation of x_k), then we have a transition back to the initial state; and (ii) if the j-th literal in $\mathfrak{c}_i$ is x_k, then we have a transition to $s_{i,j+1}$ if $j \in \{1, 2\}$, and if $j = 3$, we have a transition to dead.

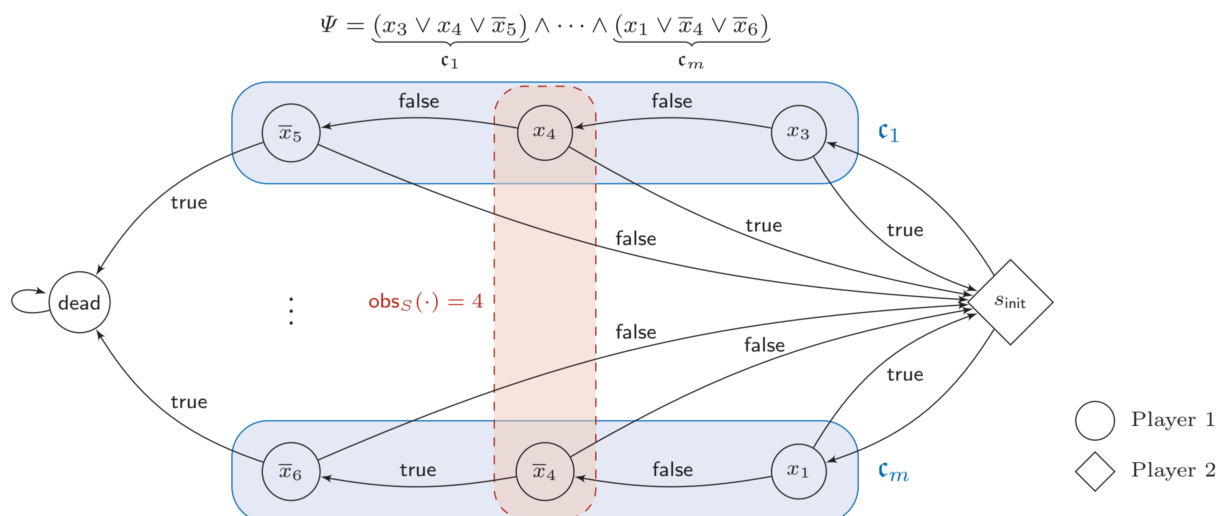


Figure 10.8: Illustration of the construction of a game from a 3-SAT formula.

In state dead both players have only one available action $\perp$, and dead is a state with only a self-loop (transition only to itself).

Observations: First, Player 1 does not observe the actions of Player 2 (i.e., Player 1 does not know which action is played by Player 2). The observation mapping for the state space for Player 1 is as follows: The set of observations is $\{0, 1, \ldots, n\}$ and we have $\mathsf{obs}_S(s_{\mathsf{init}}) = \mathsf{obs}_S(\mathsf{dead}) = 0$ and $\mathsf{obs}_S(s_{i,j}) = k$ if the j-th variable of $\mathfrak{c}_i$ is either x_k or its negation $\overline{x}_k$, i.e., the observation for Player 1 corresponds to the variables.

A pictorial description is shown in Fig. 10.8. The intuition for the above construction is as follows: Player 2 chooses a clause from the initial state s_{init}, and an observation-based memoryless strategy for Player 1 corresponds to a non-conflicting assignment to the variables. Note that Player 1 strategies are observation-based memoryless; hence, for every observation (i.e., a variable), it chooses a unique action (i.e., an assignment) and thus non-conflicting assignments are ensured. We consider $\mathcal{G}_\Psi$ with reward functions $\mathsf{wt}, \mathsf{wt}_1, \mathsf{wt}_2$ as follows: wt_2 assigns reward 1 to all transitions; wt and wt_1 assigns reward 1 to all transitions other than the self-loop at state dead, which is assigned reward 0. We ask the decision questions with $\nu = 1$. Observe that the answer to the decision problems for both mean-payoff and ratio objectives is "Yes" iff the state dead can be avoided by Player 1 (because if dead is reached, then the game stays in dead forever, violating both the mean-payoff as well as the ratio objective). We now present the two directions of the proof.

Satisfiable implies dead **is not reached.** We show that if Ψ is satisfiable, then Player 1 has an observation-based memoryless strategy π^* to ensure that dead is never reached. Consider a satisfying assignment A for Ψ, then the strategy π^* for Player 1 is as follows: Given an observation k, if A assigns true to variable x_k, then the strategy π^* chooses action true for observation k, otherwise it chooses action false. Since the assignment A satisfies all clauses, it follows that for every $1 \leqslant i \leqslant m$, there exists $s_{i,j}$ such that the strategy π^* for Player 1 ensures that the transition to s_{init} is chosen. Hence the state dead is never reached, and both the mean-payoff and ratio objectives are satisfied.

If dead **is not reached, then** Ψ **is satisfiable.** Consider an observation-based memoryless strategy π^* for Player 1 that ensures that dead is never reached. From the strategy π^* we obtain an assignment A as follows: if for observation k, the strategy π^* chooses true, then the assignment A chooses true for variable x_k, otherwise it chooses false. Since π^* ensures that dead is not reached, it means for every $1 \leqslant i \leqslant m$, that there exists $s_{i,j}$ such that the transition to s_{init} is chosen (which ensures that $\mathfrak{c}_i$ is satisfied by A). Thus since π^* ensures dead is not reached, the assignment A is a satisfying assignment for Ψ.

Thus, it follows that the answers to the decision problems are "Yes" iff Ψ is satisfiable, and this establishes the NP-hardness result. $\square$

The NP **upper bounds.** We now present the NP upper bounds for our decision problems. Recall that according to our definitions of strategies, the polynomial witness for the decision problem is a memoryless strategy (i.e., if the answer to the decision problem is "Yes", then there is a witness memoryless strategy π for Player 1). Such a strategy π can be guessed in polynomial time. Once the memoryless strategy is guessed and fixed, we need to show that there is a polynomial-time verification procedure:

Mean-payoff objectives: Once the memoryless strategy for Player 1 is fixed, the game problem reduces to a 1-player game where there is only Player 2. The verification problem hence reduces to the path problem in directed graphs analyzed and shown to be solvable in polynomial time by Theorem 10.1 in Section 10.4.1.

Ratio objectives: Again, once the memoryless strategy for Player 1 is fixed, the game problem reduces to a decision problem on directed graphs. The same reduction from ratio objectives

to mean-payoff objectives introduced in Section 2.3.1 can be applied. Theorem 10.1 hence gives a polynomial-time verification algorithm for our ratio objectives.

We summarize the result in the following theorem.

Theorem 10.3. *The decision problems for partial-observation games with mean-payoff objectives and ratio objectives, i.e., whether* $\sup_{\pi\in\Pi_{\mathscr{G}}^{M}} \inf_{\sigma\in\Sigma_{\mathscr{G}}} \mathsf{MP}(\mathsf{wt}, \mathscr{P}(s^1,\pi,\sigma)) \geqslant \nu$ *respectively* $\sup_{\pi\in\Pi_{\mathscr{G}}^{M}} \inf_{\sigma\in\Sigma_{\mathscr{G}}} \mathsf{Ratio}(\mathsf{wt}_1, \mathsf{wt}_2, \mathscr{P}(s^1,\pi,\sigma)) \geqslant \nu$*, are* NP*-complete.*

Remark 10.6. The NP-completeness of Theorem 10.3 also holds with the following extensions on objectives:

1. The reward functions wt, wt_1, wt_2 map to d-dimensional vectors of rewards, and the decision problems are with respect to a threshold vector $\vec{\nu}$.

2. Player 2 must also satisfy a conjunction of $\mathsf{Safe}(X)$ and $\mathsf{Live}(Y)$ objectives (see Section 2.3.1).

The result holds, as the NP-hardness follows from the proof of Theorem 10.3 by taking $d = 1$, $X = \varnothing$ and $Y = S$. The NP-membership follows similarly to that used in the proof of Theorem 10.3, by guessing a memoryless strategy for Player 1. The problem reduces to satisfying a conjunction of objectives in a multi-graph here, and Item 2 of Theorem 10.1 provides the required polynomial time bound.

10.5.3 Reduction of Competitive Synthesis to a Graph Game

We now turn our attention to competitive synthesis problems in the real-time scheduling context. More specifically, given a taskset $\mathcal{T}$, we consider two particular synthesis questions:

1. In *synthesis for the worst-case average utility*, the goal is to construct an on-line scheduling algorithm that has the largest worst-case average utility possible. Recall the notation $V(\rho^{\sigma}_{\mathcal{A}}, k)$ for the cumulative utility in the first k time slots of an on-line scheduling algorithm $\mathcal{A}$ with schedule $\rho^{\sigma}_{\mathcal{A}}$ under the released job sequence σ. Formally, the task is to construct an on-line scheduling algorithm $\mathcal{A}$ such that, for any online scheduling algorithm $\mathcal{A}'$,

$$\inf_{\sigma\in\mathcal{J}} \liminf_{k\to\infty} \frac{1}{k} V(\rho^{\sigma}_{\mathcal{A}}, k) \geqslant \inf_{\sigma\in\mathcal{J}} \liminf_{k\to\infty} \frac{1}{k} V(\rho^{\sigma}_{\mathcal{A}'}, k),$$

where $\mathcal{J}$ is the set of admissible job sequences.

2. In *competitive synthesis*, the task is to construct an on-line scheduling algorithm with the largest possible competitive ratio. That is, we are looking for an on-line algorithm $\mathcal{A}$ such that, for any on-line algorithm $\mathcal{A}'$, we have $\mathcal{CR}_{\mathcal{J}}(\mathcal{A}) \geqslant \mathcal{CR}_{\mathcal{J}}(\mathcal{A}')$, where $\mathcal{CR}_{\mathcal{J}}(\mathcal{A})$ is the competitive ratio of algorithm $\mathcal{A}$ under the set $\mathcal{J}$ of admissible job sequences (see Eq. (10.1)) in Section 10.2 for the definition of $\mathcal{CR}_{\mathcal{J}}$).

As in the competitive analysis case of Section 10.4, it suffices to consider only on-line scheduling algorithms encoded as LTSs (see Remark 10.1). In the following, we consider that $\mathcal{J} = \Sigma^{\omega}$, that is, there are no restrictions on the released job sequences. In Remark 10.7 below, we outline how the results can be extended to additional safety, liveness, and limit-average automata constraining $\mathcal{J}$ (see also Section 10.3.2). Finally, we conclude with a note on the *worst-case utility ratio*, namely the worst-case limiting average utility of the best online algorithm over the worst-case limiting average utility achievable by a clairvoyant algorithm (for possibly different job sequences).

Synthesis for worst-case average utility. Given a taskset, we can compute the worst-case average utility that can be achieved by any on-line scheduling algorithm. For this, we construct a *non-deterministic* finite-state LTS $L_{\mathscr{G}} = (S_{\mathscr{G}}, s_{\mathscr{G}}, \Sigma, \Pi, \Delta_{\mathscr{G}})$ with an associated reward function $r_{\mathscr{G}}$ that can simulate all possible on-line algorithms. Such an LTS has already been introduced in Section 10.3 for the clairvoyant algorithm. Note that the latter implements memoryless strategies, as all required history information is encoded in the state.

We can interpret such a non-deterministic LTS as a perfect-information graph game $\mathscr{G} = \langle S_{\mathscr{G}}, \Sigma_1, \Sigma_2, \delta \rangle$, where Σ_1 (the actions of Player 1) correspond to the output actions Π in $L_{\mathscr{G}}$, and Σ_2 (the actions of Player 2) correspond to the input actions Σ in $L_{\mathscr{G}}$. That is, Player 2 (the adversary) chooses the released tasks, while Player 1 chooses the actual transitions in δ actually taken.

Thus, we indeed have a perfect-information game, and every memoryless strategy for Player 1 corresponds to a scheduling algorithm and vice-versa (i.e., every scheduling algorithm is a memoryless strategy of Player 1 in the game $\mathscr{G}$). The weight function w for the mean-payoff objective of $\mathscr{G}$ is identical to the reward function $r_{\mathscr{G}}$, and the start state s^1 is the initial state $s_{\mathscr{G}}$ of $L_{\mathscr{G}}$. The worst-case utility of a given on-line algorithm, corresponding to a memoryless strategy

$\pi \in \Pi^M_{\mathscr{G}}$, is hence

$$\inf_{\sigma \in \Sigma_{\mathscr{G}}} \mathsf{MP}(\mathsf{wt}, \mathscr{P}(s^1, \pi, \sigma))$$

and the worst-case utility of the optimal on-line algorithm is given by

$$\sup_{\pi \in \Pi^M_{\mathscr{G}}} \inf_{\sigma \in \Sigma_{\mathscr{G}}} \mathsf{MP}(\mathsf{wt}, \mathscr{P}(s^1, \pi, \sigma)). \tag{10.3}$$

Using the results of Theorem 10.2, we obtain the following theorem.

Theorem 10.4. *The following assertions hold:*

1. *Whether there exists an on-line algorithm with worst-case average utility at least ν can be decided in* NP $\cap$ coNP *in general; and if $V_{\max}$ is bounded by the size of the non-deterministic LTS, then the decision problem can be solved in polynomial time.*

2. *An on-line algorithm with optimal worst-case average utility can be constructed in time $O(|S_{\mathscr{G}}| \cdot m \cdot V_{\max})$, where $|S_{\mathscr{G}}|$ (resp. m) is the number of states (resp. transitions) of the non-deterministic LTS $L_{\mathscr{G}}$.*

Competitive Synthesis. Given a taskset and a rational $\nu \in \mathbb{Q}$, the competitive synthesis problem asks to determine whether there exists an on-line scheduling algorithm that achieves a competitive ratio of at least ν, and to determine the optimal competitive ratio ν^*. Recall the non-deterministic LTS $L_{\mathscr{G}} = (S_{\mathscr{G}}, s_{\mathscr{G}}, \Sigma, \Pi, \Delta_{\mathscr{G}})$ and reward function $r_{\mathscr{G}}$ in the synthesis for the worst-case average utility. For solving the competitive synthesis problem, we construct a partial-observation game $\mathscr{G}_{\mathcal{CR}}$ as follows: $\mathscr{G}_{\mathcal{CR}} = \langle S_{\mathscr{G}} \times S_{\mathscr{G}}, \Sigma_1, \Sigma_2 \times \Sigma_1, \delta, \mathsf{O}_S, \mathsf{O}_\Sigma \rangle$, where $\Sigma_1 = \Pi$ and $\Sigma_2 = \Sigma$. Intuitively, we construct a product game with two components, where Player 1 only observes the first component (the on-line algorithm) and makes the choice of the transition $\alpha_1 \in \Sigma_1$ there; Player 2 is in charge of choosing the input $\alpha_2 \in \Sigma_2$ and also the transition $\alpha_1' \in \Sigma_1$ in the second component (the clairvoyant algorithm). However, due to partial observation, Player 1 does not observe the choice of the transitions of the clairvoyant algorithm.

Formally, the appropriate transition and the observation mapping are defined as follows:

(i) Transition function $\delta : (S_{\mathscr{G}} \times S_{\mathscr{G}}) \times \Sigma_1 \times (\Sigma_2 \times \Sigma_1) \to (S_{\mathscr{G}} \times S_{\mathscr{G}})$ with

$$\delta((s_1, s_2), \alpha_1, (\alpha_2, \alpha_1')) = (\delta(s_1, \alpha_1, \alpha_2), \delta(s_2, \alpha_1', \alpha_2)).$$

(ii) The observation for states for Player 1 maps every state to the first component, i.e., $\mathsf{obs}_S((s_1, s_2)) = s_1$, and the observation for actions for Player 1 maps every action (α_2, α_1') of Player 2 to its first component α_2 as well (i.e., the input from Player 2), i.e., $\mathsf{obs}_\Sigma((\alpha_2, \alpha_1')) = \alpha_2$.

The two reward functions needed for solving the ratio objective in the game are defined as follows: The reward function wt_1 gives reward according to $r_{\mathscr{G}}$ applied to the transitions of the first component. The reward function wt_2 assigns the reward according to $r_{\mathscr{G}}$ applied to the transitions of the second component. Note that this construction ensures that we compare the utility of an on-line algorithm (transitions of the first component chosen by Player 1) and an off-line algorithm (chosen by Player 2 using the second component) that operate on the *same* input sequence.

It follows that an on-line algorithm with competitive-ratio at least ν exists iff $\sup_{\pi \in \Pi_{\mathscr{G}}^M} \inf_{\sigma \in \Sigma_{\mathscr{G}}} \mathsf{Ratio}(\mathsf{wt}_1, \mathsf{wt}_2, \mathscr{P}(s^1, \pi, \sigma)) \geqslant \nu$, where $s^1 = (s_{\mathscr{G}}, s_{\mathscr{G}})$ is the start state derived from the LTS $L_{\mathscr{G}}$. By Theorem 10.3, the decision problem is in NP in the size of the game $\mathscr{G}_{C\mathcal{R}}$. Since the strategy of Player 1 can directly be translated to an on-line scheduling algorithm, the solution of the synthesis problem follows from the witness strategy for Player 1. We hence obtain the following theorem.

Theorem 10.5. *For the class of scheduling problems defined in Section 10.2, the decision problem of whether there exists an on-line scheduler with a competitive ratio at least a rational number ν is in* NP *in the size of the LTS constructed from the scheduling problem.*

Finally, finding the optimal competitive ratio ν^* (and a scheduling algorithm ensuring it) is possible by searching for $\sup\{\nu \in \mathbb{Q} :$ the answer to the decision problem is yes$\}$.

Remark 10.7. Using the reduction of Theorem 10.5 together with Remark 10.6, we obtain that the competitive synthesis problem in the presence of safety, liveness, and limit-average constraints specified as constrained automata is in NP in the size of the synchronous product of the corresponding LTSs.

Synthesis for worst-case utility ratio. We conclude our considerations regarding synthesis with the worst-case utility ratio problem, namely, determining the worst-case limiting average utility of the best online algorithm over the worst-case limiting average achievable by a clairvoyant algorithm. In sharp contrast to the competitive ratio, the job sequences used by the on-line and

off-line algorithm for computing this utility ratio may be different. Formally, we are interested in determining an online scheduling algorithm $\mathcal{A}$ that maximizes the following expression:

$$\mathcal{UR} = \liminf_{k\to\infty} \frac{\inf_{\sigma\in\mathcal{J}} V(\rho^{\sigma}_{\mathcal{A}}, k)}{\inf_{\sigma\in\mathcal{J}} V(\rho^{\sigma}_{\mathcal{C}}, k)}. \tag{10.4}$$

The numerator of $\mathcal{UR}$ corresponds to the synthesis for the worst case average utility problem, whose solution is given by Eq. (10.3) in the respective game. Similarly, the denominator is given by the following objective in the same game:

$$\inf_{\sigma\in\Pi_{\mathscr{G}}} \sup_{\pi\in\Sigma_{\mathscr{G}}} \mathsf{MP}(\mathsf{wt}, \mathscr{P}(s^1, \pi, \sigma)). \tag{10.5}$$

Herein, the input sequence is fixed (by choosing a strategy for Player 1) *before* the job sequence is fixed (by choosing a strategy for Player 2, possibly non-memoryless). According to the determinacy guaranteed by Theorem 10.2, Eq. (10.3) and Eq. (10.5) are equal, hence $\mathcal{UR} = 1$: The worst case average utility of the optimal online and the clairvoyant algorithm coincide.

Remark 10.8 (Complexity with respect to the taskset). Theorem 10.4 and Theorem 10.5 establish complexity upper bounds for the synthesis for worst-case utility, and competitive synthesis problems as a function of the size of the non-deterministic LTS $L_{\mathscr{G}}$. In general, the size of $L_{\mathscr{G}}$ is exponential in the bit representation of the taskset $\mathcal{T}$. Hence, if the input to our algorithms is the taskset $\mathcal{T}$, the polynomial upper bounds of Theorem 10.4 and Theorem 10.5 translate to exponential upper bounds in the size of $\mathcal{T}$.

Bibliography

[dim] "DIMACS Implementation Challenges".

[Wal, 2003] "T. J. Watson Libraries for Analysis (WALA)," https://github.com, 2003.

[Git, 2008] "GitHub Home," https://github.com, 2008.

[SPE, 2008] "SPECjvm2008 Benchmark Suit," http://www.spec.org/jvm2008/, 2008.

[Abboud and Vassilevska Williams, 2014] Amir Abboud and Virginia Vassilevska Williams, "Popular Conjectures Imply Strong Lower Bounds for Dynamic Problems," In *FOCS*, pages 434–443, 2014.

[Abboud and Williams, 2014] Amir Abboud and Virginia Vassilevska Williams, "Popular Conjectures Imply Strong Lower Bounds for Dynamic Problems," In *FOCS*, pages 434–443, 2014.

[Abboud *et al.*, 2015] Amir Abboud, Virginia Vassilevska Williams, and Huacheng Yu, "Matching Triangles and Basing Hardness on an Extremely Popular Conjecture," In *STOC*, pages 41–50, 2015.

[Abdulla *et al.*, 2014] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas, "Optimal Dynamic Partial Order Reduction," POPL, 2014.

[Abdulla *et al.*, 2015] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas, "Stateless Model Checking for TSO and PSO," In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035*, pages 353–367, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[Abeni and Buttazzo, 1998] Luca Abeni and Giorgio Buttazzo, "Integrating multimedia applications in hard real-time systems," In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, RTSS '98, pages 4–13, December 1998.

[Aho and Hopcroft, 1974] Alfred V. Aho and John E. Hopcroft, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974.

[Akiba *et al.*, 2013] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida, "Fast Exact Shortest-path Distance Queries on Large Networks by Pruned Landmark Labeling," In *SIGMOD'13*, SIGMOD '13, pages 349–360, 2013.

[Akiba *et al.*, 2012] Takuya Akiba, Christian Sommer, and Ken-ichi Kawarabayashi, "Shortest-Path Queries for Complex Networks: Exploiting Low Tree-width Outside the Core," In *15th International Conference on Extending Database Technology (EDBT)*, 2012.

[Almagor *et al.*, 2013] Shaull Almagor, Udi Boker, and Orna Kupferman, "Formalizing and Reasoning about Quality," In *ICALP*, pages 15–27, 2013.

[Alon and Schieber, 1987] Noga Alon and Baruch Schieber, "Optimal preprocessing for answering on-line product queries," Technical report, Tel Aviv University, 1987.

[Altisen *et al.*, 2002] Karine Altisen, Gregor Gößler, and Joseph Sifakis, "Scheduler Modeling Based on the Controller Synthesis Paradigm," *Real-Time Systems*, 23(1-2):55–84, 2002.

[Alur *et al.*, 2005] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. W. Reps, and M. Yannakakis, "Analysis of recursive state machines," *ACM Trans. Program. Lang. Syst.*, 2005.

[Alur *et al.*, 2006] R. Alur, S. La Torre, and P. Madhusudan, "Modular strategies for recursive game graphs," *Theor. Comput. Sci.*, 2006.

[Alur *et al.*, 2016] Rajeev Alur, Ahmed Bouajjani, and Javier Esparza, "Model Checking Procedural Programs," In *Handbook of Model Checking*. Springer, 2016.

[Alur *et al.*, 1999] Rajeev Alur, Sampath Kannan, and Mihalis Yannakakis, "Communicating Hierarchical State Machines," ICAL, 1999.

[Amdahl, 1967] Gene M Amdahl, "Validity of the single processor approach to achieving large

scale computing capabilities," In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[Arnborg *et al.*, 1987] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski, "Complexity of Finding Embeddings in a k-Tree," *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987.

[Arnborg and Proskurowski, 1989] Stefan Arnborg and Andrzej Proskurowski, "Linear time algorithms for NP-hard problems restricted to partial k-trees ," *Discrete Appl Math*, 1989.

[Arnold, 1996] Robert S. Arnold, *Software Change Impact Analysis*, IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[Aspvall *et al.*, 1979] Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan, "A linear-time algorithm for testing the truth of certain quantified boolean formulas," *IPL*, 8(3):121 – 123, 1979.

[Atig *et al.*, 2008] Mohamed Faouzi Atig, Benedikt Bollig, and Peter Habermehl, "Emptiness of Multi-pushdown Automata Is 2ETIME-Complete," In *Proceedings of the 12th International Conference on Developments in Language Theory*, DLT, pages 121–133, 2008.

[Aydin *et al.*, 2004] Hakan Aydin, Rami Melhem, Daniel Mossé, and Pedro Mejía-Alvarez, "Power-Aware Scheduling for Periodic Real-Time Tasks," *IEEE Transactions on Computers*, 53(5):584–600, May 2004.

[Babich and Jazayeri, 1978] Wayne A. Babich and Mehdi Jazayeri, "The method of attributes for data flow analysis," *Acta Informatica*, 10(3), 1978.

[Ball and Larus, 1993] Thomas Ball and James R. Larus, "Branch Prediction For Free," In *PLDI*, pages 300–313, 1993.

[Banachowski, 1980] Lech Banachowski, "A complement to Tarjan's result about the lower bound on the complexity of the set union problem," *Information Processing Letters*, 11(2):59 – 65, 1980.

[Baruah *et al.*, 1992] Sanjoy Baruah, Gilad Koren, Decao Mao, Bhubaneswar Mishra, Arvind

Raghunathan, Lou Rosier, Dennis Shasha, and Fuxing Wang, "On the Competitiveness of On-Line Real-Time Task Scheduling," *Real-Time Systems*, 4(2):125–144, 1992.

[Baruah *et al.*, 1991] Sanjoy Baruah, Gilad Koren, Bhubaneswar Mishra, Arvind Raghunathan, Lou Rosier, and Dennis Shasha, "On-line Scheduling in the Presence of Overload," In *Proceedings of the 32nd annual Symposium on Foundations of Computer Science*, FOCS '91, pages 100–110, Oct 1991.

[Baruah and Haritsa, 1997] Sanjoy K. Baruah and Jayant R. Haritsa, "Scheduling for Overload in Real-Time Systems," *IEEE Transactions on Computers*, 46:1034–1039, September 1997.

[Baruah and Hickey, 1998] Sanjoy K. Baruah and Mary Ellen Hickey, "Competitive On-Line Scheduling of Imprecise Computations," *IEEE Transactions on Computers*, 47(9):1027–1032, September 1998.

[Bauer *et al.*, 2013] Reinhard Bauer, Tobias Columbus, Ignaz Rutter, and Dorothea Wagner, "Search-Space Size in Contraction Hierarchies.," In *ICALP 13*, pages 93–104, 2013.

[Bellman, 1958] R. Bellman, "On a Routing Problem," *Quarterly of Applied Mathematics*, 1958.

[Berkelaar *et al.*] Michel Berkelaar, Kjell Eikland, and Peter Notebaert, *lpsolve : Open source (Mixed-Integer) Linear Programming system*, Version 5.0.0.0, May 2004.

[Bern *et al.*, 1987] M.W Bern, E.L Lawler, and A.L Wong, "Linear-time computation of optimal subgraphs of decomposable graphs," *J Algorithm*, 1987.

[Bertele and Brioschi, 1972] Umberto Bertele and Francesco Brioschi, *Nonserial Dynamic Programming*, Academic Press, Inc., Orlando, FL, USA, 1972.

[Bjorklund *et al.*, 2004] H. Bjorklund, S. Sandberg, and S. Vorobyov, "A Combinatorial Strongly Subexponential Strategy Improvement Algorithm for Mean Payoff Games," In *MFCS'04*, pages 673–685, 2004.

[Blackburn *et al.*, 2006] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B.

Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann, "The DaCapo benchmarks: java benchmarking development and analysis," *SIGPLAN Not.*, 41(10):169–190, 2006.

[Blackburn, 2006] Stephen M. et al. Blackburn, "The DaCapo Benchmarks: Java Benchmarking Development and Analysis," In *OOPSLA*, 2006.

[Bloem *et al.*, 2009a] R. Bloem, K. Chatterjee, T. A. Henzinger, and B. Jobstmann, "Better Quality in Synthesis through Quantitative Objectives," In *CAV*, pages 140–156, 2009.

[Bloem *et al.*, 2009b] R. Bloem, K. Greimel, T. A. Henzinger, and B. Jobstmann, "Synthesizing robust systems," In *FMCAD*, pages 85–92, 2009.

[Bodden, 2012] Eric Bodden, "Inter-procedural Data-flow Analysis with IFDS/IDE and Soot," In *SOAP*, New York, NY, USA, 2012. ACM.

[Bodden *et al.*, 2011] Eric Bodden, Andreas Sewe, Jan Sinschek, Mira Mezini, and Hela Oueslati, "Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders," In *ICSE '11*, pages 241–250. ACM, 2011.

[Bodlaender *et al.*, 1998] Hans Bodlaender, Jens Gustedt, and Jan Arne Telle, "Linear-time Register Allocation for a Fixed Number of Registers," In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA, pages 574–583, 1998.

[Bodlaender, 1988] Hans L. Bodlaender, "Dynamic programming on graphs with bounded treewidth," In *ICALP*, LNCS. Springer, 1988.

[Bodlaender, 1993] Hans L. Bodlaender, "A Tourist Guide through Treewidth.," *Acta Cybern.*, 1993.

[Bodlaender, 1996] Hans L. Bodlaender, "A Linear-Time Algorithm for Finding Tree-Decompositions of Small Treewidth," *SIAM J. Comput.*, 1996.

[Bodlaender, 1998] Hans L. Bodlaender, "A partial k-arboretum of graphs with bounded treewidth," *TCS*, 1998.

[Bodlaender *et al.*, 2013] Hans L. Bodlaender, Pal Grønas Drange, Markus S. Dregi, Fedor V. Fomin, Daniel Lokshtanov, and Michal Pilipczuk, "An $O(c^k n)$ 5-Approximation Algorithm

for Treewidth," In *Proceedings of the 2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, FOCS, 2013.

[Bodlaender, 1994] HansL. Bodlaender, "Dynamic algorithms for graphs with treewidth 2," In *Graph-Theoretic Concepts in Computer Science*, LNCS. Springer, 1994.

[Bodlaender, 2005] HansL. Bodlaender, "Discovering Treewidth," In *SOFSEM 2005: Theory and Practice of Computer Science*, volume 3381 of *LNCS*. Springer, 2005.

[Bodlaender and Hagerup, 1995] HansL. Bodlaender and Torben Hagerup, "Parallel algorithms with optimal speedup for bounded treewidth," volume 27, pages 1725–1746. 1995.

[Boker *et al.*, 2011] U. Boker, K. Chatterjee, T. A. Henzinger, and O. Kupferman, "Temporal Specifications with Accumulative Values," In *LICS*, 2011.

[Boker *et al.*, 2015] Udi Boker, Thomas A. Henzinger, and Jan Otop, "The Target Discounted-Sum Problem," In *Proceedings of the 2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, LICS, pages 750–761, 2015.

[Bonifaci and Marchetti-Spaccamela, 2012] Vincenzo Bonifaci and Alberto Marchetti-Spaccamela, "Feasibility Analysis of Sporadic Real-Time Multiprocessor Task Systems," *Algorithmica*, pages 230–241, 2012.

[Borodin and El-Yaniv, 1998] Allan Borodin and Ran El-Yaniv, *Online Computation and Competitive Analysis*, Cambridge University Press, 1998.

[Bouajjani *et al.*, 1997] Ahmed Bouajjani, Javier Esparza, and Oded Maler, "Reachability Analysis of Pushdown Automata: Application to Model-Checking," In *Proceedings of the 8th International Conference on Concurrency Theory*, CONCUR, pages 135–150, 1997.

[Bouajjani *et al.*, 2005] Ahmed Bouajjani, Javier Esparza, Stefan Schwoon, and Jan Strejček, "Reachability Analysis of Multithreaded Software with Asynchronous Communication," FSTTCS, 2005.

[Bouajjani *et al.*, 2003a] Ahmed Bouajjani, Javier Esparza, and Tayssir Touili, "A generic approach to the static analysis of concurrent programs with procedures," In *POPL*, 2003.

[Bouajjani *et al.*, 2003b] Ahmed Bouajjani, Javier Esparza, and Tayssir Touili, "A Generic

Approach to the Static Analysis of Concurrent Programs with Procedures," In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, 2003.

[Bouyer *et al.*, 2008] Patricia Bouyer, Uli Fahrenberg, Kim G. Larsen, Nicolas Markey, and Jiří Srba, "Infinite Runs in Weighted Timed Automata with Energy Constraints," In *Formal Modeling and Analysis of Timed Systems*, volume 5215 of *Lecture Notes in Computer Science*, pages 33–47. Springer Berlin Heidelberg, 2008.

[Bouyer *et al.*, 2014] Patricia Bouyer, Nicolas Markey, and Raj Mohan Matteplackel, "Averaging in LTL," In *CONCUR*, pages 266–280, 2014.

[Bozzelli *et al.*, 2006] Laura Bozzelli, Salvatore La Torre, and Adriano Peron, "Verification of Well-Formed Communicating Recursive State Machines," In *VMCAI*, 2006.

[Brim *et al.*, 2011] L. Brim, J. Chaloupka, L. Doyen, R. Gentilini, and J. F. Raskin, "Faster algorithms for mean-payoff games," *Formal Methods in System Design*, 38(2):97–118, 2011.

[Brosius] Dave Brosius, "Java Agent for Memory Measurements".

[Burgstaller *et al.*, 2004] Bernd Burgstaller, Johann Blieberger, and Bernhard Scholz, "On the Tree Width of Ada Programs," In *Reliable Software Technologies - Ada-Europe 2004*. 2004.

[Burns, 1991] Steven M. Burns, "Performance Analysis and Optimization of Asynchronous Circuits," Technical report, 1991.

[Callahan *et al.*, 1986] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon, "Interprocedural Constant Propagation," In *CC*. ACM, 1986.

[Carré, 1971] B. A. Carré, "An Algebra for Network Routing Problems," *IMA Journal of Applied Mathematics*, 7(3):273–294, 1971.

[Carroll and Heiser, 2010] Aaron Carroll and Gernot Heiser, "An Analysis of Power Consumption in a Smartphone.," In *USENIX*, 2010.

[Černý *et al.*, 2011] Pavol Černý, Krishnendu Chatterjee, Thomas A. Henzinger, Arjun Radhakrishna, and Rohit Singh, *Quantitative Synthesis for Concurrent Programs*, 2011.

[Černý *et al.*, 2012] Pavol Černý, Thomas A. Henzinger, and Arjun Radhakrishna, "Simulation Distances," *Theor. Comput. Sci.*, 413(1):21–35, 2012.

[Cerný *et al.*, 2013] Pavol Cerný, Thomas A. Henzinger, and Arjun Radhakrishna, "Quantitative abstraction refinement," In *POPL*, pages 115–128, 2013.

[Chatterjee *et al.*, 2010a] K. Chatterjee, L. Doyen, and T. A. Henzinger, "Quantitative languages," *ACM Trans. Comput. Log.*, 11(4), 2010.

[Chatterjee and Lacki, 2013] K. Chatterjee and J. Lacki, "Faster Algorithms for Markov Decision Processes with Low Treewidth," In *CAV*, 2013.

[Chatterjee *et al.*, 2015a] Krishnendu Chatterjee, Andreas, and Yaron Velner, "Quantitative Interprocedural Analysis," In *POPL*, 2015.

[Chatterjee *et al.*, 2010b] Krishnendu Chatterjee, Laurent Doyen, Herbert Edelsbrunner, Thomas A. Henzinger, and Philippe Rannou, "Mean-payoff Automaton Expressions," *CoRR*, abs/1006.1492, 2010.

[Chatterjee *et al.*, 2010c] Krishnendu Chatterjee, Laurent Doyen, and Thomas A. Henzinger, "Expressiveness and Closure Properties for Quantitative Languages," *LMCS*, 6(3), 2010.

[Chatterjee *et al.*, 2016a] Krishnendu Chatterjee, Amir Kafshdar Goharshady, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis, "Algorithms for algebraic path properties in concurrent systems of constant treewidth components," In *POPL*, pages 733–747, 2016.

[Chatterjee *et al.*, 2014a] Krishnendu Chatterjee, Monika Henzinger, Sebastian Krinninger, Veronika Loitzenbauer, and Michael A. Raskin, "Approximating the minimum cycle mean," *Theor. Comput. Sci.*, 2014.

[Chatterjee *et al.*, 2015b] Krishnendu Chatterjee, Thomas A. Henzinger, Barbara Jobstmann, and Rohit Singh, "Measuring and Synthesizing Systems in Probabilistic Environments," In *JACM*, pages 380–395, 2015.

[Chatterjee *et al.*, 2014b] Krishnendu Chatterjee, Thomas A Henzinger, and Jan Otop, "Nested Weighted Automata," Technical report, IST Austria, 2014.

[Chatterjee *et al.*, 2015c] Krishnendu Chatterjee, Thomas A. Henzinger, and Jan Otop, "Nested

Weighted Automata," In *Proceedings of the 2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, LICS, pages 725–737, 2015.

[Chatterjee *et al.*, 2016b] Krishnendu Chatterjee, Thomas A. Henzinger, and Jan Otop, "Nested Weighted Limit-Average Automata of Bounded Width," In *41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016)*, volume 58 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:14, 2016.

[Chatterjee *et al.*, 2016c] Krishnendu Chatterjee, Thomas A. Henzinger, and Jan Otop, "Quantitative Automata Under Probabilistic Semantics," In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS, pages 76–85, 2016.

[Chatterjee *et al.*, 2016d] Krishnendu Chatterjee, Thomas A. Henzinger, and Jan Otop, *Quantitative Monitor Automata*, pages 23–38, Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[Chatterjee *et al.*, 2015d] Krishnendu Chatterjee, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis, "Faster Algorithms for Quantitative Verification in Constant Treewidth Graphs," In *CAV*, 2015.

[Chatterjee *et al.*, 2016e] Krishnendu Chatterjee, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis, "Optimal Reachability and a Space-Time Tradeoff for Distance Queries in Constant-Treewidth Graphs," In *24th Annual European Symposium on Algorithms, ESA 2016, August 22-24, 2016, Aarhus, Denmark*, pages 28:1–28:17, 2016.

[Chatterjee *et al.*, 2015e] Krishnendu Chatterjee, Rasmus Ibsen-Jensen, Andreas Pavlogiannis, and Prateesh Goyal, "Faster Algorithms for Algebraic Path Properties in Recursive State Machines with Constant Treewidth," In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 97–109, 2015.

[Chatterjee *et al.*, 2013] Krishnendu Chatterjee, Alexander Kößler, and Ulrich Schmid, "Automated Analysis of Real-Time Scheduling using Graph Games," In *Proceedings of the 16th ACM international conference on Hybrid Systems: Computation and Control*, HSCC '13, pages 163–172, New York, NY, USA, 2013. ACM.

[Chatterjee *et al.*, 2017] Krishnendu Chatterjee, Bernhard Kragl, Samarth Mishra, and Andreas Pavlogiannis, “Faster Algorithms for Weighted Recursive State Machines.,” In *ESOP*, 2017.

[Chatterjee *et al.*, 2014c] Krishnendu Chatterjee, Andreas Pavlogiannis, Alexander Kößler, and Ulrich Schmid, “A Framework for Automated Competitive Analysis of On-line Scheduling of Firm-Deadline Tasks,” RTSS ’14, pages 118–127, 2014.

[Chatterjee *et al.*, 2015f] Krishnendu Chatterjee, Andreas Pavlogiannis, and Yaron Velner, “Quantitative Interprocedural Analysis,” In *POPL*, 2015.

[Chatterjee *et al.*, 2012] Krishnendu Chatterjee, Mickael Randour, and Jean-François Raskin, *Strategy Synthesis for Multi-Dimensional Quantitative Objectives*, pages 115–131, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[Chatterjee and Velner, 2012] Krishnendu Chatterjee and Yaron Velner, “Mean-Payoff Pushdown Games,” In *LICS*, pages 195–204, 2012.

[Chatterjee and Velner, 2013] Krishnendu Chatterjee and Yaron Velner, *Hyperplane Separation Technique for Multidimensional Mean-Payoff Games*, pages 500–515, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[Chaudhuri and Zaroliagis, 1995] Shiva Chaudhuri and Christos D. Zaroliagis, “Shortest Paths in Digraphs of Small Treewidth. Part I: Sequential Algorithms,” *Algorithmica*, 1995.

[Chaudhuri, 2008] Swarat Chaudhuri, “Subcubic Algorithms for Recursive State Machines,” In *POPL*, New York, NY, USA, 2008. ACM.

[Choi *et al.*, 1993] Jong-Deok Choi, Michael Burke, and Paul Carini, “Efficient Flow-sensitive Interprocedural Computation of Pointer-induced Aliases and Side Effects,” In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’93, pages 232–245. ACM, 1993.

[Chugh *et al.*, 2008] Ravi Chugh, Jan W. Voung, Ranjit Jhala, and Sorin Lerner, “Dataflow Analysis for Concurrent Programs Using Datarace Detection,” In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2008.

[Clarke *et al.*, 1999a] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled, *Model Checking*, MIT Press, Cambridge, MA, USA, 1999.

[Clarke *et al.*, 1999b] E.M. Clarke, O. Grumberg, M. Minea, and D. Peled, "State space reduction using partial order techniques," *STTT*, 2(3):279–287, 1999.

[Cochet-terrasson *et al.*, 1998] Jean Cochet-terrasson, Guy Cohen, Stephane Gaubert, Michael Mc Gettrick, and Jean pierre Quadrat, "Numerical Computation of Spectral Elements in Max-Plus Algebra," 1998.

[Columbus, 2012] Tobias Columbus, "Search Space Size in Contraction Hierarchies," Master's thesis, Karlsruhe Institute of Technology, 2012.

[Cormen *et al.*, 2009] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, The MIT Press, 3rd edition, 2009.

[Courcelle, 1990] Brouno Courcelle, "Graph Rewriting: An Algebraic and Logic Approach," In *Handbook of Theoretical Computer Science (Vol. B)*. MIT Press, Cambridge, MA, USA, 1990.

[Cousot and Cousot, 1977a] P. Cousot and R Cousot, "Static determination of dynamic properties of recursive procedures," In E.J. Neuhold, editor, *IFIP Conf. on Formal Description of Programming Concepts*, 1977.

[Cousot and Cousot, 1977b] Patrick Cousot and Radhia Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," In *POPL*, pages 238–252, 1977.

[Cruz, 1991] Rene L. Cruz, "A calculus for network delay. I. Network elements in isolation," *IEEE Transactions on Information Theory*, 37(1):114–131, Jan 1991.

[D'Antoni *et al.*, 2016] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh, *Qlose: Program Repair with Quantitative Objectives*, pages 383–401, 2016.

[Dasdan and Gupta, 1998] A. Dasdan and R.K. Gupta, "Faster maximum and minimum mean cycle algorithms for system-performance analysis," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(10):889–899, Oct 1998.

[Dasdan *et al.*, 1998] Ali Dasdan, Sandra S. Irani, and Rajesh K. Gupta, "An Experimental Study of Minimum Mean Cycle Algorithms," Technical report, 1998.

[De *et al.*, 2011] Arnab De, Deepak D'Souza, and Rupesh Nasre, "Dataflow Analysis for Datarace-free Programs," In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ESOP'11, pages 196–215. Springer-Verlag, 2011.

[Dertouzos, 1974] Michael L. Dertouzos, "Control Robotics: The Procedural Control of Physical Processes," In *IFIP Congress*, pages 807–813, 1974.

[Devadas *et al.*, 2010] Vinay Devadas, Fei Li, and Hakan Aydin, "Competitive analysis of online real-time scheduling algorithms under hard energy constraint," *Real-Time Syst.*, 46(1):88–120, September 2010.

[Dijkstra, 1959] Edsger. W. Dijkstra, "A note on two problems in connexion with graphs.," *Numerische Mathematik*, 1959.

[Doyle and Rivest, 1976] Jon Doyle and Ronald L. Rivest, "Linear expected time of a simple union-find algorithm," *Information Processing Letters*, 5(5):146 – 148, 1976.

[Droste and Meinecke, 2010] M. Droste and I. Meinecke, "Describing Average- and Longtime-Behavior by Weighted MSO Logics," In *MFCS*, pages 537–548, 2010.

[Droste *et al.*, 2009] Manfred Droste, Werner Kuich, and Heiko Vogler, *Handbook of Weighted Automata*, Springer Publishing Company, Incorporated, 1st edition, 2009.

[Droste and Meinecke, 2012] Manfred Droste and Ingmar Meinecke, "Weighted automata and weighted MSO logics for average and long-time behaviors," *Inf. Comput.*, 220:44–59, 2012.

[Duesterwald *et al.*, 1995] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa, "Demand-driven Computation of Interprocedural Data Flow," POPL, 1995.

[Dufour *et al.*, 2008] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky, "A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications," In *SIGSOFT FSE*, pages 59–70, 2008.

[Ehrenfeucht and Mycielski, 1979] A. Ehrenfeucht and J. Mycielski, "Positional strategies for mean payoff games," *International Journal of Game Theory*, 8(2), 1979.

[Elberfeld *et al.*, 2010] M. Elberfeld, A. Jakoby, and T. Tantau, "Logspace Versions of the Theorems of Bodlaender and Courcelle," In *FOCS*, 2010.

[Englert and Westermann, 2007] Matthias Englert and Matthias Westermann, "Considering suppressed packets improves buffer management in QoS switches," In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, pages 209–218, 2007.

[Esparza *et al.*, 2000] Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon, "Efficient Algorithms for Model Checking Pushdown Systems," In *Proceedings of the 12th International Conference on Computer Aided Verification*, CAV '00, pages 232–247, London, UK, UK, 2000. Springer-Verlag.

[Farzan *et al.*, 2013] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski, "Inductive Data Flow Graphs," POPL, 2013.

[Farzan and Madhusudan, 2007] Azadeh Farzan and P. Madhusudan, "Causal Dataflow Analysis for Concurrent Programs," In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS, 2007.

[Feige *et al.*, 2005] Uriel Feige, MohammadTaghi Hajiaghayi, and James R. Lee, "Improved Approximation Algorithms for Minimum-weight Vertex Separators," STOC, pages 563–572, 2005.

[Ferdinand *et al.*, 99] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt, "Cache Behavior Prediction by Abstract Interpretation," *Sci. Comput. Program.*, 99.

[Ferrara *et al.*, 2005] Andrea Ferrara, Guoqiang Pan, and Moshe Y. Vardi, *Treewidth in Verification: Local vs. Global*, pages 489–503, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[Filar and Vrieze, 1997] J. Filar and K. Vrieze, *Competitive Markov Decision Processes*, Springer-Verlag, 1997.

[Fink, 1992] Eugene Fink, *A Survey of Sequential and Systolic Algorithms for the Algebraic Path Problem*, Research report (University of Waterloo. Faculty of Mathematics). Faculty of Mathematics, University of Waterloo, 1992.

[Fischer and Meyer, 1971] Michael J. Fischer and Albert R. Meyer, "Boolean Matrix Multiplication and Transitive Closure," In *SWAT (FOCS)*. IEEE Computer Society, 1971.

[Flanagan and Godefroid, 2005] Cormac Flanagan and Patrice Godefroid, "Dynamic Partial-order Reduction for Model Checking Software," In *POPL*, 2005.

[Floyd, 1962] Robert W. Floyd, "Algorithm 97: Shortest path," *Communications of the ACM*, 1962.

[Ford, 1956] Lester R. Ford, "Network Flow Theory," Report P-923, The Rand Corporation, 1956.

[Gabow and Tarjan, 1985] Harold N. Gabow and Robert Endre Tarjan, "A linear-time algorithm for a special case of disjoint set union," *Journal of Computer and System Sciences*, 30(2):209 – 221, 1985.

[Galil and Italiano, 1991] Zvi Galil and Giuseppe F. Italiano, "Data Structures and Algorithms for Disjoint Set Union Problems," *ACM Comput. Surv.*, 23(3):319–344, 1991.

[Galperin and Wigderson, 1983] Hana Galperin and Avi Wigderson, "Succinct representations of graphs," *Information and Control*, 56(3):183 – 198, 1983.

[Garey and Johnson, 1979] Michael R. Garey and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., New York, NY, USA, 1979.

[Gerez *et al.*, 1992] S. H. Gerez, S. M. Heemstra de Groot, and O. E. Herrmann, "A polynomial time algorithm for the computation of the iteration-period bound in recursive data flow graphs," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 39(1):49–52, Jan 1992.

[Giegerich *et al.*, 1981] Robert Giegerich, Ulrich Möncke, and Reinhard Wilhelm, "Invariance

of Approximate Semantics with Respect to Program Transformations," In *3rd Conference of the European Co-operation in Informatics (ECI)*, 1981.

[Godefroid, 1996] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, Springer-Verlag, Secaucus, NJ, USA, 1996.

[Godefroid, 1997] Patrice Godefroid, "Model Checking for Programming Languages Using VeriSoft," In *POPL*, 1997.

[Godefroid, 2005] Patrice Godefroid, "Software Model Checking: The VeriSoft Approach," *FMSD*, 26(2):77–101, 2005.

[Golestani, 1991] S. Jamaloddin Golestani, "A Framing Strategy for Congestion Management," *IEEE Journal on Selected Areas in Communications*, 9(7):1064–1077, Sepember 1991.

[Graham *et al.*, 1989] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik, *Concrete mathematics: A foundation for computer science*, Addison-Wesley, 1989.

[Grove and Torczon, 1993] Dan Grove and Linda Torczon, "Interprocedural Constant Propagation: A Study of Jump Function Implementation," In *PLDI*. ACM, 1993.

[Grunwald and Srinivasan, 1993] Dirk Grunwald and Harini Srinivasan, "Data Flow Equations for Explicitly Parallel Programs," In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP, 1993.

[Gupta and Palis, 2001] Bhaskar Das Gupta and Michael A. Palis, "Online real-time preemptive scheduling of jobs with deadlines on multiple machines," *Journal of Scheduling*, 4(6):297–312, 2001.

[Gustedt *et al.*, 2002] Jens Gustedt, OleA. Mæhle, and JanArne Telle, "The Treewidth of Java Programs," In *Algorithm Engineering and Experiments*, LNCS. Springer, 2002.

[Hagerup, 2000] Torben Hagerup, "Dynamic algorithms for graphs of bounded treewidth," *Algorithmica*, 2000.

[Halin, 1976] Rudolf Halin, "S-functions for graphs," *Journal of Geometry*, 1976.

[Harel and Tarjan, 1984] D. Harel and R. Tarjan, “Fast Algorithms for Finding Nearest Common Ancestors,” *SIAM Journal on Computing*, 1984.

[Harel *et al.*, 1997] David Harel, Orna Kupferman, and MosheY. Vardi, “On the complexity of verifying concurrent transition systems,” In *CONCUR*. 1997.

[Haritsa *et al.*, 1990] Jayant R. Haritsa, Michael J. Carey, and Miron Livny, “On Being Optimistic About Real-time Constraints,” In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS ’90, pages 331–343, New York, NY, USA, 1990. ACM.

[Hart *et al.*, 1968] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael, “A formal basis for the heuristic determination of minimum cost paths,” *IEEE Trans. on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[Hartmann and Orlin, 1993] Mark Hartmann and James B. Orlin, “Finding minimum cost to time ratio cycles with small integral transit times,” *NETWORKS*, 23:567–574, 1993.

[Hennessy and Patterson, 2006] John L. Hennessy and David A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*, Morgan Kaufmann, 2006.

[Henzinger *et al.*, 2015] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak, “Unifying and Strengthening Hardness for Dynamic Problems via the Online Matrix-Vector Multiplication Conjecture,” In *STOC*, pages 21–30, 2015.

[Henzinger and Otop, 2013] Thomas A. Henzinger and Jan Otop, “From Model Checking to Model Measuring,” In *CONCUR*, pages 273–287, 2013.

[Henzinger and Otop, 2014] Thomas A. Henzinger and Jan Otop, “Model Measuring for Hybrid Systems,” In *Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control*, HSCC, pages 213–222, 2014.

[Henzinger *et al.*, 2014] Thomas A. Henzinger, Jan Otop, and Roopsha Samanta, “Lipschitz Robustness of Finite-state Transducers,” In *34th International Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS 2014)*, volume 29 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 431–443, Dagstuhl, Germany, 2014.

[Henzinger *et al.*, 2016] Thomas A. Henzinger, Jan Otop, and Roopsha Samanta, *Lipschitz Robustness of Timed I/O Systems*, pages 250–267, 2016.

[Hind, 2001] Michael Hind, "Pointer Analysis: Haven'T We Solved This Problem Yet?," In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 54–61. ACM, 2001.

[Hopcroft, 2007] John E. Hopcroft, *Introduction to Automata Theory, Languages, and Computation*, Pearson Addison Wesley, 3rd edition, 2007.

[Horwitz, 1997] Susan Horwitz, "Precise Flow-insensitive May-alias Analysis is NP-hard," *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, 1997.

[Horwitz *et al.*, 1995] Susan Horwitz, Thomas Reps, and Mooly Sagiv, "Demand Interprocedural Dataflow Analysis," *SIGSOFT Softw. Eng. Notes*, 1995.

[Howard, 1960] Ronald A. Howard, *Dynamic Programming and Markov Processes*, MIT˜Press, Cambridge, Massachusetts, 1960.

[Huang, 2015] Jeff Huang, "Stateless Model Checking Concurrent Programs with Maximal Causality Reduction," In *PLDI*, 2015.

[Ito and Parhi, 1995] Kazuhito Ito and Keshab K. Parhi, "Determining the Minimum Iteration Period of an Algorithm," *J. VLSI Signal Process. Syst.*, 11(3):229–244, 1995.

[Johnson, 1977] Donald B. Johnson, "Efficient Algorithms for Shortest Paths in Sparse Networks," *J. ACM*, 1977.

[Kahlon and Gupta, 2007] Vineet Kahlon and Aarti Gupta, "On the Analysis of Interacting Pushdown Systems," In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 303–314, 2007.

[Kahlon *et al.*, 2013] Vineet Kahlon, Sriram Sankaranarayanan, and Aarti Gupta, "Static analysis for concurrent programs with applications to data race detection," *International Journal on Software Tools for Technology Transfer*, 15(4):321–336, 2013.

[Kahlon *et al.*, 2009] Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang, "Static Data Race Detection for Concurrent Programs with Asynchronous Calls," In *Proceedings of the the*

7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE '09, pages 13–22, 2009.

[Karp, 1972] Richard M. Karp, "Reducibility among Combinatorial Problems," In *Complexity of Computer Computations*. Springer US, 1972.

[Karp, 1978] Richard M. Karp, "A characterization of the minimum cycle mean in a digraph," *Discrete Mathematics*, 23(3):309 – 311, 1978.

[Karp and Orlin, 1981] Richard M. Karp and James B. Orlin, "Parametric shortest path algorithms with an application to cyclic staffing," *Discrete Applied Mathematics*, 3(1):37 – 45, 1981.

[Khachiyan, 1979] Leonid G. Khachiyan, "A polynomial algorithm in linear programming," *Doklady Akademii Nauk SSSR*, 244, 1979.

[Kildall, 1973] Gary A. Kildall, "A Unified Approach to Global Program Optimization," In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL, 1973.

[Kleene, 1956] S. C. Kleene, "Representation of Events in Nerve Nets and Finite Automata," *Automata Studies*, 1956.

[Kloks, 1994] Ton Kloks, *Treewidth, Computations and Approximations*, LNCS. Springer, 1994.

[Knoop and Steffen, 1992] Jens Knoop and Bernhard Steffen, "The Interprocedural Coincidence Theorem," In *CC*, 1992.

[Knoop *et al.*, 1996] Jens Knoop, Bernhard Steffen, and Jürgen Vollmer, "Parallelism for Free: Efficient and Optimal Bitvector Analyses for Parallel Programs," *ACM Trans. Program. Lang. Syst.*, 1996.

[Koren and Shasha, 1995] Gilad Koren and Dennis Shasha, "D^{over}: An Optimal On-Line Scheduling Algorithm for Overloaded Uniprocessor Real-Time Systems," *SIAM Journal on Computing*, 24(2):318–339, April 1995.

[Koutsoupias, 2011] Elias Koutsoupias, "Scheduling without payments," In *Proceedings of the*

4th international conference on Algorithmic game theory, SAGT'11, pages 143–153, Berlin, Heidelberg, 2011. Springer-Verlag.

[Krause, 2013] Philipp Klaus Krause, "Optimal Register Allocation in Polynomial Time," In Ranjit Jhala and Koen De Bosschere, editors, *Compiler Construction*, Lecture Notes in Computer Science. 2013.

[Krause, 2014] Philipp Klaus Krause, "The Complexity of Register Allocation," *Discrete Appl. Math.*, 168:51–59, 2014.

[Kuck *et al.*, 1981] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence Graphs and Compiler Optimizations," In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 207–218, 1981.

[Kwek and Mehlhorn, 2003] Stephen Kwek and Kurt Mehlhorn, "Optimal Search for Rationals," *Inf. Process. Lett.*, 86(1):23–26, 2003.

[La Torre *et al.*, 2008] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato, "Context-bounded Analysis of Concurrent Queue Systems," TACAS, 2008.

[Lacki, 2013] J. Lacki, "Improved Deterministic Algorithms for Decremental Reachability and Strongly Connected Components," *ACM Transactions on Algorithms*, 2013.

[Lal *et al.*, 2012] Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri, "A Solver for Reachability Modulo Theories," CAV, 2012.

[Lal and Reps, 2008] Akash Lal and Thomas Reps, "Solving Multiple Dataflow Queries Using WPDSs," In *Proceedings of the 15th International Symposium on Static Analysis*, SAS, pages 93–109, 2008.

[Lal and Reps, 2009] Akash Lal and Thomas Reps, "Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis," *Form. Methods Syst. Des.*, 2009.

[Lal *et al.*, 2005] Akash Lal, Thomas W. Reps, and Gogul Balakrishnan, "Extended Weighted Pushdown Systems," In *CAV*, pages 434–448, 2005.

[Lal *et al.*, 2008] Akash Lal, Tayssir Touili, Nicholas Kidd, and Thomas Reps, "Interprocedural Analysis of Concurrent Programs Under a Context Bound," TACAS, 2008.

[Lamport, 1979] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[Lamport, 1978] Leslie Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, 21(7):558–565, July 1978.

[Landi and Ryder, 1991] William Landi and Barbara G. Ryder, "Pointer-induced Aliasing: A Problem Classification," In *POPL*. ACM, 1991.

[Landi and Ryder, 1992] William Landi and Barbara G. Ryder, "A Safe Approximate Algorithm for Interprocedural Aliasing," In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 235–248. ACM, 1992.

[Lawler, 1976] Eugene Lawler, *Combinatorial Optimization: Networks and Matroids*, Saunders College Publishing, 1976.

[Le Gall, 2014] François Le Gall, "Powers of Tensors and Fast Matrix Multiplication," In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, ISSAC, pages 296–303, 2014.

[Lee, 2002] Lillian Lee, "Fast Context-free Grammar Parsing Requires Fast Boolean Matrix Multiplication," *J. ACM*, 49(1):1–15, 2002.

[Lehmann, 1977] Daniel J. Lehmann, "Algebraic structures for transitive closure," *Theoretical Computer Science*, 1977.

[Lhoták and Hendren, 2003] Ondrej Lhoták and Laurie J. Hendren, "Scaling Java Points-to Analysis Using SPARK," In *CC*, pages 153–169, 2003.

[Lhoták and Hendren, 2006] Ondřej Lhoták and Laurie Hendren, "Context-Sensitive Points-to Analysis: Is It Worth It?," In *Proceedings of the 15th International Conference on Compiler Construction*, CC, pages 47–64, 2006.

[Liggett and Lippman, 1969] T. A. Liggett and S. A. Lippman, "Stochastic games with perfect information and time average payoff," *Siam Review*, 11:604–607, 1969.

[Locke, 1986] Carey Douglass Locke, *Best-effort Decision-making for Real-time Scheduling*, PhD thesis, CMU, Pittsburgh, PA, USA, 1986.

[Lu and Jantsch, 2007] Zhonghai Lu and Axel Jantsch, "Admitting and ejecting flits in wormhole-switched networks on chip.," *IET Computers & Digital Techniques*, 1, 2007.

[Madan Musuvathi, 2007] Tom Ball Madan Musuvathi, Shaz Qadeer, "CHESS: A systematic testing tool for concurrent software," Technical report, November 2007.

[Madani, 2002] Omid Madani, "Polynomial Value Iteration Algorithms for Deterministic MDPs," In *Proceedings of the 18th Conference on Uncertainty in Artificial Intelligence*, UAI '02, pages 311–318, 2002.

[Madhusudan and Parlato, 2011] P. Madhusudan and Gennaro Parlato, "The Tree Width of Auxiliary Storage," POPL, 2011.

[Mahr, 1984] B. Mahr, "Iteration and Summability in Semirings," In R.A. Cuninghame-Green R.E. Burkard and U. Zimmermann, editors, *Algebraic and Combinatorial Methods in Operations ResearchProceedings of the Workshop on Algebraic Structures in Operations Research*, volume 95 of *North-Holland Mathematics Studies*, pages 229 – 256. North-Holland, 1984.

[Martin, 1975] Donald A. Martin, "Borel Determinacy," *Annals of Mathematics*, 102(2):pp. 363–371, 1975.

[Mathur *et al.*, 1998] Anmol Mathur, Ali Dasdan, and Rajesh K. Gupta, "Rate Analysis for Embedded Systems," *ACM Trans. Des. Autom. Electron. Syst.*, 3(3):408–436, 1998.

[Mattern, 1989] Friedemann Mattern, "Virtual Time and Global States of Distributed Systems," In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[Mazurkiewicz, 1987] A Mazurkiewicz, "Trace Theory," In *Advances in Petri Nets 1986, Part II on Petri Nets: Applications and Relationships to Other Models of Concurrency*, pages 279–324. Springer-Verlag New York, Inc., 1987.

[Mitchell and Sevitsky, 2007] Nick Mitchell and Gary Sevitsky, "The causes of bloat, the limits of health," In *OOPSLA*, pages 245–260, 2007.

[Mitchell *et al.*, 2006] Nick Mitchell, Gary Sevitsky, and Harini Srinivasan, "Modeling Runtime Behavior in Framework-Based Applications," In *ECOOP*, 2006.

[Mohri, 2002] Mehryar Mohri, “Semiring Frameworks and Algorithms for Shortest-distance Problems,” *J. Autom. Lang. Comb.*, 7(3):321–350, 2002.

[Moore, 1959] Edward F. Moore, “The shortest path through a maze,” In *Proceedings of the International Symposium on the Theory of Switching, and Annals of the Computation Laboratory of Harvard University*. Harvard University Press, 1959.

[Müller-Olm and Seidl, 2004] Markus Müller-Olm and Helmut Seidl, “Precise interprocedural analysis through linear algebra,” In *POPL*, pages 330–341, 2004.

[Musuvathi and Qadeer, 2007] Madanlal Musuvathi and Shaz Qadeer, “Iterative Context Bounding for Systematic Testing of Multithreaded Programs,” *SIGPLAN Not.*, 42(6):446–455, 2007.

[Musuvathi *et al.*, 2008] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu, “Finding and Reproducing Heisenbugs in Concurrent Programs,” In *OSDI*, 2008.

[Naeem and Lhoták, 2008] Nomair A. Naeem and Ondrej Lhoták, “Typestate-like analysis of multiple interacting objects,” In *OOPSLA*, 2008.

[Naeem *et al.*, 2010] Nomair A. Naeem, Ondřej Lhoták, and Jonathan Rodriguez, “Practical Extensions to the IFDS Algorithm,” CC, 2010.

[Nisan *et al.*, 2007] Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay V. Vazirani, *Algorithmic Game Theory*, Cambridge University Press, New York, NY, USA, 2007.

[Novark *et al.*, 2009] Gene Novark, Emery D. Berger, and Benjamin G. Zorn, “Efficiently and precisely locating memory leaks and bloat,” In *PLDI*, pages 397–407, 2009.

[Obdrzálek, 2003] Jan Obdrzálek, “Fast Mu-Calculus Model Checking when Tree-Width Is Bounded,” In *CAV*, 2003.

[Orlin and Ahuja, 1992] James B. Orlin and Ravindra K. Ahuja, “New scaling algorithms for the assignment and minimum mean cycle problems,” *Math. Program.*, 1992.

[Palepu *et al.*, 2017] Vijay Krishna Palepu, Guoqing Xu, and James A. Jones, “Dynamic Dependence Summaries,” *ACM Trans. Softw. Eng. Methodol.*, 25(4):30:1–30:41, 2017.

[Palis, 2004] Michael A. Palis, “Competitive Algorithms for Fine-Grain Real-Time Scheduling,” In *Proceedings of the 25th IEEE Real-Time Systems Symposium*, RTSS ’04, pages 129–138. IEEE Computer Society, 2004.

[Papadimitriou, 1979] Christos H. Papadimitriou, “Efficient search for rationals,” *Information Processing Letters*, 8(1):1 – 4, 1979.

[Papadimitriou, 1993] Christos H. Papadimitriou, *Computational Complexity*, Addison-Wesley, 1993.

[Peled, 1993] Doron Peled, “All from One, One for All: On Model Checking Using Representatives,” In *CAV*, 1993.

[Planken *et al.*] Leon R. Planken, Mathijs M. de Weerdt, and Roman P.J. van der Krogt, “Computing all-pairs shortest paths by leveraging low treewidth,” In *Proceedings of the Twenty-first International Conference on Automated Planning and Scheduling (ICAPS-11)*. AAAI Press.

[Qadeer and Rehof, 2005] Shaz Qadeer and Jakob Rehof, “Context-Bounded Model Checking of Concurrent Software,” TACAS, 2005.

[Rajkumar *et al.*, 1997] Ragunathan Rajkumar, Chen Lee, John Lehoczky, and Dan Siewiorek, “A resource allocation model for QoS management,” In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, RTSS ’97, pages 298 –307, 1997.

[Ramalingam, 1994] G. Ramalingam, “The Undecidability of Aliasing,” *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, 1994.

[Reed, 1992] Bruce A. Reed, “Finding Approximate Separators and Computing Tree Width Quickly,” In *STOC*, 1992.

[Rehof and Fähndrich, 2001] Jakob Rehof and Manuel Fähndrich, “Type-base Flow Analysis: From Polymorphic Subtyping to CFL-reachability,” In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 54–66, 2001.

[Reps, 1995a] Thomas Reps, “Shape Analysis As a Generalized Path Problem,” In *Proceedings*

of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM '95, pages 1–11. ACM, 1995.

[Reps, 1997] Thomas Reps, "Program Analysis via Graph Reachability," ILPS, 1997.

[Reps, 2000] Thomas Reps, "Undecidability of Context-sensitive Data-dependence Analysis," *ACM Trans. Program. Lang. Syst.*, 22(1):162–186, 2000.

[Reps *et al.*, 1995a] Thomas Reps, Susan Horwitz, and Mooly Sagiv, "Precise Interprocedural Dataflow Analysis via Graph Reachability," In *POPL*, 1995.

[Reps *et al.*, 1995b] Thomas Reps, Susan Horwitz, and Mooly Sagiv, "Precise Interprocedural Dataflow Analysis via Graph Reachability," In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61. ACM, 1995.

[Reps *et al.*, 1994] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay, "Speeding Up Slicing," *SIGSOFT Softw. Eng. Notes*, 19(5):11–20, 1994.

[Reps *et al.*, 2007] Thomas Reps, Akash Lal, and Nick Kidd, "Program Analysis Using Weighted Pushdown Systems," In *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*, LNCS. 2007.

[Reps *et al.*, 2005] Thomas Reps, Stefan Schwoon, Somesh Jha, and David Melski, "Weighted Pushdown Systems and Their Application to Interprocedural Dataflow Analysis," *Sci. Comput. Program.*, 2005.

[Reps, 1995b] ThomasW. Reps, "Demand Interprocedural Program Analysis Using Logic Databases," In *Applications of Logic Databases*, volume 296. 1995.

[Rice, 1953] H. G. Rice, "Classes of Recursively Enumerable Sets and Their Decision Problems," *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[Robertson and Seymour, 1995] Neil Robertson and P. D. Seymour, "Graph Minors. XIII: The Disjoint Paths Problem," *J. Comb. Theory Ser. B*, 63(1):65–110, 1995.

[Robertson and Seymour, 1984] Neil Robertson and P.D Seymour, "Graph minors. III. Planar tree-width," *Journal of Combinatorial Theory, Series B*, 1984.

[Rodríguez *et al.*, 2015] César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening, "Unfolding-based Partial Order Reduction," In *CONCUR*, 2015.

[Roy, 1959] B. Roy, "Transitivité et connexité," *C. R. Acad. Sci. Paris*, 249:216–218, 1959.

[Sagiv *et al.*, 1996] Mooly Sagiv, Thomas Reps, and Susan Horwitz, "Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation," *Theor. Comput. Sci.*, 1996.

[Samanta *et al.*, 2013] Roopsha Samanta, Jyotirmoy V. Deshmukh, and Swarat Chaudhuri, *Robustness Analysis of String Transducers*, pages 427–441, 2013.

[Samanta *et al.*, 2014] Roopsha Samanta, Oswaldo Olivo, and E. Allen Emerson, *Cost-Aware Automatic Program Repair*, pages 268–284, 2014.

[Schwoon, 2002] Stefan Schwoon, *Model-Checking Pushdown Systems*, PhD thesis, Technischen Universität München, 2002.

[Şerbănuţă *et al.*, 2013] Traian Florin Şerbănuţă, Feng Chen, and Grigore Roşu, "Maximal Causal Models for Sequentially Consistent Systems," In *RV*, 2013.

[Shacham *et al.*, 2009] Ohad Shacham, Martin T. Vechev, and Eran Yahav, "Chameleon: adaptive selection of collections," In *PLDI*, pages 408–418, 2009.

[Shang *et al.*, 2012] Lei Shang, Xinwei Xie, and Jingling Xue, "On-demand Dynamic Summary-based Points-to Analysis," In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 264–274. ACM, 2012.

[Shankar *et al.*, 2008] Ajeet Shankar, Matthew Arnold, and Rastislav Bodík, "Jolt: lightweight dynamic analysis and removal of object churn," In *OOPSLA*, 2008.

[Shapley, 1953] Lloyd S. Shapley, "Stochastic Games," *Proceedings of the National Academy of Sciences*, 39(10):1095–1100, 1953.

[Sipser, 1996] Michael Sipser, *Introduction to the Theory of Computation*, International Thomson Publishing, 1st edition, 1996.

[Sridharan and Bodík, 2006a] Manu Sridharan and Rastislav Bodík, "Refinement-based Context-sensitive Points-to Analysis for Java," *SIGPLAN Not.*, 41(6):387–400, 2006.

[Sridharan and Bodík, 2006b] Manu Sridharan and Rastislav Bodík, "Refinement-based context-sensitive points-to analysis for Java," In *PLDI*, pages 387–400, 2006.

[Sridharan *et al.*, 2013] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav, "Aliasing in Object-Oriented Programming," chapter Alias Analysis for Object-oriented Programs, pages 196–232. 2013.

[Sridharan *et al.*, 2005] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík, "Demand-driven Points-to Analysis for Java," In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 59–76. ACM, 2005.

[Strassen, 1969] Volker Strassen, "Gaussian Elimination is Not Optimal," *Numer. Math.*, 13(4):354–356, 1969.

[Suwimonteerabuth *et al.*, 2008] Dejvuth Suwimonteerabuth, Javier Esparza, and Stefan Schwoon, "Symbolic Context-Bounded Analysis of Multithreaded Java Programs," SPIN, 2008.

[Tang *et al.*, 2015] Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei, "Summary-Based Context-Sensitive Data-Dependence Analysis in Presence of Callbacks," In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 83–95, 2015.

[Tarjan, 1972] Robert Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM Journal on Computing*, 1(2):146–160, 1972.

[Tarjan, 1975] Robert E Tarjan, "Solving Path Problems on Directed Graphs.," Technical report, Stanford, CA, USA, 1975.

[Thorup, 1998] Mikkel Thorup, "All Structured Programs Have Small Tree Width and Good Register Allocation," *Information and Computation*, 1998.

[Tiwari *et al.*, 1994] Vivek Tiwari, Sharad Malik, and Andrew Wolfe, "Power analysis of embedded software: a first step towards software power minimization," *IEEE Trans. VLSI Syst.*, 2(4):437–445, 1994.

[Turing, 1936] Alan M. Turing, “On Computable Numbers, with an Application to the Entscheidungsproblem,” *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[Vallée-Rai *et al.*, 1999] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan, “Soot - a Java bytecode optimization framework,” In *CASCON ’99*. IBM Press, 1999.

[van Dijk *et al.*, 2006a] Thomas van Dijk, Jan-Pieter van den Heuvel, and Wouter Slob, “Computing treewidth with LibTW,” Technical report, University of Utrecht, 2006.

[van Dijk *et al.*, 2006b] Thomas van Dijk, Jan-Pieter van den Heuvel, and Wouter Slob, “Computing treewidth with LibTW,” Technical report, University of Utrecht, 2006.

[Vassilevska Williams and Williams, 2010] Virginia Vassilevska Williams and Ryan Williams, “Subcubic Equivalences between Path, Matrix and Triangle Problems,” In *FOCS*, pages 645–654, 2010.

[Velner, 2012] Yaron Velner, *The Complexity of Mean-Payoff Automaton Expression*, pages 390–402, 2012.

[Velner, 2014] Yaron Velner, “Finite-memory Strategy Synthesis for Robust Multidimensional Mean-payoff Objectives,” In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS, pages 79:1–79:10, 2014.

[Velner, 2015] Yaron Velner, “Robust Multidimensional Mean-Payoff Games are Undecidable,” In *Foundations of Software Science and Computation Structures: 18th International Conference, FOSSACS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*, pages 312–327, 2015.

[Velner *et al.*, 2015a] Yaron Velner, Krishnendu Chatterjee, Laurent Doyen, Thomas A. Henzinger, Alexander Rabinovich, and Jean-François Raskin, “The Complexity of Multi-Mean-Payoff and Multi-Energy Games,” *Information and Computation (to appear)*, 2015.

[Velner *et al.*, 2015b] Yaron Velner, Krishnendu Chatterjee, Laurent Doyen, Thomas A. Hen-

zinger, Alexander Rabinovich, and Jean-François Raskin, "The complexity of multi-mean-payoff and multi-energy games," *Information and Computation*, 241:177 – 196, 2015.

[Viterbi, 1967] A. Viterbi, "Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm," *IEEE Trans. Inf. Theor.*, 13(2):260–269, 1967.

[Wagner *et al.*, 1994] Tim A. Wagner, Vance Maverick, Susan L. Graham, and Michael A. Harrison, "Accurate Static Estimators for Program Optimization," In *PLDI*, 1994.

[Wang *et al.*, 2008] Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta, "Peephole Partial Order Reduction," In *TACAS*, 2008.

[Warshall, 1962] Stephen Warshall, "A Theorem on Boolean Matrices," *J. ACM*, 1962.

[Wikipedia, 2015] Wikipedia, "List of performance analysis tools," 2015.

[Wilhelm *et al.*, 2008] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

[Wu and Larus, 1994] Youfeng Wu and James R. Larus, "Static branch frequency and program profile analysis," In *MICRO 27*, pages 1–11. ACM, 1994.

[Xu *et al.*, 2010a] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky, "Finding Low-utility Data Structures," In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 174–186, 2010.

[Xu *et al.*, 2009a] Guoqing Xu, Atanas Rountev, and Manu Sridharan, *Scaling CFL-Reachability-Based Points-To Analysis Using Context-Sensitive Must-Not-Alias Analysis*, pages 98–122, Springer Berlin Heidelberg, 2009.

[Xu *et al.*, 2009b] Guoqing (Harry) Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky, "Go with the flow: profiling copies to find runtime bloat," In *PLDI*, 2009.

[Xu *et al.*, 2010b] Guoqing (Harry) Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky, "Finding low-utility data structures," In *PLDI*, 2010.

[Xu and Rountev, 2008] Guoqing (Harry) Xu and Atanas Rountev, "Precise memory leak detection for java software using container profiling," In *ICSE*, pages 151–160, 2008.

[Xu and Rountev, 2010] Guoqing (Harry) Xu and Atanas Rountev, "Detecting inefficiently-used containers to avoid bloat," In *PLDI*, pages 160–173, 2010.

[Yan *et al.*, 2011a] Dacong Yan, Guoqing Xu, and Atanas Rountev, "Demand-driven Context-sensitive Alias Analysis for Java," In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 155–165. ACM, 2011.

[Yan *et al.*, 2011b] Dacong Yan, Guoqing Xu, and Atanas Rountev, "Demand-driven Context-sensitive Alias Analysis for Java," In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA, 2011.

[Yannakakis, 1990] Mihalis Yannakakis, "Graph-theoretic Methods in Database Theory," In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS, pages 230–242, 1990.

[Yano *et al.*, 2013] Yosuke Yano, Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida, "Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths," In *CIKM'13*, pages 1601–1606, 2013.

[Yao, 1985] Andrew C. Yao, "On the Expected Performance of Path Compression Algorithms," *SIAM Journal on Computing*, 14(1):129–133, 1985.

[Young *et al.*, 1991] Neal E. Young, Robert E. Tarjant, and James B. Orlin, "Faster parametric shortest path and minimum-balance algorithms," *Networks*, 21(2):205–221, 1991.

[Yuan and Eugster, 2009] Hao Yuan and Patrick Eugster, "An Efficient Algorithm for Solving the Dyck-CFL Reachability Problem on Trees," In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP, pages 175–189, 2009.

[Yuan *et al.*, 1997] Xin Yuan, Rajiv Gupta, and Rami Melhem, "Demand-Driven Data Flow

Analysis for Communication Optimization," *Parallel Processing Letters*, 07(04):359–370, 1997.

[Zadeck, 1984] Frank Kenneth Zadeck, "Incremental Data Flow Analysis in a Structured Program Editor," SIGPLAN, 1984.

[Zhang *et al.*, 2015] Naling Zhang, Markus Kusano, and Chao Wang, "Dynamic Partial Order Reduction for Relaxed Memory Models," In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 250–259, New York, NY, USA, 2015. ACM.

[Zhang *et al.*, 2013] Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su, "Fast Algorithms for Dyck-CFL-reachability with Applications to Alias Analysis," PLDI. ACM, 2013.

[Zhang and Su, 2017] Qirun Zhang and Zhendong Su, "Context-sensitive Data-dependence Analysis via Linear Conjunctive Language Reachability," In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL, pages 344–358, 2017.

[Zhang *et al.*, 2014] Xin Zhang, Ravi Mangal, Mayur Naik, and Hongseok Yang, "Hybrid top-down and bottom-up interprocedural analysis," In *PLDI*, 2014.

[Zheng and Rugina, 2008] Xin Zheng and Radu Rugina, "Demand-driven Alias Analysis for C," In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 197–208. ACM, 2008.

[Zimmermann, 2011] U. Zimmermann, *Linear and Combinatorial Optimization in Ordered Algebraic Structures*, Annals of Discrete Mathematics. Elsevier Science, 2011.

[Zwick and Paterson, 1996] Uri Zwick and Mike Paterson, "The complexity of mean payoff games on graphs," *Theoretical Computer Science*, 158(1–2):343 – 359, 1996.