Parameterized and Algebro-geometric Advances in Static Program Analysis

by

Amir Kafshdar Goharshady

November 2020

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 in Theoretical Computer Science

IST AUSTRIA

Institute of Science and Technology

The thesis of Amir Kafshdar Goharshady, titled *"Parameterized and Algebro-geometric Advances in Static Program Analysis"*, is approved by:

Supervisor: Krishnendu Chatterjee, IST Austria, Klosterneuburg, Austria

Signature: ________________________

Committee Member: Thomas Henzinger, IST Austria, Klosterneuburg, Austria

Signature: ________________________

Committee Member: Jean-François Raskin, Université Libre de Bruxelles, Belgium

Signature: ________________________

Defense Chair: Christoph Lampert, IST Austria, Klosterneuburg, Austria

Signature: ________________________

signed page is on file

IST Austria Thesis, ISSN: 2663-337X

Abstract

In this thesis, we consider several of the most classical and fundamental problems in static analysis and formal verification, including invariant generation, reachability analysis, termination analysis of probabilistic programs, data-flow analysis, quantitative analysis of Markov chains and Markov decision processes, and the problem of data packing in cache management.

We use techniques from parameterized complexity theory, polyhedral geometry, and real algebraic geometry to significantly improve the state-of-the-art, in terms of both scalability and completeness guarantees, for the mentioned problems. In some cases, our results are the first theoretical improvements for the respective problems in two or three decades.

Dedication

To the dearly-loved Fatemeh, who now has to live with the fact that I am a doctor, too.

Acknowledgments

First and foremost, I am extremely grateful to my advisor, Krishnendu Chatterjee. As I often say in all sorts of occasions, I feel very lucky to be advised by him. He offered me unparalleled opportunities during the past five years and constantly helped me in every endeavor, not only in exploring various fields and ideas and quenching my scientific curiosity, but also in career choices, supervising interns, grant applications, conference presentations, job interviews, and every other aspect of my scientific and professional life. I owe a similar debt of gratitude to Calin Guet, Thomas Henzinger, Laura Kovács, Jean-François Raskin and Uli Wagner, who provided great inspiration and support during my whole PhD. I am also thankful to Julian Fischer and Christoph Lampert who kindly agreed to chair my qualifying exam and thesis defense sessions. Another special acknowledgment goes to Fatemeh Mohammadi for, among many other things, familiarizing me with algebraic geometry.

In the past years, two of my primary co-authors were Andreas Pavlogiannis and Hongfei Fu. We spent countless hours together figuring out algorithms, writing papers, catching deadlines, and responding to reviews in conference rebuttals. Our discussions shaped much of my research and their influence is clear in every chapter of this thesis. On the same note, I would be remiss not to mention Mohsen Alambardar, Minghzhang Huang, Rasmus Ibsen-Jensen, Ali Shakiba, and Yaron Velner.

Another category of people who should rightfully be mentioned here are those with whom I had in-depth technical discussions over the years to which I owe much of my research capability, even though our discussions were mostly on topics unrelated to this thesis. This includes Guy Avni, Sergiy Bogomolov, Mirco Giacobbe, Christian Hilbe, Josef "Pepa" Tkadlec, Viktor Toman, Raimundo Saona Urmeneta, and Đorđe Žikelić.

I had the much-appreciated opportunity to do two research internships during my PhD. The first was hosted in 2017 by César Sánchez at the IMDEA Software Institute, and the second in 2019 by Sam Blackshear and David Dill at Facebook. I wholeheartedly thank them for the pleasure of working together and hope that the resulting papers get published soon (or at least eventually). On the other hand, I also supervised a number of interns at IST Austria, who ended up contributing significantly to my research: Nastaran Okati, Arash Pourdamghani, Ali Asadi, Parsa Mirtaheri, Kiarash Mohammadi, Pratyush Agarwal, Shreya Pathak and Mehrdad Karrabi (named in the order of joining IST).

I would also like to give special thanks to Mohammadreza Hooshmandasl, who was the very first person who involved me in CS research back in 2014, and who has been a fabulous mentor and source of great wisdom in the years since. A few other people also had a very significant role in my mathematical development during high school and early years of higher education, including Saeid Alikhani, Qasem Barid Loqmani, Mohammad Farshi, Hosein Khorshidi, Madjid Mirzavaziri, Mohammad Moshtaqioun, Mansour Qadiri Herati and Abolfazl Shahzadeh-Fazeli. I am grateful to them all.

A warm thank you goes to my parents, Maryam Hajizadeh Saffar and Mohammadreza Goharshady, and to my brother, Ehsan. The latter insisted vigorously on having a dedicated sentence for himself in this acknowledgments section, so here it is.

Several of the experts and administrative people at IST Austria deserve special thanks and recognition for going way above and beyond their responsibilities to help me settle in Austria, attend conferences, go on internships, and obtain grants. This includes Paul Cech, May Chan, Vlad Cozac, Elisabeth Hacker, Louise Jottrand, Niall O'Brien, Maria-Oana Rosu, Sarah Seider and Uli Seiss.

Finally, I am grateful to Facebook, IBM, and the Austrian Academy of Sciences (ÖAW) who financially supported my PhD through their respective PhD Fellowship programs and to the Royal Commission for the Exhibition of 1851 who offered me a very generous research fellowship.

About the Author

Amir Goharshady completed a BSc in Computer Science and a BSc in Mathematics, both at the Yazd University of Iran, and obtained a Graduate Diploma in Mathematics from the University of London and an MSc in Computer Science (Systems) from the Georgia Institute of Technology. Prior to his PhD, Amir was deeply involved in Mathematics and Programming Olympiads and won a total of 14 medals in national and international competitions. He joined IST Austria in 2015.

Amir's main research interests include Parameterized Complexity, Formal Methods, Verification, and Graph Algorithms. His research has been published in the top venues of the field, such as CAV, POPL, PLDI, TOPLAS, IJCAI, ESOP, CONCUR and OOPSLA, and recognized by the Khwarizmi Prize (Iran's highest state award for young researchers), two IEEE Computer Society Best Student Paper Awards (2019 and 2020), and PhD Fellowships from Facebook, IBM, and the Austrian Academy of Sciences (ÖAW).

List of Publications

The following is a list of all publications during the PhD period. In accordance with the cultural norms in mathematics and theoretical computer science, author names appear in alphabetical order, except that co-authors who are located in countries where the order matters in evaluation/graduation decisions are put first.

[1] Asadi, A., Chatterjee, K., **Goharshady, A. K.**, Mohammadi, K., and Pavlogiannis, A. **Faster Algorithms for Quantitative Analysis of MCs and MDPs with Small Treewidth**. In *18th International Symposium on Automated Technology for Verification and Analysis* **(ATVA)**, 2020.

[2] Chatterjee, K., Fu, H., **Goharshady, A. K.**, and Goharshady, E. K. **Polynomial Invariant Generation for Non-deterministic Recursive Programs**. In *41st ACM Conference on Programming Language Design and Implementation* **(PLDI)**, 2020.

[3] Chatterjee, K., **Goharshady, A. K.**, Ibsen-Jensen, R., and Pavlogiannis, A. **Optimal and Perfectly Parallel Algorithms for On-demand Data-flow Analysis**. In *29th European Symposium on Programming* **(ESOP)**, 2020.

[4] **Goharshady, A. K.** and Mohammadi, F. **An Efficient Algorithm for Computing Network Reliability in Small Treewidth**. *Reliability Engineering and System Safety*, 2020.

[5] Chatterjee, K., **Goharshady, A. K.**, Goyal, P., Ibsen-Jensen, R., and Pavlogiannis, A. **Faster Algorithms for Dynamic Algebraic Queries in Basic RSMs with Constant Treewidth**. *ACM Transactions on Programming Languages and Systems* **(TOPLAS)**, 2019.

[6] Chatterjee, K., Fu, H., and **Goharshady, A. K.** **Non-polynomial Worst-case Analysis of Recursive Programs**. *ACM Transactions on Programming Languages and Systems* **(TOPLAS)**, 2019.

[7] Huang, M., Fu, H., Chatterjee, K., and **Goharshady, A. K.** **Modular Verification for Almost-Sure Termination of Probabilistic Programs**. In *34th ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications* **(OOPSLA)**, 2019.

[8] Wang, P., Fu, H., **Goharshady, A. K.**, Chatterjee, K., Qin, X., and Shi, W. **Cost Analysis of Nondeterministic Probabilistic Programs**. In *40th ACM Conference on Programming Language Design and Implementation* **(PLDI)**, 2019.

[9] Chatterjee, K., **Goharshady, A. K.**, and Pourdamghani, A. **Probabilistic Smart Contracts: Secure Randomness on the Blockchain**. In *IEEE International Conference on Blockchain and Cryptocurrency* **(ICBC)**, 2019.

[10] Chatterjee, K., **Goharshady, A. K.**, and Pourdamghani, A. **Hybrid Mining: Exploiting Blockchain's Computational Power for Distributed Problem Solving**. In *34th ACM Symposium on Applied Computing* **(SAC)**, 2019.

[11] Chatterjee, K., **Goharshady, A. K.**, and Goharshady, E. K. **The Treewidth of Smart Contracts**. In *34th ACM Symposium on Applied Computing* **(SAC)**, 2019.

[12] Chatterjee, K., **Goharshady, A. K.**, Okati, N., and Pavlogiannis, A. **Efficient Parameterized Algorithms for Data Packing**. In *46th ACM Symposium on Principles of Programming Languages* **(POPL)**, 2019.

[13] Chatterjee, K., **Goharshady, A. K.**, Ibsen-Jensen, R., and Velner, Y. **Ergodic Mean-payoff Games for the Analysis of Attacks in Cryptocurrencies**. In *29th International Conference on Concurrency Theory* **(CONCUR)**, 2018.

[14] Chatterjee, K., **Goharshady, A. K.**, Ibsen-Jensen, R., and Pavlogiannis, A. **Algorithms for Algebraic Path Properties in Concurrent Systems of Constant**

Treewidth Components. *ACM Transactions on Programming Languages and Systems* **(TOPLAS)**, 2018.

[15] **Goharshady, A. K.**, Behrouz, A., and Chatterjee, K. **Secure Credit Reporting on the Blockchain**. In *IEEE International Symposium on Blockchain and its Applications*, 2018.

[16] Chatterjee, K., Fu, H., **Goharshady, A. K.**, and Okati, N. **Computational Approaches for Stochastic Shortest Path on Succinct MDPs**. In *27th International Joint Conference on Artificial Intelligence* **(IJCAI)**, 2018.

[17] Chatterjee, K., **Goharshady, A. K.**, and Velner, Y. **Quantitative Analysis of Smart Contracts**. In *27th European Symposium on Programming* **(ESOP)**, 2018.

[18] Chatterjee, K., **Goharshady, A. K.**, and Pavlogiannis, A. **JTDec: A Tool for Tree Decompositions in Soot**. In *15th International Symposium on Automated Technology for Verification and Analysis* **(ATVA)**, 2017.

[19] Chatterjee, K., Fu, H., and **Goharshady, A. K. Non-polynomial Worst-case Analysis of Recursive Programs**. In *29th International Conference on Computer Aided Verification* **(CAV)**, 2017.

[20] Chatterjee, K., Fu, H., and **Goharshady, A. K. Termination Analysis of Probabilistic Programs through Positivstellensatz's**. In *28th International Conference on Computer Aided Verification* **(CAV)**, 2016.

[21] Chatterjee, K., **Goharshady, A. K.**, Ibsen-Jensen, R., and Pavlogiannis, A. **Algorithms for Algebraic Path Properties in Concurrent Systems of Constant Treewidth Components**. In *43rd ACM Symposium on Principles of Programming Languages* **(POPL)**, 2016.

Table of Contents

List of Abbreviations

a.s. Almost-sure / Almost-surely

BSCC Bottom Strongly Connected Component

CC Connected Component

CFG Control Flow Graph

FPT Fixed-Parameter Tractable / Fixed-Parameter Tractability

IRW Inductive Reachability Witness

LP Linear Programming

LRSM Linear Ranking Supermartingale

MC Markov Chain

MDP Markov Decision Process

PRSM Polynomial Ranking Supermartingale

RSM Ranking Supermartingale

SCC Strongly Connected Component

SDP Semi-definite Programming

SI Strategy Iteration

QP Quadratic Programming

UIRW Universal Inductive Reachability Witness

VI Value Iteration

1 Introduction

1.1 Prologue

Formal Verification and Safety-criticality. This thesis focuses on various aspects of formal verification. On a high level, an ultimate goal of the field of verification is to obtain efficient and automated algorithms that can formally/mathematically prove the correctness or incorrectness of a program with respect to a well-defined specification. This is especially vital given that many modern software systems perform safety-critical operations, i.e. their malfunction can cause irreparable harm, either in the form of catastrophic financial losses or even in the form of life loss. Recent examples of such malfunctions include tragedies such as the Boeing crashes in 2018-2019, in which hundreds of people were killed by a software bug, and high-profile heists such as the DAO hack of 2016, in which attackers exploited a programming error to steal more than $60,000,000 from an Ethereum smart contract. When dealing with safety-critical software, it is of utmost importance that we can formally and mathematically verify its correctness and show that no such errors can happen in any execution of the program under any circumstance. This is easier said than done. There are significant challenges in verifying real-world software, and this thesis is an attempt to handle some of these challenges. Specifically, we focus on static program analysis, i.e. approaches that aim to verify the correctness of a program by analyzing its source code without actually executing it.

The Challenge of Big Code. A primary challenge in verifying modern programs is their sheer size. Many of our modern software systems are huge. They contain millions or even billions of lines of code. For example, Facebook has a mono-repository with more than 10 million lines of inter-connected code that is written by virtually every engineer who has ever worked for the company. Google, Twitter, Microsoft, and Uber have a similar approach, in which huge amounts of code are put into a single version-controlled repository with a significant volume of code change every hour, and an almost real-time continuous integration. In case of Google, the repository is more than a billion lines long. At this scale, some of the most basic correctness checks, such as checking for absence of null pointer dereferencing, have become intractable and problems that were assumed to be "solved" for many years have to be revisited.

The Challenge of Undecidability. Another challenge, which is very familiar to all computer scientists, is the inherent undecidability of many verification problems. As early as 1936, Turing proved that halting, i.e. the problem of automatedly checking whether an input program terminates, is undecidable. This means that there are no automated algorithms that can solve this problem in finite time. Rice famously extended this undecidability result to virtually every other useful verification task, including safety and reachability.

Given the two challenges above, automated formal verification might seem hopeless, but nothing can be further from the truth. The field is actually one of the most active, useful and promising areas of theoretical computer science. In this thesis, building on the previous achievements of our field, of which a long list is available in the References section, we will study two approaches for facing the above challenges.

Parameterization. The first approach is to use tools and techniques from parameterized complexity theory. The primary idea is to consider a verification task which may be quite expensive, i.e. either NP-hard, or hard-to-approximate or even solvable in high-degree polynomial time, and then identify underlying structural properties in the input instance that can help ameliorate the high complexity, and exploit these structures for faster algorithms. This might sound like using heuristics, but the main difference is that we actually show the existence of these structural properties in real-world instances. For example, the main property exploited in this thesis is the low-treewidth property of the control-flow graphs. It is mathematically proven by Thorup that every structured program has this property. As such, faster algorithms that are obtained by assuming this property are actually proven to be applicable to our real-world programs. We use such algorithms in Chapters 3–5 to scale up analyses ranging from data-flow to cache management. The primary utility of this approach is that it leads to scalable linear-time algorithms that can easily handle systems with millions or even billions of nodes/lines of code. See Section 2.2 for a more formal treatment of parameterized algorithms.

Algebro-geometric Methods. Our second approach is to handle the undecidability challenge by focusing on specific families of programs. Choosing such families is an art and needs a careful balancing act. There is a clear trade-off between the generality and expressiveness of the chosen family on the one hand, and the complexity of verifying programs in the family on the other. In this thesis, we focus on polynomial imperative programs with real variables and non-determinism. In other words, we consider programs in which all assignments and guards consist of polynomial expressions. See Section 2.5 for a formal definition. This family covers many programs used in diverse applications ranging from AI to scientific computation to network routing. Moreover, its expressiveness is further supported by the well-known Weierstrass theorem which establishes that every continuous function can be approximated as closely as desired by a polynomial. We use tools and theorems from polyhedral and algebraic geometry (Section 2.6) in order to obtain sound and (semi-)complete algorithms for three classical problems, namely Invariant Generation (Chapter 6), Reachability (Chapter 7), and Probabilistic Termination (Chapter 8) over polynomial programs. Our algorithms reduce these classical problems to quadratic programming in case of invariants and reachability, and to semi-definite programming in case of termination. These are both well-studied optimization problems with many efficient industrial solvers. Moreover, semi-definite programming is solvable in polynomial time. As such, our approaches provide the first applicable methods with completeness guarantees for these classical problems over polynomial programs.

1.2 Outline

Chapter 2 provides preliminary tools and mathematical theorems which form the theoretical basis of our algorithms in next parts of the thesis. Specifically, it presents previously-known results in parameterized complexity, real algebraic geometry, and martingale theory. It also provides new theorems with the aim of making these results applicable in our verification use-cases. Despite its name, this chapter is probably one of the most mathematically dense parts of this thesis. The goal has been to create a clear-cut separation between the mathematical results on the one hand, and algorithmic results on the other. As such, most mathematical results are put in this chapter.

Generally speaking, the results in this thesis can be divided in two parts:

- *Parameterized Approaches:* Chapters 3–5 each provide faster parameterized algorithms for a classical problem or a family of closely-related problems in static analysis. This includes Data-flow Analyses (Chapter 3), Quantitative Analyses of Markov Chains and Markov Decision Processes (Chapter 4) and Faster Algorithms for Optimal Cache Management through Data Packing (Chapter 5). The exciting point about these chapters is that they provide fast (linear-time) parameterized algorithms that can scale up to handle huge real-world programs/instances with billions of lines/nodes.

- *Algebro-geometric Approaches:* Chapters 6–8 consider polynomial programs and focus on the following classical verification problems:

  - *Safety:* Chapter 6 considers invariant generation. Our goal in this chapter is to provide automated approaches that find over-approximations of the set of states that can be reached in executions of an input program. The simple intuition is that given a set B of buggy (undesirable) states, if we can find fine-enough over-approximations of reachable states and show that they do not intersect B, we have effectively proven the correctness of our program with respect to B, i.e. we have proven that none of the errors in B can happen in the executions of the program.

  - *Reachability:* In Chapter 7, we consider the natural dual of safety and focus on the problem of proving the incorrectness of a given program. Specifically, given a set B of buggy states, we focus on synthesizing formal proofs that B can indeed be reached in a given program.

  - *Termination:* Finally, in Chapter 8, we consider what is inarguably one of the most classical problems in all of computer science, namely termination. Given a possibly-probabilistic program as input, we focus on automatically synthesizing formal proofs that the program eventually halts, i.e. it does not run forever.

  The common thread among these three chapters is that they all use theorems from polyhedral geometry and real algebraic geometry such as positivstellensätze (positive locus theorems). This has significant benefits compared to previous methods: (i) they provide completely automated push-button solutions, (ii) they provide strong completeness guarantees, in each case showing that if a polynomial safety/reachability/termination proof exists, then our algorithms are guaranteed to find it, and finally (iii) they lead to significant speed-ups over classical solutions. For safety and reachability, we provide polynomial-time reductions to quadratic programming, which is a well-studied problem in optimization with many efficient solvers. In case of termination, we go even further and provide reductions to linear and semi-definite programming, leading to an end-to-end approach that takes polynomial-time. In contrast, previous approaches for these problems reduced them to the existential theory of reals and applied extremely expensive quantifier elimination methods. As such, when it comes to polynomial programs, for all three classical problems we are providing the first ever applicable methods with completeness guarantees.

For the sake of elegance, we present our reachability and safety results in terms of non-probabilistic programs (transition systems) with non-determinism. These results can be trivially extended to probabilistic programs, as well. However, when it comes to the problem of termination, probabilistic programs are significantly trickier and hence Chapter 8 puts them front and center. Naturally, any result obtained over the more general probabilistic framework is automatically applicable to non-probabilistic cases, as well.

The goal has been to ensure that each chapter is self-contained as much as possible. The only exception is that we avoid repeating algorithmic insights. As such, Chapter 8 is better read after Chapter 7.

1.3 Summary of Contributions

In short, the contributions in each chapter of this thesis can be summarized as follows:

- In Chapter 3, we consider the general problem of same-context interprocedural data-flow analysis. This contains well-known analyses such as null pointer, reaching definitions, live variables and dead code as special cases.

- *Theoretical Contribution.* Using parameterization by treewidth, we provide an algorithm that can answer data-flow queries in $O(1)$ after pre-processing the program in $O(n)$, where n is the number of lines of code. In contrast, the best previously-known runtime bound on the approaches for solving this problem was $O(n^2)$. Moreover, we show that our algorithms are optimal in terms of both space and time and are also perfectly parallelizable. This is the first theoretical improvement in the runtime of this widely-studied problem in more than two decades.
- *Practical Contribution.* In practice, this is the first algorithm for interprocedural data-flow analysis that can scale to handle huge code-bases with millions or billions of lines of code. Our extensive experimental results show that our algorithm consistently beats the runtimes of previous methods by several orders of magnitude.

• In Chapter 4, we consider several classical quantitative analysis problems, namely Hitting Probabilities, Discounted Sum, and Mean Payoff, over Markov Chains (MCs) and Markov Decision Processes (MDPs). We parameterize the problems based on the treewidth of the MC/MDP.

  - *Theoretical Contribution.* We provide linear-time algorithms for solving the three mentioned quantitative analyses over MCs with bounded treewidth. This is in contrast to the general case of the problem which currently takes $\Omega(n^{2.37})$, where n is the number of states in the MC. Hence, we are providing an improvement factor of $\Omega(n^{1.37})$. Using strategy iteration, we obtain the same improvement for MDPs.
  - *Practical Contribution.* We show that our approaches beat the runtimes of state-of-the-art verification and optimization tools by at least one order of magnitude over MCs/MDPs with small treewidth.

• In Chapter 5, we consider Data Packing, which is a central problem in cache management and of utmost importance in compiler optimization. The problem was notoriously difficult and known to be NP-hard and hard-to-approximate within any non-trivial factor unless P=NP.

  - *Theoretical Contribution.* We take a parameterized approach, based on the treewidth of the so-called "access hypergraphs". We show that there exists a constant q, depending on cache parameters, such that Data Packing admits a linear-time parameterized algorithm if the access hypergraph of order q has constant treewidth. Despite the fact that Data Packing was studied since the 1980s, this is the first ever positive theoretical result and exact algorithm for Data Packing. All previous results were either hardness results or heuristics. Moreover, we significantly enrich the fine-grained complexity landscape of Data Packing by providing even stronger hardness results. See Figure 5.4.
  - *Practical Contribution.* We provide extensive experimental results, showing that over benchmarks from a variety of commonly-used algorithms, our approach leads to between 15 to 31 percent fewer cache misses. This is a huge improvement and extremely important given the considerable role of cache misses in runtime overheads of all programs.

- In Chapter 6, we consider the classical problem of Invariant Generation over polynomial programs. We focus on generating inductive invariants that are conjunctions of polynomial inequalities.
  - *Theoretical Contribution.* We provide a polynomial-time sound and semi-complete reduction from the invariant generation problem to quadratic programming. The reduction uses tools from real algebraic geometry, specifically Putinar's Positivstellensatz, and leads to a completely automated push-button approach for invariant generation. Notably, it provides completeness guarantees without resorting to the existential theory of the reals or quantifier elimination. Moreover, the overall runtime of our approach is subexponential, beating not only the previous complete methods for polynomial programs, but also those of the much more special case of linear/affine programs.
  - *Practical Contribution.* We provide experimental results on a variety of benchmarks from the literature demonstrating that our approach is the first applicable method with completeness guarantees. Specifically, there are instances that could

  not be handled by any previous incomplete approach. Moreover, the only previous complete approach for this problem cannot even handle toy programs such as our running example.

- In Chapter 7, we turn our focus to the problem of Reachability, which is classical in verification and the dual of invariant generation.

  – *Theoretical Contribution.* We define the novel notion of Inductive Reachability Witnesses (IRWs) which serve as succinct inductive proofs of reachability. We then focus on synthesizing linear and polynomial IRWs and obtain polynomial-time reductions to quadratic programming. Our main mathematical tools in these reductions are Farkas' Lemma, Positivstellensätze, and Hilbert's Strong Nullstellensatz. Just as in the previous case, our approaches provide completeness guarantees without using quantifier elimination.

  – *Practical Contribution.* We provide experimental results over standard benchmarks and show that, due to our completeness guarantees, our approach can handle programs and prove reachability properties that were beyond the reach of all previous methods. Specifically, it is noteworthy that our approach can easily prove reachability by long paths and identify deep bugs.

- Finally, in Chapter 8, we consider the fundamental problem of proving termination. In this chapter, we focus on polynomial programs with both probabilistic behavior and non-determinism.

  – *Theoretical Contribution.* We define the notion of Polynomial Ranking Supermartingale (PRSM) and show that it serves as a sound proof method for both almost-sure and finite termination. We also show that difference-bounded PRSMs can be used for obtaining concentration bounds on the termination time of a program. Finally, we provide sound and semi-complete approaches for synthesizing PRSMs using the positivstellensätze of Putinar and Schmüdgen. Unlike the previous cases, we provide a polynomial-time reduction to semi-definite programming.

This leads to a completely automated push-button approach for termination analysis that runs in polynomial time.

– *Practical Contribution.* We provide experimental results demonstrating that our approach is able to prove termination and synthesize bounds on the expected runtimes of various probabilistic programs with different types of non-linear behavior.

1.4 Awards

The research presented in this thesis has won a number of awards, including an IBM PhD Fellowship, a Facebook PhD Fellowship, and a DOC Fellowship of the Austrian Academy of Sciences (ÖAW). The works on polynomial programs have led to a Khwarizmi research prize, which is Iran's highest state honor for young researchers. Chapters 5 and 8 have each won an IEEE Computer Society Best Student Paper Award (Lance Stafford Larson Prize) in 2019 and 2020, respectively. Chapter 3 has led to a Research Fellowship of the Royal Society for the Exhibition of 1851 and has been nominated for an EATCS[*] Best Paper Award at ESOP.

[*]European Association for Theoretical Computer Science

2

Preliminaries

2.1 Notation

Throughout this document, we use the following notation:

Sets. We use $\mathbb{Z}, \mathbb{N}, \mathbb{Z}^{\geq 0}, \mathbb{Q}, \mathbb{R}, \mathbb{R}^{\geq 0}$ to denote the sets of integers, positive integers, non-negative integers, rational numbers, real numbers, and non-negative real numbers respectively.

Graphs. A graph is a pair $G = (V, E)$ in which V is a set of vertices and E is a multi-set of edges. Each edge is either *undirected* in which case it is a set $\{u, v\} \subseteq V$, or *directed*, in which case it is a pair $(u, v) \in V \times V$ of vertices. Unless otherwise stated, we allow self-loops and multiple edges between the same pair of vertices. A graph is undirected (resp. directed) if all of its edges are undirected (resp. directed). Given a graph $G = (V, E)$, its underlying undirected graph is defined as $\underline{G} = (V, \underline{E})$, in which

$$\underline{E} := \{\{u, v\} \in V \times V \mid \{u, v\} \in E \ \vee \ (u, v) \in E\}.$$

For each vertex $u \in V$, we define the neighbors, successors and predecessors of u as follows:

$$\mathrm{N}(u) := \{v \in V \mid \{u, v\} \in E\},$$

$$\mathrm{Succ}(u) := \{v \in V \mid (u, v) \in E\},$$

$$\text{Pred}(u) := \{v \in V \mid (v, u) \in E\}.$$

A *path* of length n in G is a sequence $\langle v_0, v_1, \dots, v_n \rangle \in V^*$ in which we have $\{v_i, v_j\} \in E$ for every $0 \leq i < n$. A *directed* path is a sequence $\langle v_0, v_1, \dots, v_n \rangle \in V^*$ such that $(v_i, v_j) \in E$ for every $0 \leq i < n$. A *simple* (directed) path is a (directed) path in which each vertex appears at most once. We write $v_0 \pi v_n$ to denote the existence of a path from v_0 to v_n and $v_0 \rightsquigarrow v_n$ to denote the existence of a directed path. The *distance* between u and v (resp. from u to v) is the length of the shortest path (resp. directed path) starting at u and ending at v. We denote it by $\text{d}_G(u, v)$ (resp. $\vec{\text{d}}_G(u, v)$). We drop the subscript G when it is clear from the context. In an undirected graph $G = (V, E)$, we define the *connected component* of a vertex $u \in V$ as $\text{CC}(u) := \{v \in V \mid u \pi v\}$. Similarly, in a directed graph G, we define the *strongly connected component* of $u \in V$ as $\text{SCC}(u) := \{v \in V \mid u \rightsquigarrow v \wedge v \rightsquigarrow u\}$. A graph is (strongly) connected if it has exactly one (strongly) connected component.

Trees and Directed Trees. A *tree* is a connected undirected graph $T = (V_T, E_T)$ in which there is a unique simple path between every pair of vertices. A *directed* tree is a directed graph $T = (V_T, E_T)$ together with a distinguished *root* vertex $r \in V_t$ such that:

- Its underlying graph $\underline{T}$ is a tree, and
- For every $(u, v) \in E_T$ we have $\text{d}_{\underline{T}}(r, u) = 1 + \text{d}_{\underline{T}}(r, v)$.

If $(u, v) \in E_T$, we say that v is the *parent* of u and u is a *child* of v. Moreover, u is an *ancestor* of v if $v \rightsquigarrow u$. In this case, v is a *descendant* of u. Note that each vertex is an ancestor and descendant of itself. We say that a vertex $l \in V_T$ is a leaf if it has no children.

Polynomials. Given a set X of variables, we denote by $\mathbb{R}[X]$ the set of polynomials over X with real coefficients. For a polynomial $p \in \mathbb{R}[X]$, we use $\deg(p)$ to denote its degree.

2.2 Parameterized Complexity and FPT

Several of the algorithms presented in this thesis are parameterized and exploit the underlying structural properties of their input instances to obtain the solution faster. As such, we now provide a short overview of parameterized problems and Fixed-Parameter Tractability (FPT). For a more detailed treatment, see [Downey and Fellows, 2012].

The idea behind parameterized complexity is to study the runtime of algorithms not only based on the size of their input, but also based on an additional parameter k. The parameter k can basically be anything. It might be part of the input, the output, or some structural property of the instance under study. We now provide a formal definition:

Parameterized Problems [Downey and Fellows, 2012]. Given a finite alphabet Σ, a *parameterized instance* is a pair $(s, k) \in \Sigma^* \times \mathbb{Z}^{\geq 0}$ consisting of an input string s and a non-negative integer k, which is called the "parameter". A *parameterized language* or *parameterized problem* is simply a set $L \subseteq \Sigma^* \times \mathbb{Z}^{\geq 0}$ of parameterized instances. As in classical complexity theory, a terminating *parameterized algorithm* A decides L iff it gets as input a parameterized instance (s, k) and accepts if and only if $(s, k) \in L$.

XP [Downey and Fellows, 2012]. A parameterized problem L is in XP (is slicewise-polynomial) if there exists a parameterized algorithm that decides L in time $O(n^{f(k)})$, where n is the length of the input string, k is the parameter, and f is any computable function.

Intuitively, if L is in XP, then it can be solved efficiently, i.e. in polynomial time, over instances that have a small parameter.

Example 2.1. *Consider the problem L of deciding whether a graph $G = (V, E)$ with n vertices and m edges has a vertex cover of size at most k. A vertex cover is a set S of vertices, such that each edge has at least one of its endpoints in S. This is a classical NP-complete problem. However, if we consider k as the parameter, we can simply obtain an algorithm that goes through all possible $\binom{n}{k}$ combinations of vertices and checks whether they are vertex covers. This takes $O(m \cdot n^k)$ time and is hence an XP algorithm. Therefore, instances in which the parameter k is small can be solved in polynomial time. However, note that the degree of the polynomial depends on k. Hence, the scalability of our algorithm is seriously hindered even with small increases in k. This discussion directly leads to the following notion:*

FPT [Downey and Fellows, 2012]. A parameterized problem L is Fixed-Parameter Tractable (FPT) if there exists a parameterized algorithm that decides L in time $O(n^c \cdot f(k))$, where n

is the size of the input, k is the parameter, f is an arbitrary computable function, and c is a constant not depending on either n or k.

Example 2.2. *Consider the same problem L as in Example 2.1. Note that for every edge $\{u, v\} \in E$, at least one of its endpoints u or v must be included in the vertex cover S. Moreover, if we know that a vertex is included in S, we can remove all of its adjacent edges. In other words, for any edge $\{u, v\} \in E$, we have*

$$(G, k) \in L \Leftrightarrow (G[V \setminus \{u\}], k-1) \in L \ \vee \ (G[V \setminus \{v\}], k-1) \in L.$$

Moreover, we know that $(G, 0) \in L$ iff G has no edges. This directly leads to a recursive algorithm that at each step chooses an arbitrary edge of the graph and recurses on removing either of its endpoints. As base cases, it accepts if the graph runs out of edges, and rejects if the parameter k becomes 0 but there are more edges remaining. It is easy to see that our algorithm has a runtime of $O((n + m) \cdot 2^k)$ and is hence an FPT algorithm. Note that for every fixed value of the parameter k, this algorithm solves the problem in linear time.

2.3 Tree Decompositions and Treewidth

In parameterized complexity, *Treewidth* [Robertson and Seymour, 1984] is one of the most widely-used parameters for graph problems. It is a measure of tree-likeness. Only trees and forests have a treewidth of 1. Moreover, if a graph G has treewidth t, then it can be "decomposed" into "bags" of vertices, each of size at most $t + 1$, such that the bags themselves are connected in a tree-like manner. If t is small, this enables us to perform bottom-up dynamic programming on G in a similar way to trees. Given this insight, the importance of treewidth comes from the fact that many NP-hard graph problems are fixed-parameter tractable with respect to the treewidth of their input instance [Bodlaender, 1994]. In other words, they are solvable in polynomial, even linear, time when the input is restricted to graphs with bounded treewidth. Moreover, bounded-treewidth graphs contain many well-studied families as special cases [Bodlaender, 1998]. This includes cacti, series-parallel graphs, outer-planar graphs, and, crucial to our results in the next chapter, control-flow graphs of structured programs [Thorup, 1998].

In this section, we provide a quick overview of tree decompositions and treewidth. For an in-depth treatment see [Cygan et al., 2015]. In the definitions below, we assume that all of our graphs are undirected. For directed/mixed graphs $G = (V, E)$, we use their underlying graph $\underline{G} = (V, \underline{E})$ instead.

Tree Decompositions. Given a graph $G = (V, E)$, a *tree decomposition* of G is a tree $T = (V_T, E_T)$ together with a sequence $\langle B_i \rangle_{i \in V_T}$ of subsets of vertices of G associated with each node $i \in V_T$ of the tree, such that the following conditions are met:

(i) Every vertex appears in some B_i, i.e. $\bigcup_{i \in V_T} B_i = V$;

(ii) Every edge appears in some B_i, i.e. $\forall e \in \underline{E} \;\; \exists i \in V_T \;\; e \subseteq B_i$. Equivalently, for every edge, there exists a B_i that contains both its endpoints.

(iii) For every vertex $v \in V$, the set $T_v = \{i \in V_T \mid v \in B_i\}$ of all nodes i of the tree T that contain v in their corresponding B_i, forms a connected subtree of T. Equivalently, if $i, j, k \in V_T$ and i is on the unique path between j and k in T, then $B_j \cap B_k \subseteq B_i$.

To avoid confusion, we reserve the word "vertex" for vertices of G and use the word "node" for vertices of T. Moreover, we call each B_i a "bag".

Treewidth. The width of a tree decomposition $(T, \langle B_i \rangle)$ is the size of its largest bag minus one, i.e. $\max_{i \in V_T} |B_i| - 1$. The treewidth of a graph G is the smallest width among all tree decompositions of G and is denoted $\mathrm{tw}(G)$.

Example 2.3. *Figure 2.1 shows a graph G and a tree decomposition of G. This tree decomposition has a width of* 2 *and is optimal. Hence, the treewidth of G is 2.*

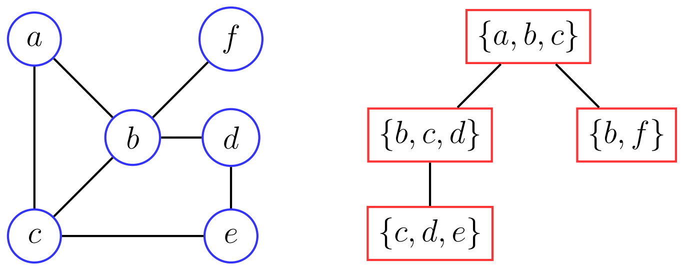


Figure 2.1: A graph G and one of its optimal tree decompositions $(T, \langle B_i \rangle)$.

Separation. The key structural property that we exploit in low-treewidth graphs is a separation property. Let $G = (V, E)$ and $V_1, V_2 \subseteq V$. The pair (V_1, V_2) is called a *separation* of G if (i) $V_1 \cup V_2 = V$, and (ii) no edge connects a vertex in $V_1 \setminus V_2$ to a vertex in $V_2 \setminus V_1$ or vice versa. If (V_1, V_2) is a separation, the set $V_1 \cap V_2$ is called a *separator*. The intuition behind this definition is that any path between V_1 and V_2 has to go through at least one of the vertices in the separator $V_1 \cap V_2$.

Example 2.4. *Figure 2.2 shows a separation of a graph G into (V_1, V_2). Note that any path from V_1 to V_2 has to go through their separator/intersection.*

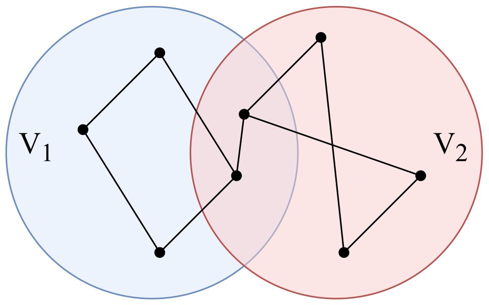


Figure 2.2: A separation of a graph G into two parts: V_1 and V_2.

The following lemma states the fundamental separation property of tree decompositions.

Lemma 2.1 (Separation Property [Cygan et al., 2015]). *Let $(T, \langle B_i \rangle)$ be a tree decomposition of $G = (V, E)$ and $e = (i, j) \in E_T$. If we remove e, the tree T breaks into two connected components, T^i and T^j, respectively containing i and j. Let $V_i = \bigcup_{i' \in T^i} B_{i'}$ and $V_j = \bigcup_{j' \in T^j} B_{j'}$. Then (V_i, V_j) is a separation of G and its corresponding separator is $V_i \cap V_j = B_i \cap B_j$.*

Informally, the lemma above means that any path in the graph G can be traced in the tree decomposition T by two types of moves: (i) going to an adjacent vertex in the same bag, or (ii) going to the same vertex in an adjacent bag.

Example 2.5. *Consider the same graph and tree decomposition as in Figure 2.1. We can trace the path $\langle f, b, c, e\rangle$ of G in the tree decomposition. This is illustrated in Figure 2.3 in which the subscripts show the trace, i.e. $\langle f_1, b_2, b_3, c_4, c_5, c_6, e_7\rangle$. Note that each move is either to an adjacent vertex in the same bag, e.g. b_3 to c_4, or to the same vertex in an adjacent bag, e.g. c_4 to c_5.*

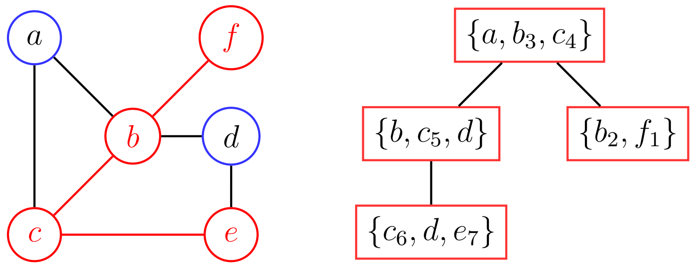


Figure 2.3: A path in a graph G and its trace in the tree decomposition.

To simplify the algorithms that exploit tree decompositions, we now define the notions of labeling and nice tree decomposition.

Vertex-Nice Tree Decompositions. A *vertex-nice* (or simply *nice*) tree decomposition of a graph G is a directed tree decomposition $(T, \langle B_i\rangle)$ in which a specific node $r \in V_T$ is designated at root and every node $i \in V_T$ is "labeled" by a subgraph G_i of G such that the following rules are obeyed:

1. If i is a leaf in T, then $B_i = \emptyset$ and $G_i = (\emptyset, \emptyset)$.

2. Otherwise, i satisfies one of the following cases:

   - *Join Node.* The node i has two children, i_1 and i_2, $B_i = B_{i_1} = B_{i_2}$ and $G_i = G_{i_1} \cup G_{i_2}$.

   - *Introduce Node.* The node i has a single child i_1 and $B_i = B_{i_1} \cup \{v\}$ for some vertex $v \notin B_{i_1}$. In this case, we say that i introduces v. Moreover, we have

$G_i = G_{i_1} \cup \{v\} \cup \{e \in \underline{E} \mid e \subseteq B_i\}$. In other words, the label graph G_i is obtained from G_{i_1} by adding the vertex v and all the edges that go between v and vertices in the bag B_i.

- *Forget Vertex Node.* The node i has a single child i_1 and $B_i = B_{i_1} \setminus \{v\}$ for some vertex $v \in B_{i_1}$. We say that i forgets v. Moreover, $G_i = G_{i_1}$.

Intuitively, the label graph G_i is the subgraph of G consisting of all the vertices that are introduced in the subtree of T rooted at i and all edges between these vertices.

Example 2.6. *Figure 2.4 shows a nice tree decomposition of the graph G in Figure 2.1. This tree decomposition was obtained by simply adding intermediate transition nodes to the tree decomposition of Figure 2.1. The leftmost node is the root. Leaf bags are shown in dotted lines. Introduce bags are shown in blue, forget bags in red, and join bags in black.*

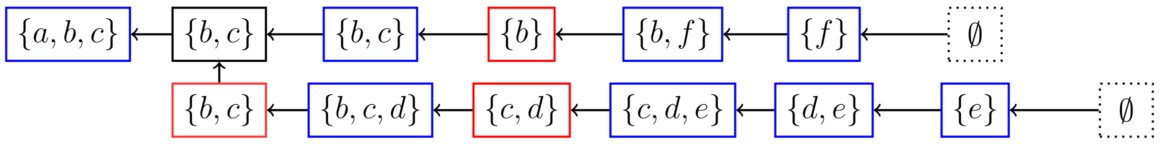

$$\begin{tikzcd}
\{a,b,c\} & \{b,c\} \arrow[l] & \{b,c\} \arrow[l] & \{b\} \arrow[l] & \{b,f\} \arrow[l] & \{f\} \arrow[l] & \emptyset \arrow[l] & \\
& \{b,c\} \arrow[u] & \{b,c,d\} \arrow[l] & \{c,d\} \arrow[l] & \{c,d,e\} \arrow[l] & \{d,e\} \arrow[l] & \{e\} \arrow[l] & \emptyset \arrow[l]
\end{tikzcd}$$

Figure 2.4: A nice tree decomposition of the graph in Figure 2.1.

Intuitively, in a nice tree decomposition, each bag differs from its neighbors in at most one vertex. This enables dynamic programming routines to re-use much of the values computed in children of a bag when computing new values in that bag. However, sometimes even a nice tree decomposition changes the label graph too fast. This is especially the case when considering introduce nodes, that suddenly add a vertex and possibly many edges. In these cases, we need the more refined notion of edge-nice decompositions, that introduce vertices and edges separately:

Edge-Nice Tree Decompositions. An *edge-nice* tree decomposition of a graph G is a directed tree decomposition $(T, \langle B_i \rangle)$ in which a specific node $r \in V_T$ is designated as the root and every node $i \in V_T$ is labeled by a subgraph G_i of G, such that the following rules are obeyed:

1. If i is a leaf in T, then $B_i = \emptyset$ and $G_i = (\emptyset, \emptyset)$.

2. Otherwise, i satisfies one of the following cases:

   - *Join Node.* The node i has two children, i_1 and i_2, $B_i = B_{i_1} = B_{i_2}$ and $G_i = G_{i_1} \cup G_{i_2}$.

   - *Introduce Vertex Node.* The node i has a single child i_1 and $B_i = B_{i_1} \cup \{v\}$ for some vertex $v \notin B_{i_1}$. In this case, we say that i introduces v. Moreover, $G_i = G_{i_1} \cup \{v\}$, i.e. G_i is the graph resulting from adding v as an isolated vertex to G_{i_1}.

   - *Introduce Edge Node.* Similar to the previous case, i has a single child i_1. This time, $B_i = B_{i_1}$, but G_i is defined as the graph resulting from adding a new edge e to G_{i_1}. All endpoints of e must be present in B_i. We say that i introduces e.

   - *Forget Vertex Node.* The node i has a single child i_1 and $B_i = B_{i_1} \setminus \{v\}$ for some vertex $v \in B_{i_1}$. We say that i forgets v. Moreover, $G_i = G_{i_1}$.

3. Each edge is introduced exactly once.

Intuitively, the label graph G_i is the subgraph of G consisting of all the vertices and edges that are introduced in the subtree of T rooted at i.

Remark 2.1. *The notion of label subgraphs G_i is solely defined for theoretical purposes and used in our proofs of correctness. In practice, our implementations always avoid the overhead of constructing G_i's.*

Example 2.7. *Figure 2.5 shows an edge-nice tree decomposition of the graph G of Figure 2.1. The leftmost node is the root. In each node i of the tree, its label subgraph G_i is illustrated and the vertices of the bag B_i are shown in red. Intuitively, an edge-nice tree decomposition constructs the graph in small increments and the bag B_i contains the vertices that can participate in the incremental change.*

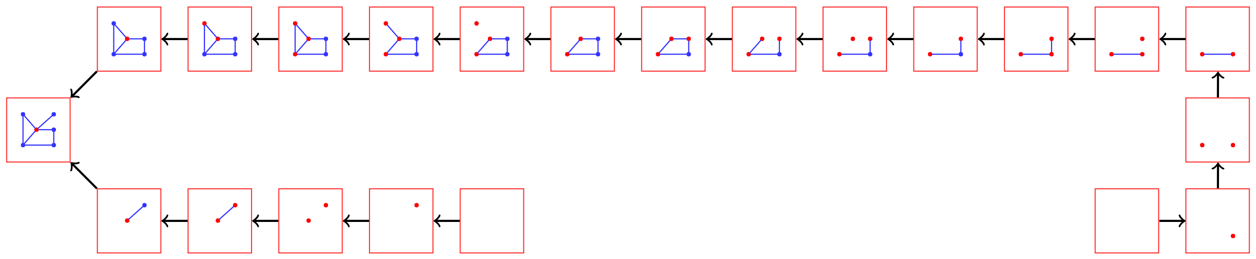


Figure 2.5: An edge-nice tree decomposition of the graph in Figure 2.1.

Computing the Treewidth. For general graphs, the problem of computing the treewidth is NP-hard [Arnborg et al., 1987]. Nevertheless, it is fixed-parameter tractable based on the treewidth itself. Indeed, [Bodlaender, 1996] provides a linear-time FPT algorithm for this problem. One of our major use-cases for treewidth is as a parameter of the control-flow graphs of structured programs. In this case, a linear-time algorithm was provided in [Thorup, 1998], which produces a tree decomposition of width at most 7 by a single pass over the program parse tree. As both of these algorithms run in linear time, they naturally lead to tree decompositions with linearly many bags. Finally, it is well-known that any tree decomposition can be made (edge-)nice in linear time. This is achieved by adding intermediate bags as in Figure 2.4. See [Cygan et al., 2015] for more details. Based on these points, whenever we consider treewidth-based parameterized algorithms, we also assume that a linearly-sized nice or edge-nice tree decomposition is given as part of the input.

2.4 Stochastic Processes and Martingales

In this section, we provide an overview of some basic tools from probability theory that will be used in the next chapters. Please refer to [Williams, 1991] for a more formal and detailed treatment of the material in this section.

Probability Distributions. A *discrete probability distribution* over a countable set U is a function $p : U \to [0,1]$ such that $\sum_{u \in U} p(u) = 1$. The *support* of p is defined as $\mathrm{supp}(p) := \{u \in U \mid p(u) > 0\}$.

Probability Spaces. A *probability space* is a triple $(\Omega, \mathcal{F}, \mathbb{P})$, where Ω is a non-empty set (called the *sample space*), $\mathcal{F}$ is a *σ-algebra* over Ω (i.e. a collection of subsets of Ω that contains the empty set $\emptyset$ and is closed under complementation and countable union) and $\mathbb{P}$ is a *probability measure* on $\mathcal{F}$, i.e. a function $\mathbb{P}\colon \mathcal{F} \to [0,1]$ such that (i) $\mathbb{P}(\Omega) = 1$ and (ii) for all set-sequences $A_1, A_2, \dots \in \mathcal{F}$ that are pairwise-disjoint (i.e. $A_i \cap A_j = \emptyset$ whenever $i \neq j$) it holds that $\sum_{i=1}^{\infty} \mathbb{P}(A_i) = \mathbb{P}\left(\bigcup_{i=1}^{\infty} A_i\right)$. Elements of $\mathcal{F}$ are called *events*. An event $A \in \mathcal{F}$ holds *almost-surely* (a.s.) if $\mathbb{P}(A) = 1$.

Random Variables. A *random variable* X from a probability space $(\Omega, \mathcal{F}, \mathbb{P})$ is an $\mathcal{F}$-measurable function $X\colon \Omega \to \mathbb{R} \cup \{-\infty, +\infty\}$, i.e. a function such that for all $d \in \mathbb{R} \cup \{-\infty, +\infty\}$, the set $\{\omega \in \Omega \mid X(\omega) < d\}$ belongs to $\mathcal{F}$. X is *bounded* if there exists a real number $M > 0$ such that for all $\omega \in \Omega$, we have $X(\omega) \in \mathbb{R}$ and $|X(\omega)| \leq M$.

Expectation. The *expected value* of a random variable X from a probability space $(\Omega, \mathcal{F}, \mathbb{P})$, denoted by $\mathbb{E}(X)$, is defined as the Lebesgue integral of X with respect to $\mathbb{P}$, i.e. $\mathbb{E}(X) := \int X \, \mathrm{d}\mathbb{P}$. The precise definition of Lebesgue integral is somewhat technical and is omitted here (See [Williams, 1991, Chapter 5] for a formal definition). If *range* $X = \{d_0, d_1, \dots\}$ is countable, then we have $\mathbb{E}(X) = \sum_{k=0}^{\infty} d_k \cdot \mathbb{P}(X = d_k)$.

Filtrations and Stopping Times. A *filtration* of a probability space $(\Omega, \mathcal{F}, \mathbb{P})$ is an infinite sequence $\{\mathcal{F}_n\}_{n \in \mathbb{Z}^{\geq 0}}$ of σ-algebras over Ω such that $\mathcal{F}_n \subseteq \mathcal{F}_{n+1} \subseteq \mathcal{F}$ for all $n \in \mathbb{Z}^{\geq 0}$. Intuitively, a filtration models the information available at any given point of time. A *stopping time* with respect to $\{\mathcal{F}_n\}_{n \in \mathbb{Z}^{\geq 0}}$ is a random variable $R : \Omega \to \mathbb{Z}^{\geq 0} \cup \{+\infty\}$ such that for every $n \in \mathbb{Z}^{\geq 0}$, the event $R \leq n$ belongs to $\mathcal{F}_n$.

Conditional Expectation. Let X be any random variable from a probability space $(\Omega, \mathcal{F}, \mathbb{P})$ such that $\mathbb{E}(|X|) < +\infty$. Then, given any σ-algebra $\mathcal{G} \subseteq \mathcal{F}$, there exists a random variable (from $(\Omega, \mathcal{F}, \mathbb{P})$), denoted by $\mathbb{E}(X|\mathcal{G})$, such that:

(E1) $\mathbb{E}(X|\mathcal{G})$ is $\mathcal{G}$-measurable, and

(E2) $\mathbb{E}\left(|\mathbb{E}(X|\mathcal{G})|\right) < +\infty$, and

(E3) for all $A \in \mathcal{G}$, we have $\int_A \mathbb{E}(X|\mathcal{G}) \, \mathrm{d}\mathbb{P} = \int_A X \, \mathrm{d}\mathbb{P}$.

The random variable $\mathbb{E}(X|\mathcal{G})$ is called the *conditional expectation* of X given $\mathcal{G}$. The random variable $\mathbb{E}(X|\mathcal{G})$ is a.s. unique in the sense that if Y is another random variable satisfying (E1)–(E3), then $\mathbb{P}(Y = \mathbb{E}(X|\mathcal{G})) = 1$. We refer to [Williams, 1991, Chapter 9] for details. Intuitively, $\mathbb{E}(X|\mathcal{G})$ is the expectation of X, when assuming the information in $\mathcal{G}$.

Discrete-Time Stochastic Processes. A *discrete-time stochastic process* is a sequence $\Gamma = \{X_n\}_{n\in\mathbb{Z}^{\geq 0}}$ of random variables where X_n's are all from some probability space $(\Omega, \mathcal{F}, \mathbb{P})$. The process Γ is *adapted to* a filtration $\{\mathcal{F}_n\}_{n\in\mathbb{Z}^{\geq 0}}$ if for all $n \in \mathbb{Z}^{\geq 0}$, X_n is $\mathcal{F}_n$-measurable. Intuitively, the random variable X_i models some value at the i-th step of the process.

Difference-Boundedness. A discrete-time stochastic process $\Gamma = \{X_n\}_{n\in\mathbb{Z}^{\geq 0}}$ adapted to a filtration $\{\mathcal{F}_n\}_{n\in\mathbb{Z}^{\geq 0}}$ is *difference-bounded* if there exists a $c \in (0, +\infty)$ such that for all $n \in \mathbb{Z}^{\geq 0}$, $|X_{n+1} - X_n| \leq c$ almost-surely.

Martingales and Supermartingales. A discrete-time stochastic process $\Gamma = \{X_n\}_{n\in\mathbb{Z}^{\geq 0}}$ adapted to a filtration $\{\mathcal{F}_n\}_{n\in\mathbb{Z}^{\geq 0}}$ is a *martingale* (resp. *supermartingale*) if for every $n \in \mathbb{Z}^{\geq 0}$, $\mathbb{E}(|X_n|) < +\infty$ and it holds a.s. that $\mathbb{E}(X_{n+1}|\mathcal{F}_n) = X_n$ (resp. $\mathbb{E}(X_{n+1}|\mathcal{F}_n) \leq X_n$).

Intuitively, a martingale (resp. supermartingale) is a discrete-time stochastic process in which for an observer who has seen the values of $X_0, \ldots, X_n$, the expected value at the next step, i.e. $\mathbb{E}(X_{n+1}|\mathcal{F}_n)$, is equal to (resp. no more than) the last observed value X_n. Also, note that in a martingale the observed values for $X_0, \ldots, X_{n-1}$ do not matter given that $\mathbb{E}(X_{n+1}|\mathcal{F}_n) = X_n$. In contrast, in a supermartingale, the only requirement is that $\mathbb{E}(X_{n+1}|\mathcal{F}_n) \leq X_n$ and hence $\mathbb{E}(X_{n+1}|\mathcal{F}_n)$ may depend on $X_0, \ldots, X_{n-1}$. Also, note that $\mathcal{F}_n$ might contain more information than just the observations of X_i's.

Example 2.8. *Consider an unbiased and discrete random walk, in which we start at a position X_0, and at each second walk one step to either left or right with equal probability. Let X_n denote our position after n seconds. It is easy to verify that $\mathbb{E}[X_{n+1}|X_0, \ldots, X_n] = \frac{1}{2} \cdot (X_n - 1) + \frac{1}{2} \cdot (X_n + 1) = X_n$. Hence, this random walk is a martingale. Note that by definition, every martingale is also a supermartingale. As another example, consider the classical gambler's ruin: a gambler starts with Y_0 dollars of money and bets continuously until he loses all of his money. If the bets are unfair, i.e. the expected value of his money* *after a bet is less than its expected value before the bet, then the sequence $\{Y_n\}_{n\in\mathbb{Z}^{\geq 0}}$ is a supermartingale. In this case, Y_n is the gambler's total money after n bets. On the other hand, if the bets are fair, then $\{Y_n\}$ is a martingale.*

Example 2.9 (Pólya's Urn [Mahmoud, 2008])**.** *As a more interesting example, consider an urn that initially contains R_0 red and B_0 blue marbles ($R_0 + B_0 > 0$). At each step, we take one marble from the urn, chosen uniformly at random, look at its color and then add two marbles of that color to the urn. Let B_n, R_n and M_n respectively be the number of red, blue and all marbles after n steps. Also, let $\beta_n = \frac{B_n}{M_n}$ and $\rho_n = \frac{R_n}{M_n}$ be the proportion of marbles that are blue (resp. red) after n steps. Let $\mathcal{F}_n$ model the observations until the n-th step. The process described above leads to the following equations:*

$$M_{n+1} = 1 + M_n,$$

$$\mathbb{E}(B_{n+1}|\mathcal{F}_n) = \mathbb{E}(B_{n+1}|B_1,\ldots,B_n) = \frac{B_n}{M_n}\cdot(B_n+1) + \frac{R_n}{M_n}\cdot B_n,$$

$$\mathbb{E}(R_{n+1}|\mathcal{F}_n) = \mathbb{E}(R_{n+1}|B_1,\ldots,B_n) = \frac{R_n}{M_n}\cdot(R_n+1) + \frac{B_n}{M_n}\cdot R_n.$$

Note that we did not need to care about observing R_i's, M_i's, β_i's or ρ_i's, because they can be uniquely computed in terms of B_i's. More generally, an observer can observe only B_i's, or only R_i's, or only β_i's or ρ_i's and can then compute the rest using this information. Based on the equations above, we have:

$$\mathbb{E}(\beta_{n+1}|\mathcal{F}_n) = \frac{B_n}{M_n}\cdot\frac{B_n+1}{M_n+1} + \frac{M_n-B_n}{M_n}\cdot\frac{B_n}{M_n+1} = \frac{B_n}{M_n} = \beta_n,$$

$$\mathbb{E}(\rho_{n+1}|\mathcal{F}_n) = \frac{R_n}{M_n}\cdot\frac{R_n+1}{M_n+1} + \frac{M_n-R_n}{M_n}\cdot\frac{R_n}{M_n+1} = \frac{R_n}{M_n} = \rho_n.$$

Hence, both $\{\beta_n\}_{n\in\mathbb{Z}^{\geq 0}}$ and $\{\rho_n\}_{n\in\mathbb{Z}^{\geq 0}}$ are martingales. Informally, this means that the expected proportion of blue marbles in the next step is exactly equal to their observed proportion in the current step. This might be counter-intuitive. For example, consider a state where 99% of the marbles are blue. Then, it is more likely that we will add a blue marble in the next state. However, this is mitigated by the fact that adding a blue marble changes the proportions much less dramatically than adding a red marble.

2.5 Transition Systems

The second half of this thesis focuses on safety, reachability and termination analyses over polynomial programs. In this section, we provide detailed definitions for programs and fix our notation. We model our programs as transition systems.

Valuations. Let $\mathbf{V}$ be a finite set of *variables*. A *valuation* over $\mathbf{V}$ is a function $\nu : \mathbf{V} \to \mathbb{R}$ that assigns a real value to every variable. We denote the set of all valuations over $\mathbf{V}$ by $\mathbb{R}^{\mathbf{V}}$. We sometimes use a valuation ν over a set $\mathbf{V}' \subset \mathbf{V}$ of variables as a valuation over $\mathbf{V}$. In such cases, we assume that $\nu(v) = 0$ for every $v \in \mathbf{V} \setminus \mathbf{V}'$. Given a valuation ν, a variable v and $x \in \mathbb{R}$, we write $\nu[v \leftarrow x]$ to denote a valuation ν' such that $\nu'(v) = x$ and ν' agrees with ν for every other variable.

Polynomial Arithmetic Expressions. A *polynomial arithmetic expression* $\mathfrak{e}$ over $\mathbf{V}$ is an expression built from the variables in $\mathbf{V}$, real constants, and the arithmetic operations of addition, subtraction and multiplication.

Propositional Polynomial Predicates. A *propositional polynomial predicate* is a propositional formula built from (i) *atomic assertions* of the form $\mathfrak{e}_1 \bowtie \mathfrak{e}_2$, where $\mathfrak{e}_1$ and $\mathfrak{e}_2$ are polynomial arithmetic expressions, and $\bowtie \in \{<, \leq, \geq, >\}$ and (ii) propositional connectives $\vee$, $\wedge$ and $\neg$. The satisfaction relation $\models$ between a valuation ν and a propositional polynomial predicate ϕ is defined in the natural way, i.e. by substituting the variables with their values in ν and evaluating the resulting boolean expression.

Transition Systems. A *transition system* (or simply *system*) is a tuple $S = (\mathbf{V}, \mathbf{L}, \ell_0, I, \mathbf{\Theta})$, in which $\mathbf{V}$ is a finite set of variables, $\mathbf{L}$ is a finite set of *locations* or *labels*, $\ell_0 \in \mathbf{L}$ is the *initial* or *starting* location, I is an assertion over $\mathbf{V}$ which defines the set of possible initial valuations, and $\mathbf{\Theta}$ is a finite set of *transitions*. Each transition $\theta \in \mathbf{\Theta}$ is of the form $\theta = (\ell, \ell', \varphi, \mu)$ where $\ell, \ell' \in \mathbf{L}$ are the *pre* and *post* locations, φ is an assertion over $\mathbf{V}$ that serves as the *transition condition*, and $\mu : \mathbb{R}^{\mathbf{V}} \to \mathbb{R}^{\mathbf{V}}$ is an *update function*. For brevity, in the sequel, we assume that we have fixed a system $S = (\mathbf{V}, \mathbf{L}, \ell_0, I, \mathbf{\Theta})$ which is under study. For a location $\ell \in \mathbf{L}$, we write $\mathbf{\Theta}_\ell$ to denote the set of transitions out of ℓ. We say that a system is β-branching if $|\mathbf{\Theta}_\ell| \leq \beta$ for every location ℓ.

$I : x \geq 0 \ \wedge \ y \geq 0 \ \wedge \ z \geq 0$

```
a:  while  $x \geq y$:
b:      □  $(x, y) := (x+1, y+2)$
c:      □  $(x, y, z) := (x+z, y+z, z-1)$
d:
```

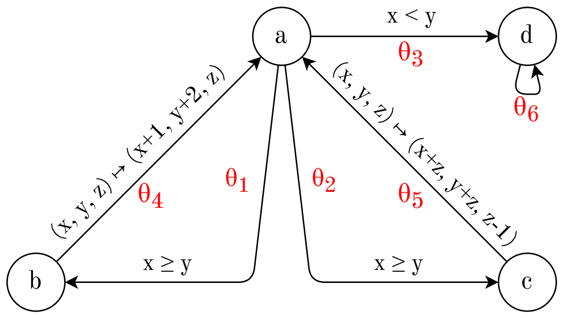


Figure 2.6: A Simple Program (left) and its Representation as a Transition System (right)

Example 2.10. *Consider the program in Figure 2.6 (left), in which* □ *denotes non-determinism choice between transitioning to* b *or* c*. The transition system in Figure 2.6 (right) represents this program. Note that we have* $\ell_0 = a$ *and assume the initial valuations satisfy* $x, y, z \geq 0$*. Each transition is labeled by its name, condition and update function. For brevity, we drop the condition whenever it is* ***true*** *and also drop the update function when it is the identity function.*

States. A state in S is a pair $\sigma = (\ell, \nu) \in \mathbf{L} \times \mathbb{R}^{\mathbf{V}}$, consisting of a location and a valuation for the variables. We denote the set of all states by Σ. A subset $\Sigma' \subseteq \Sigma$ of states is called bounded if the set of valuations that appear in the elements of Σ' is bounded.

Successors. A state $\sigma' = (\ell', \nu')$ is called a *successor* of a state $\sigma = (\ell, \nu)$ if there exists a transition $\theta = (\ell, \ell', \varphi, \mu) \in \mathbf{\Theta}$ such that $\nu \models \varphi$ and $\nu' = \mu(\nu)$. *For theoretical elegance, we assume that every state has at least one successor.* In practice, when modeling a program as a transition system, there might be states in which the program terminates. In such cases, the corresponding transition system will remain in the final state, i.e. we assume that there is a transition from the final state to itself that does not change the value of any variable.

Example 2.11. *In Figure 2.6 (right), the state* $(b, 1, 1, 2)$*, i.e. the state at location* b *for which the values of* x *and* y *are* 1 *and the value of* z *is* 2*, is a successor of* $(a, 1, 1, 2)$ *through* θ_1*. Similarly,* $(a, 2, 3, 2)$ *is a successor of* $(b, 1, 1, 2)$ *through* θ_4*. Also note that there is a transition* θ_6 *from* d *to itself, handling the case where the program terminates as described above.*

Runs. A *run* of the system $S = (\mathbf{V}, \mathbf{L}, \ell_0, I, \mathbf{\Theta})$ is an *infinite* sequence $\mathsf{r} = \langle \sigma_i, \theta_i \rangle_{i=0}^{\infty} = \langle (\ell_i, \nu_i), \theta_i \rangle_{i=0}^{\infty}$, where each $\sigma_i \in \Sigma$ is a state consisting of a location $\ell_i \in \mathbf{L}$ and a valuation $\nu_i \in \mathbb{R}^{\mathbf{V}}$, and each $\theta_i = (\ell_i, \ell_{i+1}, \varphi_i, \mu_i) \in \mathbf{\Theta}$ is a transition from ℓ_i to ℓ_{i+1}, such that:

- r starts in the initial location ℓ_0;
- $\nu_0 \models I$, i.e. the initial valuation satisfies the assertion I;
- For every i, we have $\nu_i \models \varphi_i$ and $\nu_{i+1} = \mu_i(\nu_i)$, i.e. σ_{i+1} is a successor of σ_i through θ_i.

Semi-runs. A *semi-run* is defined similarly to a run, except that it does not have to start at ℓ_0 or satisfy I. A *path* of length n is a finite prefix $\pi = \sigma_0, \theta_0, \ldots, \sigma_{n-1}, \theta_{n-1}, \sigma_n$ of a run. Note that a path must always end at a state. Similarly, a *semi-path* is a finite prefix of a semi-run that ends at a state.

Non-determinism. The system S is called *deterministic* if there is exactly one possible transition at every state. Formally, S is deterministic if for every $\sigma = (\ell, \nu) \in \Sigma$, there exists exactly one $\theta \in \mathbf{\Theta}$ such that $\theta = (\ell, \ell', \varphi, \mu)$ and $\nu \models \varphi$. Otherwise, S is *non-deterministic.*

To model probabilistic programs (Chapter 8), we extend the notion of transition systems:

Probabilistic Transition Systems. A *probabilistic transition system* is a tuple $S = (\mathbf{V}, \mathbf{L}, \mathbf{L}_p, \ell_0, I, \theta)$ where every part is the same as in transition systems, except that $\mathbf{L}$ is partitioned into two sets $\mathbf{L}_p \subseteq \mathbf{L}$ and $\mathbf{L} \setminus \mathbf{L}_p$, and $\mathbf{\Theta}$ is a finite set of *probabilistic transitions.* Each transition $\theta \in \mathbf{\Theta}$ is of the form $\theta = (\ell, \ell', p, \varphi, \mu)$ in which

- $\ell, \ell' \in \mathbf{L}$ are the pre and post locations;
- If $\ell \in \mathbf{L}_p$, then $p \in [0, 1]$ is the probability of the transition, otherwise $p = \star$, denoting non-determinism;
- φ is an assertion over $\mathbf{V}$ that serves as the transition condition. If $\ell \in \mathbf{L}_p$, then $\varphi = \mathbf{true}$.
- $\mu : \mathbb{R}^{\mathbf{V}} \to \Delta(\mathbb{R}^{\mathbf{V}})$ assigns to each valuation $\nu \in \mathbb{R}^{\mathbf{V}}$ in ℓ a (discrete or continuous) distribution over the valuations in ℓ'.

Additionally, for all $\ell \in \mathbf{L}_p$, we have $\sum_{\theta \in \mathbf{\Theta}_\ell} \theta.p = 1$.

Informally, we are extending transition systems in two ways: First, we are allowing the update function to probabilistically choose the next valuation, i.e. provide a distribution over the next valuation. Second, we are considering probabilistic locations $\mathbf{L}_p$ in which the next transition is chosen probabilistically instead of non-deterministically.

Runs. A run of the probabilistic transition system $S = (\mathbf{V}, \mathbf{L}, \mathbf{L}_p, \ell_0, I, \theta)$ is an infinite sequence $\mathsf{r} = \langle \sigma_i, \theta_i \rangle_{i=0}^{\infty} = \langle (\ell_i, \nu_i), \theta_i \rangle_{i=0}^{\infty}$ where each $\sigma_i \in \Sigma$ is a state consisting of a location $\ell_i \in \mathbf{L}$ and a valuation $\nu_i \in \mathbb{R}^{\mathbf{V}}$, and each $\theta_i = (\ell_i, \ell_{i+1}, p_i, \varphi_i, \mu_i) \in \mathbf{\Theta}$ is a transition from ℓ_i to ℓ_{i+1} such that:

- r starts at the initial location ℓ_0;
- $\nu_0 \models I$;
- For every i, we have $\nu_i \models \varphi_i$ and $\nu_{i+1} \in \text{supp}(\mu_i(\nu_i))$. In other words, σ_{i+1} is a probabilistic successor of σ_i through θ_i.

Semi-runs, paths and semi-paths are defined in the same way as in the non-probabilistic case.

Schedulers. Let $\daleth$ be the set of paths of a probabilistic transition system $S = (\mathbf{V}, \mathbf{L}, \mathbf{L}_p, \ell_0, I, \theta)$. A scheduler is a function s that assigns to every path $\pi \in \daleth$ ending in state at a location $\ell \notin \mathbf{L}_p$, a transition $\mathsf{s}(\pi) \in \mathbf{\Theta}_\ell$. A run $\mathsf{r} = \langle (\ell_i, \nu_i), \theta_i \rangle_{i=0}^{\infty}$ is *compatible* with s if for every $n \in \mathbb{Z}^{\geq 0}$, we have $\ell_n \notin \mathbf{L}_p \Rightarrow \theta_n = \mathsf{s}(\langle \ell_0, \nu_0, \theta_0, \dots, \ell_n, \nu_n \rangle)$. It is well-known that a scheduler s naturally induces a unique probability measure $\mathbb{P}^{\mathsf{s}}$ over the runs of S. See [Chakarov and Sankaranarayanan, 2013] for details. We use the notation $\mathbb{E}^{\mathsf{s}}$ to denote expected values of random variables under $\mathbb{P}^{\mathsf{s}}$.

2.6 Sätze for Positivity and Nonnegativity of Polynomials

In this section, we provide an overview of several classical theorems in linear programming and real algebraic geometry and obtain corollaries which make them useful in the static analysis of polynomial programs. Generally speaking, the use-case of these corollaries in our problems is that they help reduce entailment conditions over polynomial inequalities to solving systems of quadratic constraints (quadratic programming).

2.6.1 Farkas' Lemma and Handelman's Theorem

Farkas' Lemma is a classical tool in polyhedral geometry and linear programming. Below, we provide a presentation of this lemma that follows [Colón et al., 2003].

Lemma 2.2 (Farkas' Lemma [Farkas, 1902, Matousek and Gärtner, 2007])**.** *Consider a set* $\mathbf{V} = \{v_1, \ldots, v_r\}$ *of real-valued variables and the following system of* m *linear inequalities over* $\mathbf{V}$*:*

$$\Phi : \begin{cases} a_{1,0} + a_{1,1} \cdot v_1 + \ldots + a_{1,r} \cdot v_r \geq 0 \\ \vdots \\ a_{m,0} + a_{m,1} \cdot v_1 + \ldots + a_{m,r} \cdot v_r \geq 0 \end{cases}$$

When Φ *is satisfiable, it entails a given linear inequality*

$$\psi : c_0 + c_1 v_1 + \ldots + c_r v_r \geq 0$$

if and only if ψ *can be written as a non-negative linear combination of* $1 \geq 0$ *and the inequalities in* Φ, *i.e. if and only if there exist* non-negative *real numbers* $y_0, y_1, \ldots, y_m$, *such that:*

$$c_0 = y_0 + \sum_{i=1}^{m} y_i \cdot a_{i,0} \quad , \quad c_1 = \sum_{i=1}^{m} y_i \cdot a_{i,1} \quad , \quad \ldots \quad , \quad c_r = \sum_{i=1}^{m} y_i \cdot a_{i,r}.$$

Moreover, Φ *is unsatisfiable if and only if* $-1 \geq 0$ *can be derived as above.*

Notation. Given a set $X \subseteq \mathbb{R}^{\mathbf{V}}$, we write $\overline{X}$ to denote the closure of X, i.e. the smallest closed subset of $\mathbb{R}^{\mathbf{V}}$ that contains X. Similarly, for Φ defined as below, we use the notation $\overline{\Phi}$ to denote the system of linear inequalities obtained by replacing every $\bowtie_i$ in Φ with $\geq$.

We often find ourselves in situations where Φ consists of both strict and non-strict linear inequalities. We should hence use the following variant/corollary of Lemma 2.2:

Corollary 2.1. *Consider a set* $\mathbf{V} = \{v_1, \ldots, v_r\}$ *of real-valued variables and the following system of* m *linear inequalities over* $\mathbf{V}$*:*

$$\Phi : \begin{cases} a_{1,0} + a_{1,1} \cdot v_1 + \ldots + a_{1,r} \cdot v_r \ \ \bowtie_1 \ \ 0 \\ \vdots \\ a_{m,0} + a_{m,1} \cdot v_1 + \ldots + a_{m,r} \cdot v_r \ \ \bowtie_m \ \ 0 \end{cases}$$

in which $\bowtie_i \in \{>, \geq\}$. *When* Φ *is satisfiable, it entails a given non-strict linear inequality*

$$\psi : c_0 + c_1 v_1 + \ldots + c_r v_r \geq 0$$

if and only if ψ *can be written as a non-negative linear combination of* $1 \geq 0$ *and the inequalities in* Φ, *i.e. if and only if there exist* non-negative *real numbers* $y_0, y_1, \ldots, y_m$, *such that:*

$$c_0 = y_0 + \sum_{i=1}^{m} y_i \cdot a_{i,0} \quad , \quad c_1 = \sum_{i=1}^{m} y_i \cdot a_{i,1} \quad , \quad \ldots \quad , \quad c_r = \sum_{i=1}^{m} y_i \cdot a_{i,r}.$$

Moreover, Φ *is unsatisfiable if and only if either* $-1 \geq 0$ *can be derived as above, or* $0 > 0$ *can be derived as above with the extra requirement that* $\sum_{\bowtie_i \in \{>\}} y_i > 0$ *(i.e. in order to derive a strict inequality, we should use at least one of the strict inequalities in* Φ *with non-zero coefficient).*

Proof. For the first part, suppose that ψ is entailed by Φ, hence $\{x \in \mathbb{R}^{\mathbf{V}} \mid c_0 + c_1 \cdot x_1 + \ldots + c_r \cdot x_r \geq 0\} \supseteq \{x \in \mathbb{R}^{\mathbf{V}} \mid x \models \Phi\}$. The former is a closed set, hence it also includes the closure of the latter, which is the set of points that satisfy $\overline{\Phi}$. Hence, we can apply Lemma 2.2 to $\overline{\Phi}$ and ψ to obtain the desired result.

For the second part, if Φ is satisfiable, then obviously no non-negative combination of inequalities in Φ can sum up to a contradiction such as $0 > 0$ or $-1 \geq 0$. If $\overline{\Phi}$ is not satisfiable, then by Lemma 2.2, we can obtain $-1 \geq 0$. The only remaining case is if $\overline{\Phi}$ is satisfiable but Φ is not. Reorder the inequalities in Φ so that the non-strict inequalities appear first. Then, consider the smallest i for which the first i inequalities in Φ are unsatisfiable. Let $\Phi[1 \ldots i]$ denote the first i inequalities. Based on our ordering, we know that the $i-$th inequality is strict and of the form $a_{i,0} + a_{i,1} \cdot v_1 + \ldots + a_{i,r} \cdot v_r > 0$. Given that $\Phi[1 \ldots i]$ is unsatisfiable, we know that $\{x \in \mathbb{R}^{\mathbf{V}} \mid x \models \Phi[1 \dots i-1]\} \subseteq \{x \in \mathbb{R}^{\mathbf{V}} \mid a_{i,0} + a_{i,1} \cdot v_1 + \dots + a_{i,r} \cdot v_r \leq 0\}$. Therefore, by the first part above, we can write

$$a_{i,0} + a_{i,1} \cdot v_1 + \dots + a_{i,r} \cdot v_r \leq 0$$

as a non-negative combination of the first $i-1$ inequalities. Moreover, the $i-$th inequality is:

$$a_{i,0} + a_{i,1} \cdot v_1 + \dots + a_{i,r} \cdot v_r > 0$$

Summing up these two, we get $0 > 0$. □

We now turn to Handelman's theorem, which characterizes positive polynomials over polytopes. Before presenting the theorem, we first need the notion of monoids:

Monoid. Consider a set $\mathbf{V} = \{v_1, \dots, v_r\}$ of real-valued variables and the following system of m linear inequalities over $\mathbf{V}$:

$$\Phi : \begin{cases} a_{1,0} + a_{1,1} \cdot v_1 + \dots + a_{1,r} \cdot v_r \bowtie_1 0 \\ \vdots \\ a_{m,0} + a_{m,1} \cdot v_1 + \dots + a_{m,r} \cdot v_r \bowtie_m 0 \end{cases}$$

in which $\bowtie_i \in \{>, \geq\}$. Let g_i be the LHS of the i-th inequality, i.e. $g_i(v_1, \dots, v_r) := a_{i,0} + a_{i,1} \cdot v_1 + \dots + a_{i,r} \cdot v_r$. The monoid of Φ is defined as:

$$\textsc{Monoid}(\Phi) := \left\{ \prod_{i=1}^{m} g_i^{\kappa_i} \;\middle|\; \forall 1 \leq i \leq m, \quad \kappa_i \in \mathbb{N} \cup \{0\} \right\}.$$

Basically, $\textsc{Monoid}(\Phi)$ is the set of all polynomials that can be obtained by multiplying the linear expressions on the LHS of Φ together. Note that each such expression can appear zero or multiple times in the multiplication. Specifically, it is noteworthy that $1 \in \textsc{Monoid}(\Phi)$.

Theorem 2.1 ([Handelman, 1988]). *Consider a set* $\mathbf{V} = \{v_1, \ldots, v_r\}$ *of real-valued variables and the following system of* m *linear inequalities over* $\mathbf{V}$*:*

$$\Phi : \begin{cases} a_{1,0} + a_{1,1} \cdot v_1 + \ldots + a_{1,r} \cdot v_r \geq 0 \\ \vdots \\ a_{m,0} + a_{m,1} \cdot v_1 + \ldots + a_{m,r} \cdot v_r \geq 0 \end{cases}$$

If Φ *is satisfiable,* $\textsc{Sat}(\Phi)$ *is compact, and* Φ *entails a given polynomial inequality* $g(v_1, \ldots, v_r) > 0$, *then there exist* $y_1, y_2, \ldots, y_u \in [0, \infty)$ *and* $h_1, h_2, \ldots, h_u \in \textsc{Monoid}(\Phi)$ *such that:*

$$g = \sum_{i=1}^{u} y_i \cdot h_i.$$

As was the case with Farkas' Lemma, it is useful to have a variant of Handelman's theorem that can handle strict inequalities in Φ. We present such a variant, which is a direct corollary of Theorem 2.1 and characterizes strongly positive polynomials over bounded, but not necessarily closed, polyhedra. Before doing so, we need the notion of strong positivity:

Strong Positivity. Let $X \subseteq \mathbb{R}^{\mathbf{V}}$ be a set of valuations and $g \in \mathbb{R}[\mathbf{V}]$ a polynomial over $\mathbf{V}$. We say that g is *strongly* positive over X, and write $X \models g \gg 0$ (or simply $g \gg 0$ when X is clear from context), if $\inf_{x \in X} g(x) > 0$. The real value $\delta := \inf_{x \in X} g(x)$ is called the *positivity gap* or *positivity witness* of g over X. Moreover, g is strongly greater than h, denoted $g \gg h$, iff $g - h \gg 0$.

Corollary 2.2. *Consider a set* $\mathbf{V} = \{v_1, \ldots, v_r\}$ *of real-valued variables and the following system of* m *linear inequalities over* $\mathbf{V}$*:*

$$\Phi : \begin{cases} a_{1,0} + a_{1,1} \cdot v_1 + \ldots + a_{1,r} \cdot v_r \bowtie_1 0 \\ \vdots \\ a_{m,0} + a_{m,1} \cdot v_1 + \ldots + a_{m,r} \cdot v_r \bowtie_m 0 \end{cases}$$

in which $\bowtie_i \in \{>, \geq\}$. *If* Φ *is satisfiable and* $\textsc{Sat}(\Phi)$ *is* bounded, *then* Φ *entails a given strong polynomial inequality* $g(v_1, \ldots, v_r) \gg 0$, *or in other words* $\textsc{Sat}(\Phi) \models g(v_1, \ldots, v_r) \gg 0$, *if and only if there exist constants* $y_0 \in (0, \infty)$ *and* $y_1, y_2, \ldots, y_u \in [0, \infty)$, *and polynomials*

$h_1, h_2, \dots, h_u \in \textsc{Monoid}(\Phi)$ *such that:*

$$g = y_0 + \sum_{i=1}^{u} y_i \cdot h_i. \tag{2.1}$$

Proof. It is obvious that every g in the form of (2.1) is strongly positive over $\textsc{Sat}(\Phi)$, given that Φ trivially entails $g \geq y_0 > 0$.

We now prove the other side. Suppose Φ entails $g \gg 0$. Let $\delta > 0$ be the positivity gap of g over $\textsc{Sat}(\Phi)$ and choose δ', y_0 such that $0 < y_0 < \delta' < \delta$. So, $\textsc{Sat}(\Phi) \subseteq \textsc{Sat}(g > \delta')$ and hence $\textsc{Sat}(\overline{\Phi}) = \overline{\textsc{Sat}(\Phi)} \subseteq \overline{\textsc{Sat}(g > \delta')} = \textsc{Sat}(g \geq \delta')$. Therefore, $\overline{\Phi}$ entails $g - \delta' \geq 0$. So, it also entails $g - y_0 > 0$. Applying Theorem 2.1 to $\overline{\Phi}$ and $g - y_0$, we have:

$$g - y_0 = \sum_{i=1}^{u} y_i \cdot h_i$$

which is equivalent to Equation (2.1). □

2.6.2 Positivstellensätze

A *Positivstellensatz* (German for "positive locus theorem", plural: Positivstellensätze) is a theorem in real algebraic geometry that characterizes positive polynomials over semi-algebraic sets. The most commonly-used satz in this thesis is Putinar's positivstellensatz.

Theorem 2.2 (Putinar's Positivstellensatz [Putinar, 1993])**.** *Consider a set* $\mathbf{V} = \{v_1, \dots, v_r\}$ *of real-valued variables and the following system of* m *polynomial inequalities over* $\mathbf{V}$*:*

$$\Phi : \begin{cases} g_1(v_1, \dots, v_r) \geq 0, \\ \qquad\quad \vdots \\ g_m(v_1, \dots, v_r) \geq 0 \end{cases}$$

where $g_1, \dots, g_m \in \mathbb{R}[\mathbf{V}]$ *are polynomials. If there exists a* g_i *such that the set* $\textsc{Sat}(g_i \geq 0)$ *is compact, and* Φ *entails a given polynomial inequality* $g(v_1, \dots, v_r) > 0$ *then there exist* *polynomials* $h_0, h_1, \ldots, h_m \in \mathbb{R}[\mathbf{V}]$ *such that*

$$g = h_0 + \sum_{i=1}^{m} h_i \cdot g_i$$

and every h_i *is a sum of squares, i.e.* $h_i = \sum h_{i,j}^2$ *for some polynomials* $h_{i,j} \in \mathbb{R}[\mathbf{V}]$.

Putinar's positivstellensatz provides a characterization of all polynomials g that are positive over the closed and bounded set $\textsc{Sat}(\Phi)$. As a corollary, given a set of atomic non-negativity assumptions $g_i(x) \geq 0$, in order to find all polynomials g that are positive under these assumptions, we only need to look into polynomials of form (2.2):

Corollary 2.3. *Let* $\mathbf{V}, g, g_1, \ldots, g_m$ *and* Φ *be as above. Then* $g(x) > 0$ *for all* $x \models \Phi$ if and only if:

$$g = \epsilon + h_0 + \sum_{i=1}^{m} h_i \cdot g_i \tag{2.2}$$

where $\epsilon > 0$ *is a real number and each polynomial* h_i *is the sum of squares of some polynomials in* $\mathbb{R}[\mathbf{V}]$.

Proof. It is obvious that if (2.2) holds, then $g(x) > 0$ for all $x \models \Phi$. We prove the other side. Let $g(x) > 0$ for all $x \models \Phi$. Given that $\textsc{Sat}(\Phi)$ is compact and g continuous, $g(\textsc{Sat}(\Phi))$ must also be compact and hence closed. Therefore, $\delta := \inf_{x \models \Phi} g(x) > 0$. Let $\epsilon = \delta/2$, then $g(x) - \epsilon > 0$ for all $x \models \Phi$. Applying Putinar's Positivstellensatz to $g - \epsilon$ leads to the desired result. □

Note that Putinar's positivstellensatz automatically provides a criterion for satisfiability of Φ. Consider the polynomial inequality $-1 > 0$. This inequality is false, and is hence entailed by Φ if and only if Φ is unsatisfiable. As in the previous cases, we need a variant of this theorem that can handle strict inequalities in Φ. We now obtain such a variant.

Corollary 2.4. *Consider a set* $\mathbf{V} = \{v_1, \dots, v_r\}$ *of real-valued variables and the following system of* m *polynomial inequalities over* $\mathbf{V}$*:*

$$\Phi : \begin{cases} g_1(v_1, \dots, v_r) \bowtie_1 0, \\ \vdots \\ g_m(v_1, \dots, v_r) \bowtie_m 0 \end{cases}$$

in which every $g_i \in \mathbb{R}[\mathbf{V}]$ *is a polynomial and every* $\bowtie_i \in \{>, \geq\}$*. Also, assume that there is some* i *such that the set* $\textsc{Sat}(g_i \geq 0)$ *is compact, or equivalently* $\textsc{Sat}(g_i \bowtie_i 0)$ *is bounded. If* Φ *is satisfiable, then it entails a* strong *polynomial inequality* $g(v_1, \dots, v_r) \gg 0$*, if and only if there exist a constant* $y_0 \in (0, \infty)$ *and polynomials* $h_0, \dots, h_m \in \mathbb{R}[\mathbf{V}]$ *such that*

$$g = y_0 + h_0 + \sum_{i=1}^{m} h_i \cdot g_i \tag{2.3}$$

and every h_i *is a sum of squares, i.e.* $h_i = \sum h_{i,j}^2$ *for some polynomials* $h_{i,j} \in \mathbb{R}[\mathbf{V}]$.

Proof. It is obvious that any polynomial g that can be represented as in Equation (2.3) is strongly positive over $\textsc{Sat}(\Phi)$ and has a positivity gap of at least $y_0 > 0$.

We now prove the other side. Suppose Φ is satisfiable and entails $g \gg 0$ with positivity gap δ, and choose $0 < y_0 < \delta' < \delta$. We have $\textsc{Sat}(\Phi) \subseteq \textsc{Sat}(g > \delta')$ so $\textsc{Sat}(\overline{\Phi}) = \overline{\textsc{Sat}(\Phi)} \subseteq \overline{\textsc{Sat}(g > \delta')} = \textsc{Sat}(g \geq \delta') \subseteq \textsc{Sat}(g > y_0)$. Hence, $\overline{\Phi}$ entails $g - y_0 > 0$. Applying Theorem 2.2 to $\overline{\Phi}$ and $g - y_0$ leads to the desired result.

□

A more general positivstellensatz, which we will use in Chapter 8 is the one due to Schmüdgen. To present this satz, we should first define the concept of preordering.

Preordering. Consider a set $\mathbf{V} = \{v_1, \dots, v_r\}$ of real-valued variables and the following system of m polynomial inequalities over $\mathbf{V}$:

$$\Phi : \begin{cases} g_1(v_1, \dots, v_r) \geq 0, \\ \vdots \\ g_m(v_1, \dots, v_r) \geq 0 \end{cases}$$

where $g_1, \ldots, g_m \in \mathbb{R}[\mathbf{V}]$ are polynomials. The *preordering* $\mathsf{PO}(\Phi)$ is defined as follows:

$$\mathsf{PO}(\Phi) := \left\{ \sum_{w \in \{0,1\}^m} h_w \cdot \prod_{i=1}^{m} g_i^{w_i} \;\mid\; \text{every } h_w \in \mathbb{R}[\mathbf{V}] \text{ is a sum of squares} \right\}.$$

Here, w_i denotes the i-th component of the word w. Intuitively, the preordering of Φ is the set of all polynomials in $\mathbb{R}[\mathbf{V}]$ that can be written as a combination of products of g_i's, in which the multiplier of each product is a sum-of-squares polynomial.

Theorem 2.3 (Schmüdgen's Positivstellensatz [Schmüdgen, 1991])**.** *Consider a set* $\mathbf{V} = \{v_1, \ldots, v_r\}$ *of real-valued variables and the following system of* m *polynomial inequalities over* $\mathbf{V}$*:*

$$\Phi : \begin{cases} g_1(v_1, \ldots, v_r) \geq 0, \\ \quad\vdots \\ g_m(v_1, \ldots, v_r) \geq 0 \end{cases}$$

where $g_1, \ldots, g_m \in \mathbb{R}[\mathbf{V}]$ *are polynomials. If* $\textsc{Sat}(\Phi)$ *is compact and* Φ *entails a given polynomial inequality* $g(v_1, \ldots, v_r) > 0$*, then* $g \in \mathsf{PO}(\Phi)$.

Note that the compactness requirement in Schmüdgen's positivstellensatz is weaker than the one in Putinar's positivstellensatz.

2.6.3 Hilbert's Nullstellensatz

Hilbert's Nullstellensatz (German for "zero locus theorem") is a profound theorem that establishes a fundamental relationship between geometry and algebra, and is arguably the basis of the entire field of algebraic geometry. We use this satz for solving our satisfiability problems. Specifically, corollary 2.4 provides a characterization of strongly positive polynomials over a semi-algebraic set $\textsc{Sat}(\Phi)$ if it is non-empty. However, we also need a criterion for unsatisfiability of Φ. Given that Φ may contain both strict and non-strict inequalities, the situation is trickier than Theorem 2.2. To obtain such a characterization, we use Hilbert's Strong Nullstellensatz for reals:

Theorem 2.4 (Strong Nullstellensatz [Atiyah and Macdonald, 1969])**.** *Consider a set* $\mathbf{V} = \{v_1, \dots, v_r\}$ *of real-valued variables and let* $g_1, \dots, g_m, g \in \mathbb{R}[\mathbf{V}]$ *be polynomials over* $\mathbf{V}$. *Then exactly one of the following statements holds:*

- *There exists a valuation* $\nu \in \mathbb{R}^{\mathbf{V}}$, *such that* $g_1(\nu) = g_2(\nu) = \dots = g_m(\nu) = 0$, *but* $g(\nu) \neq 0$.
- *There exist a non-negative integer* α *and polynomials* $h_1, \dots, h_m \in \mathbb{R}[\mathbf{V}]$ *such that*

$$\sum_{i=1}^{m} h_i \cdot g_i = g^{\alpha}.$$

We now have the required tools for characterizing unsatisfiable Φ's.

Theorem 2.5. *Consider a set* $\mathbf{V} = \{v_1, \dots, v_r\}$ *of real-valued variables and the following system of* m *polynomial inequalities over* $\mathbf{V}$*:*

$$\Phi : \Big\{ g_1(v_1, \dots, v_r) \bowtie_1 0, \quad \dots, \quad g_m(v_1, \dots, v_r) \bowtie_m 0$$

in which every $g_i \in \mathbb{R}[\mathbf{V}]$ *is a polynomial and every* $\bowtie_i \in \{>, \geq\}$. Φ *is unsatisfiable, if and only if at least one of the following statements holds:*

(i) There exist a constant $y_0 \in (0, \infty)$ *and sum-of-square polynomials* $h_0, \dots, h_m \in \mathbb{R}[\mathbf{V}]$ *such that*

$$-1 = y_0 + h_0 + \sum_{i=1}^{m} h_i \cdot g_i.$$

(ii) There exist a non-negative integer α *and polynomials* $h_1, \dots, h_m \in \mathbb{R}[\mathbf{V}^*]$ *for* $\mathbf{V}^* = \mathbf{V} \cup \{w_1, \dots, w_m\}$, *such that for some* $1 \leq j \leq m$ *with* $\bowtie_j \in \{>\}$, *we have*

$$w_j^{2\cdot\alpha} = \sum_{i=1}^{m} h_i \cdot (g_i - w_i^2)$$

Proof. If Φ is satisfiable, then it cannot entail $-1 > 0$, so (i) is impossible. We now show that (ii) implies unsatisfiability of Φ as well. Define $\tilde{g}_i(v_1, \dots, v_r, w_1, \dots, w_m) := g_i(v_1, \dots, v_r) - w_i^2$.

So, we have

$$w_j^{2\cdot\alpha} = \sum_{i=1}^{m} h_i \cdot \tilde{g}_i.$$

Moreover, $g_j^\alpha = \left(\tilde{g}_j + w_j^2\right)^\alpha = \sum_{i=0}^{\alpha} \binom{\alpha}{i} \tilde{g}_j^i \cdot w_j^{2\cdot(\alpha-i)} = w_j^{2\cdot\alpha} + h'_j \cdot \tilde{g}_j$ for some $h'_j \in \mathbb{R}[\mathbf{V}^*]$. So, letting $h''_i = h_i$ for $i \neq j$ and $h''_j = h_j + h'_j$, we have

$$g_j^\alpha = \sum_{i=1}^{m} h''_i \cdot (g_i - w_i^2)$$

Let $\nu \in \mathbb{R}^{\mathbf{V}} \cap \textsc{Sat}(\Phi)$. We extend ν to $\nu^* \in \mathbb{R}^{\mathbf{V}^*}$ as follows: for every w_i, let $\nu^*(w_i) = \sqrt{\nu(g_i)}$. So, we have $\nu^*(g_i - w_i^2) = 0$, and hence the RHS of the equation above is 0 at ν^*. On the other hand, we have $\nu^*(g_j^\alpha) = \nu(g_j^\alpha) = (\nu(g_j))^\alpha > 0$. This contradiction shows that Φ is unsatisfiable.

We now prove the other side. Suppose that Φ is unsatisfiable. If $\overline{\Phi}$ is unsatisfiable, then it entails $-1.5 > 0$ and hence we can apply Theorem 2.2 to write $-1.5 = h_0 + \sum_{i=1}^{m} h_i \cdot g_i$ for some sum-of-squares polynomials h_i, which is equivalent to $-1 = 0.5 + h_0 + \sum_{i=1}^{m} h_i \cdot g_i$, hence leading to case (i) above. The only remaining case is if $\overline{\Phi}$ is satisfiable but Φ is not. Reorder the inequalities in Φ so that the non-strict inequalities appear first. Let j be the smallest index for which $\Phi[1 \dots j]$, i.e. the set of first j inequalities in Φ, is unsatisfiable. By definition, $\Phi[1 \dots j-1]$ is satisfiable and hence $\overline{\textsc{Sat}(\Phi[1 \dots j-1])} = \textsc{Sat}(\overline{\Phi}[1 \dots j-1])$. Moreover, since $\Phi[1 \dots j] = \Phi[1 \dots j-1] \wedge (g_j > 0)$ is unsatisfiable, we know that $\Phi[1 \dots j-1]$ entails $g_j \leq 0$. In other words, $\textsc{Sat}(\Phi[1 \dots j-1]) \subseteq \textsc{Sat}(g_j \leq 0)$. Taking closures from both sides shows that $\overline{\Phi}[1 \dots j-1]$ entails $g_j \leq 0$. So, $\overline{\Phi}[1 \dots j]$ entails $g_j = 0$. Define $\tilde{g}_i(v_1, \dots, v_r, w_1, \dots, w_m) := g_i(v_1, \dots, v_r) - w_i^2$. We claim there is no valuation $\nu^* \in \mathbb{R}^{\mathbf{V}^*}$ such that for all $1 \leq i \leq j$, $\tilde{g}_i(\nu^*) = 0$, but $g_j(\nu^*) \neq 0$. To prove this, suppose that such a valuation exists, and let ν be its restriction to $\mathbf{V}$. For each $1 \leq i \leq j$, since $\tilde{g}_i(\nu^*) = 0$, we have $g_i(\nu) \geq 0$. Moreover, $g_j(\nu) = g_j(\nu^*) \neq 0$. This is a contradiction with the previously proven fact that $\overline{\Phi}[1 \dots j]$ entails $g_j = 0$. Applying the strong nullstellensatz (Theorem 2.5) to the $\tilde{g}_i$'s and g_j, we conclude that there exist a non-negative integer α and polynomials

$\tilde{h}_1, \ldots, \tilde{h}_j \in \mathbb{R}[\mathbf{V}^*]$ such that

$$g_j^\alpha = \sum_{i=1}^{j} \tilde{h}_i \cdot \tilde{g}_i$$

Note that $g_j^\alpha = \left(\tilde{g}_j + w_j^2\right)^\alpha = \sum_{i=0}^{\alpha} \binom{\alpha}{i} \tilde{g}_j^i \cdot w_j^{2\cdot(\alpha-i)} = w_j^{2\cdot\alpha} + h'_j \cdot \tilde{g}_j$ for some $h'_j \in \mathbb{R}[\mathbf{V}^*]$. Defining $h_i = \tilde{h}_i$ for all $i \neq j$, and $h_j = \tilde{h}_j - h'_j$, we get

$$w_j^{2\cdot\alpha} = \sum_{i=1}^{j} h_i \cdot \tilde{g}_i = \sum_{i=1}^{j} h_i \cdot (g_i - w_i^2).$$

□

2.7 Encoding Sum-of-Squares Polynomials in QP

Given that several of the sätze in the previous section require sum-of-square polynomials, in our algorithms we also need to encode the property "$h \in \mathbb{R}[X]$ is a sum-of-squares" as a system of quadratic constraints (a quadratic programming instance). In this section, we show how this can be achieved using classical mathematical techniques.

Lemma 2.3. *Given a polynomial $h \in \mathbb{R}[X]$ as input, the problem of deciding whether h is a sum of squares, i.e. whether h can be written as $\sum_i f_i^2$ for some polynomials $f_i \in \mathbb{R}[X]$, can be reduced in polynomial time to solving a system of quadratic equalities.*

We provide a reduction using the following two classical theorems:

Theorem 2.6 (Corollary 7.2.9 in [Horn and Johnson, 1990])**.** *A polynomial $h \in \mathbb{R}[X]$ of even degree d is a sum-of-squares* if and only if *there exists a k-dimensional symmetric positive semi-definite matrix Q such that $h = y^T Q y$, where k is the number of monomials of degree no greater than $d/2$ and y is a column vector consisting of every such monomial.*

Theorem 2.7 ([Higham, 2009, Golub and Van Loan, 1996])**.** *A symmetric square matrix Q is positive semi-definite* if and only if *it has a Cholesky decomposition of the form $Q = LL^T$ where L is a lower-triangular matrix with non-negative diagonal entries.*

Given the two theorems above, our reduction uses the following procedure for generating quadratic equations that are equivalent to the assertion that h is a sum-of-squares:

The Reduction. The algorithm generates the set $\mathbf{M}_{\lfloor d/2 \rfloor}$ of monomials of degree at most $\lfloor d/2 \rfloor$ over X. It then orders these monomials arbitrarily into a vector y and symbolically computes the equality

$$h = y^T L L^T y \tag{2.4}$$

where L is a lower-triangular matrix whose every non-zero entry is a new variable in the system. We call these variables l-variables. For every $l_{i,i}$, i.e. every l-variable that appears on the diagonal of L, the algorithm adds the constraint $l_{i,i} \geq 0$ to the quadratic system. Then, it translates Equation (2.4) into quadratic equations over l-variables and the coefficients of h and by equating the coefficients of corresponding terms on the two sides of (2.4). The resulting system encodes the property that h is a sum-of-squares.

Example 2.12. *Let $X = \{a, b\}$ be the set of variables and $h \in \mathbb{R}[X]$ a quadratic polynomial, i.e. $h(a, b) = t_1 + t_2 \cdot a + t_3 \cdot b + t_4 \cdot a^2 + t_5 \cdot a \cdot b + t_6 \cdot b^2$. We aim to encode the property that h is a sum-of-squares as a system of quadratic equalities and inequalities. To do so, we first generate all monomials of degree at most $\lfloor d/2 \rfloor = 1$, which are 1, a and b. Hence, we let $y = \begin{bmatrix} 1 & a & b \end{bmatrix}^T$. We then generate a lower-triangular matrix L whose every non-zero entry is a new variable:*

$$L = \begin{bmatrix} l_1 & 0 & 0 \\ l_2 & l_3 & 0 \\ l_4 & l_5 & l_6 \end{bmatrix}.$$

We also add the inequalities $l_1 \geq 0, l_3 \geq 0$ and $l_6 \geq 0$ to our system. Now, we write the equation $h = y^T L L^T y$ and compute it symbolically:

$$h = \begin{bmatrix} 1 & a & b \end{bmatrix} \begin{bmatrix} l_1 & 0 & 0 \\ l_2 & l_3 & 0 \\ l_4 & l_5 & l_6 \end{bmatrix} \begin{bmatrix} l_1 & l_2 & l_4 \\ 0 & l_3 & l_5 \\ 0 & 0 & l_6 \end{bmatrix} \begin{bmatrix} 1 \\ a \\ b \end{bmatrix},$$

which leads to: $t_1 + t_2 \cdot a + t_3 \cdot b + t_4 \cdot a^2 + t_5 \cdot a \cdot b + t_6 \cdot b^2 = l_1^2 + 2 \cdot l_1 \cdot l_2 \cdot a + 2 \cdot l_1 \cdot l_4 \cdot b + (l_2^2 + l_3^2) \cdot a^2 + (2 \cdot l_2 \cdot l_4 + 2 \cdot l_3 \cdot l_5) \cdot a \cdot b + (l_4^2 + l_5^2 + l_6^2) \cdot b^2$.

Note that both sides of the equation above are polynomials over $\{a, b\}$, *hence they are equal iff their corresponding coefficients are equal. So, we get the following quadratic equalities over the* t*-variables and* l*-variables:* $t_1 = l_1^2, t_2 = 2 \cdot l_1 \cdot l_2, \ldots, t_6 = l_4^2 + l_5^2 + l_6^2$. *This concludes the construction of our quadratic system.*

Remark 2.2. *Note that the QP instance in the reduction above is by definition a semi-definite programming instance given that* Q *is a symmetric positive semi-definite matrix.*

3

Faster Algorithms for Data-Flow Analysis

This chapter originally appeared in the following publications:

[•] Chatterjee, K., **Goharshady, A. K.**, Ibsen-Jensen, R., and Pavlogiannis, A. **Optimal and Perfectly Parallel Algorithms for On-demand Data-flow Analysis**. In *29th European Symposium on Programming* **(ESOP)**, 2020.

[•] Chatterjee, K., **Goharshady, A. K.**, Goyal, P., Ibsen-Jensen, R., and Pavlogiannis, A. **Faster Algorithms for Dynamic Algebraic Queries in Basic RSMs with Constant Treewidth**. *ACM Transactions on Programming Languages and Systems* **(TOPLAS)**, 2019.

[•] Chatterjee, K., **Goharshady, A. K.**, Ibsen-Jensen, R., and Pavlogiannis, A. **Algorithms for Algebraic Path Properties in Concurrent Systems of Constant Treewidth Components**. *ACM Transactions on Programming Languages and Systems* **(TOPLAS)**, 2018.

[•] Chatterjee, K., **Goharshady, A. K.**, Ibsen-Jensen, R., and Pavlogiannis, A. **Algorithms for Algebraic Path Properties in Concurrent Systems of Constant Treewidth Components**. In *43rd ACM Symposium on Principles of Programming Languages* **(POPL)**, 2016.

3.1 Introduction

Data-flow Analysis. Interprocedural data-flow analyses, such as live variables analysis, alias analysis and null pointers analysis, are ubiquitous in verification and compiler optimization. The most widely-used framework for interprocedural data-flow analysis is *IFDS*, which considers distributive data-flow functions over a finite domain of *data facts*. *On-demand* data-flow analyses restrict the focus of the analysis on specific program locations and data facts. This setting provides a natural split between (i) an *offline (or preprocessing) phase*, where the program is partially analyzed and analysis summaries are created, and (ii) an *on-line (or query) phase*, where analysis queries arrive on demand and the summaries are used to speed up answering queries.

Summary of Our Results. We exploit the fact that flow graphs of programs have low treewidth to develop faster algorithms that are *space and time optimal* for many common same-context data-flow analyses, in both the preprocessing and the query phase. We also use treewidth to develop query solutions that are *perfectly parallelizable*, i.e. the total work for answering each query can be split to a number of threads such that each thread performs only a constant amount of work. Finally, we report on an implementation of our algorithms and perform a series of on-demand analysis experiments on standard benchmarks. Our experimental results show a drastic speed-up of the queries after only a lightweight preprocessing phase, which significantly outperforms existing techniques.

Applications. Static program analysis is a fundamental approach for both analyzing program correctness and performing compiler optimizations. Static data-flow analyses associate with each program location a set of data-flow facts which must (may) hold under all (some) program executions. These facts are then used to reason about program correctness, report erroneous behavior, and optimize program execution. Static data-flow analyses have numerous applications, such as in detection of null pointer dereferencing [Nanda and Sinha, 2009, Shang et al., 2012], in detecting privacy and security issues (e.g. taint analysis and SQL injection analysis) [Arzt et al., 2014, Gould et al., 2004], as well as in compiler optimizations (e.g. constant propagation, reaching definitions, and register allocation) [Reps et al., 1995, Grove and Torczon, 1993, Sagiv et al., 1996, Appel and Palsberg, 2003].

Interprocedural Analysis and the IFDS framework. Data-flow analyses fall in two classes: *intraprocedural* and *interprocedural*. In the former, each procedure of the program is analyzed in isolation, ignoring the interaction between procedures which occurs due to parameter passing/return. In the latter, all procedures of the program are analyzed together, accounting for such interactions, which leads to results of increased precision, and hence is often preferable to intraprocedural analysis [Reps, 2000, Rountev et al., 2006, Späth et al., 2019]. To filter out false results, interprocedural analyses typically employ call-context sensitivity, which ensures that the underlying execution paths respect the calling context of procedure invocations, i.e. if an execution path follows a procedure call from `f` to `g`, when `g` terminates, the execution path will continue to the corresponding call site of `f`. One of the most widely-used frameworks for interprocedural data-flow analysis is the framework of Interprocedural Finite Distributive Subset (IFDS) problems [Reps et al., 1995], which offers a unified formulation of a wide class of interprocedural data-flow analyses as a reachability problem. This elegant algorithmic formulation of data-flow analysis has been a topic of active study, leading to subsequent practical improvements [Horwitz et al., 1995, Naeem et al., 2010, Arzt et al., 2014, Rapoport et al., 2015, Schubert et al., 2019].

On-demand analysis. Exhaustive data-flow analysis is computationally expensive and often unnecessary. Hence, a topic of great interest in the community is that of *on-demand* data-flow analysis [Horwitz et al., 1995, Reps, 1995, Reps, 1998, Naeem et al., 2010]. On-demand analyses have several applications, such as (quoting from [Horwitz et al., 1995, Reps, 1998]) (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) side-stepping incremental updating problems, and (v) offering demand analysis as a user-level operation.

Example 3.1. *As a toy motivating example, consider the partial program shown in Figure 3.1, compiled with a just-in-time compiler that uses speculative optimizations. Whether the compiler must compile the expensive function* `h` *depends on whether x is null in line 6. Performing a null-pointer analysis from the entry of* `f` *reveals that x might be null in line 6. Hence, if the decision to compile* `h` *relies only on an offline static analysis,* `h` *is always compiled, even when not needed.*

```
1 void f(int b){
2   int *x = NULL, *y = NULL;
3   if(b > 1)
4     y = &b;
5   g(x,y);
6   if(x==NULL)
7     h();
8 }
```

```
 9 void g(int *&x, int *y){
10   x=y;
11 }

12 void h(){
13   //An expensive
14   //function
15 }
```

Figure 3.1: A partial C++ program.

Now consider the case where the execution of the program is in line 4, and at this point the compiler decides on whether to compile `h`*. It is clear that given this information, x cannot be null in line 6 and thus* `h` *does not have to be compiled. As we have seen above, this decision can not be made based on offline analysis. On the other hand, an* on-demand *analysis starting from the current program location will correctly conclude that x is not null in line 6. Note however, that this decision is made by the compiler during runtime. Hence, such an on-demand analysis is useful only if it can be performed extremely fast. It is also highly desirable that the time for running this analysis is predictable, so that the compiler can decide whether to run the analysis or simply compile* `h` *proactively.*

Our techniques in this chapter answer the above challenges rigorously. Our approach exploits the treewidth of control flow graphs as the parameter in order to obtain vastly more efficient parameterized algorithms for data-flow analysis.

Treewidth of Programs. As shown in [Thorup, 1998], control flow graphs of `goto`-free structured programs in many classic programming languages have a treewidth of at most 7. The low treewidth of CFGs has also been confirmed experimentally for programs written in Java [Gustedt et al., 2002], C [Klaus Krause et al., 2019], Ada [Burgstaller et al., 2004] and Solidity [Chatterjee et al., 2019b].

Problem statement. We focus on on-demand data-flow analysis in IFDS [Reps et al., 1995, Horwitz et al., 1995, Reps, 1998]. The input consists of a supergraph G of n vertices, a data-fact domain D and a data-flow transformer function M. Edges of G capture control-flow within each procedure, as well as procedure invocations and returns. The set D defines the domain of the analysis, and contains the data facts to be discovered by the analysis for each program location. The function M associates with every edge (u, v) of G a data-flow transformer $M(u, v) : 2^D \to 2^D$. In words, $M(u, v)$ defines the set of data facts that hold at v in some execution that transitions from u to v, given the set of data facts that hold at u.

On-demand analysis brings a natural separation between (i) an *offline (or preprocessing) phase*, where the program is partially analyzed, and (ii) an *online (or query) phase*, where on-demand queries are handled. The task is to preprocess the input in the offline phase, so that in the online phase, the following types of on-demand queries are answered efficiently:

1. A *pair query* has the form (u, d_1, v, d_2), where u, v are vertices of G in the same procedure, and d_1, d_2 are data facts. The goal is to decide if there exists a same-context execution that starts in u and ends in v, and given that the data fact d_1 held at the beginning of the execution, the data fact d_2 holds at the end. These are known as *same-context* queries and are very common in data-flow analysis [Chaudhuri, 2008, Reps et al., 1995].

2. A *single-source* query has the form (u, d_1), where u is a vertex of G and d_1 is a data fact. The goal is to compute for every vertex v that belongs to the same procedure as u, all the data facts that might hold in v as witnessed by same-context executions that start in u and assuming that d_1 holds at the beginning of each such execution.

Previous Results. The on-demand analysis problem admits a number of solutions that lie in the preprocessing/query spectrum. On the one end, the preprocessing phase can be disregarded, and every on-demand query be treated anew. Since each query starts a separate instance of IFDS, the time to answer it is $O(n \cdot |D|^3)$, for both pair and single-source queries [Reps et al., 1995]. On the other end, all possible queries can be pre-computed and cached in the preprocessing phase in time $O(n^2 \cdot |D|^3)$, after which each query costs time proportional to the size of the output (i.e., $O(1)$) for pair queries and $O(n \cdot |D|)$ for single-source queries). Note that this full preprocessing also incurs a cost $O(n^2 \cdot |D|^2)$ in space for storing the cache table, which is often prohibitive. On-demand analysis was more thoroughly studied in [Horwitz et al., 1995]. The main idea is that, instead of pre-computing the answer to all possible queries, the analysis results obtained by handling each query are memoized to a cache table, and are used for speeding up the computation of subsequent queries. This is a heuristic-based approach that often works well in practice, however, the only guarantee provided is that of *same-worst-case-complexity*, which states that in the worst case, the algorithm uses $O(n^2 \cdot |D|^3)$ time and $O(n^2 \cdot |D|^2)$ space, similarly to the complete preprocessing case. This guarantee is inadequate for runtime applications such as the example of Figure 3.1, as it would require either (i) to run a full analysis, or (ii) to run a partial analysis which might wrongly conclude that `h` is reachable, and thus compile it. Both cases incur a large runtime overhead, either because we run a full analysis, or because we compile an expensive function.

Our Contributions. We provide algorithms for on-demand IFDS analyses that have strong worst-case time complexity guarantees and thus lead to more predictable performance than mere heuristics. Our contributions in this chapter are as follows:

1. We develop an algorithm that, given a program represented as a supergraph of size n and a data fact domain D, solves the on-demand same-context IFDS problem while spending (i) $O(n \cdot |D|^3)$ time in the preprocessing phase, and (ii) $O(\lceil |D| / \log n \rceil)$ time for a pair query and $O(n \cdot |D|^2 / \log n)$ time for a single-source query in the query phase. Observe that when $|D| = O(1)$, the preprocessing and query times are proportional to the size of the input and outputs, respectively, and are thus *optimal*[*]. In addition, our algorithm uses $O(n \cdot |D|^2)$ space at all times, which is proportional to the size of the input, and is thus *space optimal.* Hence, our algorithm not only improves upon previous state-of-the-art solutions, but also ensures optimality in both time and space.

2. We also show that after our one-time preprocessing, each query is *perfectly parallelizable*, i.e., every bit of the output can be produced by a single thread in $O(1)$ time. This makes our techniques particularly useful to speculative optimizations, since the analysis is guaranteed to take constant time and thus incur little runtime overhead. Although the parallelization of data-flow analysis has been considered before [Lee and Ryder, 1992, Rodriguez and Lhoták, 2011], this is the first time to obtain solutions that span beyond

[*]Note that we count the input itself as part of the space usage.

heuristics and offer theoretical guarantees. Moreover, this is a rather surprising result, given that general IFDS is known to be P-complete.

3. We provide experimental results showing that after only a lightweight preprocessing, we obtain a significant speedup in the query phase compared to standard on-demand techniques in the literature. Also, our parallel implementation achieves a speedup close to the theoretical optimal, which illustrates that the perfect parallelization of the problem is realized by our approach in practice.

3.2 The IFDS Framework

IFDS [Reps et al., 1995] is a ubiquitous and general framework for interprocedural data-flow analyses that have finite domains and distributive flow functions. It encompasses a wide variety of analyses, including truly-live variables, copy constant propagation, possibly-uninitialized variables, secure information-flow, and gen/kill or bitvector problems such as reaching definitions, available expressions and live variables [Reps et al., 1995, Bodden, 2012]. IFDS obtains *interprocedurally precise* solutions. In contrast to intraprocedural analysis, in which precise denotes "meet-over-all-paths", interprocedurally precise solutions only consider valid paths, i.e. paths in which when a function reaches its end, control returns back to the site of the most recent call [Sharir and Pnueli, 1981]. In this section, we provide a comprehensive overview of the IFDS framework.

Model of computation. In this chapter, we consider the standard RAM model with word size $W = \Theta(\log n)$, where n is the size of our input. In this model, one can store W bits in one word (aka "word tricks") and arithmetic and bitwise operations between pairs of words can be performed in $O(1)$ time. In practice, word size is a property of the machine and not the analysis. Modern machines have words of size at least 64. Since the size of real-world input instances never exceeds 2^{64}, the assumption of word size $W = \Theta(\log n)$ is well-realized in practice and no additional effort is required by the implementer to account for W in the context of data flow analysis.

Flow Graphs and Supergraphs. In IFDS, a program with k procedures is specified by a *supergraph*, i.e. a graph $G = (V, E)$ consisting of k flow graphs $G_1, \ldots, G_k$, one for each procedure, and extra edges modeling procedure-calls. Flow graphs represent procedures in the usual way, i.e. they contain one vertex v_i for each statement i and there is an edge from v_i to v_j if the statement j may immediately follow the statement i in an execution of the procedure. The only exception is that a procedure-call statement i is represented by two vertices, a *call* vertex c_i and a *return-site* vertex r_i. The vertex c_i only has incoming edges, and the vertex r_i only has outgoing edges. There is also a *call-to-return-site* edge from c_i to r_i. The call-to-return-site edges are included for passing intraprocedural information, such as information about local variables, from c_i to r_i. Moreover, each flow graph G_l has a unique *start* vertex s_l and a unique *exit* vertex e_l.

The supergraph G also contains the following edges for each procedure-call i with call vertex c_i and return-site vertex r_i that calls a procedure l: (i) an interprocedural *call-to-start* edge from c_i to the start vertex of the called procedure, i.e. s_l, and (ii) an interprocedural *exit-to-return-site* edge from the exit vertex of the called procedure, i.e. e_l, to r_i.

Example 3.2. *Figure 3.2 shows a simple C++ program on the left and its supergraph on the right. Each statement i of the program has a corresponding vertex v_i in the supergraph, except for statement 7, which is a procedure-call statement and hence has a corresponding call vertex c_7 and return-site vertex r_7.*

Interprocedurally valid paths. Not every path in the supergraph G can potentially be realized by an execution of the program. Consider a path P in G and let P' be the sequence of vertices obtained by removing every v_i from P, i.e. P' only consists of c_i's and r_i's. Then, P is called a *same-context valid path* if P' can be generated from S in this grammar:

$$\begin{array}{ll} S & \rightarrow c_i \;\; S \;\; r_i \;\; S \quad \text{for a procedure-call statement } i \\ & \mid \;\; \varepsilon \end{array}.$$

Moreover, P is called an *interprocedurally valid path* or simply *valid* if P' can be generated from the nonterminal S' in the following grammar:

```
void f(int *&x, int *y){
        y = new int(1);
        y = new int(2);
}

int main(){
        int *x, *y;
        f(x,y);
        *x += *y;
}
```

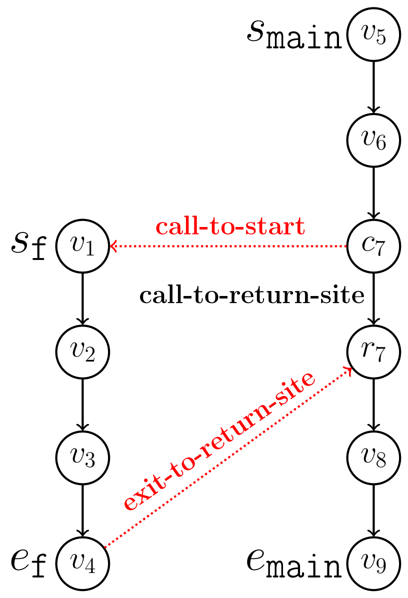


Figure 3.2: A C++ program and its supergraph.

$$\begin{aligned} S' &\to S' \ c_i \ S \qquad \text{for a procedure-call statement } i \\ &\mid \ S \end{aligned}$$

For any two vertices u, v of the supergraph G, we denote the set of all interprocedurally valid paths from u to v by $\mathsf{IVP}(u, v)$ and the set of all same-context valid paths from u to v by $\mathsf{SCVP}(u, v)$. Informally, a valid path starts from a statement in a procedure p of the program and goes through a number of procedure-calls while respecting the rule that whenever a procedure ends, control should return to the return-site in its parent procedure. A same-context valid path is a valid path in which every procedure-call ends and hence control returns back to the initial procedure p in the same context.

IFDS [Reps et al., 1995]. An IFDS problem *instance* is a tuple $I = (G, D, F, M, \sqcap)$ where:

- $G = (V, E)$ is a supergraph as above.
- D is a finite set, called the *domain*, and each $d \in D$ is called a *data flow fact*.
- The *meet operator* $\sqcap$ is either intersection or union.
- $F \subseteq 2^D \to 2^D$ is a set of *distributive flow functions* over $\sqcap$, i.e. for each function $f \in F$ and every two sets of facts $D_1, D_2 \subseteq D$, we have $f(D_1 \sqcap D_2) = f(D_1) \sqcap f(D_2)$.

- $M : E \to F$ is a map that assigns a distributive flow function to each edge of the supergraph.

Let $P = (w_i)_{i=0}^{k}$ be a path in G, $e_i = (w_{i-1}, w_i)$ and $m_i = M(e_i)$. In other words, the e_i's are the edges appearing in P and the m_i's are their corresponding distributive flow functions. The *path function* of P is defined as: $\mathsf{pf}_P := m_k \circ \cdots \circ m_2 \circ m_1$ where $\circ$ denotes function composition. The solution of I is the collection of values $\{\mathsf{MVP}_v\}_{v \in V}$:

$$\mathsf{MVP}_v := \bigsqcap_{P \in \mathsf{IVP}(s_{\mathsf{main}}, v)} \mathsf{pf}_P(\top).$$

Intuitively, the solution is defined by taking *meet-over-all-valid-paths*. If the meet operator is union, then MVP_v is the set of data flow facts that *may* hold at v, when v is reached in *some* execution of the program. Conversely, if the meet operator is intersection, then MVP_v consists of data flow facts that *must* hold at v in *every* execution of the program that reaches v. Similarly, we define the same-context solution of I as the collection of values $\{\mathsf{MSCP}_v\}_{v \in V_{\mathsf{main}}}$ defined as follows:

$$\mathsf{MSCP}_v := \bigsqcap_{P \in \mathsf{SCVP}(s_{\mathsf{main}}, v)} \mathsf{pf}_P(\top). \tag{3.1}$$

The intuition behind MSCP is similar to that of MVP, except that in MSCP_v we consider *meet-over-same-context-paths* (corresponding to runs that return to the same stack state).

Remark 3.1. *We note two points about the IFDS framework:*

- *As in [Reps et al., 1995], we only consider IFDS instances in which the meet operator is union. Instances with intersection can be reduced to union instances by dualization [Reps et al., 1995].*

- *For brevity, we are considering a global domain D, while in many applications the domain is procedure-specific. This does not affect the generality of our approach and our algorithms remain correct for the general case where each procedure has its own dedicated domain. Indeed, our implementation supports the general case.*

Succinct Representations. A distributive function $f : 2^D \to 2^D$ can be succinctly represented by a relation $R_f \subseteq (D \cup \{\mathbf{0}\}) \times (D \cup \{\mathbf{0}\})$ defined as:

$$\begin{aligned} R_f := \; & \{(\mathbf{0}, \mathbf{0})\} \\ \cup \; & \{(\mathbf{0}, b) \mid b \in f(\emptyset)\} \\ \cup \; & \{(a, b) \mid b \in f(\{a\}) - f(\emptyset)\}. \end{aligned}$$

Given that f is distributive over union, we have $f(\{d_1, \ldots, d_k\}) = f(\{d_1\}) \cup \cdots \cup f(\{d_k\})$. Hence, to specify f it is sufficient to specify $f(\emptyset)$ and $f(\{d\})$ for each $d \in D$. This is exactly what R_f does. In short, we have: $f(\emptyset) = \{b \in D \mid (\mathbf{0}, b) \in R_f\}$ and $f(\{d\}) = f(\emptyset) \cup \{b \in D \mid (d, b) \in R_f\}$. Moreover, we can represent the relation R_f as a bipartite graph H_f in which each part consists of the vertices $D \cup \{\mathbf{0}\}$ and R_f is the set of edges. For brevity, we define $D^* := D \cup \{\mathbf{0}\}$.

Example 3.3. *Let $D = \{a, b\}$. Figure 3.3 provides several examples of bipartite graphs representing distributive functions.*

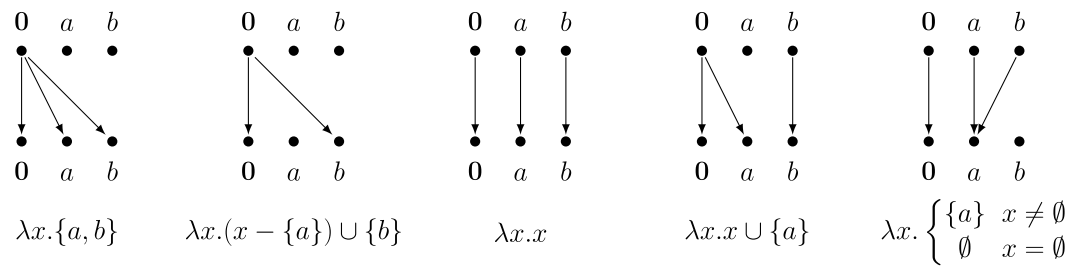

$\lambda x.\{a, b\}$
$\lambda x.(x - \{a\}) \cup \{b\}$
$\lambda x.x$
$\lambda x.x \cup \{a\}$
$\lambda x. \begin{cases} \{a\} & x \neq \emptyset \\ \emptyset & x = \emptyset \end{cases}$

Figure 3.3: Succinct representation of several distributive functions.

Bounded Bandwidth Assumption. Following [Reps et al., 1995], we assume that the bandwidth in function calls and returns is bounded by a constant. In other words, there is a small constant b, such that for every edge e that is a call-to-start or exit-to-return-site edge, every vertex in the graph representation $H_{M(e)}$ has degree b or less. This is a classical assumption in IFDS [Reps et al., 1995, Bodden, 2012] and models the fact that every parameter in a called function is only dependent on a few variables in the callee (and conversely, every returned value is only dependent on a few variables in the called function).

Composition of Distributive Functions. Let f and g be distributive functions and R_f and R_g their succinct representations. It is easy to verify that $g \circ f$ is also distributive, hence it has a succinct representation $R_{g \circ f}$. Moreover, we have $R_{g \circ f} = R_f; R_g = \{(a, b) \mid \exists c \quad (a, c) \in R_f \wedge (c, b) \in R_g\}$.

Example 3.4. *In terms of graphs, to compute $H_{g \circ f}$, we first take H_f and H_g, then contract corresponding vertices in the lower part of H_f and the upper part of H_g, and finally compute reachability from the topmost part to the bottommost part of the resulting graph. Consider $f(x) = x \cup \{a\}$, $g(x) = \{a\}$ for $x \neq \emptyset$ and $g(\emptyset) = \emptyset$, then $g \circ f(x) = \{a\}$ for all $x \subseteq D$. Figure 3.4 shows contracting of corresponding vertices in H_f and H_g (left) and using reachability to obtain $H_{g \circ f}$ (right).*

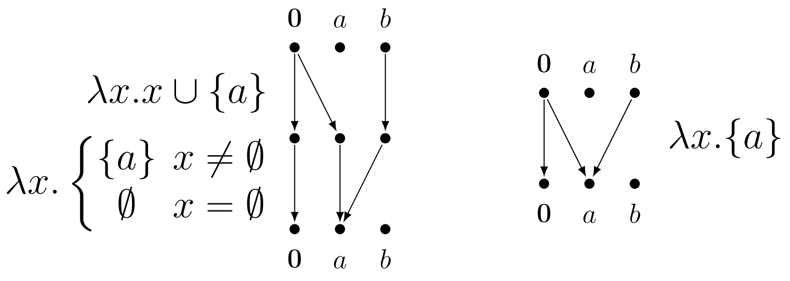


Figure 3.4: Obtaining $H_{g \circ f}$ (right) from H_f and H_g (left)

Exploded Supergraph. Given an IFDS instance $I = (G, D, F, M, \cup)$ with supergraph $G = (V, E)$, its *exploded supergraph* $\overline{G}$ is obtained by taking $|D^*|$ copies of each vertex in V, one corresponding to each element of D^*, and replacing each edge e with the graph representation $H_{M(e)}$ of the flow function $M(e)$. Formally, $\overline{G} = (\overline{V}, \overline{E})$ where $\overline{V} = V \times D^*$ and

$$\overline{E} = \left\{((u, d_1), (v, d_2)) \mid e = (u, v) \in E \ \wedge \ (d_1, d_2) \in R_{M(e)}\right\}.$$

A path $\overline{P}$ in $\overline{G}$ is (same-context) valid, if the path P in G, obtained by ignoring the second component of every vertex in $\overline{P}$, is (same-context) valid. As shown in [Reps et al., 1995], for a data flow fact $d \in D$ and a vertex $v \in V$, we have $d \in \mathsf{MVP}_v$ iff there is a valid path in $\overline{G}$ from (s_{main}, d') to (v, d) for some $d' \in \top \cup \{\mathbf{0}\}$. Hence, the IFDS problem is reduced to reachability by valid paths in $\overline{G}$. Similarly, the same-context IFDS problem is reduced to reachability by same-context valid paths in $\overline{G}$.

Example 3.5. *Consider a null pointer analysis on the program in Figure 3.2. At each program point, we want to know which pointers can potentially be null. We first model this problem as an IFDS instance. Let $D = \{\bar{x}, \bar{y}\}$, where $\bar{x}$ is the data flow fact that x might be null and $\bar{y}$ is defined similarly. Figure 3.5 shows the same program and its exploded supergraph.*

At point 8, the values of both pointers x and y are used. Hence, if either of x or y is null at 8, a null pointer error will be raised. However, as evidenced by the two valid paths shown in red, both x and y might be null at 8. The pointer y might be null because it is passed to the function `f` by value (instead of by reference) and keeps its local value in the transition from c_7 to r_7, hence the edge $((c_7, \bar{y}), (r_7, \bar{y}))$ is in $\overline{G}$. On the other hand, the function `f` only initializes y, which is its own local variable, and does not change x (which is shared with `main`).

Formal Problem Definition. We consider same-context IFDS problems in which the flow graphs G_i have a treewidth of at most t for a fixed constant t. We extend the classical notion of same-context IFDS solution in two ways: (i) we allow arbitrary start points for the analysis, i.e. we do not limit our analyses to same-context valid paths that start at s_{main}; and (ii) instead of a one-shot algorithm, we consider a two-phase process in which the algorithm first preprocesses the input instance and is then provided with a series of queries to answer. We formalize these points below. We fix an IFDS instance $I = (G, D, F, M, \cup)$ with exploded supergraph $\overline{G} = (\overline{V}, \overline{E})$.

Meet over Same-context Valid Paths. We extend the definition of MSCP by specifying a start vertex u and an initial set Δ of data flow facts that hold at u. Formally, for any

```
1 void f(int *&x, int *y) {
2         y = new int(1);
3         y = new int(2);
4 }

5 int main() {
6         int *x, *y;
7         f(x,y);
8         *x += *y;
9 }
```

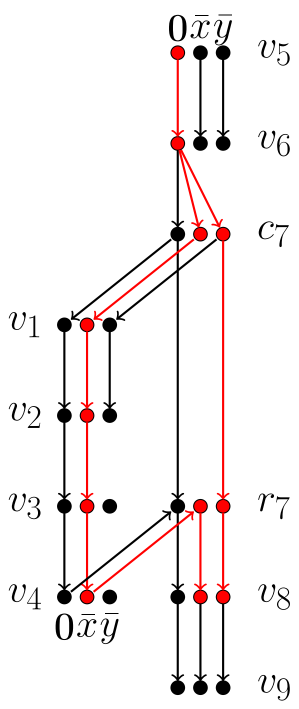


Figure 3.5: A Program (left) and its Exploded Supergraph (right).

vertex v that is in the same flow graph as u, we define:

$$\mathsf{MSCP}_{u,\Delta,v} := \bigsqcap_{P \in \mathsf{SCVP}(u,v)} \mathsf{pf}_P(\Delta). \tag{3.2}$$

The only difference between (3.2) and (3.1) is that in (3.1), the start vertex u is fixed as s_{main} and the initial data-fact set Δ is fixed as $\top$, while in (3.2), they are free to be any vertex/set.

Reduction to Reachability. As explained above, computing MSCP is reduced to reachability via same-context valid paths in the exploded supergraph $\overline{G}$. This reduction does not depend on the start vertex and initial data flow facts. Hence, for a data flow fact $d \in D$, we have $d \in \mathsf{MSCP}_{u,\Delta,v}$ iff in the exploded supergraph $\overline{G}$ the vertex (v, d) is reachable via same-context valid paths from a vertex (u, δ) for some $\delta \in \Delta \cup \{\mathbf{0}\}$. Hence, we define the following types of queries:

Pair Query. A pair query provides two vertices (u, d_1) and (v, d_2) of the exploded supergraph $\overline{G}$ and asks whether they are reachable by a same-context valid path. Hence, the answer to a pair query is a single bit. Intuitively, if $d_2 = \mathbf{0}$, then the query is simply asking if v is reachable from u by a same-context valid path in G. Otherwise, d_2 is a data flow fact and the query is asking whether $d_2 \in \mathsf{MSCP}_{u,\{d_1\}\cap D,v}$.

Single-source Query. A single-source query provides a vertex (u, d_1) and asks for all vertices (v, d_2) that are reachable from (u, d_1) by a same-context valid path. Assuming that u is in the flow graph $G_i = (V_i, E_i)$, the answer to the single source query is a sequence of $|V_i|\cdot|D^*|$ bits, one for each $(v, d_2) \in V_i \times D^*$, signifying whether it is reachable by same-context valid paths from (u, d_1). Intuitively, a single-source query asks for all pairs (v, d_2) such that (i) v is reachable from u by a same-context valid path and (ii) $d_2 \in \mathsf{MSCP}_{u,\{d_1\}\cap D,v} \cup \{\mathbf{0}\}$.

Intuition. We note the intuition behind such queries. We observe that since the functions in F are distributive over $\cup$, we have $\mathsf{MSCP}_{u,\Delta,v} = \cup_{\delta\in\Delta}\mathsf{MSCP}_{u,\{\delta\},v}$, hence $\mathsf{MSCP}_{u,\Delta,v}$ can be computed by $O(|\Delta|)$ single-source queries.

Classical Solution. The classical IFDS algorithm [Reps et al., 1995] is a work-list algorithm that computes reachability via valid paths in an iterative way. It maintains a table of reachability and updates it until convergence to the solution. As mentioned before, this leads to a runtime of $O(n \cdot |D|^3)$ per query. In the remainder of this chapter, we show how parameterization by treewidth can help for more efficient algorithms.

3.3 Treewidth-based Algorithms

3.3.1 Notation and Preliminary Lemmas

Notation. Consider a rooted tree $T = (V_T, E_T)$ with root vertex r. For an arbitrary vertex $v \in V_T$, the *depth* of v, denoted by d_v, is defined as the length of the unique path from v to r. The *depth* or *height* of T is the maximum depth among its vertices. We denote the set of ancestors of v by $\mathsf{A}_v^{\uparrow}$ and its descendants by $\mathsf{D}_v^{\downarrow}$. It is straightforward to see that for every $0 \leq d \leq \mathsf{d}_v$, the vertex v has a unique ancestor with depth d. We denote this ancestor by a_v^d. The subtree $T_v^{\downarrow}$ corresponding to v is defined as $T[\mathsf{D}_v^{\downarrow}] = (\mathsf{D}_v^{\downarrow}, E_T \cap \mathsf{D}_v^{\downarrow} \times \mathsf{D}_v^{\downarrow})$, i.e. the part of T that consists of v and its descendants. Given two vertices $u, v \in V_T$, the *lowest common* *ancestor* $\mathsf{lca}(u, v)$ of u and v is defined as $\operatorname{argmax}_{w \in \mathsf{A}_u^\uparrow \cap \mathsf{A}_v^\uparrow} \mathsf{d}_w$. In other words, $\mathsf{lca}(u, v)$ is the common ancestor of u and v with maximum depth, i.e. which is farthest from the root.

In our algorithm for treewidth-based IFDS analysis, we need to have a *balanced* tree decomposition. Moreover, we perform lowest common ancestor queries on the tree decomposition every time we need to answer an IFDS query. As such, we will rely on the following two classical lemmas:

Lemma 3.1 ([Bodlaender and Hagerup, 1998]). *Given a graph G with constant treewidth t, a binary tree decomposition with $O(n)$ bags, height $O(\log n)$ and width $O(t)$ can be computed in linear time.*

Lemma 3.2 ([Harel and Tarjan, 1984]). *Given a rooted tree T with n vertices, there is an algorithm that preprocesses T in $O(n)$ and can then answer lowest common ancestor queries, i.e. queries that provide two vertices u and v and ask for $\mathsf{lca}(u, v)$, in $O(1)$.*

3.3.2 Preprocessing

The original solution to the IFDS problem, as first presented in [Reps et al., 1995], reduces the problem to reachability over a newly constructed graph. We follow a similar approach, except that we exploit the low-treewidth property of our flow graphs at every step. Our preprocessing is described below. It starts with computing constant-width tree decompositions for each of the flow graphs. We then use standard techniques to make sure that our tree decompositions have a nice form, i.e. that they are balanced and binary. Then comes a reduction to reachability, which is similar to [Reps et al., 1995]. Finally, we precompute specific useful reachability information between vertices in each bag and its ancestors. As it turns out in the next section, this information is sufficient for computing reachability between any pair of vertices, and hence for answering IFDS queries.

Overview. Our preprocessing consists of the following steps:

(1) **Finding Tree Decompositions.** In this step, we compute a tree decomposition $(T_i, \langle B_{i,j} \rangle)$ of constant width t for each flow graph G_i. This can either be done by applying the

algorithm of [Bodlaender, 1996] directly on G_i, or by using an algorithm due to Thorup [Thorup, 1998] and parsing the program.

(2) **Balancing and Binarizing.** In this step, we balance the tree decompositions T_i using the algorithm of Lemma 3.1 and make them binary using the standard process of [Chaudhuri and Zaroliagis, 2000].

(3) **LCA Preprocessing.** We preprocess the T_i's for answering lowest common ancestor queries using Lemma 3.2.

(4) **Reduction to Reachability.** In this step, we modify the exploded supergraph $\overline{G} = (\overline{V}, \overline{E})$ to obtain a new graph $\hat{G} = (\overline{V}, \hat{E})$, such that for every pair of vertices (u, d_1) and (v, d_2), there is a path from (u, d_1) to (v, d_2) in $\hat{G}$ iff there is a *same-context valid path* from (u, d_1) to (v, d_2) in $\overline{G}$. So, this step reduces the problem of reachability via same-context valid paths in $\overline{G}$ to simple reachability in $\hat{G}$.

(5) **Local Preprocessing.** In this step, for each pair of vertices (u, d_1) and (v, d_2) for which there exists a bag B_j such that both u and v appear in B_j, we compute and cache whether $(u, d_1) \rightsquigarrow (v, d_2)$ in $\hat{G}$. We write $(u, d_1) \rightsquigarrow_{\mathsf{local}} (v, d_2)$ to denote a reachability established in this step.

(6) **Ancestors Reachability Preprocessing.** In this step, we compute reachability information between each vertex in a bag and vertices appearing in its ancestors in the tree decomposition. Concretely, for each pair of vertices (u, d_1) and (v, d_2) such that u appears in a bag B_j and v appears in a bag B_k and k is an ancestor of j, we establish and remember whether $(u, d_1) \rightsquigarrow (v, d_2)$ in $\hat{G}$ and whether $(v, d_2) \rightsquigarrow (u, d_1)$ in $\hat{G}$. As above, we use the notations $(u, d_1) \rightsquigarrow_{\mathsf{anc}} (v, d_2)$ and $(v, d_2) \rightsquigarrow_{\mathsf{anc}} (u, d_1)$.

Steps (1)–(3) above are standard and well-known processes. We now provide details of steps (4)–(6). To skip the details and read about the query phase, see Section 3.3.4 below.

Step (4): Reduction to Reachability

In this step, our goal is to compute a new graph $\hat{G}$ from the exploded supergraph $\overline{G}$ such that there is a path from (u, d_1) to (v, d_2) in $\hat{G}$ iff there is a same-context valid path from (u, d_1) to (v, d_2) in $\overline{G}$. The idea behind this step is the same as that of the *tabulation algorithm* in [Reps et al., 1995].

Summary Edges. Consider a call vertex c_l in G and its corresponding return-site vertex r_l. For $d_1, d_2 \in D^*$, the edge $((c_l, d_1), (r_l, d_2))$ is called a *summary edge* if there is a same-context valid path from (c_l, d_1) to (r_l, d_2) in the exploded supergraph $\overline{G}$. Intuitively, a summary edge summarizes the effects of procedure calls (same-context interprocedural paths) on the reachability between c_l and r_l. From the definition of *summary edges*, it is straightforward to verify that the graph $\hat{G}$ obtained from $\overline{G}$ by adding every summary edge and removing every interprocedural edge has the desired property, i.e. a pair of vertices are reachable in $\hat{G}$ iff they are reachable by a same-context valid path in $\overline{G}$. Hence, we first find all summary edges and then compute $\hat{G}$. This is shown in Algorithm 3.1.

We now describe what Algorithm 3.1 does. Let s_p be the start point of a procedure p. A *shortcut edge* is an edge $((s_p, d_1), (v, d_2))$ such that v is in the same procedure p and there is a same-context valid path from (s_p, d_1) to (v, d_2) in $\overline{G}$. The algorithm creates an empty graph $H = (\overline{V}, E')$. Note that H is implicitly represented by only saving E'. It also creates a queue Q of edges to be added to H (initially $Q = \overline{E}$) and an empty set S which will store the summary edges. The goal is to construct H such that it contains (i) *intraprocedural* edges of $\overline{G}$, (ii) summary edges, and (iii) shortcut edges.

It constructs H one edge at a time. While there is an unprocessed intraprocedural edge $e = ((u, d_1), (v, d_2))$ in Q, it chooses one such e and adds it to H (lines 5–10). Then, if (u, d_1) is reachable from (s_p, d_3) via a same-context valid path, then by adding the edge e, the vertex (v, d_2) also becomes accessible from (s_p, d_3). Hence, it adds the shortcut edge $((s_p, d_3), (v, d_2))$ to Q, so that it is later added to the graph H. Also if the new edge is itself a shortcut edge (lines 14–17), then new shortcut edges should be added to the successors of (v, d_2). Moreover, if u is the start s_p of the procedure p and v is its end e_p, then for every call vertex c_l calling the procedure p and its respective return-site r_l, we can add summary

Algorithm 3.1: Computing $\hat{G}$ in Step (4)

```
1  $Q \leftarrow \overline{E}$;
2  $S \leftarrow \emptyset$;
3  $E' \leftarrow \emptyset$;
4  **while** $Q \neq \emptyset$ **do**
5      Choose $e = ((u, d_1), (v, d_2)) \in Q$;
6      $Q \leftarrow Q - \{e\}$;
7      **if** $(u, v)$ *is an interprocedural edge, i.e. a call-to-start or exit-to-return-site edge* **then**
8          **continue**;
9      $p \leftarrow$ the procedure s.t. $u, v \in V_p$;
10     $E' \leftarrow E' \cup \{e\}$;
11     **foreach** $d_3$ s.t. $((s_p, d_3), (u, d_1)) \in E'$ or $(s_p, d_3) = (u, d_1)$ **do**
12         **if** $((s_p, d_3), (v, d_2)) \notin E' \cup Q$ **then**
13             $Q \leftarrow Q \cup \{((s_p, d_3), (v, d_2))\}$;
14     **if** $u = s_p$ **then**
15         **foreach** $(w, d_3)$ s.t. $((v, d_2), (w, d_3)) \in E'$ **do**
16             **if** $((u, d_1), (w, d_3)) \notin E' \cup Q$ **then**
17                 $Q \leftarrow Q \cup \{((u, d_1), (w, d_3))\}$;
18     **if** $u = s_p$ **and** $v = e_p$ **then**
19         **foreach** $(c_l, d_3)$ s.t. $((c_l, d_3), (u, d_1)) \in \overline{E}$ **do**
20             **foreach** $d_4$ s.t. $((v, d_2), (r_l, d_4)) \in \overline{E}$ **do**
21                 **if** $((c_l, d_3), (r_l, d_4)) \notin E' \cup Q$ **then**
22                     $S \leftarrow S \cup \{((c_l, d_3), (r_l, d_4))\}$;
23                     $Q \leftarrow Q \cup \{((c_l, d_3), (r_l, d_4))\}$;
24 $\hat{G} \leftarrow \overline{G}$;
25 **foreach** $e = ((u, d_1), (v, d_2)) \in \overline{E}$ **do**
26     **if** $u$ *and* $v$ *are not in the same procedure* **then**
27         $\hat{G} = \hat{G} - \{e\}$;
28 $\hat{G} \leftarrow \hat{G} \cup S$;
```

edges that summarize the effect of calling p (lines 18–23). Finally, lines 24–28 compute $\hat{G}$ as discussed above.

Correctness. As argued above, every edge that is added to H is either intraprocedural, a summary edge or a shortcut edge. Moreover, all such edges are added to H, because H is constructed one edge at a time and every time an edge e is added to H, all the summary/shortcut edges that might occur as a result of adding e to H are added to the queue Q and hence later to H. Therefore, Algorithm 3.1 correctly computes summary edges and the graph $\hat{G}$.

Complexity. Note that the graph H has at most $O(|E| \cdot |D^*|^2)$ edges. Addition of each edge corresponds to one iteration of the while loop at line 4 of Algorithm 3.1. Moreover, each iteration takes $O(|D^*|)$ time, because the loops at lines 11 and 15 iterate over at most $|D^*|$ possible values for d_3 (and constantly many values for w) and the loops at lines 19 and 20 have constantly many iterations due to the bounded bandwidth assumption (Section 3.2). Since $|D^*| = O(|D|)$ and $|E| = O(n)$, the total runtime of Algorithm 3.1 is $O(|n| \cdot |D|^3)$. For a more detailed analysis, see [Reps et al., 1995, Appendix].

Step (5): Local Preprocessing

In this step, we compute the set R_{local} of local reachability edges, i.e. edges of the form $((u, d_1), (v, d_2))$ such that u and v appear in the same bag b of a tree decomposition T_i and $(u, d_1) \rightsquigarrow (v, d_2)$ in $\hat{G}$. We write $(u, d_1) \rightsquigarrow_{\mathsf{local}} (v, d_2)$ to denote $((u, d_1), (v, d_2)) \in R_{\mathsf{local}}$. Note that $\hat{G}$ has no interprocedural edges. Hence, we can process each T_i separately. We use a divide-and-conquer technique similar to the kernelization method used in [Chaudhuri and Zaroliagis, 2000] (Algorithm 3.2).

Algorithm 3.2: Local Preprocessing in Step (5)

```
1  $R_{\mathsf{local}} \leftarrow \emptyset$;
2  foreach $(T_i, \langle B_{i,j} \rangle)$ do
3      computeLocalReachability($T_i, \langle B_{i,j} \rangle$);
4  Function computeLocalReachability($T, \langle B_i \rangle$)
5      Choose a leaf node $l$ of $T$;
6      $p \leftarrow$ parent of $l$;
7      foreach $u, v \in B_l,\ d_1, d_2 \in D^*$ s.t. $(u, d_1) \rightsquigarrow (v, d_2)$ in $\hat{G}[B_l \times D^*]$ do
8          $\hat{G} = \hat{G} \cup \{((u, d_1), (v, d_2))\}$;
9          $R_{\mathsf{local}} = R_{\mathsf{local}} \cup \{((u, d_1), (v, d_2))\}$;
10     if $p \neq$ null then
11         computeLocalReachability($T - l$);
12         foreach $u, v \in B_l,\ d_1, d_2 \in D^*$ s.t. $(u, d_1) \rightsquigarrow (v, d_2)$ in $\hat{G}[B_l \times D^*]$ do
13             $\hat{G} = \hat{G} \cup \{((u, d_1), (v, d_2))\}$;
14             $R_{\mathsf{local}} = R_{\mathsf{local}} \cup \{((u, d_1), (v, d_2))\}$;
```

Algorithm 3.2 processes each tree decomposition T_i separately. When processing T, it chooses a leaf node l of T and computes all-pairs reachability on the induced subgraph

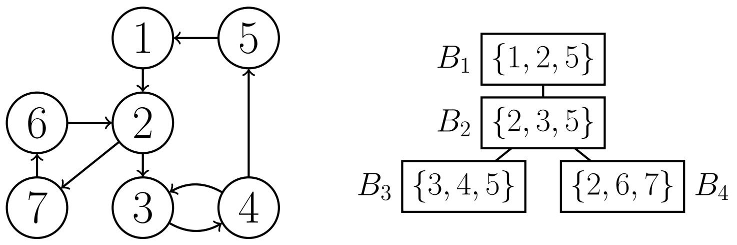


Figure 3.6: A Graph G and one of its Tree Decompositions T.

$H_l = \hat{G}[B_l \times D^*]$, consisting of vertices that appear in B_l. Then, for each pair of vertices (u, d_1) and (v, d_2) s.t. u and v appear in B_l and $(u, d_1) \rightsquigarrow (v, d_2)$ in H_l, the algorithm adds the edge $((u, d_1), (v, d_2))$ to both R_{local} and $\hat{G}$ (lines 7–9). Note that this does not change reachability relations in $\hat{G}$, given that the vertices connected by the new edge were reachable by a path before adding it. Then, if l is not the only node in T, the algorithm recursively calls itself over the tree decomposition $T - l$, i.e. the tree decomposition obtained by removing l and B_l (lines 10–11). Finally, it repeats the reachability computation on H_l (lines 12–14). The running time of the algorithm is $O(n \cdot |D^*|^3)$.

Example 3.6. *Consider the graph G and tree decomposition T given in Figure 3.6 and let $D^* = \{\mathbf{0}\}$, i.e. let $\hat{G}$ and $\bar{G}$ be isomorphic to G. Figure 3.7 illustrates the steps taken by Algorithm 3.2. In each step, a bag is chosen and a local all-pairs reachability computation is performed over the bag. Local reachability edges are added to R_{local} and to $\hat{G}$ (if they are not already in $\hat{G}$).*

We now prove the correctness and establish the complexity of Algorithm 3.2.

Correctness. We prove that when $\mathsf{computeLocalReachability}(T)$ ends, the set R_{local} contains all the local reachability edges between vertices that appear in the same bag in T. The proof is by induction on the size of T. If T consists of a single bag, then the local reachability computation on H_l (lines 7–9) fills R_{local} correctly. Now assume that T has n bags. Let $H_{-l} = \hat{G}[\cup_{i \neq l} B_i \times D^*]$. Intuitively, H_{-l} is the part of $\hat{G}$ that corresponds to other bags in T, i.e. every bag except the leaf bag B_l. After the local reachability computation at lines 7–9, (v, d_2) is reachable from (u, d_1) in H_{-l} only if it is reachable in $\hat{G}$. This is because (i) the

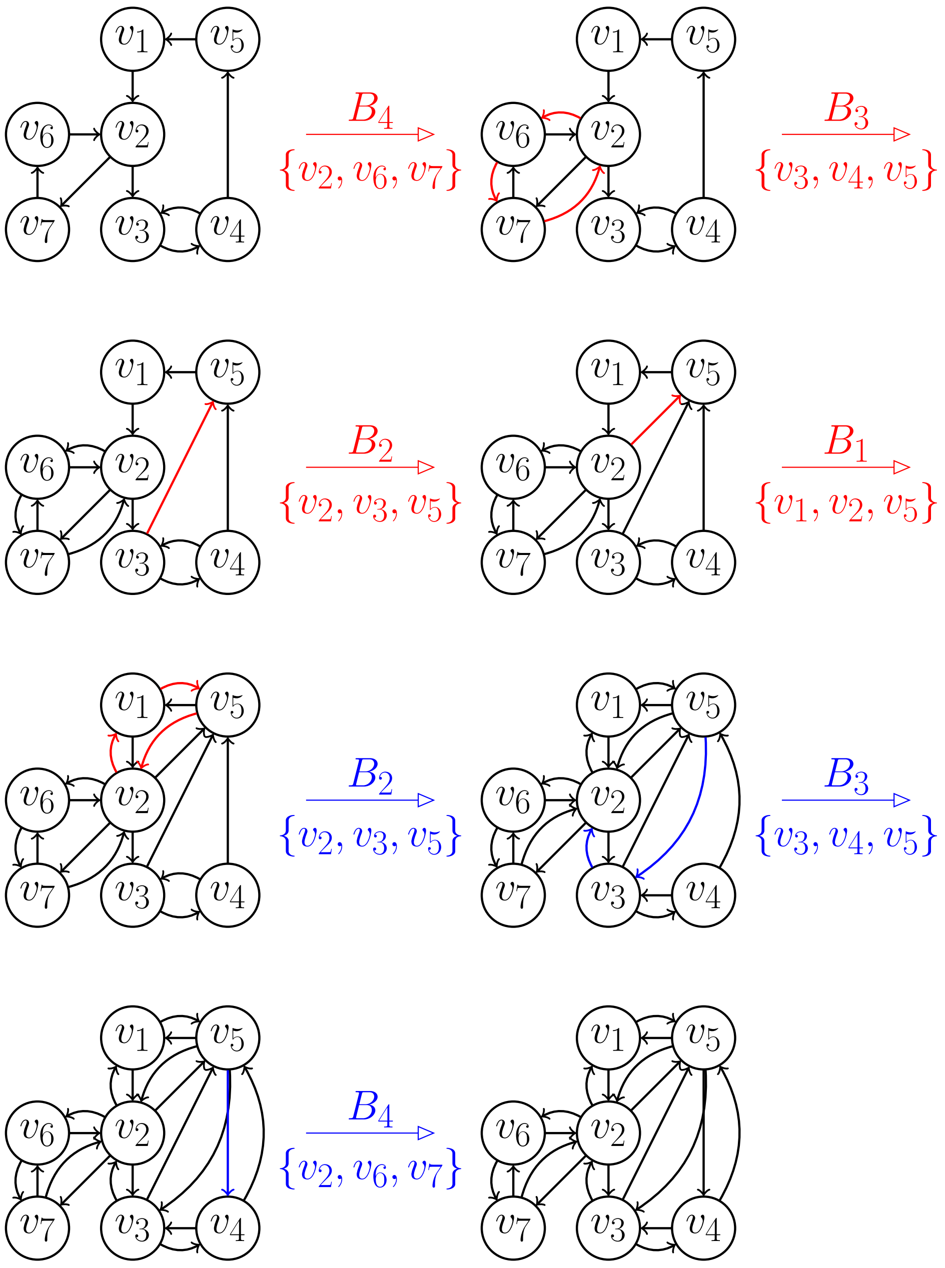


Figure 3.7: Local Preprocessing (Step 5) on the graph and decomposition of Figure 3.6

vertices of H_l and H_{-l} form a separation of $\hat{G}$ with separator $(B_l \cap B_p) \times D^*$ (Lemma 2.1) and (ii) all reachability information in H_l is now replaced by direct edges (line 8). Hence, by induction hypothesis, line 11 finds all the local reachability edges for $T - l$ and adds them to both R_{local} and $\hat{G}$. Therefore, after line 11, for every $u, v \in B_l$, we have $(u, d_1) \rightsquigarrow (v, d_2)$ in H_l iff $(u, d_1) \rightsquigarrow (v, d_2)$ in $\hat{G}$. Hence, the final all-pairs reachability computation of lines 12–14 adds all the local edges in B_l to R_{local}.

Complexity. Algorithm 3.2 performs at most two local all-pair reachability computations over the vertices appearing in each bag, i.e. $O(t \cdot |D^*|)$ vertices. Each such computation can be performed in $O(t^3 \cdot |D^*|^3)$ using standard reachability algorithms. Given that the T_i's have $O(n)$ bags overall, the total runtime of Algorithm 3.2 is $O(n \cdot t^3 \cdot |D^*|^3) = O(n \cdot |D^*|^3)$. Note that the treewidth t is a constant and hence the factor t^3 can be removed.

Step (6): Ancestors Reachability Preprocessing

This step aims to find reachability relations between each vertex of a bag and vertices that appear in the ancestors of that bag. As in the previous case, we compute a set R_{anc} and write $(u, d_1) \rightsquigarrow_{\mathsf{anc}} (v, d_2)$ if $((u, d_1), (v, d_2)) \in R_{\mathsf{anc}}$.

This step is performed by Algorithm 3.3. For each node η with corresponding bag B_η and vertex (u, d) such that $u \in B_\eta$ and each $0 \leq j < \mathsf{d}_\eta$, we maintain two sets: $F(u, d, \eta, j)$ and $F'(u, d, \eta, j)$ each containing a set of vertices whose first coordinate is in the ancestor of η at depth j. Intuitively, the vertices in $F(u, d, \eta, j)$ are reachable from (u, d). Conversely, (u, d) is reachable from the vertices in $F'(u, d, \eta, j)$. At first all F and F' sets are initialized as $\emptyset$. We process each tree decomposition T_i in a top-down manner and do the following actions at each bag:

- If a vertex u appears in both η and its parent p, then the reachability data computed for (u, d) at p can also be used in η. So, the algorithm copies this data (lines 4–7).
- If $(u, d_1) \rightsquigarrow_{\mathsf{local}} (v, d_2)$, then this reachability relation is saved in F and F' (lines 10–11). Also, any vertex that is reachable from (v, d_2) is reachable from (u, d_1), too. So,

the algorithm adds $F(v, d_2, \eta, j)$ to $F(u, d_1, \eta, j)$ (line 13). The converse happens to F' (line 14).

Algorithm 3.3: Ancestors Preprocessing in Step (6)

```
1  foreach $T_i, \langle B_{i,j} \rangle$ do
2      foreach $\eta \in V_{T_i}$ in top-down order do
3          $p \leftarrow$ parent of $\eta$;
4          foreach $u \in B_\eta \cap B_p, d \in D^*$ do
5              foreach $0 \leq j < \mathsf{d}_\eta$ do
6                  $F(u, d, \eta, j) \leftarrow F(u, d, p, j)$;
7                  $F'(u, d, \eta, j) \leftarrow F'(u, d, p, j)$;
8          foreach $u, v \in B_\eta, d_1, d_2 \in D^*$ do
9              if $(u, d_1) \rightsquigarrow_{\mathsf{local}} (v, d_2)$ then
10                 $F(u, d_1, \eta, \mathsf{d}_\eta) \leftarrow F(u, d_1, \eta, \mathsf{d}_\eta) \cup \{(v, d_2)\}$;
11                 $F'(v, d_2, \eta, \mathsf{d}_\eta) \leftarrow F'(v, d_2, \eta, \mathsf{d}_\eta) \cup \{(u, d_1)\}$;
12                 foreach $0 \leq j < \mathsf{d}_\eta$ do
13                     $F(u, d_1, \eta, j) \leftarrow F(u, d_1, \eta, j) \cup F(v, d_2, \eta, j)$;
14                     $F'(v, d_2, \eta, j) \leftarrow F'(v, d_2, \eta, j) \cup F'(u, d_1, \eta, j)$
```

Correctness. After the execution of Algorithm 3.3, $(v, d_2) \in F(u, d_1, \eta, j)$ iff (i) (v, d_2) is reachable from (u, d_1) and (ii) $u \in B_\eta$ and $v \in B_{\mathsf{a}_\eta^j}$, i.e. v appears in the ancestor of η at depth j. Similarly, $(u, d_1) \in F'(v, d_2, \eta, j)$ iff (i) (v, d_2) is reachable from (u, d_1) and (ii) $v \in B_\eta$ and $u \in B_{\mathsf{a}_\eta^j}$. We provide a proof for correctness of F, the case with F' can be handled similarly. Assume that conditions (i) and (ii) hold and let $P : (u, d_1) \rightsquigarrow (v, d_2)$ be a path in the graph. We use induction on the number l of bags between η and a_η^j. Formally, $l := \mathsf{d}_\eta - j$. If $l = 0$, then $(u, d_1) \rightsquigarrow_{\mathsf{local}} (v, d_2)$ and hence (v, d_2) is added to $F(u, d_1, \eta, j)$ at line 10. Otherwise, there is a vertex $(w, d_3) \in P$ such that $w \in B_\eta \cap B_p$ (Lemma 2.1). Therefore, $(u, d_1) \rightsquigarrow_{\mathsf{local}} (w, d_3)$ and by induction hypothesis $(v, d_2) \in F(w, d_3, p, j)$. Therefore, (v, d_2) is added to $F(w, d_3, \eta, j)$ at line 6 and then to $F(u, d_1, \eta, j)$ at line 13. The other side is easy to check.

Complexity. The algorithm considers $O(n)$ bags in line 2. For each bag, it considers $O(t \cdot |D^*|)$ different combinations of u, d in line 4. For each combination, it updates $O(\mathsf{d}_\eta)$ values in lines 5–7. Note that each F or F' set has a size of at most $t \cdot |D^*| = O(|D^*|)$. Moreover, given that the tree decompositions T_i are balanced, we have $\mathsf{d}_\eta = O(\log n)$. Hence, the total runtime of this part of the algorithm is $O(n \cdot |D^*|^2 \cdot \log n)$. Similarly, in line 8, the algorithm considers $O(t^2) = O(1)$ combinations of u, v and $O(|D^*|^2)$ combinations of d_1, d_2 and performs $O(\log n)$ updates for each of them (lines 12–14). Hence, the total runtime of this part and the whole algorithm is $O(n \cdot |D^*|^3 \cdot \log n)$.

3.3.3 Word Tricks

We now show how to reduce the time complexity of Algorithm 3.3 from $O(n \cdot |D^*|^3 \cdot \log n)$ to $O(n \cdot |D^*|^3)$ using word tricks. The idea is to pack the F and F' sets of Algorithm 3.3 into words, i.e. represent them by a binary sequence.

Given a node $\eta \in V_{T_i}$, we define δ_η as the sum of sizes of bags corresponding to all ancestors of η. The tree decompositions are balanced, so η has $O(\log n)$ ancestors. Moreover, the width is t, hence $\delta_\eta = O(t \cdot \log n) = O(\log n)$ for every node η. We perform a top-down pass of each tree decomposition T_i and compute δ_η for each η.

For every node η, $u \in B_\eta$ and $d_1 \in D^*$, we store $F(u, d_1, \eta, -)$ as a binary sequence of length $\delta_\eta \cdot |D^*|$. The first $|B_\eta| \cdot |D^*|$ bits of this sequence correspond to $F(u, d_1, \eta, \mathsf{d}_\eta)$. The next $|B_p| \cdot |D^*|$ correspond to $F(u, d_1, \eta, \mathsf{d}_b - 1)$, and so on. We use a similar encoding for F'. Using this encoding, Algorithm 3.3 can be rewritten by word tricks and bitwise operations as follows:

- Lines 5–6 copy $F(u, d, p, -)$ into $F(u, d, \eta, -)$. However, we have to shift and align the bits, so these lines can be replaced by

$$F(u, d, \eta, -) \leftarrow F(u, d, p, -) \ll |B_\eta| \cdot |D^*|;$$

- Line 10 sets a single bit to 1.

- Lines 12–13 perform a union, which can be replaced by the bitwise OR operation. Hence, these lines can be replaced by

$$F(u, d_1, \eta, -) \leftarrow F(u, d_1, \eta, -) \textbf{ OR } F(v, d_2, \eta, -);$$

- Computations on F' can be handled similarly.

Note that we do not need to compute R_{anc} explicitly given that our queries can be written in terms of the F and F' sets. It is easy to verify that using these word tricks, every W operations in lines 6, 7, 13 and 14 are replaced by one or two bitwise operations on words. Hence, the overall runtime of Algorithm 3.3 is reduced to $O\left(\frac{n \cdot |D^*|^3 \cdot \log n}{W}\right) = O(n \cdot |D^*|^3)$.

3.3.4 Answering Queries

We now describe how to answer pair and single-source queries using the data saved in the preprocessing phase.

Answering a Pair Query. Our algorithm answers a pair query from a vertex (u, d_1) to a vertex (v, d_2) as follows:

(i) If u and v are not in the same flow graph, return 0 (no).

(ii) Otherwise, let G_i be the flow graph containing both u and v. Let η_u be the root node of u in T_i, i.e. the highest node such that $u \in B_{\eta_u}$, and η_v be the root node of v. Compute $\eta := \mathsf{lca}(\eta_u, \eta_v)$.

(iii) If there exists a vertex $w \in B_\eta$ and $d_3 \in D^*$ such that $(u, d_1) \rightsquigarrow_{\mathsf{anc}} (w, d_3)$ and $(w, d_3) \rightsquigarrow_{\mathsf{anc}} (v, d_2)$, return 1 (yes), otherwise return 0 (no).

Correctness. If there is a path $P : (u, d_1) \rightsquigarrow (v, d_2)$, then we claim P must pass through a vertex (w, d_3) with $w \in B_\eta$. If $\eta = \eta_u$ or $\eta = \eta_v$, the claim is obviously true. Otherwise, consider the path $P' : \eta_u \rightsquigarrow \eta_v$ in the tree decomposition T_i. This path passes through η (by definition of η). Let $e = \{\eta, \eta'\}$ be an edge of P'. Applying the separation property (Lemma 2.1) to e, proves that P must pass through a vertex (w, d_3) with $w \in B_\eta \cap B_{\eta'} \subseteq B_\eta$. Moreover, η is an ancestor of both η_u and η_v, hence we have $(u, d_1) \rightsquigarrow_{\mathsf{anc}} (w, d_3)$ and $(w, d_3) \rightsquigarrow_{\mathsf{anc}} (v, d_2)$.

Complexity. Computing the lowest common ancestor takes $O(1)$ time. Checking all possible vertices (w, d_3) takes $O(t \cdot |D^*|) = O(|D|)$. This runtime can be decreased to $O\left(\left\lceil \frac{|D|}{\log n} \right\rceil\right)$ by word tricks.

Answering a Single-source Query. Consider a single-source query from a vertex (u, d_1) with $u \in V_i$. We can answer this query by performing $|V_i| \times |D^*|$ pair queries, i.e. by performing one pair query from (u, d_1) to (v, d_2) for each $v \in V_i$ and $d_2 \in D^*$. Since $|D^*| = O(|D|)$, the total complexity is $O\left(|V_i| \cdot |D| \cdot \left\lceil \frac{|D|}{\log n} \right\rceil\right)$ for answering a single-source query. Using a more involved preprocessing method, we can slightly improve this time to $O\left(\frac{|V_i| \cdot |D|^2}{\log n}\right)$. See [Chatterjee et al., 2020c] for more details. Based on the results above, we now present our main theorem:

Theorem 3.1. *Given an IFDS instance $I = (G, D, F, M, \cup)$, our algorithm preprocesses I in time $O(n \cdot |D|^3)$ and can then answer each pair query and single-source query in time*

$$O\left(\left\lceil \frac{|D|}{\log n} \right\rceil\right) \quad \textit{and} \quad O\left(\frac{n \cdot |D|^2}{\log n}\right), \quad \textit{respectively.}$$

3.3.5 Parallelizability and Optimality

We now turn our attention to parallel versions of our query algorithms, as well as cases where the algorithms are optimal.

Parallelizability. Assume we have k threads in our disposal.

1. Given a pair query of the form (u, d_1, v, d_2), let η_u (resp. η_v) be the root node of u (resp. v), and $\eta = \mathsf{lca}(\eta_u, \eta_v)$ the lowest common ancestor of η_u and η_v. We partition the set $B_\eta \times D^*$ into k subsets $\{A_i\}_{1 \le i \le k}$. Then, thread i handles the set A_i, as follows: for every pair $(w, d_3) \in A_i$, the thread sets the output to 1 (yes) iff $(u, d_1) \rightsquigarrow_{\mathsf{anc}} (w, d_3)$ and $(w, d_3) \rightsquigarrow_{\mathsf{anc}} (v, d_2)$.

2. Recall that a single source query (u, d_1) is answered by breaking it down to $|V_i| \times |D^*|$ pair queries, where $G_i = (V_i, E_i)$ is the flow graph containing u. Since all such pair queries are independent, we parallelize them among k threads, and further parallelize each pair query as described above.

With word tricks, parallel pair and single-source queries require $O\left(\left\lceil \frac{|D|}{k\cdot\log n} \right\rceil\right)$ and $O\left(\left\lceil \frac{n\cdot|D|}{k\cdot\log n} \right\rceil\right)$ time, respectively. Hence, for large enough k, each query requires only $O(1)$ time, and we achieve *perfect parallelism*.

Optimality. Observe that when $|D| = O(1)$, i.e. when the domain is small, our algorithm is *optimal*: the preprocessing runs in $O(n)$, which is proportional to the size of the input, and the pair query and single-source query run in times $O(1)$ and $O(n/\log n)$, respectively, each case being proportional to the size of the output. Small domains arise often in practice, e.g. in dead-code elimination or null-pointer analysis.

3.4 Experimental Results

We report on an experimental evaluation of our techniques and compare their performance to standard alternatives in the literature.

Benchmarks. We used 5 classical data-flow analyses in our experiments, including reachability (for dead-code elimination), possibly-uninitialized variables analysis, simple uninitialized variables analysis, liveness analysis of the variables, and reaching-definitions analysis. We followed the specifications in [Horwitz et al., 1995] for modeling the analyses in IFDS. We used real-world Java programs from the DaCapo benchmark suite [Blackburn et al., 2006], obtained their flow graphs using Soot [Vallée-Rai et al., 2010] and applied our JTDec tool [Chatterjee et al., 2017b] for computing balanced tree decompositions. Given that some of these benchmarks are prohibitively large, we only considered their main Java packages, i.e. packages containing the starting point of the programs. We experimented with a total of 22 benchmarks, which, together with the 5 analyses above, led to a total of 110 instances. Our instance sizes, i.e. number of vertices and edges in the exploded supergraph, range from 22 to $190,591$. See [Chatterjee et al., 2020c] for details.

Implementation and comparison. We implemented both variants of our approach, i.e. sequential and parallel, in C++. We also implemented the parts of the classical IFDS algorithm [Reps et al., 1995] and its on-demand variant [Horwitz et al., 1995] responsible for same-context queries. All of our implementations closely follow the pseudocodes of our algorithms and the ones in [Reps et al., 1995, Horwitz et al., 1995], and no additional optimizations are applied. We compared the performance of the following algorithms for randomly-generated queries:

- *SEQ.* The sequential variant of our algorithm.
- *PAR.* A variant of our algorithm in which the queries are answered using perfect parallelization and 12 threads.
- *NOPP.* The classical same-context IFDS algorithm of [Reps et al., 1995], with *no preprocessing.* NOPP performs a complete run of the classic IFDS algorithm for each query.
- *CPP.* The classical same-context IFDS algorithm of [Reps et al., 1995], with *complete preprocessing.* In this algorithm, all summary edges and reachability information are precomputed and the queries simply report the values computed in the preprocessing phase.
- *OD.* The on-demand same-context IFDS algorithm of [Horwitz et al., 1995]. This algorithm does not preprocess the input. However, it remembers the information obtained in each query and uses it to speed-up the following queries. In [Horwitz et al., 1995], this algorithm was shown to be much more effective than the classical IFDS algorithm of [Reps et al., 1995].

For each instance, we randomly generated 10,000 pair queries and 100 single-source queries. In case of single-source queries, source vertices were chosen uniformly at random. For pair queries, we first chose a source vertex uniformly at random, and then chose a target vertex in the same procedure, again uniformly at random.

Experimental Setting. The results were obtained on Debian using an Intel Xeon E5-1650 processor (3.2 GHz, 6 cores, 12 threads) with 128GB of RAM. The parallel results used all 12 threads.

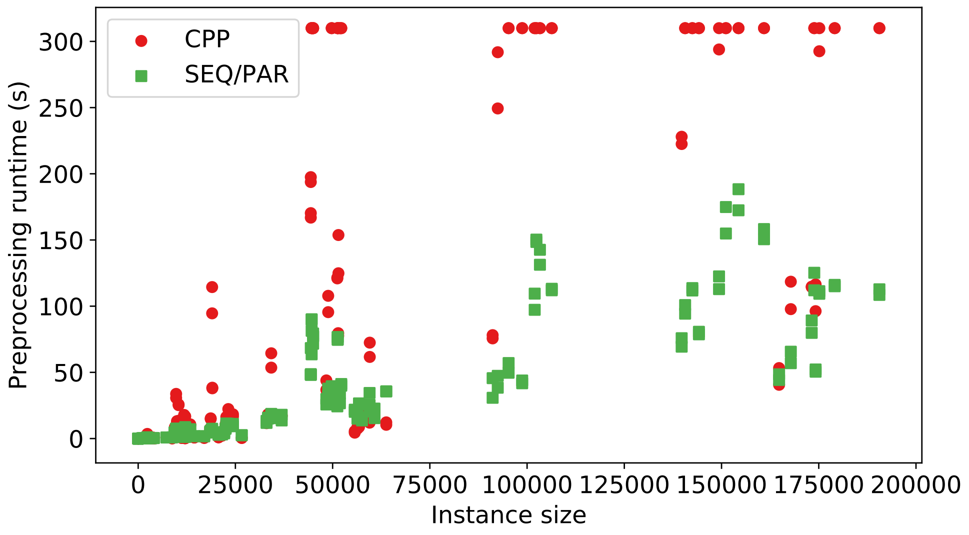


Figure 3.8: Preprocessing times of CPP and SEQ/PAR (over all instances). A dot above the 300s line denotes a timeout.

Time limit. We enforced a preprocessing time limit of 5 minutes per instance. This is in line with the preprocessing times of state-of-the-art tools on benchmarks of this size, e.g. Soot takes 2-3 minutes to generate all flow graphs for each benchmark.

Results. We found that, except for the smallest instances, our algorithm consistently outperforms all previous approaches. Our results were as follows:

- **Treewidth.** The maximum width amongst the obtained tree decompositions was 9, while the minimum was 1. Hence, our experiments confirm the results of [Gustedt et al., 2002] and show that real-world Java programs have small treewidth. See [Chatterjee et al., 2020c] for more details.

- **Preprocessing Time.** As in Figure 3.8, our preprocessing is more lightweight and scalable than CPP. Note that CPP preprocessing times out at 25 of the 110 instances, starting with instances of size $< 50,000$, whereas our approach can comfortably handle instances of size $200,000$. Although the theoretical worst-case complexity of CPP preprocessing is $O(n^2 \cdot |D|^3)$, we observed that its runtime over our benchmarks grows more slowly. We believe this is because our benchmark programs generally consist of a large number of small procedures. Hence, the worst-case behavior of CPP preprocessing, which happens on instances with large procedures, is not captured by the DaCapo benchmarks. In contrast, our preprocessing time is $O(n \cdot |D|^3)$ and having small or large procedures does not matter to our algorithms. Hence, we expect that our approach would outperform CPP preprocessing more significantly on instances containing large functions. However, as Figure 3.8 demonstrates, our approach is faster even on instances with small procedures.

- **Query Time.** As expected, in terms of pair query time, NOPP is the worst performer by a large margin, followed by OD, which is in turn extremely less efficient than CPP, PAR and SEQ (Figure 3.9, top). This illustrates the underlying trade-off between preprocessing and query-time performance. Note that both CPP and our algorithms (SEQ and PAR), answer each pair query in $O(1)$. They all have pair-query times of less than a millisecond and are indistinguishable in this case. The same trade-off appears in single-source queries as well (Figure 3.9, bottom). Again, NOPP is the worst performer, followed by OD. SEQ and CPP have very similar runtimes, except that SEQ outperforms CPP in some cases, due to word tricks. However, PAR is extremely faster, which leads to the next point.

- **Parallelization.** In Figure 3.9 (bottom right), we also observe that single-source queries are handled considerably faster by PAR in comparison with SEQ. Specifically, using 12 threads, the average single-source query time is reduced by a factor of 11.3. Hence, our experimental results achieve near-perfect parallelism and confirm that our algorithm is well-suited for parallel architectures.

Note that Figure 3.9 combines the results of all five mentioned data-flow analyses. However, the observations above hold independently for every single analysis, as well. For analysis-specific figures see our technical report at [Chatterjee et al., 2020c].

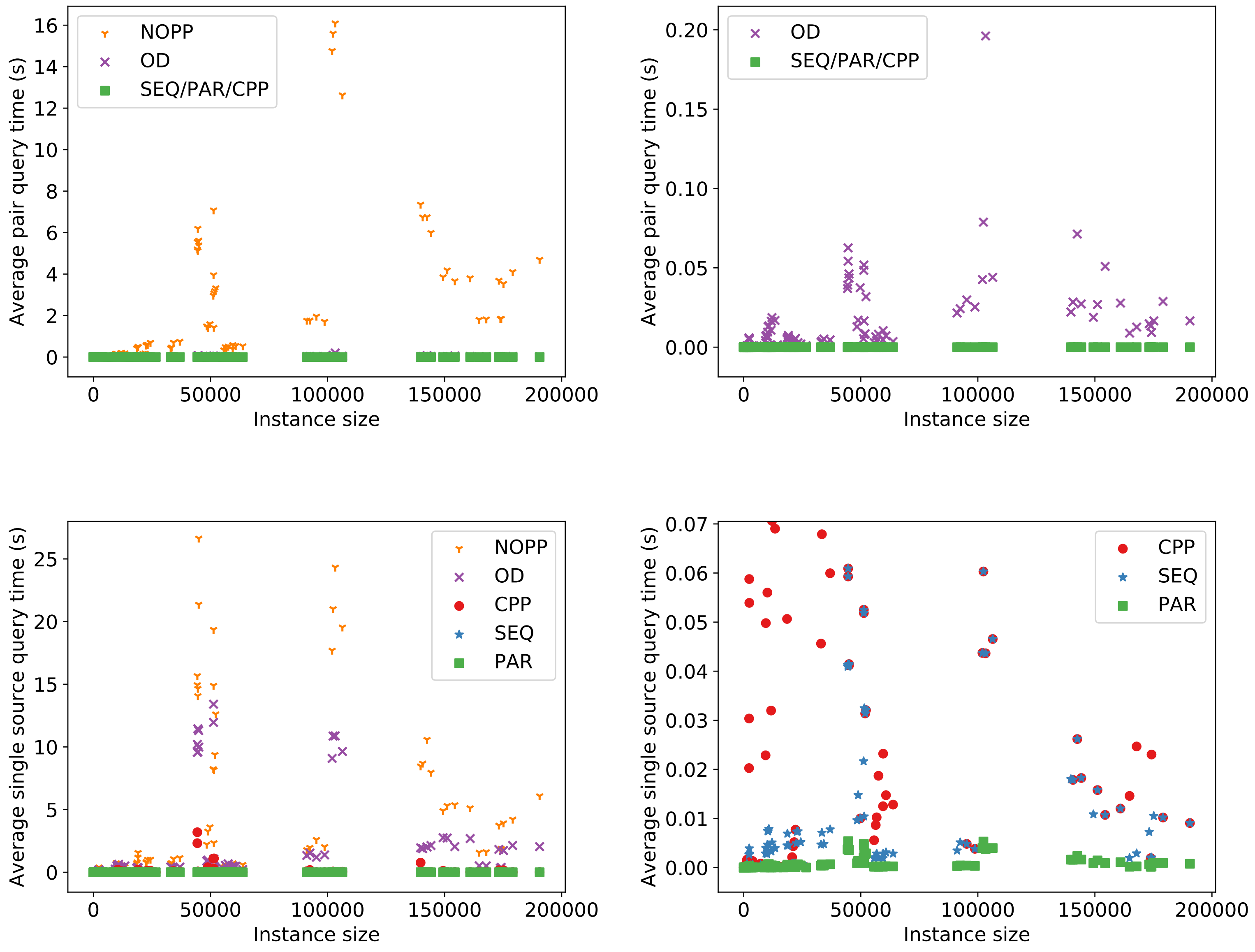


Figure 3.9: Comparison of pair query time (top row) and single source query time (bottom row) of the algorithms. Each dot represents one of the 110 instances. Each row starts with a global picture (left) and zooms into smaller time units (right) to differentiate between the algorithms. The plots above contain results over all five analyses. However, our observations hold independently for every single analysis, as well (See [Chatterjee et al., 2020c]).

4
Faster Algorithms for Quantitative Analysis of MCs and MDPs

This chapter originally appeared in the following publication:

[•] Asadi, A., Chatterjee, K., **Goharshady, A. K.**, Mohammadi, K., and Pavlogiannis, A. **Faster Algorithms for Quantitative Analysis of MCs and MDPs with Small Treewidth**. In *18th International Symposium on Automated Technology for Verification and Analysis* **(ATVA)**, 2020.

4.1 Introduction

Quantitative Analysis of MCs and MDPs. Discrete-time Markov Chains (MCs) and Markov Decision Processes (MDPs) are two standard formalisms in system analysis. Their main associated *quantitative* objectives are hitting probabilities, discounted sum, and mean payoff. Although there are many techniques for computing these objectives in general MCs/MDPs, they have not been thoroughly studied in terms of parameterized algorithms, especially when the parameter is the treewidth of the MC/MDP. This is in contrast to *qualitative* objectives for MCs, MDPs and graph games, for which treewidth-based algorithms yield significant complexity improvements.

Our Results. In this chapter, we consider the problem of computing the quantitative objectives above, parameterized by the treewidth. We obtain linear-time FPT algorithms for MCs. Specifically, for an MC with n vertices and treewidth t, our algorithm computes these quantitative objectives at every vertex in total time $O(n \cdot t^2)$. This is a huge improvement over the general case, where the best algorithms are based on Gaussian elimination and the bound on their runtime is $O(n^\omega)$. Here, ω is the matrix multiplication constant. The best known upper-bound on ω is 2.3728639 [Le Gall, 2014]. Combining our methods with classical Strategy Iteration (SI) leads to faster algorithms for solving small-treewidth MDPs. Specifically, for an MDP with n vertices and treewidth t, we obtain algorithms with a runtime of $O(\kappa \cdot n \cdot t^2)$ where κ is the number of required iterations in SI. Our experimental results (Section 4.5) show that on graphs of small treewidth, our algorithms beat the runtime of state-of-the-art model checkers and optimization suites by one order of magnitude.

MCs. One of the most standard formalisms for modeling randomness in discrete-time systems is that of discrete-time Markov Chains (MCs). MCs have immense applications in verification, and are used to express randomness both in the system and in the environment [Chatterjee et al., 2010]. The modeling power of MCs has also led to various extensions, such as parametric [Daws, 2004, Lanotte et al., 2007, Hahn et al., 2009], interval [Jonsson and Larsen, 1991, Benedikt et al., 2013] and augmented interval [Chonev, 2019] MCs. Besides the theoretical appeal, the analysis of MCs is also a core component in several model checkers [Dehnert et al., 2017, Kwiatkowska et al., 2011].

MDPs. When the system exhibits both stochastic and non-deterministic behavior, the standard model is that of Markov Decision Processes (MDPs). For example, MDPs are used to model stochastic controllers, where non-determinism models the freedom of the controller and randomness models the behavior of the system. MDPs are also a topic of active study in verification [Tappler et al., 2019, Chatterjee et al., 2018e].

Quantitative Analysis. Three of the most standard analysis objectives for MCs are the following: (a) The *hitting probabilities* objective takes as input a set of target vertices $\mathfrak{T}$ of the MC, and asks to compute for each vertex u, the probability that a random walk from u eventually hits $\mathfrak{T}$. The *discounted sum* objective takes as input a discount factor $\lambda \in (0,1)$ and a reward function R that assigns a reward to each edge of the MC. The task is to compute for each vertex u the expected reward value of a random walk starting from u, where the value of the walk is the sum of the rewards along its edges, discounted by the factor λ at each step. Finally, the *mean payoff* objective is similar to discounted sum, except that the value of a walk is the long-run average of the rewards along its edges. In MDPs, the analyses ask for a strategy that maximizes the respective quantity.

Previous Methods. Given the importance of quantitative objectives for MCs and MDPs, there have been various techniques for solving them efficiently. For MCs, the hitting probabilities and discounted sum objectives reduce to solving a system of linear equations [Norris, 1998]. For MDPs, all three objectives reduce to solving a linear program [Norris, 1998]. Besides the LP formulation, two popular approaches for solving quantitative objectives on MDPs are value iteration [Bellman, 1957] and strategy iteration [Howard, 1960]. Value iteration is the most commonly used method in verification and operates by computing optimal policies for successive finite horizons. However, this process leads only to approximations of the optimal values, and for some objectives no stopping criterion for the optimal strategy is known [Ashok et al., 2017]. In cases where such criteria are known (e.g. [Quatmann and Katoen, 2018]), the number of iterations necessary before the numbers can be rounded to provide an optimal solution can be extremely high [Chatterjee and Henzinger, 2008]. Nevertheless, value iteration has proved to be very successful in practice and is included in many probabilistic model checkers, such as [Kwiatkowska et al., 2011, Dehnert et al., 2017].

On the other hand, strategy iteration lies on the observation that given a fixed strategy, the MDP reduces to an MC, and hence one can compute the value of each vertex using existing techniques on MCs. Then, the strategy can be refined to a new strategy that improves the value of each vertex. The running time of strategy iteration can be written as $O(\kappa \cdot f)$, where κ is the number of strategy refinements and f is the time for evaluating the strategy. Although κ can be exponentially large [Fearnley, 2010], it behaves as a small constant in practice, which makes strategy iteration work well in practice [Křetínský and Meggendorfer, 2017]. Hence, both for MCs and for MDPs using strategy iteration, the performance of the algorithm largely depends on the speed of solving the respective linear system [Křetínský and Meggendorfer, 2017].

Related Works. There are many works that exploit treewidth in non-probabilistic settings, e.g., in graphs and matrices [Fomin et al., 2018, Ferrara et al., 2005]; and for qualitative analysis in probabilistic setting [Chatterjee and Łącki, 2013]. In contrast, in this work we exploit treewidth for *quantitative* analysis of probabilistic systems. The closest previous work is [Chatterjee and Łącki, 2013]. It considers the maximal end-component decomposition and the almost-sure reachability set computation in low-treewidth MDPs. Note that these are both *qualitative* objectives, and thus very different from the *quantitative* objectives we consider here, which cannot be solved by [Chatterjee and Łącki, 2013]. Specifically, the main problem solved by [Chatterjee and Łącki, 2013] is almost-sure reachability, i.e. reachability with probability 1, which is a very special qualitative case of computing hitting probabilities.

4.2 Preliminary Definitions and Notation

We now provide basic definitions and fix our notation for the rest of this chapter.

Discrete Probability Distributions. Given a finite set X, a probability distribution over X is a function $d : X \to [0, 1]$ such that $\sum_{x \in X} d(x) = 1$. We denote the set of all probability distributions over X by $\mathcal{D}(X)$.

Primal Graphs. Let S be a system of linear equations with m equations and n unknowns (variables). The primal graph $G(S)$ of S is an undirected graph with n vertices, each corresponding to one unknown in S, in which there is an edge between two unknowns x and y iff there exists an equation in S that contains both x and y with non-zero coefficients.

Markov Chains (MCs). A *Markov chain* $C = (V, E, \delta)$ consists of a finite directed graph (V, E) and a probabilistic transition function $\delta : V \to \mathcal{D}(V)$, such that for any pair u, v of vertices, we have $\delta(u)(v) > 0$ if and only if $(u, v) \in E$. In an MC C, we start a random walk from a vertex $v_0 \in V$ and at each step, being in a vertex v, we probabilistically choose one of the successors of v and go there. The probability with which a successor w is chosen is given by $\delta(v)(w)$. Let $O \subseteq V^\omega$ be a measurable set of infinite paths on V, we use the notation $Pr_{v_0}(O)$ to denote the probability that our infinite random walk starting from v_0 is a member of O. See [Gagniuc, 2017, Kemeny et al., 2012] for more detailed treatment.

Markov Decision Processes (MDPs). A *Markov decision process* $P = (V, E, V_1, V_P, \delta)$ consists of a finite directed graph (V, E), a partitioning of V into two sets V_1 and V_P, and a probabilistic transition function $\delta : V_P \to \mathcal{D}(V)$, such that for any $(u, v) \in V_P \times V$, we have $\delta(u)(v) > 0$ if and only if $(u, v) \in E$. We assume that all vertices of an MDP have at least one outgoing edge. Intuitively, an MDP is a one-player game in which we have two types of vertices: those controlled by Player 1, i.e. V_1, and those that behave probabilistically, i.e. V_P.

Strategies. In an MDP P, a *strategy* is a function $\sigma : V_1 \to V$, such that for every $v \in V_1$ we have $(v, \sigma(v)) \in E$. Informally, a strategy is a recipe for Player 1 that tells her which successor to choose based on the current state*. Given an MDP P with a strategy σ, we start a random walk from a vertex $v_0 \in V$ and at each step, being in a vertex v, choose the successor as follows: (i) if $v \in V_1$, then we go to $\sigma(v)$, and (ii) if $v \in V_P$ we act as in the case of MCs, i.e. we go to each successor w with probability $\delta(v)(w)$. As before, given a measurable set $O \subseteq V^\omega$ of infinite paths on V, we define $Pr^\sigma_{v_0}(O)$ as the probability that our infinite random walk becomes a member of O. Note that an MDP with a fixed strategy σ is basically an MC, in which for every $v \in V_1$ we have $\delta(v)(\sigma(v)) = 1$. See [Filar and Vrieze, 1996, Howard, 1960] for more detailed treatment.

*We only consider pure memoryless strategies because they are sufficient for our use-cases, i.e. there always exists an optimal strategy that is pure and memoryless [Křetínský and Meggendorfer, 2017].

4.3 Quantitative Problems

4.3.1 Definition of the Problems

We consider three classical quantitative problems: Hitting Probabilities, Discounted Sums of Rewards, and Mean Payoff. We now formalize these problems over MCs and MDPs.

Hitting Probabilities [Norris, 1998]**.** Let $C = (V, E, \delta)$ be an MC and $\mathfrak{T} \subseteq V$ a designated set of *target* vertices. We define $\mathit{Hit}(\mathfrak{T}) \subseteq V^\omega$ as the set of all infinite sequences of vertices that intersect $\mathfrak{T}$. The *Hitting probability* $\mathit{HitPr}(u, \mathfrak{T})$ is defined as $\mathit{Pr}_u(\mathit{Hit}(\mathfrak{T}))$. In other words, $\mathit{HitPr}(u, \mathfrak{T})$ is the probability of eventually reaching $\mathfrak{T}$, assuming that we start our random walk at u. In case of MDPs, we assume that the player aims to maximize the hitting probability by choosing the best possible strategy. Therefore, we define $\mathit{HitPr}(u, \mathfrak{T})$ as $\max_\sigma \mathit{Pr}^\sigma_u(\mathit{Hit}(\mathfrak{T}))$.

Discounted Sums of Rewards [Puterman, 2014]**.** Let $C = (V, E, \delta)$ be an MC and $R : E \to \mathbb{R}$ a *reward function* that assigns a real value to each edge. Also, let $\lambda \in (0, 1)$ be a *discount factor*. Given an infinite path $\pi = v_0, v_1, \ldots$ over (V, E), we define the total reward $R(\pi)$ of π as follows:

$$R(\pi) = \sum_{i=0}^{\infty} \lambda^i \cdot R(v_i, v_{i+1}) = R(v_0, v_1) + \lambda \cdot R(v_1, v_2) + \lambda^2 \cdot R(v_2, v_3) + \ldots.$$

Let $u \in V$ be a vertex, we define $\mathit{ExpDisSum}(u)$ as the expected value of the reward of our random walk if we begin it at u, i.e. $\mathit{ExpDisSum}(u) := \mathbb{E}_u[R(\pi)]$. As in the previous case, when considering MDPs, we assume that the player aims to maximize the discounted sum, hence given an MDP $P = (V, E, V_1, V_P, \delta)$, a reward function R and a discount factor λ, we define $\mathit{ExpDisSum}(u) := \max_\sigma \mathbb{E}^\sigma_u[R(\pi)]$.

Mean Payoff [Puterman, 2014, Křetínský and Meggendorfer, 2017]**.** Let C be an MC and R a reward function. Given an infinite path $\pi = v_0, v_1, \ldots$ over C, we define the n-step average reward of π as follows:

$$R(\pi[0..n]) := \frac{1}{n} \sum_{i=1}^{n} R(v_{i-1}, v_i).$$

Given a start vertex $u \in V$, the expected *long-time average* or *mean payoff* value from u is defined as $\mathit{ExpMP}(u) := \lim_{n\to\infty} \mathbb{E}_u[R(\pi[0..n])]$. In other words, $\mathit{ExpMP}(u)$ captures the expected reward per step in a random walk starting at u. For an MDP P, we define $\mathit{ExpMP}(u) := \max_\sigma \lim_{n\to\infty} \mathbb{E}_u^\sigma[R(\pi[0..n])]$. The limits in the former definitions are guaranteed to exist [Puterman, 2014, Křetínský and Meggendorfer, 2017].

Quantitative Analysis Problems. We consider the following classical problems for both MCs and MDPs:

- *Computing Hitting Probabilities:* Given a target set $\mathfrak{T}$ compute $\mathit{HitPr}(u, \mathfrak{T})$ for every vertex u.
- *Computing Expected Discounted Sums:* Given a reward function R and a discount factor $\lambda \in (0, 1)$, compute $\mathit{ExpDisSum}(u)$ for every vertex u.
- *Computing Mean Payoffs:* Given a reward function R, compute $\mathit{ExpMP}(u)$ for every vertex u.

4.3.2 Classical Algorithms

We now review some of the most well-known algorithms for handling the quantitative analysis problems above over MCs and MDPs.

Solving MCs [Norris, 1998]**.** A classical approach to the above problems for MCs is to reduce them to solving systems of linear equations. In case of hitting probabilities, we define one variable x_u for each vertex u, whose value in the solution to the system would be equal to $\mathit{HitPr}(u, \mathfrak{T})$. The system is constructed as follows:

- We add the equation $x_\mathsf{t} = 1$ for every $\mathsf{t} \in \mathfrak{T}$, and

- For every vertex $u \notin \mathfrak{T}$ with successors $u_1, \ldots, u_k$, we add the following equation:

$$x_u = \sum_{i=1}^{k} \delta(u)(u_i) \cdot x_{u_i}.$$

If every vertex can reach a target, then it is well-known that the resulting system has a unique solution in which the value assigned to each x_u is equal to $\mathit{HitPr}(u, \mathfrak{T})$. A similar approach can be used in the case of discounted sums. We define one variable y_u per vertex u and if the successors of u are $u_1, \ldots, u_k$, then we add the following equation:

$$y_u = \sum_{i=1}^{k} \delta(u)(u_i) \cdot (R(u, u_i) + \lambda \cdot y_{u_i}).$$

There are several different ways of reducing the mean payoff problem to solving systems of linear equations. One technique is presented in Section 4.4.5 below. See [Puterman, 2014] for more details.

Solving MDPs. There are two classical approaches to solving the above problems for MDPs. One is to reduce the problem to Linear Programming (LP) in a manner similar to the reduction from MC to linear systems [Feinberg, 2012]. The other approach is to use dynamic programming [Bellman, 1957]. We consider a widely-used variety of dynamic programming, called *strategy iteration* or *policy iteration* [Howard, 1960].

Strategy Iteration (SI) [Bellman, 1957]**.** In SI, we start with an arbitrary initial strategy σ_0 and attempt to find a better strategy in each step. Formally, assume that our strategy after i iterations is σ_i. Then, we compute $\mathit{val}_i(u) = \mathit{HitPr}^{\sigma_i}(u, \mathfrak{T})$ for every vertex u. This is equivalent to computing hitting probabilities in the MC that is obtained by considering our MDP together with the strategy σ_i. We use the values $\mathit{val}_i(u)$ to obtain a better strategy σ_{i+1} as follows: for every vertex $v \in V_1$ with successors $v_1, v_2, \ldots, v_k$, we set $\sigma_{i+1}(v) = \arg\max_{v_j} \mathit{val}_i(v_j)$. In case of discounted sum, we let $\mathit{val}_i(u) = \mathit{ExpDisSum}^{\sigma_i}(u)$ and $\sigma_{i+1}(v) = \arg\max_{v_j} R(v, v_j) + \lambda \cdot \mathit{val}_i(v_j)$. We repeat these steps until we reach a point where our strategy converges.

It is well-known that strategy iteration always converges to the optimal strategy, and at that point the values val_i will be the desired hitting probabilities/discounted sums [Howard, 1960, Feinberg, 2012]. Given that SI solves the classic problems above on MDPs by several calls to a procedure for solving the same problems on MCs, our runtime improvements for MCs are naturally extended to MDPs. So, in the sequel we only focus on MCs.

4.4 Treewidth-based Quantitative Analysis Algorithms

We now consider quantitative problems on MCs. As mentioned before, our improvements carry over to MDPs using SI. We build on classical state-elimination algorithms such as those used in [Daws, 2004, Hahn et al., 2010]. The main novelty of our approach is that we use the tree decompositions to obtain a suitable *order* for eliminating vertices. This specific ordering significantly reduces the runtime complexity of classical state-elimination algorithms from cubic to linear. Aside from the ordering, which is the main basis for our algorithmic improvements, the rest of this section is mostly well-known transformations on MCs. However, a new subtlety arises: while in general MCs there are several variants of elimination rules, in small-treewidth MCs we must also make sure the elimination step does not increase the treewidth or invalidate the underlying tree decomposition.

We first review state-elimination for computing hitting probabilities (Section 4.4.1). Then, in Section 4.4.2, we show how to exploit the treewidth to speed up this process and obtain a linear-time algorithm. Section 4.4.3 provides a similar speed-up for computing expected discounted sums. In Section 4.4.4, we show our most general result, i.e. solving small-treewidth systems of linear equations in linear time. While this algorithm is more general than those of Sections 4.4.2 and 4.4.3, it repeatedly applies the costly Gram-Schmidt orthogonalization process, and is hence not preferable in practice. Finally, Section 4.4.5 combines these ideas to compute expected mean payoffs in linear time.

4.4.1 State Elimination for Computing Hitting Probabilities

We begin by looking into the problem of computing hitting probabilities for general MCs without exploiting the treewidth.

Singleton Target Sets. Without loss of generality, we can assume that our target set contains a single vertex. Otherwise, we add a new vertex $\mathfrak{t}$ and add edges with probability 1 from every target vertex to $\mathfrak{t}$. This will keep the hitting probabilities intact.

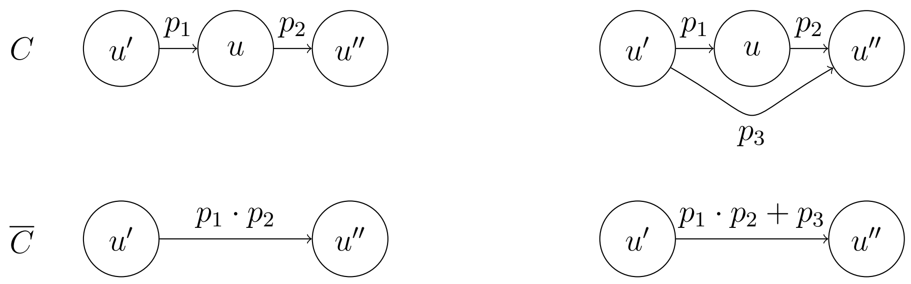


Figure 4.1: Removing a vertex u when computing hitting probabilities in MCs.

Vertex Elimination. Consider our MC $C = (V, E, \delta)$ and our target vertex $\mathfrak{t} \in V$. If there is only one vertex in the MC then there is not much to solve. We just return that $\mathit{HitPr}(\mathfrak{t}, \mathfrak{t}) = 1$. Otherwise, we take an arbitrary vertex $u \neq \mathfrak{t}$ and try to remove it from the MC to obtain a smaller MC that can in turn be solved using the same method. We should do this in a manner that does not change $\mathit{HitPr}(v, \mathfrak{t})$ for any vertex $v \neq u$. Figure 4.1 shows how to remove a vertex u from C in order to obtain a smaller MC $\overline{C} = (V \setminus \{u\}, \overline{E}, \overline{\delta})$[†]. In this figure, the vertex u' is a predecessor of u and u'' is one of its successors. The left side shows the changes when there is no edge from u' to u'' and the right side shows the other case, where $(u', u'') \in E$. Edge labels are δ values. Basically, we remove u and all of its edges, and instead add new edges from every predecessor u' to every successor u''. We also update the transition function δ by setting $\overline{\delta}(u')(u'') = \delta(u')(u'') + \delta(u')(u) \cdot \delta(u)(u'')$.

[†]We always use $\overline{C}$ to denote an MC that is obtained from C by removing one vertex. We apply this rule across our notation, e.g. $\overline{\delta}$ is the respective transition function.

It is easy to verify that for every $v \neq u$, we have $\overline{HitPr}(v, \mathfrak{t}) = HitPr(v, \mathfrak{t})$. Hence, we can compute hitting probabilities for every vertex $v \neq u$ in $\overline{C}$ instead of C. Finally, if $u_1, u_2, \ldots, u_k$ are the successors of u in C, we know that $HitPr(u, \mathfrak{t}) = \sum_{i=1}^{k} \delta(u)(u_i) \cdot HitPr(u_i, \mathfrak{t}) = \sum_{i=1}^{k} \delta(u)(u_i) \cdot \overline{HitPr}(u_i, \mathfrak{t})$. Hence, we can easily compute the hitting probability for u using this formula. A pseudocode of this approach is given in Algorithm 4.1.

Special Cases. A special case arises when there is a self-loop transition from u to u. If $\delta(u)(u) = 1$, i.e. u is an absorbing trap, then we can simply remove u, noting that $HitPr(u, \mathfrak{t}) = 0$. On the other hand if $0 < \delta(u)(u) < 1$, then we should distribute $\delta(u)(u)$ proportionately among the other successors of u because staying for a finite number of steps in the same vertex u does not change the hitting property of a path, and the probability of staying at u forever is 0.

Complexity Analysis. Removing each vertex can take at most $O(n^2)$ time, given that it has $O(n)$ predecessors and successors. We should remove $n - 1$ vertices, leading to a total runtime of $O(n^3)$, which is worse than the reduction to system of linear equations and then applying Gaussian elimination, leading to a runtime of $O(n^\omega)$. However, the runtime can be significantly improved if we remove vertices in an order that guarantees every vertex has a low degree upon removal. In the next section we show how to exploit a tree decomposition to obtain such an order.

4.4.2 Hitting Probabilities Parameterized by Treewidth

The main idea behind our algorithm is simple: we take the algorithm from the previous section and use tree decompositions to obtain an ordering for the removal of vertices. Given that we can choose any bag in T as the root, without loss of generality, we assume that the target vertex $\mathfrak{t}$ is in the root bag[‡]. We base our approach on the following lemmas:

Lemma 4.1. *Let $l \in V_T$ be a* leaf *node of the tree decomposition $(T, \langle B_i \rangle)$ of our MC C, and let $\bar{l}$ be the parent of l. If $B_l \subseteq B_{\bar{l}}$, then $(V_T \setminus \{l\}, E_T \setminus \{(l, \bar{l})\})$, together with the same bags $\langle B_i \rangle_{i \in V_T \setminus \{l\}}$, also form a valid tree decomposition for C.*

[‡]If $|\mathfrak{T}| \geq 2$, we use the same technique as in the previous section to have only one target $\mathfrak{t}$. To keep the tree decomposition valid, we add $\mathfrak{t}$ to every bag.

Algorithm 4.1: A Simple State Elimination Algorithm for Computing Hitting Probabilities in MCs.

```
Function ComputeHitProbs($C = (V, E, \delta), \mathsf{t}$):
    if $V = \{\mathsf{t}\}$ then
        $HitPr(\mathsf{t}, \mathsf{t}) \leftarrow 1$
    else
        Choose an arbitrary $u \in V \setminus \{\mathsf{t}\}$
        if $\delta(u)(u) = 1$ then
            $HitPr(u, \mathsf{t}) \leftarrow 0$
            ComputeHitProbs $((V \setminus \{u\}, E, \delta), \mathsf{t})$
        else
            $f \leftarrow \frac{1}{1-\delta(u)(u)}$
            $\delta(u)(u) \leftarrow 0$
            $E \leftarrow E \setminus \{(u, u)\}$
            foreach $u'' \in V : (u, u'') \in E$ do
                $\delta(u)(u'') \leftarrow \delta(u)(u'') \cdot f$
            foreach $u' \in V : (u', u) \in E$ do
                foreach $u'' \in V : (u, u'') \in E$ do
                    $\delta(u')(u'') \leftarrow \delta(u')(u'') + \delta(u')(u) \cdot \delta(u, u'')$
                    $E \leftarrow E \cup \{(u', u'')\}$
            ComputeHitProbs $((V \setminus \{u\}, E, \delta), \mathsf{t})$
            $HitPr(u, \mathsf{t}) \leftarrow 0$
            foreach $u'' \in V : (u, u'') \in E$ do
                $HitPr(u, \mathsf{t}) \leftarrow HitPr(u, \mathsf{t}) + \delta(u, u'') \cdot HitPr(u'', \mathsf{t})$
```

Proof. We just need to check that all the required properties of a tree decomposition hold after removal of l. Given that $B_l \subseteq B_{\bar{l}}$, any vertex that appears in B_l is also in $B_{\bar{l}}$ and hence removal of l does not cause any vertex to be unrepresented in the tree decomposition. The same applies to edges. Moreover, removing a *leaf* node cannot disconnect the previously-connected set of nodes whose bag contained a specific vertex. □

Lemma 4.2. *Let $l \in V_T$ be a node of the tree decomposition $(T, \langle B_i \rangle)$ and assume that the vertex $u \in V$ only appears in B_l, i.e. it does not appear in any other bag. Then, the vertex u has at most $|B_l|$ predecessors/successors in C.*

Proof. If u' is a predecessor/successor of u, then there is an edge between them. By definition, a tree decomposition should cover every edge. Hence, there should be a bag B_i such that $u, u' \in B_i$. By assumption, u only appears in B_l. Hence, every predecessor/successor u' must also appear in B_l. □

The Algorithm. The above lemmas provide a convenient order for removing vertices. At each step, we choose an arbitrary *leaf* node l. If there is a vertex u that appears *only* in B_l, then we eliminate u as in Figure 4.1. In this case, Lemma 4.2 guarantees that u has $O(t)$ predecessors and successors. Otherwise, $B_l \subseteq B_{\bar{l}}$ (recall that each vertex appears in a connected subtree) and we can remove l from our tree decomposition according to Lemma 4.1. A pseudocode of this approach is given in Algorithm 4.2.

Example 4.1. *Consider the graph and tree decomposition in Figure 4.2 (top right) with an arbitrary transition probability function δ and target vertex $\mathfrak{t} = 6$. The target vertex is shown in green. At each step the vertex/bag that is being removed is shown in red. An active bag whose vertices, but not itself, are considered for removal is shown in blue. On this example, our algorithm would first choose an arbitrary leaf bag, say $\{7, 9\}$ and then realize that 9 has only appeared in this bag. Hence it removes vertex 9 from the MC using the same procedure as in the previous section. In the next iteration, it chooses the bag $\{7\}$ and realizes that the set of vertices in this bag is a subset of vertices that appear in its parent. Hence, it removes this unnecessary bag. The algorithm continues similarly, until only the target vertex 6 remains, at which point the problem is trivial. Figure 4.2 shows all the steps of our algorithm. Note*

Algorithm 4.2: Computing Hitting Probabilities in an MC using a Tree Decomposition.

```
Function ComputeHitProbs($C = (V, E, \delta), \mathtt{t}, (V_T, E_T), \langle B_i \rangle$):
    if $V = \{\mathtt{t}\}$ then
        $HitPr(\mathtt{t}, \mathtt{t}) \leftarrow 1$
    else
        repeat
            Choose an arbitrary leaf node $l \in V_T$
            $\bar{l} \leftarrow$ parent of $l$
            if $B_l \subseteq B_{\bar{l}}$ then
                $V_T \leftarrow V_T \setminus \{l\}$
                $E_T \leftarrow E_T \setminus \{(l, \bar{l})\}$
            else
                Choose an arbitrary $u \in B_l \setminus B_{\bar{l}}$
                $B_l \leftarrow B_l \setminus \{u\}$
                break
        if $\delta(u)(u) = 1$ then
            $HitPr(u, \mathtt{t}) \leftarrow 0$
            ComputeHitProbs $((V \setminus \{u\}, E, \delta), \mathtt{t}, (V_T, E_T), \langle B_i \rangle)$
        else
            $f \leftarrow \frac{1}{1-\delta(u)(u)}$
            $\delta(u)(u) \leftarrow 0$
            $E \leftarrow E \setminus \{(u, u)\}$
            foreach $u'' \in B_l : (u, u'') \in E$ do
                $\delta(u)(u'') \leftarrow \delta(u)(u'') \cdot f$
            foreach $u' \in B_l : (u', u) \in E$ do
                foreach $u'' \in B_l : (u, u'') \in E$ do
                    $\delta(u')(u'') \leftarrow \delta(u')(u'') + \delta(u')(u) \cdot \delta(u, u'')$
                    $E \leftarrow E \cup \{(u', u'')\}$
            ComputeHitProbs $((V \setminus \{u\}, E, \delta), \mathtt{t}, (V_T, E_T), \langle B_i \rangle)$
            $HitPr(u, \mathtt{t}) \leftarrow 0$
            foreach $u'' \in B_l : (u, u'') \in E$ do
                $HitPr(u, \mathtt{t}) \leftarrow HitPr(u, \mathtt{t}) + \delta(u, u'') \cdot HitPr(u'', \mathtt{t})$
```

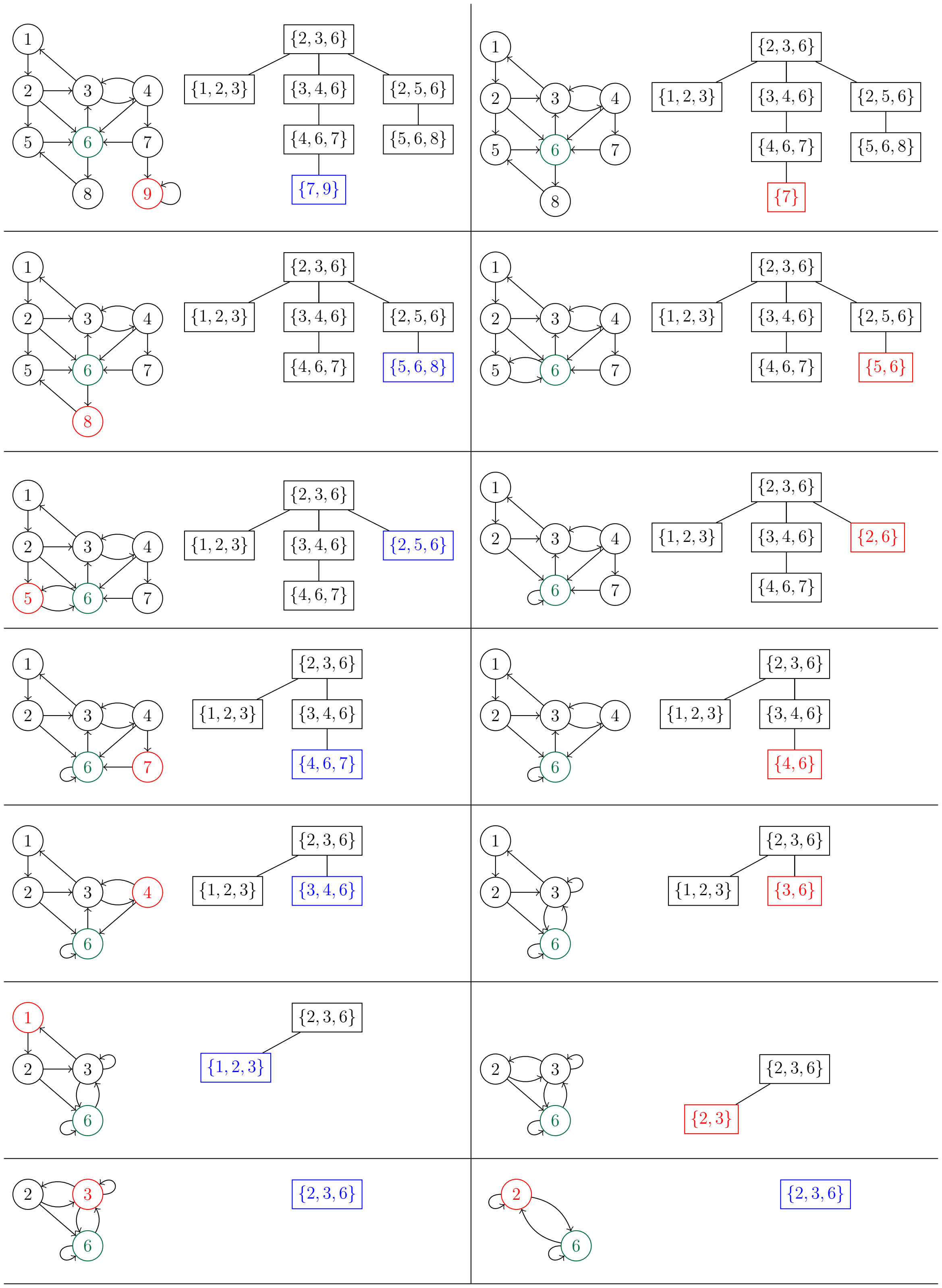


Figure 4.2: An Example Run of our Algorithm for Computing Hitting Probabilities using a Tree Decomposition.

that because the width of our tree decomposition is 2*, at each step when we are removing a vertex* u*, it has at most* 3 *neighbors (counting itself).*

Validity of the Tree Decomposition. Note that throughout this algorithm the tree decomposition remains valid, because we are only adding edges between vertices that are already in the same leaf bag B_l. Given that we remove at most $O(n)$ bags and $n-1$ vertices and that removing each vertex takes only $O(t^2)$, the total runtime is $O(n \cdot t^2)$. Hence, we have the following theorem:

Theorem 4.1. *Given an MC with* n *vertices and treewidth* t *and an optimal tree decomposition of the MC with* $O(n)$ *bags, Algorithm 4.2 computes hitting probabilities from every vertex to a designated target set in* $O(n \cdot t^2)$*.*

4.4.3 Expected Discounted Sums Parameterized by Treewidth

We use a similar approach for handling the discounted sum problem. The only difference is in how a vertex is removed.

The $\hat{\mathbf{1}}$ Gadget. Given an MC $C = (V, E, \delta)$, a tree decomposition $(T, \langle B_i \rangle)$ of C, a reward function $R : E \to \mathbb{R}$ and a discount factor $\lambda \in (0, 1)$, we first add a new vertex called $\hat{\mathbf{1}}$ to the MC. The vertex $\hat{\mathbf{1}}$ is disjoint from all other vertices and only has a single self-loop with probability 1 and reward $1 - \lambda$. In other words, we define $\delta(\hat{\mathbf{1}})(\hat{\mathbf{1}}) = 1$ and $R(\hat{\mathbf{1}}, \hat{\mathbf{1}}) = 1 - \lambda$. We also add $\hat{\mathbf{1}}$ to the vertex set of every bag. The reason behind this gadget is that we have $\mathit{ExpDisSum}(\hat{\mathbf{1}}) = (1 - \lambda) \cdot (1 + \lambda + \lambda^2 + \ldots) = 1$.

Generalizing the Probabilities. In our algorithm, the requirement that for all u, v we should have $0 \leq \delta(u)(v) \leq 1$ is unnecessary and becomes untenable, too. Therefore, we allow $\delta(u)(v)$ to have any real value, and use the linear system interpretation of C as in Section 4.3.2, i.e. instead of considering C as an MC, we consider it to be a representation of the linear system S_C defined as follows:

- For every vertex $u \in V$, the system S_C contains one unknown y_u, and

- For every vertex $u \in V$, whose successors are $u_1, u_2, \dots, u_k$, the system S_C contains an equation $\mathfrak{e}_u := y_u = \sum_{i=1}^{k} \delta(u)(u_i) \cdot (R(u, u_i) + \lambda \cdot y_{u_i})$.

As mentioned in Section 4.3.2, in the solution to S_C, the value assigned to the unknown y_u is equal to *ExpDisSum*(u) in the MC C. However, the definition above does not depend on the fact that C is an MC and can also be applied if δ has arbitrary real values.

Vertex Elimination. Now suppose that we want to remove a vertex $u \neq \hat{\mathbf{1}}$ with successors $u_1, \dots, u_k$ from C. This is equivalent to removing y_u from S_C without changing the values of other unknowns in the solution. Given that we have $y_u = \sum_{i=1}^{k} \delta(u)(u_i) \cdot (R(u, u_i) + \lambda \cdot y_{u_i})$, we can simply replace every occurrence of y_u in other equations with the right-hand-side expression of this equation. If $u' \neq u$ is a predecessor of u, then we have $y_{u'} = A + \delta(u')(u) \cdot (R(u', u) + \lambda \cdot y_u)$, where A is an expression that depends on other successors of u'. We can rewrite this equation as $y_{u'} = A + \delta(u')(u) \cdot R(u', u) + \sum_{i=1}^{k} \delta(u')(u) \cdot \delta(u)(u_i) \cdot \lambda \cdot (R(u, u_i) + \lambda \cdot y_{u_i})$. This is equivalent to obtaining a new $\overline{C}$ from C by removing the vertex u and adding the following edges from every predecessor u' of u:

- An edge $(u', \hat{\mathbf{1}})$, such that $R(u', \hat{\mathbf{1}}) = 0$ and $\delta(u')(1) = \frac{1}{\lambda} \cdot (\delta(u')(u) \cdot R(u', u))$,
- An edge (u', u_i) to every successor u_i of u, such that $R(u', u_i) = R(u, u_i)$ and $\delta(u')(u_i) = \delta(u')(u) \cdot \delta(u)(u_i) \cdot \lambda$.

This construction is shown in Figure 4.3, where each edge is labelled with its δ and R values. As shown above, using this construction the value of y_v remains the same in solutions of S_C and $S_{\overline{C}}$.

Special Cases. There are two special cases that can cause our construction to fail. However, we can avoid both of these cases using simple transformations in the graph before applying this construction. We now describe how we handle each of them:

- *Parallel Edges.* If two edges with the same direction are created between the same pair (u, v) of vertices, then we replace them with a single edge. If the δ values of initial edges were δ_1, δ_2 and their R values were r_1, r_2, we set $\delta(u)(v) = \delta_1 + \delta_2$ and

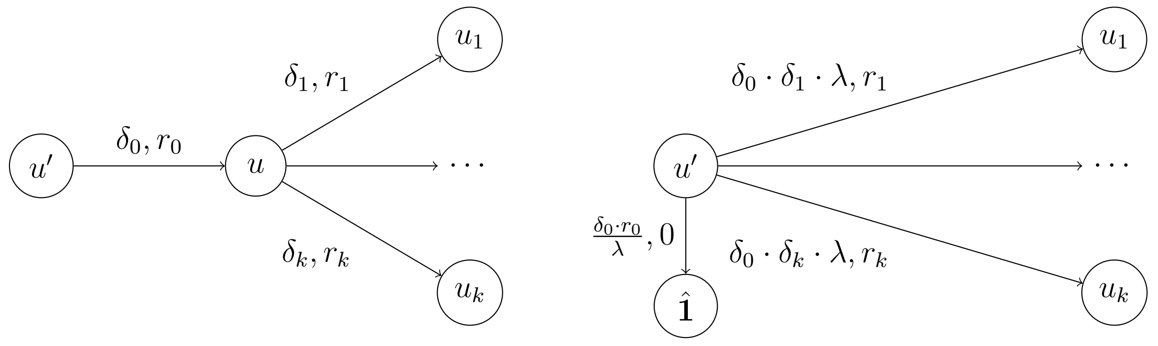

u'
δ_0, r_0
u
δ_1, r_1
u_1
$\dots$
δ_k, r_k
u_k
u'
$\delta_0 \cdot \delta_1 \cdot \lambda, r_1$
u_1
$\dots$
$\frac{\delta_0 \cdot r_0}{\lambda}, 0$
$\hat{\mathbf{1}}$
$\delta_0 \cdot \delta_k \cdot \lambda, r_k$
u_k

Figure 4.3: Removing a vertex u when computing discounted sums in MCs.

$R(u, v) = \frac{\delta_1 \cdot r_1 + \delta_2 \cdot r_2}{\delta_1 + \delta_2}$. It is straigthforward to verify that this transformation is sound, i.e. it does not change the solution of the corresponding system.

- *Self-loops.* If a self-loop (u, u) appears in our graph, this is equivalent to having an equation $\mathfrak{e}_u := y_u = R$ in the linear system, in which R is a linear expression that contains a non-zero multiple of y_u. In this case, we simplify this equation to $y_u = R'$ by moving the summand containing y_u to the left hand side and multiplying both sides by a suitable factor. We then update the outgoing edges of u in our graph to model the new system. Note that this update does not add any new edges to the graph, except possibly the edge $(u, \hat{\mathbf{1}})$ for handling leftover constant factors.

Correctness and Complexity. As in the previous section, we can solve the problem on the smaller $\overline{C}$ and then use the equation $\mathfrak{e}_u$ to compute the value of y_u in the solution to S_C. This algorithm's runtime can be analyzed exactly as before. We have to remove n vertices and each removal takes $O(n^2)$ for a total runtime of $O(n^3)$. To obtain a better algorithm that exploits tree decompositions, we can use the exact same removal order as in the previous section, leading to the same runtime, i.e. $O(n \cdot t^2)$. Note that we have added $\hat{\mathbf{1}}$ to the associated vertex set of every bag, so the tree decomposition always remains valid throughout our algorithm. Given this discussion, we have the following theorem:

Theorem 4.2. *Given an MC with n vertices and treewidth t and an optimal tree decomposition of the MC with $O(n)$ bags, the algorithm described in this section computes expected discounted sums from every vertex of the MC in $O(n \cdot t^2)$.*

4.4.4 Solving Systems of Equations Parameterized by the Treewidth of their Primal Graphs

The ideas used in the previous section can be extended to obtain faster algorithms for solving any linear system whose primal graph has a small treewidth. However, new subtleties arise, given that general linear systems might have no solution or infinitely many solutions. In contrast, the systems S_C discussed in the previous section were guaranteed to have a unique solution.

Setting. We consider a system S of m linear equations over n real unknowns as input, and assume that its primal graph $G(S)$ has treewidth t.

Variable Elimination. Our algorithm for solving S is similar to our previous algorithms, and is actually what most students are taught in junior high school. We take an arbitrary unknown x and choose an arbitrary equation $\mathfrak{e}$ in which x appears with a non-zero coefficient. We then rewrite $\mathfrak{e}$ as $x = R_x$, where R_x is a linear expression based on other unknowns. Finally, we replace every occurrence of x in other equations with R_x and solve the resulting smaller system $\overline{S}$. If $\overline{S}$ has no solutions or inifinitely many solutions, then so does S. Otherwise, we evaluate R_x in the solution of $\overline{S}$ to get the solution value for x.

Complexity of Variable Elimination. Using this algorithm, we have to remove $O(n)$ unknowns. When removing x, we might have to replace an expression of size $O(n)$, i.e. R_x, in $O(m)$ potential other equations where x has appeared. Hence, the overall runtime is $O(n^2 \cdot m)$.

Exploiting Treewidth. Given a tree decomposition $(T, \langle B_i \rangle)$ of the primal graph $G(S)$, we choose the unknows in the usual order, i.e. we always choose an unknown x that appears only in a leaf bag. If x does not appear in any equations, then we can simply remove it and then S is satisfiable iff $\overline{S}$ is satisfiable. Moreover, if S is satisfiable, then it has infinitely many solutions, given that x is not restricted. Otherwise, there is an equation $\mathfrak{e}$ in which x appears with non-zero coefficient, and hence we can rewrite this equation as $x = R_x$. Note that x has $O(t)$ neighbors in $G(S)$, given that it only appears in a leaf bag and all of its neighbors should also appear in the same bag, hence the length of R_x is $O(t)$, too. The problem is that x might have appeared in any of the other $O(m)$ equations. Hence, replacing it with R_x in every equation will lead to a runtime of $O(m \cdot t)$. We repeat this for every unknown, so our total runtime is $O(n \cdot m \cdot t)$, which is not linear.

Applying Gram-Schmidt. The crucial observation is that while x might have appeared in as many as m equations, not all of them are linearly independent. Let $\mathfrak{E}_x$ be the set of equations containing x and B_l be the leaf bag in which x appears and assume that $B_l = \{x, y_1, \dots, y_{k-1}\}$. Then the only unknowns that can appear together with x in an equation are $y_1, \dots, y_{k-1}$. In other words, all equations in $\mathfrak{E}_x$ are over B_l. Hence, we can apply the Gram-Schmidt process on $\mathfrak{E}_x$ to remove the unnecessary equations and only keep at most k equations that form an orthogonal basis (or alternatively realize that the system is unsatisfiable). A pseudocode of our approach is given in Algorithm 4.3.

Correctness and Complexity. Given that we are operating in dimension $k = O(t)$, each application of Gram-Schmidt will take $O(t^2 \cdot |\mathfrak{E}_x|)$ time. Thus, our runtime is $O((n+m) \cdot t^2)$, which is linear in the size of the system. As in previous algorithms, our approach always keeps the tree decomposition valid. Hence, we have the following theorem:

Theorem 4.3. *Given a system of m linear equations over n unknowns, its primal graph, and a tree decomposition of the primal graph with width t and $O(n+m)$ bags, our algorithm solves the system in time $O((n+m) \cdot t^2)$.*

The algorithm can easily be extended to find a basis for the solution set.

4.4.5 Mean Payoff Parameterized by Treewidth

The mean payoff problem is a bit trickier than previous cases. To solve it, we first need to define several basic notions.

Algorithm 4.3: Solving a system S of linear equations, given its primal graph $G = (V, E)$ and exploiting a tree decomposition $(T, \langle B_i \rangle)$ of G. Note that G is undirected. Lines 16–17 ensure that G always remains a supergraph of the primal graph of S and that $(T, \langle B_i \rangle)$ always remains a valid tree decomposition of G.

```
1  Function SolveLinearSystem($S, G = (V, E), (T, \langle B_i \rangle)$):
2      if $V = \{\emptyset\}$ then
3          $solution \leftarrow \emptyset$
4          return $solution$
5      else
6          repeat
7              Choose an arbitrary leaf node $l \in V_T$
8              $\bar{l} \leftarrow$ parent of $l$
9              if $B_l \subseteq B_{\bar{l}}$ then
10                 $V_T \leftarrow V_T \setminus \{l\}$
11                 $E_T \leftarrow E_T \setminus \{(l, \bar{l})\}$
12             else
13                 Choose an arbitrary $x \in B_l \setminus B_{\bar{l}}$
14                 $B_l \leftarrow B_l \setminus \{x\}$
15                 break
16         foreach $y_1, y_2 \in B_l : y_1 \neq y_2$ do
17             $E \leftarrow E \cup \{(y_1, y_2)\}$
18         $\mathfrak{E} \leftarrow$ equations in $S$ that contain $x$ with non-zero coefficient
19         $S \leftarrow S \setminus \mathfrak{E}$
20         if Gramm-Schmidt($\mathfrak{E}$) = Unsatisfiable then
21             return Unsatisfiable
22         $\mathfrak{E} \leftarrow$ Gramm-Schmidt($\mathfrak{E}$)
23         if $\mathfrak{E} = \emptyset$ then
24             if SolveLinearSystem($S, G \setminus \{x\}, (T, \langle B_i \rangle)$) = Unsatisfiable then
25                 return Unsatisfiable
26             else
27                 return Underdetermined
28         else
29             Choose an arbitrary $\mathfrak{e} \in \mathfrak{E}$ and write it as $x = R_x$
30             $\mathfrak{E} \leftarrow \mathfrak{E} \setminus \{\mathfrak{e}\}$
31             foreach $\mathfrak{e}' \in \mathfrak{E}$ do
32                 $\mathfrak{e}' \leftarrow \mathfrak{e}'[R_x/x]$ //replace every occurrence of $x$ with $R_x$
33             $S \leftarrow S \cup \mathfrak{E}$
34             if SolveLinearSystem($S, G \setminus \{x\}, (T, \langle B_i \rangle)$) $\in$
                 {Unsatisfiable, Underdetermined} then
35                 return SolveLinearSystem($S, G \setminus \{x\}, (T, \langle B_i \rangle)$)
36             else
37                 $solution \leftarrow$ SolveLinearSystem($S, G \setminus \{x\}, (T, \langle B_i \rangle)$)
38                 $solution \leftarrow solution[x \mapsto [R_x]_{solution}]$
39                 return $solution$
```

Strongly Connected Components. Given an MC $C = (V, E, \delta)$, a *Strongly Connected Component* (SCC) is a maximal subset $\alpha \subseteq V$, such that for every pair of vertices $u, v \in \alpha$, there is a path from u to v in C. An SCC β is called a *Bottom Strongly Connected Component* (BSCC) if no other SCC is reachable from β. It is well-known that every vertex belongs to a unique SCC and that there is a linear-time algorithm that computes the SCCs and BSCCs of any given MC. An MC is called *ergodic* if its vertex set consists of only a single BSCC.

Limiting Distribution [Norris, 1998]**.** Given an ergodic MC $C = (\beta, E, \delta)$ with a single BSCC β and an arbitrary vertex $u \in \beta$, we define the *limiting distribution* δ_{lim} over β as follows: $\delta_{\text{lim}}(v) := \lim_{n\to\infty} \mathbb{E}_u \left[\frac{1}{n} \cdot |\{i \mid 0 \leq i < n \wedge \pi_i = v\}|\right]$, where π is a random walk beginning at u. Informally, $\delta_{\text{lim}}(v)$ is the fraction of time that we are expected to spend in vertex v, when we start a random walk in C. Note that due to ergodicity, the starting vertex of the random walk does not matter. We can similarly define a limiting distribution δ^E_{lim} over the edges of C by letting $\delta^E_{\text{lim}}(u, v) := \delta_{\text{lim}}(u) \cdot \delta(u)(v)$.

Mean Payoff based on Limiting Distribution. From the definition above, it is easy to see that the mean payoff value *ExpMP*(u) is the same for every vertex $u \in \beta$ of the ergodic MC. More specifically, we have *ExpMP*$(u) = \sum_{(v_1,v_2)\in E} R(v_1, v_2) \cdot \delta^E_{\text{lim}}(v_1, v_2)$. Therefore, computing the *ExpMP* values is reduced to computing the limiting distribution.

Mean Payoff in Non-ergodic MCs. Now consider a general MC $C = (V, E, \delta)$ and a vertex $u \in V$. If u is in a BSCC β, then any path starting from u will never leave β. Therefore, $\mathit{ExpMP}_V(u) = \mathit{ExpMP}_B(u)$. On the other hand, if u is in a non-bottom SCC α, then the random walk beginning from u will eventually reach a BSCC almost-surely (with probability 1). Let $\beta_1, \beta_2, \ldots$ be the BSCCs of C and $b_i \in \beta_i$. Hence, given that we can ignore a finite prefix when computing mean payoffs, the expected mean payoff from u is

$$\mathit{ExpMP}(u) = \sum_i \mathit{HitPr}(u, \beta_i) \cdot \mathit{ExpMP}(b_i) = \sum_i \mathit{HitPr}(u, b_i) \cdot \mathit{ExpMP}(b_i).$$

Every vertex in β_i has the same expected mean payoff and will be reached almost-surely from every other vertex in β_i, i.e. hitting probabilities between pairs of vertices in the same BSCC β_i are always 1, hence the choice of b_i is arbitrary.

The Algorithm. We use the two observations above to compute expected mean payoffs in a given MC C. Algorithm 4.4 summarizes our approach. As explained above, the problem is now reduced to computing $\delta_{\lim}$ (Line 5) and hitting probabilities (Lines 11–12). We now explain how we handle each of these two subproblems.

Algorithm 4.4: Computing Expected Mean Payoffs in a given MC C.

```
1  Function ComputeExpMP($C = (V, E, \delta)$):
2      $\beta_1, \beta_2, \ldots \leftarrow$ BSCCs of $C$
3      Choose an arbitrary vertex $b_i$ from each $\beta_i$
4      foreach $\beta_i$ do
5          Compute $\delta_{\lim}$ for $(\beta_i, E \cap (\beta_i \times \beta_i), \delta)$
6          foreach $(v_1, v_2) \in E \cap (\beta_i \times \beta_i)$ do
7              $\delta^E_{\lim}(v_1, v_2) \leftarrow \delta_{\lim}(v_1) \cdot \delta(v_1)(v_2)$
8          $x \leftarrow \sum_{(v_1,v_2) \in E \cap (\beta_i \cdot \beta_i)} R(v_1, v_2) \cdot \delta^E_{\lim}(v_1, v_2)$
9          foreach $u \in \beta_i$ do
10             $ExpMP(u) \leftarrow x$
11     foreach $u \in V \setminus \bigcup \beta_i$ do
12         $ExpMP(u) \leftarrow \sum_i HitPr(u, b_i) \cdot ExpMP(b_i)$
```

Computing Limiting Distribution of an Ergodic MC. Let $C = (B, E, \delta)$ be an ergodic MC. We define the linear system S_C as follows:

- We add a variable x_u for each vertex $u \in B$.
- For each vertex $u \in B$ with *predecessors* $u_1, u_2, \ldots, u_k$, we add a constraint $x_u = \sum_{i=1}^{k} x_{u_i} \cdot \delta(u_i)(u)$.
- We add the constraint $\sum_{u \in B} x_u = 1$.

The system S_C has a unique solution in which the value of each x_u is equal to $\delta_{\lim}(u)$ [Norris, 1998]. Unfortunately, the last constraint includes all of the variables in the system and hence the primal graph of our system does not have constant treewidth. However, this is a minor restriction. We can consider the system S'_C obtained by ignoring the last constraint. This system is homogeneous and its primal graph is the isomorphic to (V, E) and has treewidth t. Hence, we can use the algorithm of Section 4.4.4 to find an arbitrary solution to S'_C. We can then scale all the values in our solution to satisfy the constraint $\sum_{u \in B} x_u = 1$, hence obtaining the unique solution of S_C. Therefore, Line 5 of Algorithm 4.4 takes $O(|\beta_i| \cdot t^2)$ time according to Theorem 4.3.

Computing Expected Mean Payoff for non-BSCC vertices. We can compute all the values of $ExpMP(u)$ for $u \in V \setminus \bigcup \beta_i$ (Lines 11–12) with *a single* call to our algorithm for hitting probabilities (Algorithm 4.2, Section 4.4.2). Note that Algorithm 4.2 does not rely on the premise that the function δ can only have values between 0 and 1. Hence, we can set all the b_i's as targets, but when merging them to a single target $\mathfrak{t}$, we set $\delta(b_i)(\mathfrak{t}) = ExpMP(b_i)$, which was computed in Line 10. This ensures that the value computed for $ExpMP(u)$ is exactly the RHS of Line 12 in Algorithm 4.4. Using this trick, the runtime of Lines 11–12 of our algorithm is $O(n \cdot t^2)$ as per Theorem 4.1.

Given the discussion above, we have the following theorem:

Theorem 4.4. *Given an MC with n vertices and treewidth t and an optimal tree decomposition with $O(n)$ bags, Algorithm 4.4 computes expected mean payoffs from every vertex in $O(n \cdot t^2)$.*

Remark 4.1. *In SI over MDPs with mean payoff objectives, one also needs to compute additional values, called* potentials *or* biases *[Křetínský and Meggendorfer, 2017, Puterman, 2014]. However, this computation is classically reduced to solving a system of linear equations whose primal graph is the MDP. Hence, the algorithm of Section 4.4.4 can be applied, and our improvements for computing mean payoff in MCs extend to MDPs.*

4.5 Experimental Results

In this section, we report on a C/C++ implementation of our algorithms and provide a performance comparison with previous approaches in the literature.

Compared Approaches. We consider the hitting probability and discounted sum problems for MCs and MDPs. In the case of MCs, we directly use our algorithms from Section 4.4.2 and Section 4.4.3. For MDPs, we use strategy iteration, where we use the above algorithms for the strategy evaluation step in each iteration. We compare our approach with the following alternatives:

- *Classical Approaches.* In case of MCs, we compare against a highly-optimized implementation of Gaussian elimination (Gauss). For MDPs, we consider our own implementation of value iteration (VI) and strategy iteration (SI).

- *Numerical and Industrial Optimizers.* We use Matlab and Gurobi to solve systems of linear equalities corresponding to MCs. For MDPs, we use Matlab, Gurobi and lpsolve [Berkelaar et al., 2003] to handle the corresponding LPs.

- *Probabilistic Model Checkers.* The well-known model checkers Storm [Dehnert et al., 2017] and Prism [Kwiatkowska et al., 2011] have standard procedures for computing hitting probabilities, but not for discounted sums. We therefore compare our runtimes on hitting probability instances with their runtimes.

Despite the fact that treewidth has been extensively studied in verification and model checking [Obdržálek, 2003, Ferrara et al., 2005], including for the analysis of Markov decision processes [Chatterjee and Łącki, 2013], to the best of our knowledge there are no benchmark suites consisting of low-treewidth MCs/MDPs. Previous works such do not provide any experimental results.

Motivation for Benchmarks. The main motivation to study MCs/MDPs with small treewidth is that they occur naturally in static program analysis, where a key algorithmic problem is reachability on the CFGs, e.g. as shown in Chapter 3, data-flow analyses in frameworks such as IFDS are reduced to reachability. Moreover, probability annotations of the CFG are useful in many contexts such as (i) in probabilistic programs where the branches are probabilistic; or (ii) when branch-profiling information is available that assigns probabilities to branch execution [Smith, 1998]. If we consider CFGs where all branches are deterministic or probabilistic, then we have MCs; and if there are also non-deterministic branches, then we have MDPs. In both cases, the reachability analysis in CFGs with probability annotation corresponds to the computation of hitting probabilities. Therefore, hitting probabilities can be used to answer questions like "given the branch profiles, compute the probability that a given pointer is null in some instruction". Additionally, [De Alfaro et al., 2003] shows how discounted-sum objectives are relevant in the analysis of systems, e.g. with discounted-sum reachability we can model that a later bug is better than an earlier one. It is well-established that structured programs have small treewidth, both theoretically [Thorup, 1998] and experimentally [Gustedt et al., 2002].

Benchmarks. Given the points above, we used CFGs of the 40 Java programs from the DaCapo suite [Blackburn et al., 2006] as our benchmarks. They have between 33 and 103918 vertices and transitions. To obtain MDPs, we randomly (with probability 1/2) turned each vertex into a Player 1 or a probabilistic one. We assigned random probabilities to each outgoing edge of a probabilistic vertex. To obtain MCs, we did the same, except that all vertices are probabilistic. For the hitting probabilities problem, we chose one random vertex from each connected component of the control flow graphs as a target. In case of discounted sum, we uniformly chose a discount factor between 0 and 1 for each instance, and also assigned random integral rewards between -1000 to 1000 to each edge. Finally, we used our own tool, JTDec [Chatterjee et al., 2017b], to compute tree decompositions. In each case the width of the obtained decomposition was no more than 9. See [Asadi et al., 2020b] for details of benchmarks.

Results. The runtimes are shown in Figures 4.4–4.7. Note that the y-axes are in *logarithmic scale*. For example, Figure 4.4 shows results for computing hitting probabilities in MCs, where Prism is the slowest tool by far, while our approach comfortably beats every other method. The gap is more apparent in MDPs (Figures 4.6–4.7). Overall, we see that our new algorithms consistently outperform both classical approaches like VI and SI, and highly optimized solvers and model checkers like Gurobi, Prism and Storm, by one or more orders of magnitude. Hence, the theoretical improvements are also realized in practice. See [Asadi et al., 2020b] for raw numbers.

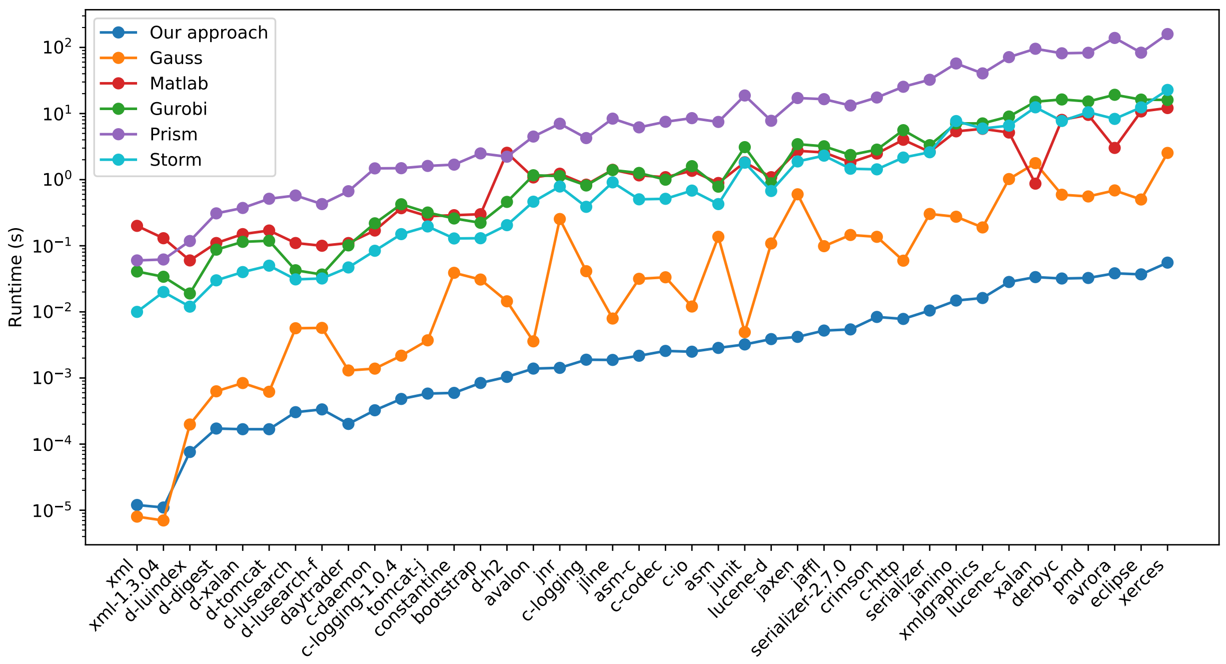


Figure 4.4: Experimental Results for Computing Hitting Probabilities in MCs.

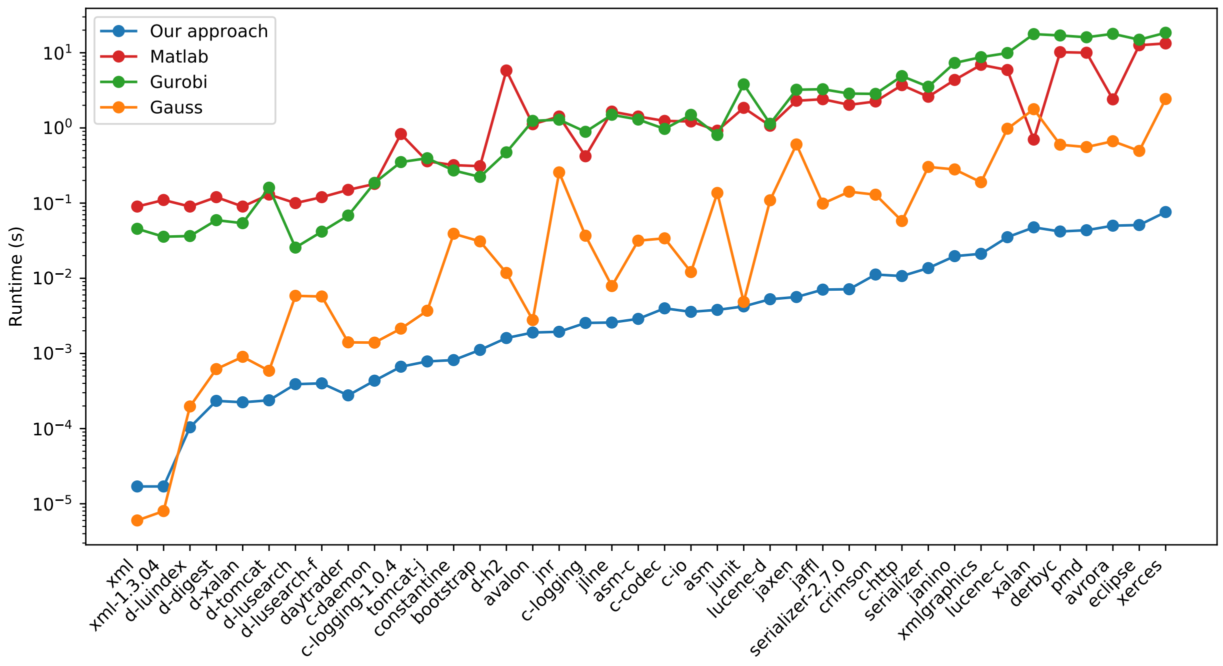


Figure 4.5: Experimental Results for Computing Expected Discounted Sums in MCs.

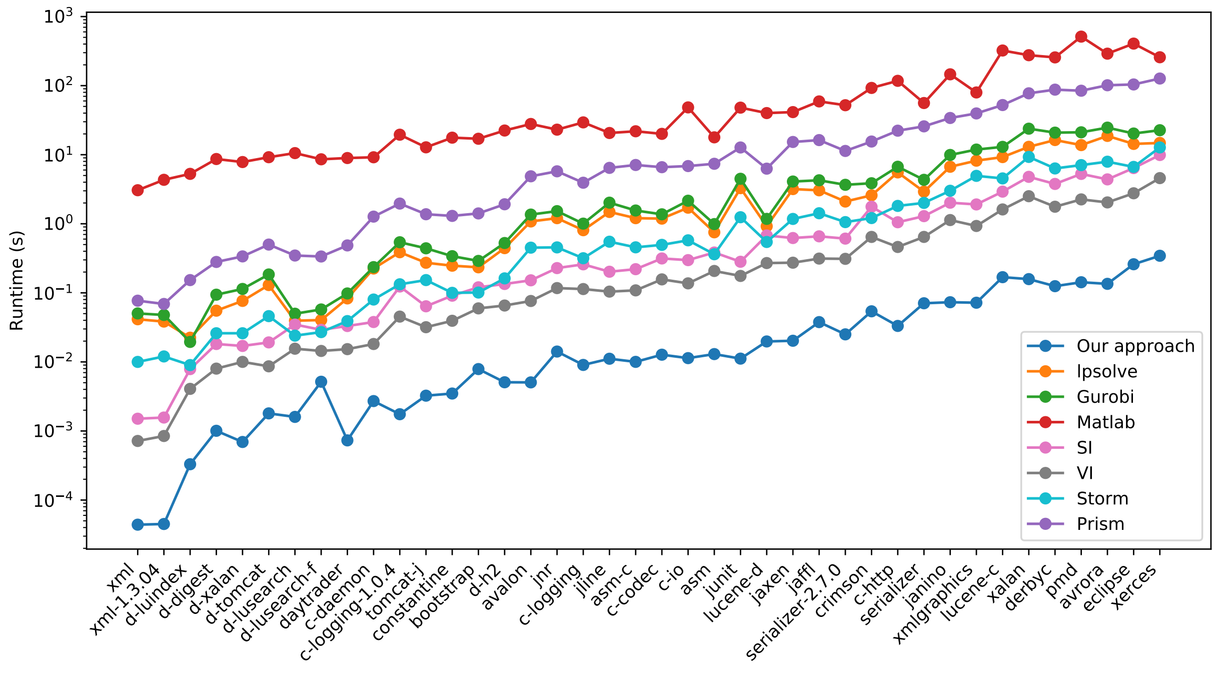


Figure 4.6: Experimental Results for Computing Hitting Probabilities in MDPs.

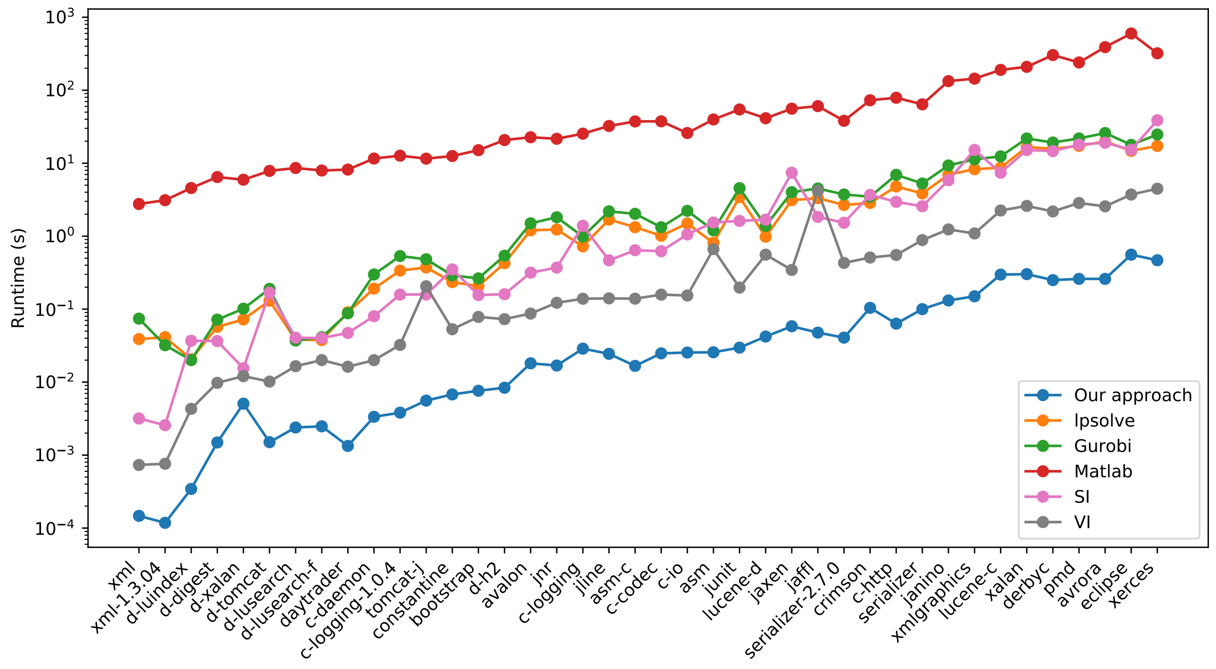


Figure 4.7: Experimental Results for Computing Expected Discounted Sums in MDPs.

5

Faster Algorithms for Data Packing

This chapter originally appeared in the following publication:

[•] Chatterjee, K., **Goharshady, A. K.**, Okati, N., and Pavlogiannis, A. **Efficient Parameterized Algorithms for Data Packing**. In *46th ACM Symposium on Principles of Programming Languages* (**POPL**), 2019.

5.1 Introduction

Background. There is a huge gap between the speeds of modern caches and main memories, and therefore cache misses account for a considerable loss of efficiency in programs. The predominant technique to address this issue has been *Data Packing*: data elements that are frequently accessed within time proximity are packed into the same cache block, thereby minimizing accesses to the main memory. In this chapter, we consider the algorithmic problem of Data Packing on a two-level memory system. Given a reference sequence R of accesses to data elements, the task is to partition the elements into cache blocks such that the number of cache misses on R is minimized. The problem is notoriously difficult: it is NP-hard even when the cache has size 1, and is hard to approximate for any cache size larger than 4. Therefore, all existing techniques for Data Packing are based on heuristics and lack theoretical guarantees.

Our Results. In this chapter, we present the first positive theoretical results for Data Packing, along with new and stronger negative results. We consider the problem under the lens of the underlying *access hypergraphs*, which are hypergraphs of affinities between the data elements, where the order of an access hypergraph corresponds to the size of the affinity group. We study the problem parameterized by the treewidth of access hypergraphs. Our main results are as follows: we show there is a number q^* depending on the cache parameters such that (a) if the access hypergraph of order q^* has constant treewidth, then there is a *linear-time* algorithm for Data Packing; and (b) the Data Packing problem remains NP-hard even if the access hypergraph of order $q^* - 1$ has constant treewidth. Thus, we establish a fine-grained dichotomy depending on a single parameter, namely, the highest order among access hypegraphs that have constant treewidth; and establish the optimal value q^* of this parameter. Finally, we present an experimental evaluation of a prototype implementation of our algorithm. Our results demonstrate that, in practice, access hypergraphs of many commonly-used algorithms have small treewidth. We compare our approach with several state-of-the-art heuristic-based algorithms and show that our algorithm leads to significantly fewer cache-misses.

5.2 Paging and Packing

We consider the problem of Data Packing over a two-level memory system consisting of a small cache and a large main memory. Given a reference sequence of memory accesses to data elements, the goal is to organize the data elements into blocks in order to minimize cache misses. Intuitively, putting contemporaneously-accessed elements in the same block reduces the number of cache misses, but existing heuristic-based results do not present any theoretical guarantees. In this chapter, we consider this problem from a theoretical perspective and establish its complexity by presenting exact algorithms and stronger hardness results. We start with an overview of previous results and a formal definition of the problem.

5.2.1 Overview of the Problems and Previous Results

Cache Management. Consider a memory system with an associative cache and a main memory. Data items are stored in the main memory and organized into sets of a small size, which are called *blocks (or pages)*. All data items have the same size and all blocks can hold the same number of data items. The cache has a small capacity and can hold a few blocks at any given time. Whenever a program needs to access a data element, its corresponding block must be present in the cache before the access can happen. Therefore, if the block is not already in the cache, it will be copied into the cache from the main memory, potentially by evicting another block. This copying process is called a *cache miss*, and given the considerably slower speed of the main memory, cache misses are very time-consuming and lead to significant overhead [Wulf and McKee, 1995]. Therefore, the problem of cache management, i.e. minimizing the number of cache misses, is of great importance in compilers and operating systems. Cache management can naturally be divided in two parts [Calder et al., 1998]: (i) deciding on how to replace the blocks in the cache, i.e. which block to evict when the cache is full and a miss occurs and (ii) deciding on the placement scheme of the data items inside blocks. These problems are respectively called *Paging (or choosing a replacement policy)* [Sleator and Tarjan, 1985] and *Data Packing* [Thabit, 1982, Lavaee, 2016].

Paging (Replacement Policy). In paging, given a data placement scheme that divides the data items into blocks and a so-called *reference sequence* of accesses to data elements, the problem is to choose a block to be evicted each time a cache miss occurs. The goal is to do this in a way that minimizes the total number of cache misses over the reference sequence [Panagiotou and Souza, 2006]. An algorithm that chooses the block to be evicted is called a *replacement policy.* Common replacement policies include *FIFO*, which evicts the oldest block in the cache, and *LRU*, which evicts the least recently used block [Borodin et al., 1995, Lavaee, 2016]. Note that both FIFO and LRU can also be applied in the online setting, i.e. when the algorithm does not know the entire sequence in advance and can only observe accesses as they are made. In the offline case, where the entire reference sequence is given in the beginning, the optimal replacement policy is to evict the block whose first use is furthest in the future [Borodin et al., 1995]. This is called the *optimal offline policy (OOP).* We primarily focus on LRU as the replacement policy, because it is the one that is most commonly used in practice [Zhong et al., 2004]. However, most of our results extend to FIFO and OOP, too.

Data Packing. The other aspect of cache management, which is the focus of this chapter, is Data Packing [Thabit, 1982]. Consider a cache with a capacity of m blocks, where each block can store p data items. Given a reference sequence R of length N of accesses to n distinct data items and a replacement policy, Data Packing asks for the optimal placement of data items into blocks in order to minimize the number of cache misses. The parameters m and p are considered to be small constants, and the complexity is studied wrt n and N which are large. Data Packing is an extremely hard problem and is known to be hard to approximate within any non-trivial factor, i.e., any factor significantly less than N, unless P=NP [Lavaee, 2016].

Heuristics and Affinity. Given the hardness of cache management and Data Packing, the research in this area has been mostly focused on developing heuristics. The intuition behind many of these heuristics is to exploit the underlying affinities between data elements or blocks by trying to place elements that are commonly accessed together in the same block or evicting the block that is less frequently accessed in conjunction with the rest of the blocks in the cache [Zhong et al., 2004, Ding and Kandemir, 2014, Calder et al., 1998, Ding and Kennedy, 1999, Han and Tseng, 2006]. Some approaches, such as [Zhang et al., 2006], provide more sophisticated heuristics and construct a hierarchy of affinities. However, none of the existing heuristics provide any theoretical guarantees.

Access Graphs. The concept of access graph [Borodin et al., 1995] has been introduced to model the affinities between data elements or blocks. An access graph is simply a graph in which there is a vertex corresponding to every data item and two vertices are connected by an edge if their respective items appear consecutively in the reference sequence. Access graphs might be weighted to model how many times every pair of elements have appeared consecutively. Similar structures and extensions of access graphs to access hypergraphs have been introduced in [Thabit, 1982, Lavaee, 2016] where they are called proximity (hyper)graphs. Moreover, most of the heuristic-based approaches also consider variants of the notion of access graphs.

Cache Misses vs Cache Hits. We consider the Data Packing problem, which asks to *minimize* the cache *misses*. Its natural dual problem is to *maximize* cache *hits*. While the two problems are equivalent in case of exact algorithms, an approximation algorithm for maximum cache hits does not necessarily lead to an approximation for minimum cache misses [Lavaee, 2016]. For example, if in an access sequence of length N we have $N - \sqrt{N}$ cache hits and $\sqrt{N}$ cache misses, an approximation of $N - \sqrt{N}$ hits can lead to an arbitrarily bad approximation of $\sqrt{N}$ cache misses. In practice, cache misses occur much less frequently than cache hits, but contribute significantly to the runtime overhead of programs. Thus, approximation of cache misses is more important than approximation of cache hits, and the Data Packing problem is defined in terms of cache misses.

Previous Results on Cache Management. To the best of our knowledge, all theoretical results on minimizing cache misses are negative or hardness results. We summarize some of the main results in this area. Given a reference sequence R of length N and a cache with a capacity of m blocks, the following results have been shown:

(i) In [Petrank and Rawitz, 2002], the authors considered the general problem of Cache-conscious Data Placement, consisting of both Paging and Data Packing. They showed the problem is NP-hard and unless P=NP, it cannot even be approximated within a factor of $O(N^{1/2-\epsilon})$.

(ii) In the same paper, it was shown that any algorithm that does not process the entire sequence, but instead relies on pairwise affinity information on data items, such as the access graph, cannot find a solution within a factor of $m - 3$ from the optimal, even with unbounded time.

(iii) In [Lavaee, 2016], it was shown that Data Packing is NP-hard and hard to approximate within a factor of $O(N^{1-\epsilon})$ unless P = NP.

Given these hardness results, Data Packing is usually handled by heuristic-based algorithms that do not provide any theoretical guarantee. The only positive theoretical result deals with approximating maximum cache hits:

(iv) In [Lavaee, 2016] it was established that the dual problem of Data Packing with the goal of maximizing cache hits, instead of minimizing cache misses, is approximable within a constant factor. However, this does not approximate the optimal number of cache misses.

5.2.2 Formal Definitions

In this section, we formalize the problem of data packing and fix our notation. We also formally present previously-known hardness results. The problem was first studied in [Thabit, 1982]. Here, we present an adaptation of its definition as formalized in [Lavaee, 2016].

Notation. A hypergraph is a pair $G = (V, E)$ consisting of a finite set V of vertices and a multi-set E of hyperedges. Each hyperedge $e \in E$ is a subset of V. Let $G = (V, E)$ be a (hyper)graph, and $X \subseteq V$, then we denote by $G[X]$, the induced subgraph of G over X, i.e. $G[X] = (X, \{e \in E \mid e \subseteq X\})$. Given two (hyper)graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, we define their union and intersection in the natural way, i.e. $G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2)$ and $G_1 \cap G_2 = (V_1 \cap V_2, E_1 \cap E_2)$. If $\mathcal{F}$ is a family of sets, we write $\cup\mathcal{F}$ (resp. $\cap\mathcal{F}$) to denote $\cup_{A\in\mathcal{F}} A$ (resp. $\cap_{A\in\mathcal{F}} A$). Given two functions $f, g : A \to \mathbb{Z}$, equality and summation are defined in a pointwise manner, i.e. $f \equiv g \Leftrightarrow \forall a \in A; \quad f(a) = g(a)$, and for any $a \in A$, we have $(f+g)(a) = f(a) + g(a)$. Given a function $f : A \to B$ and a subset $A' \subseteq A$, we use $f_{|A'}$ to denote the restriction of f to A'. This restriction is a function of the form $f_{|A'} : A' \to B$ that agrees with f on every point in A'. For a set X, we write $P(X)$ to denote the power set of X, i.e. the set of all subsets of X.

Data Placement Schemes. Given a set D of size n of *data items* and a positive integer p, a data placement scheme σ is a partitioning of D into blocks of size at most p. We call p the *packing factor*. It is often useful to think of σ as an equivalence relation on D whose equivalence classes are the blocks. Hence, following the usual notation, we write $x\sigma y$ to denote that x and y are in the same block, $[x]_\sigma$ to denote the block of σ that contains the data element x and D/σ to denote the set of blocks or equivalence classes of σ.

Replacement Policies. Given a set D of n data items, a cache of size m, a data placement scheme σ, and a sequence $R \in D^N$ of accesses to data items, a replacement policy is a function that decides which block must be evicted from the cache at each time. Formally, a replacement policy is a function $\pi : \{0, 1, 2, \dots, N\} \to P(D/\sigma)$ that assigns to each time point i, the set of blocks that are present in the cache right before the access $R[i]$. Any such policy must satisfy the following:

- $\pi(0) = \emptyset$, i.e. the cache must be empty at the beginning;
- For all $1 \le i \le N$, $|\pi(i)| \le m$, i.e. there are at most m blocks in the cache at each time;
- For all $1 \le i \le N$, $|\pi(i) \setminus \pi(i-1)| \le 1$ and $|\pi(i-1) \setminus \pi(i)| \le 1$, i.e. at most one block can be added to the cache and at most one block can be evicted at each step;
- For all $1 \le i \le N$, $R[i] \in \cup\pi(i)$, i.e. the block containing an access $R[i]$ must be in the cache right before that access.

Remark 5.1. *Note that the replacement policy only matters when the cache has a size of at least* 2*. When the cache has unit size, there is always a unique choice for the block that must be evicted.*

Cache Misses. Given a data placement scheme σ and a replacement policy π as above, the number of cache misses caused by σ and π over R is defined as the number of times a new block is loaded into the cache. Formally, $\text{misses}(\sigma, \pi) = |\{i \mid 1 \leq i \leq N, \pi(i) \setminus \pi(i-1) \neq \emptyset\}|$.

The LRU Policy. Due to its popularity, we assume throughout this paper that the replacement policy is LRU, i.e. the Least-Recently-Used block is always evicted from the cache. However, most of our results carry over to First-In-First-Out (FIFO) and the Optimal Offline Policy (OOP), as well. Recall that FIFO evicts the oldest block in the cache and OOP evicts the block that is going to be used furthest in the future.

Remark 5.2. *LRU can cause at most m times as many cache misses as OOP, where m is the number of blocks that can fit into the cache. In practice, it usually leads to between* 2 *to* 3 *times as many cache misses [Panagiotou and Souza, 2006].*

We are now ready to define our main problem:

The Data Packing Optimization Problem. Consider a memory subsystem that consists of n distinct data elements and a fully-associative cache with a capacity of m blocks and a packing factor of p. Given a sequence R of length N of references to data elements, the Data Packing problem asks for a data placement scheme σ that minimizes the number of cache misses incurred by the reference sequence R, using LRU as the replacement policy. We denote an instance of the Data Packing problem by $I = (n, m, p, R)$.

Parameters. In the sequel, we consider the parameters m and p to be small constants and try to find polynomial algorithms in terms of N and n.

The Hardness of Data Packing. Note that we are considering the problem of minimizing cache misses, not that of maximizing cache hits. While the two problems are equivalent in terms of exact algorithms, approximating the minimal number of cache misses is much harder than approximating the maximal number of cache hits. The latter problem admits a polynomial-time constant-factor approximation [Lavaee, 2016]. In contrast, the following theorem shows that the former problem is hard to even approximate.

Theorem 5.1 ([Lavaee, 2016])**.** *Assuming either LRU, FIFO or OOP as the replacement policy, we have the following hardness results:*

- *For any m and any $p \geq 3$, Data Packing is NP-hard.*
- *Unless P=NP, for any $m \geq 5$, $p \geq 2$ and any constant $\epsilon > 0$, there is no polynomial algorithm that can approximate the Data Packing problem within a factor of $O(N^{1-\epsilon})$.*

We now define the concepts of access graph and access hypergraph. Various similar notions have been defined in the past, and are sometimes called affinity graphs or proximity graphs. These hypergraphs will later serve as a basis for reducing the Data Packing problem to a graph problem.

Access Graph. Given a sequence R of length N of accesses to data elements from a set D of size n, the access graph of R is a simple graph $G_R = (V, E)$ in which V consists of n vertices, each corresponding to one of the data elements in D, and there is an edge between two distinct vertices iff their corresponding data elements appear consecutively somewhere in R. More formally, $\{u, v\} \in E$ iff $u \neq v$ and there exists an index i, such that $\{R[i], R[i+1]\} = \{u, v\}$.

Intuitively, one can think of the graph G_R as the structure on data elements that is respected by the access sequence R, in the sense that R can only go from a vertex in G_R to one of its neighbors. Moreover, G_R is the sparsest graph over which R is a (possibly non-simple) path.

Example 5.1. *Consider the access sequence $R = \langle a, b, c, a, b, b, d, b, d, e, c, b, f\rangle$. There are 6 data elements in this sequence and its access graph G_R is shown in Figure 5.1. Note that R is a path on this graph and every edge appears somewhere along R, hence no subgraph of G_R has the same property.*

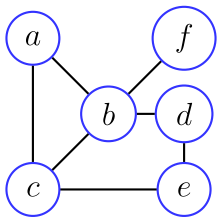


Figure 5.1: The access graph G_R of $R = \langle a, b, c, a, b, b, d, b, d, e, c, b, f\rangle$

We now extend the concept of access graphs to higher order affinity relations between data items, resulting in access hypergraphs.

Hypergraphs and Ordered Hypergraphs. A hypergraph $G = (V, E)$ consists of a set V of vertices and a multiset E of hyperedges. Each hyperedge $e \in E$ is in turn a *subset* of the vertices of G. An ordered hypergraph $G = (V, E)$ consists of a set V of vertices and a set E of ordered hyperedges. Each ordered hyperedge $e \in E$ is a *sequence* of distinct vertices of G, i.e. a hyperedge together with an order on its vertices. Intuitively, hypergraphs are natural extensions of graphs, where each edge can connect more than two vertices. Given a hypergraph G, its primal graph G^p is a graph on the same set V of vertices, where two vertices u and v are connected by an edge iff there exists a hyperedge $e \in E$ containing both u and v. We shall simply refer to hypergraphs and hyperedges as graphs and edges when there is no fear of confusion.

Access Hypergraph. Given a natural number q and an access sequence R as above, the access hypergraph $G^q_R = (V, E)$ is a hypergraph defined as follows:

- There are n vertices in V, each corresponding to one data element;
- For each data access $R[i]$, there is a corresponding hyperedge e_i in E. The hyperedge e_i consists of $R[i]$ and the $q - 1$ *distinct* data elements that are accessed right before $R[i]$. If there are less than $q - 1$ such elements, e_i will include all of them. Concretely, e_i is defined as follows: $e_i := \{R[j] \mid j \leq i \wedge |\{R[j], R[j+1], \ldots, R[i]\}| \leq q\}$.

We call q the *order* of the access hypergraph. It is easy to verify that removing repeated edges from the access hypergraph G^2_R leads to the access graph G_R.

Example 5.2. *Consider the access sequence $R = \langle a, b, c, a, b, b, d, b, d, e, c, b, f \rangle$. Letting $q = 3$, the corresponding access hypergraph G^3_R of order 3 consists of the following hyperedges (sometimes there are multiple copies of the same hyperedge, as shown below. We consider these to be distinct hyperedges):*

$$\{a\}, \{a, b\}, \{a, b, c\} \times 4, \{a, b, d\} \times 3, \{b, d, e\}, \{c, d, e\}, \{b, c, e\}, \{b, c, f\}.$$

Figure 5.2 shows the segments of the sequence that correspond to edges in G^3_R.

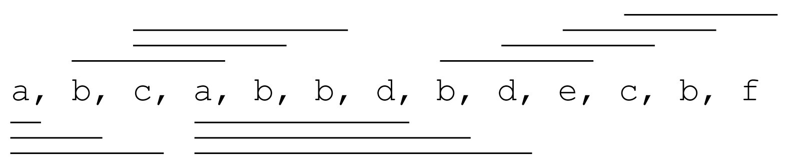


Figure 5.2: Segments of R corresponding to edges in the hypergraph G^3_R

Ordered Access Hypergraphs. Given an access sequence R as above, the ordered access hypergraph $\hat{G}^q_R$ is defined similarly to G^q_R, except that each hyperedge is ordered in the natural way, i.e. in the order of appearance of its corresponding data elements in R. Formally, for every access $R[i]$, there is a corresponding ordered hyperedge e_i in $\hat{G}^q_R$. The ordered hyperedge e_i is a sequence $\langle v_1, v_2, \ldots, v_l \rangle$ of vertices of $\hat{G}^q_R$ such that $v_l = R[i]$, v_{l-1} is the first distinct data element accessed before $R[i]$, v_{l-2} is the second distinct element, etc. Moreover, l is the maximum between q and the number of distinct elements accessed up until $R[i]$.

Example 5.3. *Consider the access sequence $R = \langle a, b, c, a, b, b, d, b, d, e, c, b, f \rangle$. The access hypergraph G^3_R was shown in Example 5.2. We now construct the ordered hyperedges of $\hat{G}^3_R$. Intuitively, we start from any access $R[i]$ in R and go back until we see 3 different data elements. These data elements will form the ordered hyperedge e_i corresponding to $R[i]$. This is illustrated in Figure 5.3. Note that the elements in an ordered hyperedge e_i are ordered by their last access time before or at $R[i]$, e.g. see the hyeperedge $\langle a, d, b \rangle$ in Figure 5.3.*

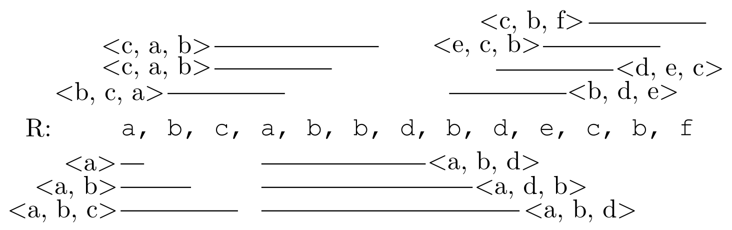


Figure 5.3: Ordered Hyperedges of G and the segments in R to which they correspond

5.3 Summary of Our Results

Treewidth in Data Packing. We will show that Data Packing can be reduced to a graph problem. In many cases when a graph arises from a structured process, the treewidth of the graph is not very large [Bodlaender, 1998]. For Data Packing, the access graphs arise from structured program accessing data from a well-defined data structure. Thus, it is natural to study the problem of Data Packing in terms of the treewidth property of the arising graphs, as we do in the sequel.

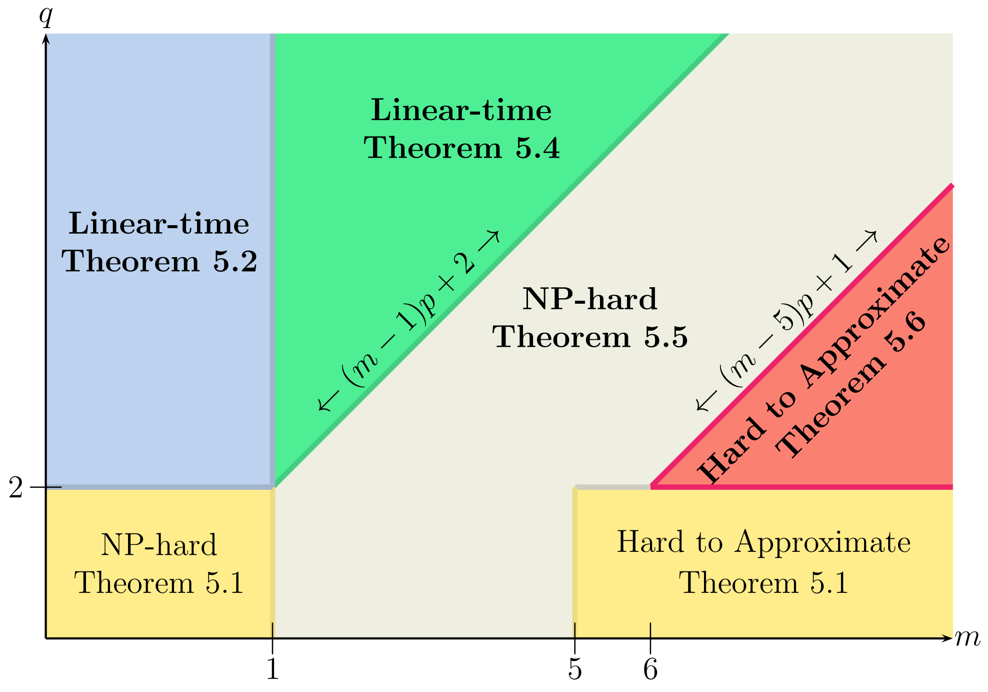


Figure 5.4: The complexity of Data Packing for $p \geq 3$. Here m is the cache size and q is the highest order for which the access hypergraph has constant treewidth. Theorem 5.1 was established in [Lavaee, 2016]. The rest of the picture is filled by this paper and our results are shown in bold face.

Our Contributions. Our contributions include (a) polynomial algorithms for Data Packing in constant treewidth access (hyper)graphs, (b) stronger hardness results, and (c) experimental results demonstrating that our approach leads to considerably fewer cache misses in comparison with previously-known heuristic-based approaches. Concretely, consider that the cache has size m, every block can hold p data items and the reference sequence is of length N

with n distinct items. We consider the access hypergraph of order q. We show that the Data Packing problem can be reduced to a graph partitioning problem of the access (hyper)graph and study whether the treewidth parameter can be exploited for polynomial-time algorithms. Our main results, assuming constant m and p, are as follows:

1. *Results on Access Graphs.* We first consider $q = 2$. Note that order-2 access hypergraphs are basically access graphs. We establish the following results:

    - *Linear-time algorithm.* We present a linear-time algorithm for Data Packing when the access graph is of constant treewidth and $m = 1$ (Theorem 5.2).
    - *Hardness of the exact problem.* The Data Packing problem remains NP-hard for $m \geq 2$ and $p \geq 3$ even if the underlying access graph is a tree (which has treewidth 1) (Theorem 5.3).
    - *Hardness of approximation.* Unless P=NP, for any $m \geq 6, p \geq 2$ and any constant $\epsilon > 0$, the Data Packing problem is hard to approximate within a factor of $O(N^{1-\epsilon})$ even if the underlying access graph is a tree (Theorem 5.3).

2. *Results on Access Hypergraphs.* We then consider access hypergraphs of higher order. Let $q^* = (m-1) \cdot p + 2$. Note that q^* depends only on the cache parameters, and not on n or N. We establish the following results:

    - *Linear-time algorithm.* We present a linear-time algorithm for Data Packing when the access hypergraph of order q^* has constant treewidth (Theorem 5.4).
    - *Hardness of the exact problem.* For $m \geq 2$ and $p \geq 3$, the Data Packing problem remains NP-hard even if the access hypergraph of order $q^* - 1$ has constant treewidth (Theorem 5.5).
    - *Hardness of approximation.* Unless P=NP, for $m \geq 6$ and $p \geq 2$ and any constant $\epsilon > 0$, the Data Packing problem is hard to approximate within a factor of $O(N^{1-\epsilon})$ even if the access hypergraph of order $q^* - 4 \cdot p - 1$ has constant treewidth (Theorem 5.6).

Note that while constant treewidth has been exploited to obtain polynomial-time algorithms for NP-complete graph problems such as Vertex Cover and Hamiltonian Cycle, we show that for Data Packing the constant treewidth property does not always help, and the problem remains hard even when the access hypergraph of order $q^* - 1$ has constant treewidth. Our hardness result and linear-time algorithm present a sharp boundary (or fine-grained dichotomy) that shows when the treewidth can be exploited. Concretely, the hardness of the Data Packing problem can be captured by a single parameter, namely, the highest order amongst access hypergraphs that have bounded treewidth. We establish the optimal value q^* of this parameter which is the necessary and sufficient condition for existence of efficient parameterized algorithms that exploit treewidth.

3. *Experimental results.* We present an experimental evaluation of a prototype implementation of our algorithm, and experimental results on a variety of benchmarks from linear algebra, sorting algorithms, dynamic programming, recursive algorithms, string matching, computational geometry and algorithms on tree data-structures. Our results show that the access hypergraphs of most of the benchmarks have small treewidth. We compare our approach with several state-of-the-art heuristic-based algorithms. The experimental results show that on average our optimal algorithms obtain 15-30% imporvement over the previous heuristic-based approaches.

Significance. In summary, we present the first positive theoretical results for Data Packing, i.e., for cache-miss minimization. We also enrich the complexity landscape as shown in Figure 5.4. Only the results of Theorem 5.1 were known before, and all other results (which are shown in bold) are established in the present work.

5.4 Algorithms and Hardness Results based on Treewidth of Access Graphs

We now consider the problem of Data Packing when parameterized by the treewidth of the underlying access graph. In Section 5.4.1, we provide a linear-time algorithm when $m = 1$ and the access graph has constant treewidth. Note that this problem is NP-hard for general access graphs, as demonstrated by Theorem 5.1. Then, in Section 5.4.2 we show that for $m \geq 2$ the problem remains NP-hard and hard-to-approximate even when the access graph is a tree, i.e. has treewidth 1.

5.4.1 Algorithm for $m = 1$ and Constant-treewidth Access Graph

We are given a Data Packing instance $I = (n, 1, p, R)$, its access graph G_R and an edge-nice tree decomposition $(T, \langle B_i \rangle)$ of the access graph with width t and $O(n \cdot t)$ nodes. We first reduce the problem of Data Packing to a graph problem over G_R and then provide a linear-time fixed-parameter algorithm for solving the graph problem. We start by defining the Minimum-weight p-partitioning problem.

p-partitionings. Given an integer $p > 0$ and a graph $G = (V, E)$, a p-partitioning of G is a partitioning ψ of the set V of vertices such that each partition set has a size of at most p. In other words, a p-partitioning of G is a data placement scheme where the vertices of G are the data elements and p is the packing factor.

Cross Edges. Given a p-partitioning ψ of the graph $G = (V, E)$, an edge $e = \{u, v\} \in E$ is called a cross edge if its two endpoints are in different partition sets, i.e. if $[u]_\psi \neq [v]_\psi$.

Minimum-weight p-partitioning. Given a simple graph $G = (V, E)$, a weight function $w : E \to \mathbb{N}$ and a positive integer p, the Minimum-weight p-partitioning problem asks for a p-partitioning of G in which the total weight of cross edges is minimized.

Reduction of Data Packing to Minimum-weight p-partitioning. We now reduce the Data Packing problem to Minimum-weight p-partitioning. Given an instance $I = (n, 1, p, R)$ of Data Packing, we consider the access graph $G_R = (V, E)$ and define the weight function

$w_R : E \to \mathbb{N}$ as $w_R(\{u,v\}) := |\{i \mid \{R[i], R[i+1]\} = \{u,v\}\}|$. Informally, the weight of an edge is the number of times its two endpoints have appeared consecutively in R. The reduction is now complete.

Lemma 5.1. *The optimal number of cache misses in a Data Packing instance $I = (n, 1, p, R)$ is 1 plus the total weight of cross edges in a Minimum-weight p-partitioning of G_R with weight function w_R.*

Proof. Every p-partitioning ψ of G_R is a data placement scheme for I and vice versa. Given that $m = 1$, the replacement policy does not matter (Remark 5.1) and a cache miss occurs each time R accesses a new block. If we consider R as a path on G_R, a cache miss occurs at the very beginning and then each time this path goes from one equivalence class of ψ to another. Therefore, the number of cache misses of ψ is 1 plus the total weight of cross edges in ψ. $\square$

Example 5.4. *Consider the access sequence $R = \langle \underline{a}, \underline{b}, \underline{c}, a, \underline{b}, b, d, b, d, \underline{e}, \underline{c}, \underline{b}, \underline{f} \rangle$ of Example 5.1 and the Data Packing instance $I = (6, 1, 2, R)$, i.e. each block can store up to 2 data elements. Figure 5.5 shows the graph G_R in which every edge is weighted by the number of times it is traversed in R. An optimal 2-partitioning of G_R is shown in which vertices of the same color are in the same partition. The total weight of cross edges in this partitioning is 7. The corresponding data placement scheme is $\{\{a,c\}, \{b,d\}, \{e\}, \{f\}\}$ which leads to 8 cache misses on R. The cache misses are underlined.*

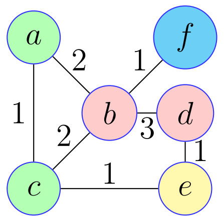


Figure 5.5: An optimal 2-partitioning

We will provide an algorithm for solving the Minimum-weight p-partitioning problem on a graph G using an optimal edge-nice tree decomposition of G. Our algorithm employs a bottom-up dynamic programming technique. We first need several basic concepts to define the algorithm.

States over a Set of Vertices. Given a graph $G = (V, E)$, a natural number p and a subset $A \subseteq V$ of vertices, a state over A is a pair $s = (\varphi, sz)$ such that (i) φ is a partitioning of A in which every equivalence class has a size of at most p, and (ii) sz is a size enlargement function $sz : A/\varphi \to \{0, \dots, p-1\}$ that maps each equivalence class $[v]_\varphi$ to a number which is at most $p - |[v]_\varphi|$. Intuitively, the idea is to take A to be one of the bags in the tree decomposition and later extend a state over A to a p-partitioning of G by adding the vertices in $V \setminus A$. So, a state over A partitions the vertices of A into sets of size at most p and for each partition $[v]_\varphi$ fixes the exact number $sz([v]_\varphi)$ of vertices from $V \setminus A$ that should be added to $[v]_\varphi$. We denote the set of all states over A by $S_{A,p}$ or simply S_A when p is clear from the context.

Realization. We say that a p-partitioning ψ realizes the state $s = (\varphi, sz)$ over A, if (i) the restriction of ψ to A is equal to φ, i.e. $\psi_{|A} = \varphi$ and (ii) for all vertices $v \in A$, $sz([v]_\varphi) = |[v]_\psi| - |[v]_\varphi|$. Intuitively, ψ realizes s if (i) ψ partitions the vertices in A in the same manner as φ and (ii) if a partition $[v]_\psi$ of ψ intersects A, then $[v]_\psi$ contains as many vertices from outside of A as fixed by sz.

Example 5.5. *Figure 5.6 shows all 14 possible states over the set $A = \{a, b, c\}$ of vertices with $p = 2$. In each case, each row denotes one partition set and hence the order of rows and the order of squares in a row does not matter. Empty squares correspond to the possibility of extension of the set, as defined by sz. The optimal 2-partitioning ψ presented in Figure 5.5 realizes the highlighted state in Figure 5.6, because ψ puts a and c in the same partition and puts b in a partition of size 2, whose other member, d, comes from outside the set $\{a, b, c\}$.*

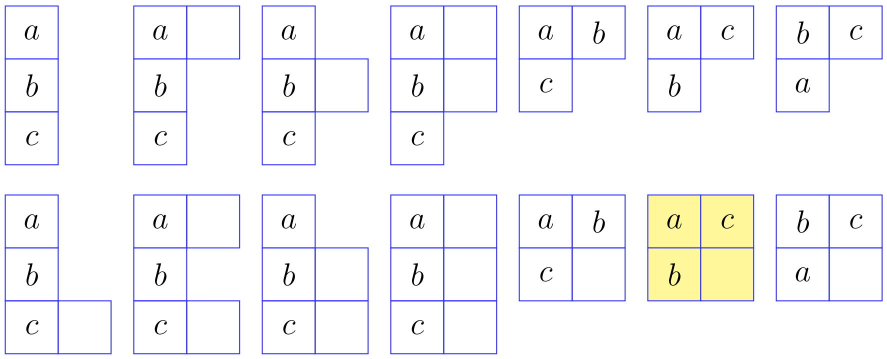


Figure 5.6: All possible states over $A = \{a, b, c\}$ with $p = 2$

Compatibility. We say that two states s and s', respectively over the sets A and A', are compatible if there exists a p-partitioning that realizes both of them. We use the notation $s \doteq s'$ to show compatibility.

Example 5.6. *Intuitively, two states are compatible if they can fit into each other. Figure 5.7 shows the states realized by the 2-partitioning of Figure 5.5 above over the sets $A = \{a, b, c\}$ and $A' = \{d, e, f\}$ and how they can be fitted together to create the entire 2-partitioning.*

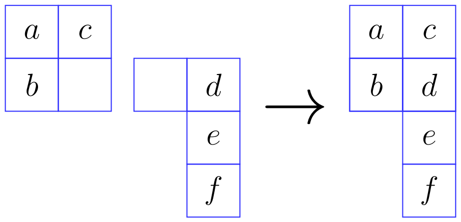


Figure 5.7: Two compatible states over $A = \{a, b, c\}$ and $A' = \{d, e, f\}$

The Algorithm. We are now ready to describe our algorithm in detail. Given a graph G, a weight function w and an optimal edge-nice tree decomposition T of G, our algorithm performs a bottom-up dynamic programming on T. This is broken into three steps, which are described below. We use the graph of Figure 5.5 and its edge-nice tree decomposition in Figure 5.8 as examples.

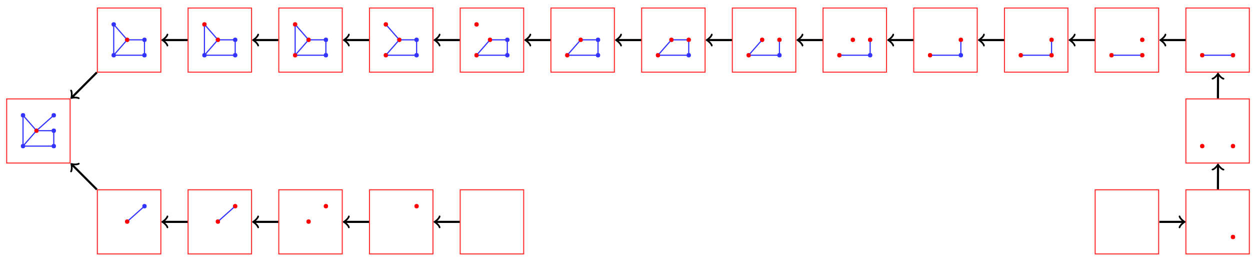


Figure 5.8: An edge-nice tree decomposition of the graph in Figure 5.5. The root r is the leftmost node.

Step 0: Initialization. We define several variables at each node of our tree T. These variables are meant to be computed in a bottom-up manner. Concretely, for every $i \in V_T$ and every state s over the bag B_i, we define a variable $\boldsymbol{dp}[i, s]$ and initialize it to $+\infty$.

Invariant. Formally, our algorithm satisfies the following invariant for every $\boldsymbol{dp}$ variable right after the end of its computation:

$\boldsymbol{dp}[i, s] =$ The minimum total weight of cross edges over all p-partitionings of G_i that realize s.

Intuitively, we are considering the states over the bag B_i and extending them by adding vertices that were introduced in the subtree of i in T.

Step 1: Computation of $\boldsymbol{dp}$. The algorithm starts from the bottom of the tree T and computes the $\boldsymbol{dp}$ variables bottom up, i.e. with an order such that for every node $i \in V_T$ the $\boldsymbol{dp}$ variables at its children are computed before the $\boldsymbol{dp}$ variables of i. For every node $i \in V_T$ and state $s = (\varphi, sz) \in S_{B_i}$, we show how $\boldsymbol{dp}[i, s]$ is computed based on type of the node i:

(1.1) if i is a Leaf: $\boldsymbol{dp}[i, s] = 0$;

(1.2) if i is a Join node with children i_1 and i_2:

$$\boldsymbol{dp}[i, s] = \min_{sz_1 + sz_2 \equiv sz} \boldsymbol{dp}[i_1, (\varphi, sz_1)] + \boldsymbol{dp}[i_2, (\varphi, sz_2)];$$

Note that the summation and equality above are pointwise.

(1.3) if i is an Introduce Vertex node, introducing v, with a single child i_1:

$$\boldsymbol{dp}[i,s] = \boldsymbol{dp}[i_1, (\varphi_{|B_{i_1}}, sz_{|B_{i_1}})];$$

(1.4) if i is an Introduce Edge node, introducing e, with a single child i_1:

$$\boldsymbol{dp}[i,s] = \boldsymbol{dp}[i_1, s] + w(e, \varphi),$$

where $w(e, \varphi)$ is equal to $w(e)$ if e is a cross edge in φ and zero otherwise;

(1.5) if i is a Forget Vertex node, forgetting v, with a single child i_1:

$$\boldsymbol{dp}[i,s] = \min_{s' \in S_{B_{i_1}} \wedge s' \doteq s} \boldsymbol{dp}[i_1, s'].$$

Recall that $\doteq$ denotes compatibility.

Step 2: Computing the Output. The algorithm computes the output, i.e. the optimal weight of a p-partitioning, using the values stored at $\boldsymbol{dp}$ variables. If r is the root node of T, then the algorithm outputs the following value: $\min_{s \in S_{B_r}} \boldsymbol{dp}[r,s]$.

This concludes our algorithm. A simple pseudocode of our approach is provided in Algorithm 5.1. We now prove the correctness of our algorithm.

Lemma 5.2. *Algorithm 5.1 correctly computes the total weight of cross edges in a Minimum-weight p-partitioning.*

Proof. We prove this lemma in two steps. First, we show that the invariant defined above holds after computing $\boldsymbol{dp}[i,s]$ assuming that it was satisfied for all $\boldsymbol{dp}$ variables in the children of i (Correctness of Step 1). Then, assuming that the invariant holds for $\boldsymbol{dp}$ variables at the root, we show that the output is the total weight of an optimal p-partitioning (Correctness of Step 2).

Intuitively, the invariant says that if we only consider the graph G_i, i.e. the part of G that was introduced in the subtree of T rooted at i, and those p-partitionings of G_i that

Algorithm 5.1: Computing the total weight of cross edges in an optimal p-partitioning

```
function Main ($G, T, \langle B_i \rangle, w, p$);
Input  : A graph $G = (V, E)$, an edge-nice tree-decomposition $(T, \langle B_i \rangle)$ of $G$, a
         weight function $w : E \to \mathbb{N}$ and a positive integer $p$.
Output: Total weight of cross-edges in an optimal $p$-partitioning of $G$ wrt $w$.
initialize $dp[,] \leftarrow +\infty$;
$r \leftarrow T.root$;
compute_dp($r$);
return $\min_{s \in S_{B_r}} dp[r, s]$;

function compute_dp ($i$);
Input  : A node $i$ of $T$
Result : Fills in $dp[i, s]$ for all $s \in S_{B_i}$
forall $i' \in i.children$ do
    compute_dp($i'$);
if $i$ is a leaf then
    $dp[i, s_\emptyset] \leftarrow 0$;
else if $i$ is a join node then
    $i_1 \leftarrow i.children[1]$;
    $i_2 \leftarrow i.children[2]$;
    forall $s = (\varphi, sz) \in S_{B_i}$ do
        $dp[i, s] \leftarrow \min_{sz_1 + sz_2 \equiv sz} dp[i_1, (\varphi, sz_1)] + dp[i_2, (\varphi, sz_2)]$;
else if $i$ is an introduce vertex node then
    $i_1 \leftarrow i.children[1]$;
    forall $s = (\varphi, sz) \in S_{B_i}$ do
        $dp[i, s] \leftarrow dp[i_1, (\varphi_{|B_{i_1}}, sz_{|B_{i_1}})]$;
else if $i$ is an introduce edge node, introducing $e = (u, v)$ then
    $i_1 \leftarrow i.children[1]$;
    forall $s = (\varphi, sz) \in S_{B_i}$ do
        $dp[i, s] \leftarrow dp[i_1, s]$;
        if $[u]_\varphi \neq [v]_\varphi$ then
            $dp[i, s] \leftarrow dp[i, s] + w(e)$
else if $i$ is a forget vertex node, forgetting $v$ then
    $i_1 \leftarrow i.children[1]$;
    forall $s = (\varphi, sz) \in S_{B_i}$ do
        $dp[i, s] \leftarrow +\infty$;
        for $j \leftarrow 0$ to $p - 1$ do
            $\varphi' = \varphi \cup \{\{v\}\}$;
            $sz' = sz \cup \{(\{v\}, j)\}$;
            $dp[i, s] = \min\{dp[i, s], dp[i_1, (\varphi', sz')]\}$;
        forall $Y \in \varphi$ do
            if $|Y| < p \quad \wedge \quad sz(Y) \geq 1$ then
                $\varphi' = \varphi \cup \{Y \cup \{v\}\} \setminus \{Y\}$;
                $sz' = sz \cup \{(Y \cup \{v\}, sz(Y) - 1)\} \setminus \{(Y, sz(Y)\}$;
                $dp[i, s] = \min\{dp[i, s], dp[i_1, (\varphi', sz')]\}$;
```

realize the state s, then $\boldsymbol{dp}[i, s]$ holds the minimum total weight of cross edges among these p-partitionings.

Correctness of Step 1. As in the algorithm, we break this part into several cases:

(1.1) *Computations at Leaves.* The node i is a leaf in T, hence G_i is the empty graph and B_i is the empty set. Therefore, S_{B_i} contains a single trivial state $s_\emptyset$ and we have $\boldsymbol{dp}[i, s_\emptyset] = 0$ because the total weight of cross edges in an empty graph is zero.

(1.2) *Computations at Join Nodes.* The node i is a join node with children i_1 and i_2. We want to compute $\boldsymbol{dp}[i, s]$ where $s = (\varphi, sz)$. Therefore, we only consider those p-partitionings that realize s. Given that $B_i = B_{i_1} = B_{i_2}$, φ imposes itself on both B_{i_1} and B_{i_2}. However, each partition in φ must be extended by a number of vertices as defined by sz. These vertices must come from either G_{i_1} or G_{i_2} and must not already be present in B_i. According to the separation lemma (Lemma 2.1), the only vertices that are in both G_{i_1} and G_{i_2} are precisely those of B_i. Hence, each new vertex comes either from G_{i_1} or G_{i_2} but not from both. Therefore, we should minimize our total cross edge weights wrt $\boldsymbol{dp}$ variables of the form $\boldsymbol{dp}[i_1, (\varphi, sz_1)]$ and $\boldsymbol{dp}[i_2, (\varphi, sz_2)]$ where $sz_1 + sz_2 \equiv sz$. The function sz_1 defines the number of vertices that should be added from $G_{i_1} - B_i$ to each partition of φ and sz_2 does the same for $G_{i_2} - B_i$. Formally, if we let $w(\varphi)$ be the total weight of cross edges caused by φ in $G_{i_1} \cap G_{i_2} = G_{i_1}[B_i] \cap G_{i_2}[B_i]$, then we should let:

$$\boldsymbol{dp}[i, s] = \boldsymbol{dp}[i, (\varphi, sz)] = \min_{sz_1 + sz_2 \equiv sz} \boldsymbol{dp}[i_1, (\varphi, sz_1)] + \boldsymbol{dp}[i_2, (\varphi, sz_2)] - w(\varphi).$$

The reason we are subtracting $w(\varphi)$ at the end is that the weights of its corresponding edges are taken into account twice, i.e. once in each of $\boldsymbol{dp}[i_1, (\varphi, sz_1)]$ and $\boldsymbol{dp}[i_2, (\varphi, sz_2)]$.

We now show it is always the case that $w(\varphi) = 0$. If an edge contributes to $w(\varphi)$, then it must be present in both G_{i_1} and G_{i_2}. However, by property (3) of an edge-nice tree-decomposition, each edge is introduced exactly once. Hence, G_{i_1} and G_{i_2} do not share any edges and $w(\varphi) = 0$. Therefore, by setting:

$$\boldsymbol{dp}[i, s] = \boldsymbol{dp}[i, (\varphi, sz)] = \min_{sz_1 + sz_2 \equiv sz} \boldsymbol{dp}[i_1, (\varphi, sz_1)] + \boldsymbol{dp}[i_2, (\varphi, sz_2)],$$

we satisfy the invariant.

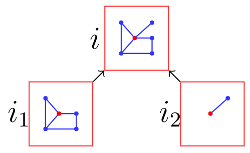


Figure 5.9: In a join node i, G_{i_1} and G_{i_2} do not share any edges and their shared vertices are in B_i.

(1.3) *Computations at Introduce Vertex Nodes.* The node i is an introduce vertex node. So, it has a single child i_1 and $B_i = B_{i_1} \cup \{v\}$ for some $v \notin B_{i_1}$. We know that the vertex v cannot possibly appear in G_{i_1} because every vertex appears in a connected subtree of T and $v \notin B_{i_1}$. Hence, G_i is obtained by adding v as an isolated vertex to G_{i_1}. Again, we want to compute $\boldsymbol{dp}[i, s]$ and should hence only consider the p-partitionings that realize s. Given that $B_{i_1} \subset B_i$, s imposes a unique compatible state on B_{i_1}. Moreover, G_i has no new edges in comparison with G_{i_1}, so the total weight of cross edges should only be computed in G_{i_1}. Hence, we let

$$\boldsymbol{dp}[i, s] = \boldsymbol{dp}[i, (\varphi, sz)] = \boldsymbol{dp}[i_1, (\varphi_{|B_{i_1}}, sz_{|B_{i_1}})].$$

Intuitively, this is equivalent to removing v from its partition and then computing the $\boldsymbol{dp}$ in i_1.

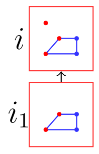


Figure 5.10: In an introduce vertex node i, the newly introduced vertex is isolated and there are no new edges.

(1.4) *Computations at Introduce Edge Nodes.* The node i has a single child i_1 and $B_i = B_{i_1}$. Moreover, the only difference between G_i and G_{i_1} is in a single edge e. When computing $\boldsymbol{dp}[i, s]$, the state s forces itself on $B_{i_1} = B_i$. Hence we should let:

$$\boldsymbol{dp}[i, s] = \boldsymbol{dp}[i_1, s] + w(e, s)$$

where $w(e, s)$ is the contribution of the edge e to the total weight of cross edges in s. It is zero if the two sides of e are put in the same partition set by s and is equal to $w(e)$ otherwise.

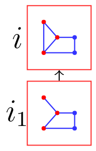


Figure 5.11: A new edge is introduced in the node i. The states are only dependent on vertices and hence are the same over B_i and B_{i_1}. However, we have to account for the weight of the new edge.

(1.5) *Computations at Forget Vertex Nodes.* In this case the node i has a single child i_1 and $B_i = B_{i_1} \setminus \{v\}$ for some $v \in B_{i_1}$. However, $G_i = G_{i_1}$. Hence, when computing $\boldsymbol{dp}[i, s]$, it is sufficient to take the minimum among the values of $\boldsymbol{dp}$ variables of all states s' over B_{i_1} that are compatible with s. More precisely, we let

$$\boldsymbol{dp}[i, s] = \min_{s' \in S_{B_{i_1}} \wedge s' \doteq s} \boldsymbol{dp}[i_1, s'].$$

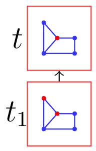


Figure 5.12: When a vertex v is forgotten by i, we have $G_i = G_{i_1}$, but $B_i = B_{i_1} \setminus \{v\}$.

Correctness of Step 2. Given that r is the root node of T, we have $G_r = G$. Since every p-partitioning of G realizes some state over B_r, it follows that the optimal weight of a p-partitioning is $\min_{s \in S_{B_r}} \boldsymbol{dp}[r, s]$. This concludes the proof.

□

Remark 5.3. *Algorithm 5.1 computes the total weight of cross edges in a Minimum-weight p-partitioning. As is common with dynamic programming algorithms, an optimal p-partitioning itself can be obtained by keeping track of the choices made during the computation of* `dp` *variables, i.e. keeping track of the cases that led to the minimal values in each computation.*

We now establish the complexity of our approach and present the main theorem of this section.

Number of States. For a fixed p, let C_k^p denote the number of different possible states over a set of size k, i.e. $C_k^p := |S_{\{1,2,\ldots,k\}}|$. We write C_k instead of C_k^p when p can be inferred from the context. We now establish bounds on the value of C_k. Note that these bounds only depend on p and k.

Lemma 5.3. $C_k \leq \frac{(p+k-1)!}{(p-1)!} = O\left(e^{p-k} \cdot k^{k+0.5} \cdot \left(\frac{p+k}{p-1}\right)^{p-1}\right)$.

Proof. Obviously, $C_1 = p$. We prove that $C_k \leq (p+k-1) \cdot C_{k-1}$ and the desired inequality follows by a simple induction.

Consider a state $s = (\varphi, sz)$ over $\{1, 2, \ldots, k\}$. Either k is in a singleton partition in s or it is put together with some other elements of $\{1, 2, \ldots, k-1\}$. In the former case, removing k leads to a state s' over $\{1, 2, \ldots, k-1\}$ that is compatible with s. In the latter, removing k and incrementing $sz([k]_\varphi)$ leads to a similarly compatible s'. Therefore, each state s over $\{1, 2, \ldots, k\}$ can be obtained by taking a state s' over $\{1, 2, \ldots, k-1\}$ and adding k to it either (i) as a separate partition with any sz value, or (ii) inside another partition that has an sz value of at least 1 and decrementing its sz value. Given a state s', there are p ways of doing (i), corresponding to the different values that can be assigned to $sz(\{k\})$, and at most $k-1$ ways of doing (ii), because there are at most $k-1$ partitions in s'. Hence,

$$C_k \leq (p+k-1) \cdot C_{k-1} \leq (p+k-1) \cdot \frac{(p+k-2)!}{(p-1)!} = \frac{(p+k-1)!}{(p-1)!}.$$

For the last part, we have $\frac{(p+k-1)!}{(p-1)!} = \binom{p+k-1}{p-1} \cdot k!$. It is well-known that $\binom{n}{r} \leq \left(\frac{e \cdot n}{r}\right)^r$ for all $n \geq r > 0$ and hence $\binom{p+k-1}{p-1} \leq \left(\frac{e \cdot (p+k-1)}{p-1}\right)^{p-1}$. By Stirling's approximation we have $k! \sim \sqrt{2\pi k} \cdot \left(\frac{k}{e}\right)^k$. Combining the two and ignoring the constants, we get the desired result: $\frac{(p+k-1)!}{(p-1)!} = O\left(e^{p-k} \cdot k^{k+0.5} \cdot \left(\frac{p+k}{p-1}\right)^{p-1}\right)$. □

Lemma 5.4. $C_k \leq \left(\frac{0.792 \cdot p \cdot k}{\ln(k+1)}\right)^k = O\left((0.792 \cdot p \cdot k)^k\right)$.

Proof. Let B_k be the k-th Bell number, i.e. the number of different partitions of $\{1, 2, \ldots, k\}$. Given a partition φ, we can form the enlargement function sz in at most p^k ways, i.e. there are at most k partitions and we have at most p choices for the sz value of each partition. Hence, $C_k \leq p^k \cdot B_k$. In [Berend and Tassa, 2010], it is established that $B_k \leq \left(\frac{0.792 \cdot k}{\ln(k+1)}\right)^k$. The desired result follows. □

Theorem 5.2. *Given a Data Packing instance $I = (n, 1, p, R)$ as input, where n is the number of distinct data elements, p is the packing factor, R is the reference sequence with a length of N and the cache has unit size, the Data Packing problem, i.e. finding the minimal number of cache misses, can be solved in linear time, i.e. in time $O(N + n \cdot t^2 \cdot C_t \cdot p^t)$, when the underlying access graph G_R has treewidth $t - 1$.*

Proof. Given a Data Packing instance $I = (n, 1, p, R)$, we first apply the reduction of Lemma 5.1 which takes $O(N)$. We then use Algorithm 5.1 to solve the resulting Minimum-weight p-partitioning problem. The correctness of this algorithm was established in Lemma 5.2. The only remaining part is to find the runtime of Algorithm 5.1. Note that the time spent for computing edge-nice tree decompositions is dominated by the rest of our runtime.

The algorithm needs values of $\boldsymbol{dp}$ variables for all nodes of the tree decomposition which are at most $O(n \cdot t)$. We obtain upper-bounds for the runtime of our algorithm on each type of node:

- *Leaves.* There is a single state at each leaf and its $\boldsymbol{dp}$ is zero. Hence we spend $O(1)$ at each leaf.

- *Join Nodes.* At a join node i, there are at most C_t states and for each state $s = (\varphi, sz)$ we have to look into the states corresponding to every possible size enlargement function

$sz_1 \leq sz$. As in the proof of Lemma 5.4, there are at most p^t such functions. Creating each corresponding state takes $O(t)$. Hence, we spend $O(t \cdot C_t \cdot p^t)$ at each join node.

- *Introduce Vertex Nodes.* At a node i, there are C_t states and we spend $O(t)$ computing the unique corresponding state over B_{i_1}. Thus, each introduce vertex node takes $O(t \cdot C_t)$.
- *Introduce Edge Nodes.* This case is similar to the previous one and takes $O(t \cdot C_t)$.
- *Forget Vertex Nodes.* At a node i, there are C_t states and for each of them we have to look into all its compatible states over B_{i_1}. Note that such compatible states can be obtained either by putting the vertex v in its own partition set, which can have any size between 1 and p, or by adding it to the partition set of another vertex in B_i. Hence, there are at most $p+t$ such states and the total processing time of a forget vertex node is $O(t \cdot (p+t) \cdot C_t)$.

Note that the runtime for join nodes dominates the rest. Given that there are $O(n \cdot t)$ nodes in total, the whole computation takes $O(n \cdot t^2 \cdot C_t \cdot p^t)$ time. Finally, the algorithm spends $O(C_t)$ time computing the final result using the ***dp*** values at the root. □

Corollary 5.1. *We have the following upperbounds on the runtime of Algorithm 5.1:*

$$O\left(N + n \cdot t^{t+2.5} \cdot p^t \cdot e^{p-t} \cdot \left(\frac{p+t}{p-1}\right)^{p-1}\right),$$

and

$$O\left(N + n \cdot t^{t+2} \cdot p^{2 \cdot t} \cdot (0.792)^t\right).$$

Proof. The bounds above can be obtained by applying Lemmas 5.3 and 5.4 to C_t in Theorem 5.2. □

Remark 5.4. *By exploiting treewidth, we provided a* linear-time *algorithm for finding the exact solution to the Data Packing problem when $m = 1$. Note that in the general case, i.e. without considering parameterization by treewidth, this problem is NP-hard as mentioned in Theorem 5.1.*

Remark 5.5. *We assumed LRU as the replacement policy. However, given that the replacement policy does not matter when the cache has unit size (Remark 5.1), our algorithm is applicable to any replacement policy, including FIFO and OOP.*

5.4.2 Hardness of Data Packing on Trees

In this section, we provide a reduction from the general problem of Data Packing to the special case where the access graph is a tree, i.e. has treewidth 1. This reduction leads to hardness results that enhance those of [Lavaee, 2016] by showing that the problem remains hard even on trees. This indicates that although considering constant treewidth access graphs led to efficient algorithms for the case of $m = 1$, constant treewidth access graphs alone are not sufficient for $m \geq 2$.

Theorem 5.3 (Hardness of Data Packing on Trees)**.** *Given a Data Packing instance* $I = (n, m, p, R)$*, we have the following hardness results:*

- Hardness of the Exact Problem. *For any* $m \geq 2$ *and any* $p \geq 3$*, Data Packing is NP-hard even if the underlying access graph* G_R *is a tree.*

- Hardness of Approximation. *Unless P=NP, for any* $m \geq 6, p \geq 2$ *and any constant* $\epsilon > 0$*, there is no polynomial approximation algorithm for the Data Packing problem with an approximation factor of* $O(N^{1-\epsilon})$ *even if the access graph* G_R *is a tree.*

Proof. We provide a linear-time reduction that transforms a Data Packing instance $I = (n, m, p, R)$ to another instance $I' = (n + (m+1)p, m+1, p, R')$ such that the access graph $G_{R'}$ is a tree. Both hardness results can then be obtained by applying this reduction to the hardness results of Theorem 5.1.

Given I, we introduce $(m+1) \cdot p$ new data elements $d_1, d_2, \ldots, d_{(m+1)\cdot p}$. Let X be the sequence

$$d_1, d_2, \ldots, d_{(m+1)\cdot p}, d_{(m+1)\cdot p-1}, \ldots, d_1.$$

We form the sequence R' as follows:

$$d_1, R[1], d_1, R[2], d_1, \ldots, d_1, R[N], d_1, \underbrace{X, X, \ldots, X}_{2\cdot N+m+2 \text{ times}},$$

i.e. we take R and add d_1 at its beginning, end and between every two elements of it, then we concatenate the result with $2\cdot N+m+2$ copies of X. We let $I' = (n+(m+1)\cdot p, m+1, p, R')$. Note that the cache in I' has one spot more than the cache of I.

By construction, $G_{R'}$ is a tree, because it consists of a path $d_1, \ldots, d_{(m+1)\cdot p}$ and every other vertex of the graph is only connected to d_1. We now show that the optimal number of cache misses in I' is exactly $m+1$ plus the optimal number of cache misses in I.

Let σ be an optimal data placement scheme for I', then σ must necessarily put the d_i's in exactly $m+1$ blocks, otherwise each X in the sequence R' will lead to at least one cache miss for a total of at least $2\cdot N+m+2$. On the other hand, putting the d_i's in $m+1$ blocks leads to at most $2\cdot N+m+1$ cache misses, even if all accesses before the X's are missed. In particular, σ does not put any element of R in the same block as d_1. Therefore, σ first leads to a cache miss on the first access to d_1, then keeps d_1 in the cache forever. Hence, σ fills one spot of the cache with the block of d_1 and has m spots left for scheduling R. Finally, σ loads the other m blocks that contain some d_i's but not d_1. Hence, the number of cache misses caused by σ is 1 (for the first d_1) plus the optimal number of misses in I plus m (for the X's). □

Remark 5.6. *As mentioned before, we are considering the LRU replacement policy in this paper. However, the reduction above works for the OOP replacement policy as well. Hence, the hardness results are established for both policies.*

5.5 Algorithms and Hardness Results based on Treewidth of Access Hypergraphs

In this section, we exploit constant treewidth of higher-order access hypergraphs for solving Data Packing. Section 5.5.1 extends our linear-time algorithm to every m, when the access hypergraph of order $q^* := (m-1)\cdot p+2$ has constant treewidth. As indicated by Theorem 5.1, this problem is hard to even approximate in the general case. In Section 5.5.2 we argue that q^* is the optimal order for exploiting treewidth in the sense that the problem remains NP-hard even if the access hypergraph of order $q^* - 1$ has constant treewidth. This also leads to a new hardness-of-approximation result.

5.5.1 Algorithm for Constant-treewidth Access Hypergraph

In this section, we extend the algorithm of Section 5.4.1 to any cache size m, provided that the hypergraph $G_R^{q^*}$ is of constant treewidth, where $q^* = (m-1)\cdot p+2$. Note that Theorem 5.3 implies such an extension cannot be made if we only consider constant-treewidth G_R. Since we are considering $m > 1$, the paging policy should now be taken into account. We assume that the paging policy is LRU.

Intuition on Cache Misses. The main intuition behind our algorithm is the following: given an instance $I = (n, m, p, R)$ and a data placement scheme σ for I, we can deduce whether an access $R[i]$ leads to a cache miss by looking at only the $(m-1)\cdot p+1 = q^* - 1$ previous accesses to *distinct* data elements. We now formalize this intuition.

Previous Access of a Block. Consider a Data Packing instance $I = (n, m, p, R)$, a data placement scheme σ for I and an access $R[i]$. Let $\beta := [R[i]]_\sigma$ be the block of σ containing $R[i]$. We define $\texttt{prev}_\sigma(i)$ as the index of the previous access to β or 0 if no such access exists, i.e. $\texttt{prev}_\sigma(i) := \max\{j < i \mid j = 0 \vee [R[j]]_\sigma = [R[i]]_\sigma\}$.

Lemma 5.5. *Given a data placement scheme σ for I, an access $R[i]$ leads to a cache miss if and only if $\texttt{prev}_\sigma(i) = 0$ or there are at least m distinct blocks of σ whose elements appear in the range $R[\texttt{prev}_\sigma(i)+1]\dots R[i-1]$.*

Proof. We are assuming LRU as the replacement policy and the cache starts empty. If $\texttt{prev}_\sigma(i) = 0$, then $R[i]$ is the first access to its block and will definitely lead to a cache miss. We now consider the case where $\texttt{prev}_\sigma(i) \neq 0$. Let $j := \texttt{prev}_\sigma(i)$ and assume that $\beta := [R[i]]_\sigma = [R[j]]_\sigma$ is the block containing $R[i]$ and $R[j]$. By definition, none of the elements $R[j+1], \dots, R[i-1]$ belong to β. If there are at most $m-1$ blocks between $R[j+1]$

and $R[i-1]$, then $R[i]$ cannot lead to a cache miss. This is because right after the access $R[j]$, the block β is present in the cache and is the most recently used block of the cache. Hence, in order for it to be evicted, at least m other blocks must be accessed. On the other hand, if there are at least m blocks between $R[j+1]$ and $R[i-1]$, then all of these blocks will be loaded into the cache and hence β will be evicted before the access $R[i]$ leading to an eventual cache miss on $R[i]$. □

Corollary 5.2. *Given a data placement scheme σ, an access $R[i]$ and the q^*-1 distinct elements that were accessed before $R[i]$ (or all of the previous distinct elements if there is less than q^*-1 of them) in the order of their last access time, one can deduce whether $R[i]$ leads to a cache miss.*

Proof. If $R[\texttt{prev}_\sigma(i)]$ is one of these previous elements, then we can simply check whether at least m different blocks appear between $R[\texttt{prev}_\sigma(i)]$ and $R[i]$. Otherwise, either $R[i]$ is the first access to its block or all the q^*-1 elements are appearing between $R[\texttt{prev}_\sigma(i)]$ and $R[i]$. In the first case $R[i]$ leads to a cache miss. In the second case, by pigeonhole principle, there are at least m blocks between the two elements $R[\texttt{prev}_\sigma(i)]$ and $R[i]$ and hence there is a cache miss at $R[i]$. □

Remark 5.7. *Note that the previous access to the block containing a data element $R[i]$ might be an access to $R[i]$ itself. Hence, $R[i]$ might itself appear in the q^*-1 distinct elements that were accessed before $R[i]$.*

As in Section 5.4.1, we are going to reduce Data Packing to a graph problem and then exploit treewidth to obtain a linear-time algorithm. Corollary 5.2 suggests that in order to detect cache misses, one only needs to consider the ordered access hypergraph of order $q^* = (m-1)\cdot p + 2$, i.e. $\hat{G}_R^{q^*}$. However, in order to address the corner case mentioned in Remark 5.7, we define an ordered hypergraph G by a slight change to the edges of $\hat{G}_R^{q^*}$ and then reduce Data Packing to a graph problem over G.

The Ordered Hypergraph G. We define the ordered hypergraph G as having the same vertices and edges as the ordered access hypergraph $\hat{G}_R^{q^*}$, except in the following case:

- Given an access $R[i]$ to a data element d, let $R[j]$ be the last access before $R[i]$ to the same data element d. If there are at most q^* distinct data elements accessed in the range $R[j+1] \dots R[i-1]$, then the edge e_i corresponding to $R[i]$, will also contain $R[j]$ (in its natural position according to the order of vertices in e_i).

Example 5.7. *Consider the access sequence $R = \langle d, c, a, b, c \rangle$ and let $m = p = 2$. Hence, we have $q^* = (m-1) \cdot p + 2 = 4$. In the graph $\hat{G}_R^{q^*}$ the edge corresponding to the second c is $e_5 = \langle d, a, b, c \rangle$. However, there are less than q^* distinct data elements appearing between the two accesses to c, i.e. there are only two such elements, namely, a and b. Hence, in G, the previous access to c appears in this edge as well. Therefore, in G, the edge e_5 is of the form $e_5 = \langle d, c, a, b, c \rangle$.*

The intuition behind the way G is defined comes from Corollary 5.2 and Remark 5.7. The idea is to have the edge e_i contain all the data necessary to decide whether a cache-miss will happen at the access $R[i]$. We now formalize this concept.

Missed Edges. Given an ordered hyperedge e_i of G and a data placement scheme σ, we can deduce whether a cache miss happens at $R[i]$ using Corollary 5.2, because the edge e_i contains an ordered list of at least $(m-1) \cdot p + 1 = q^* - 1$ distinct data elements that were accessed right before $R[i]$. We say that an ordered hyperedge e_i is missed in σ, if the corresponding $R[i]$ is a cache miss.

Identifying Missed Edges. Consider the data placement scheme σ as a p-partitioning of vertices of G. Based on Lemma 5.5 and Corollary 5.2, an ordered hyperedge $e_i = \langle v_1, \dots, v_l \rangle$ is missed iff the sequence of vertices $\langle v_l, v_{l-1}, \dots, v_1 \rangle$ in G visits at least m distinct partitions before getting back to the partition $[v_l]_\sigma$ or if it never comes back. A short pseudocode to determine whether a hyperedge is missed is provided in Algorithm 5.2. Note that this determination can be done in $O(m \cdot p)$ and only depends on the p-partitioning of $\{v_1, \dots, v_l\}$.

We now define our graph problem as follows:

Minimum-miss p-partitioning. Given a hypergraph $G = (V, E)$ with ordered hyperedges, partition V into sets of size at most p in a manner that minimizes the number of missed edges.

Algorithm 5.2: Checking if an ordered hyperedge is missed

```
1 function missed_edge ($e, \sigma$);
  Input  : An ordered hyperedge $e = \langle v_1, \dots, v_q \rangle$ and a $p$-partitioning $\sigma$
  Output: Whether $e$ is missed in $\sigma$
2 $\beta \leftarrow [v_q]_\sigma$;
3 $VisitedBlocks \leftarrow \emptyset$;
4 for $j \leftarrow q - 1$ downto 1 do
5     if $|VisitedBlocks| < m \wedge [v_j]_\sigma = \beta$ then
6         return false;
7     else
8         $VisitedBlocks \leftarrow VisitedBlocks \cup \{[v_j]_\sigma\}$;
9 return true;
```

As a direct result of the previous discussion, we have the following lemma:

Lemma 5.6. *The optimal number of cache misses in a Data Packing instance $I = (n, m, p, R)$ is equal to the optimal number of missed edges in a p-partitioning of G.*

Proof. Any data placement scheme σ for I is also a p-partitioning of G. As shown above, σ misses an edge e_i in G iff it causes a cache miss at $R[i]$ in I. Hence, the number of cache misses caused by σ in I is equal to the number of missed edges caused by σ in G. □

States over a Set of Vertices. We define states in the exact same manner as in Section 5.4.1, i.e. a state over a set A of vertices is a pair $s = (\varphi, sz)$ consisting of an equivalence relation φ and a size enlargement function sz. The concepts of realization and compatibility are also defined similarly.

The Algorithm. We now provide a linear-time algorithm for solving the Minimum-miss p-partitioning problem, assuming that the hypergraph G has constant treewidth. The algorithm is an extension of the one provided in Section 5.4.1. In the following, we let $(T, \langle B_i \rangle_{i \in V_T})$ be an optimal edge-nice tree decomposition of G. Our algorithm performs a bottom-up dynamic programming on T.

Step 0: Initialization. We define several variables at each node of the tree T. Concretely, for every node $i \in V_T$ and every state s over B_i, we define a variable $\boldsymbol{dp}[i, t]$, initially holding a value of $+\infty$.

Invariant. The most different aspect of our algorithm compared to Section 5.4.1 is the invariant. Formally, we require our algorithm to satisfy the following invariant for every dp variable right after the end of its computation:

$\mathit{dp}[i, s] :=$ The minimum number of missed edges over all p-partitionings of G_i that realize s.

Step 1: Computation of dp. The dp variables are computed in a bottom-up manner. Given a node $i \in V_T$ and a state $s = (\varphi, sz) \in S_{B_i}$, we show how $\mathit{dp}[i, s]$ is computed in terms of the dp variables at the children of i. This computation depends on the type of the node i.

(1.1) if i is a Leaf: $\mathit{dp}[i, s] = 0$;

(1.2) if i is a Join node with children i_1 and i_2:

$$\mathit{dp}[i, s] = \min_{sz_1 + sz_2 \equiv sz} \mathit{dp}[i_1, (\varphi, sz_1)] + \mathit{dp}[i_2, (\varphi, sz_2)];$$

(1.3) if i is an Introduce Vertex node, introducing v, with a single child i_1:

$$\mathit{dp}[i, s] = \mathit{dp}[i_1, (\varphi_{|B_{i_1}}, sz_{|B_{i_1}})];$$

(1.4) if i is an Introduce Edge node, introducing e, with a single child i_1:

$$\mathit{dp}[i, s] = \mathit{dp}[i_1, s] + \begin{cases} 1 & \text{missed\_edge}(e, \varphi) \\ 0 & \text{otherwise} \end{cases};$$

(1.5) if i is a Forget Vertex node, forgetting v, with a single child i_1:

$$\mathit{dp}[i, s] = \min_{s_1 \in S_{B_{i_1}} \wedge s_1 \doteq s} \mathit{dp}[i_1, s_1].$$

Recall that $\doteq$ denotes compatibility of states.

Step 2: Computing the Output. The algorithm computes the output, i.e. the minimum number of missed edges in a p-partitioning of G, using the values of $\boldsymbol{dp}$ variables at the root node r of T. Formally, the output is $\min_{s \in S_{B_r}} \boldsymbol{dp}[r, s]$.

This concludes our algorithm. While most of its computations are similar to Algorithm 5.1, the argument for correctness and its runtime are rather different. A pseudocode of the approach is given in Algorithm 5.3. We first prove the correctness of our approach and then establish its time complexity.

Lemma 5.7. *Algorithm 5.3 correctly computes the total number of missed edges in a Minimum-miss p-partitioning.*

Proof. Our proof heavily depends on the invariant defined above. Intuitively, the invariant says that $\boldsymbol{dp}[i, s]$ must be filled with the minimum number of edges that are missed in a p-partitiong realizing s, over the subgraph G_i of G, which consists of all the vertices and hyperedges that are introduced below i in T. We prove the lemma in two steps. First, we prove that the invariant is satisfied after computing $\boldsymbol{dp}[i, s]$, assuming that it were satisfied for all $\boldsymbol{dp}$ variables in the children of i (Correctness of Step 1). Then, we prove that assuming the invariant holds for $\boldsymbol{dp}$ variables at the root node r of T, the algorithm computes the right output (Correcntess of Step 2).

Correctness of Step 1. We break the proof into several cases:

(1.1) *Computations at Leaves.* The node i is a leaf in T. So G_i is the empty graph and hence there are no missed edges in G_i. Moreover, there is exactly one state over B_i, i.e. the trivial state $s_\emptyset$. Hence, we should let $\boldsymbol{dp}[i, s_\emptyset] = 0$.

(1.2) *Computations at Join Nodes.* A join node i has two children i_1 and i_2 with $B_i = B_{i_1} = B_{i_2}$. When computing the value of $\boldsymbol{dp}[i, s]$ for a state $s = (\varphi, sz)$, we only have to consider those states over B_{i_1} and B_{i_2} that are compatible with s. However, $B_{i_1} = B_{i_2} = B_i$, hence the partitioning φ is also imposed on B_{i_1} and B_{i_2}. The function sz specifies how many new vertices must be added to each partition of φ from G_{i_1} and G_{i_2}. Note that by the separation lemma (Lemma 2.1), the only vertices that belongs

Algorithm 5.3: Computing the number of missed edges in an optimal p-partitioning

```
function Main ($G, T, \langle B_i \rangle, p$);
Input  : A hypergraph $G = (V, E)$ with ordered hyperedges, an edge-nice
         tree-decomposition $(T, \langle B_i \rangle)$ of $G$ and a positive integer $p$.
Output: The minimum number of missed hyperedges in a $p$-partitioning of $G$
initialize $\mathit{dp}[,] \leftarrow +\infty$;
$r \leftarrow T.root$;
compute_dp($r$);
return $\min_{s \in S_{B_r}} \mathit{dp}[r, s]$;

function compute_dp ($i$);
Input  : A node $i$ of $T$
Result : Fills in $\mathit{dp}[i, s]$ for all $s \in S_{B_i}$
forall $i' \in i.children$ do
    compute_dp($i'$);
if $i$ is a leaf then
    $\mathit{dp}[i, s_\emptyset] \leftarrow 0$;
else if $i$ is a join node then
    $i_1 \leftarrow i.children[1]$;
    $i_2 \leftarrow i.children[2]$;
    forall $s = (\varphi, sz) \in S_{B_i}$ do
        $\mathit{dp}[i, s] \leftarrow \min_{sz_1 + sz_2 \equiv sz} \mathit{dp}[i_1, (\varphi, sz_1)] + \mathit{dp}[i_2, (\varphi, sz_2)]$;
else if $i$ is an introduce vertex node then
    $i_1 \leftarrow i.children[1]$;
    forall $s = (\varphi, sz) \in S_{B_i}$ do
        $\mathit{dp}[i, s] \leftarrow \mathit{dp}[i_1, (\varphi_{|B_{i_1}}, sz_{|B_{i_1}})]$;
else if $i$ is an introduce edge node, introducing $e = (u, v)$ then
    $i_1 \leftarrow i.children[1]$;
    forall $s = (\varphi, sz) \in S_{B_i}$ do
        $\mathit{dp}[i, s] \leftarrow \mathit{dp}[i_1, s]$;
        if $missed\_edge(e, \varphi)$ then
            $\mathit{dp}[i, s] \leftarrow \mathit{dp}[i, s] + 1$
else if $i$ is a forget vertex node, forgetting $v$ then
    $i_1 \leftarrow i.children[1]$;
    forall $s = (\varphi, sz) \in S_{B_i}$ do
        $\mathit{dp}[i, s] \leftarrow +\infty$;
        for $j \leftarrow 0$ to $p - 1$ do
            $\varphi' = \varphi \cup \{\{v\}\}$;
            $sz' = sz \cup \{(\{v\}, j)\}$;
            $\mathit{dp}[i, s] = \min\{\mathit{dp}[i, s], \mathit{dp}[i_1, (\varphi', sz')]\}$;
        forall $Y \in \varphi$ do
            if $|Y| < p \quad \wedge \quad sz(Y) \geq 1$ then
                $\varphi' = \varphi \cup \{Y \cup \{v\}\} \setminus \{Y\}$;
                $sz' = sz \cup \{(Y \cup \{v\}, sz(Y) - 1)\} \setminus \{(Y, sz(Y)\}$;
                $\mathit{dp}[i, s] = \min\{\mathit{dp}[i, s], \mathit{dp}[i_1, (\varphi', sz')]\}$;
```

to both G_{i_1} and G_{i_2} are already included in B_i, hence no new vertex can be in both. Therefore, we have to look into $\boldsymbol{dp}$ variables of the form $\boldsymbol{dp}[i_1, (\varphi, sz_1)]$, $\boldsymbol{dp}[i_2, (\varphi, sz_2)]$ where $sz_1 + sz_2 \equiv sz$. Concretely, we should let:

$$\boldsymbol{dp}[i, s] = \boldsymbol{dp}[i, (\varphi, sz)] = \min_{sz_1 + sz_2 \equiv sz} \boldsymbol{dp}[i_1, (\varphi, sz_1)] + \boldsymbol{dp}[i_2, (\varphi, sz_2)].$$

Note that the two graphs G_{i_1} and G_{i_2} do not share any edges as argued in Lemma 5.2.

(1.3) *Computations at Introduce Vertex Nodes.* In this case, i is a node, with a single child i_1, and introduces the vertex v. Then $v \notin G_{i_1}$ and $G_i = G_{i_1} \cup \{v\}$, i.e. G_i is obtained by adding v to G_{i_1} as an isolated vertex. Given that G_i has no new edges in comparison with G_{i_1}, it follows that the missed edges in G_i are precisely those that were missed in G_{i_1}. Also, $B_i = B_{i_1} \cup \{v\}$ and so given a state $s = (\varphi, sz)$ over B_i, there is only one compatible state over B_{i_1}, i.e. $s_1 = (\varphi_{|B_{i_1}}, sz_{|B_{i_1}})$. Therefore, we must let $\boldsymbol{dp}[i, s] = \boldsymbol{dp}[i_1, s_1]$.

(1.4) *Computations at Introduce Edge Nodes.* The node i has one child i_1, $B_i = B_{i_1}$ and $G_i = G_{i_1} \cup \{e\}$, where e is the newly introduced hyperedge. Note that, by property (2) of edge-nice tree decompositions, all vertices of e must appear in B_i. So φ gives us enough information to know whether e is a missed edge. Also, given that $B_{i_1} = B_i$, the state s forces itself on B_{i_1} and therefore, letting

$$\boldsymbol{dp}[i, s] = \boldsymbol{dp}[i_1, s] + \begin{cases} 1 & \text{missed\_edge}(e, \varphi) \\ 0 & \text{otherwise} \end{cases}$$

preserves the invariant.

(1.5) *Computations at Forget Vertex Nodes.* This case is handled in the exact same manner as in Section 5.4.1. Given that $G_i = G_{i_1}$ and $B_i \subseteq B_{i_1}$, the value of $\boldsymbol{dp}[i, s]$ should be set to the minimum value of $\boldsymbol{dp}[i_1, s_1]$ over all states s_1 that are compatible with s. Formally,

$$\boldsymbol{dp}[i, s] = \min_{s_1 \in S_{B_{i_1}} \wedge s_1 \doteq s} \boldsymbol{dp}[i_1, s_1].$$

Correctness of Step 2. Let r be the root node of T, then $G_r = G$ and every p-partitioning of G realizes exactly one state over B_r. Hence, the minimum number of missed edges in the entire graph G is $\min_{s \in S_{B_r}} \boldsymbol{dp}[r, s]$.

□

Remark 5.8. *Algorithm 5.3 computes the optimal number of missed edges in a p-partitioning of G. As is common in dynamic programming approaches, an optimal p-partitioning itself can be obtained by keeping track of the choices that led to minimum values during the computation of* $\boldsymbol{dp}$ *variables.*

We conclude this section by establishing the complexity of Algorithm 5.3.

Theorem 5.4. *Given a Data Packing instance* $I = (n, m, p, R)$ *as input, where* n *is the number of distinct data items,* p *is the packing factor,* R *is the reference sequence with a length of* N *and the cache has a capacity of* m *blocks, the Data Packing problem, i.e. finding the minimal number of cache misses, can be solved in linear time, i.e. in time* $O(n \cdot t^2 \cdot C_t \cdot p^t + N \cdot C_t \cdot (t + m \cdot p))$*, when the underlying access hypergraph* $G_R^{q^*}$ *has treewidth* $t - 1$.

Proof. Creating the ordered hypergraph G and the reduction from Data Packing to Minimum-miss p-partitioning using Lemma 5.6 take linear time, i.e. $O(N \cdot m \cdot p)$. Note that G is obtained by ordering the vertices of $G_R^{q^*}$ and then adding duplicated vertices to some of the edges, hence $\mathrm{tw}(G) = \mathrm{tw}(G_R^{q^*})$. As before, the optimal tree decomposition $(T, \langle B_i \rangle)$ can be computed in linear time. Since there are N hyperedges in G, the tree T will have $O(n \cdot k + N)$ nodes, where N of them are introduce edge nodes and $O(n \cdot t)$ of them are of the other types.

The times spent at leaves, join nodes, introduce vertex nodes and forget vertex nodes are exactly the same as those established in Theorem 5.2. In an introduce edge node, the algorithm has to compute C_t different $\boldsymbol{dp}$ values, each taking time $O(t + m \cdot p)$ due to the call to the missed_edge subprocedure. Hence, processing each introduce edge node takes $O(C_t \cdot (t + m \cdot p))$. Therefore, the total time spent on computing $\boldsymbol{dp}$ values is $O(n \cdot t^2 \cdot C_t \cdot p^t \cdot + N \cdot C_t \cdot (t + m \cdot p))$. Finally, it takes $O(C_t)$ time to compute the final answer using $\boldsymbol{dp}$ variables at the root node. □

Remark 5.9. *The runtime above is linear in n and N, given that C_t is bounded by a function of p and t. Hence, by exploiting the treewidth of G, we were able to obtain an exact linear-time algorithm for Data Packing. Note that by Theorem 5.1, the general problem, i.e. without parameterization by treewidth, is hard to even approximate.*

5.5.2 Hardness of Data Packing on Constant-treewidth Access Hypergraphs

In Section 5.4.2, we showed that Data Packing is hard even if the access graph G_R is a tree, i.e. even if G_R^2 has treewidth 1. Section 5.5.1 provided a linear-time algorithm for Data Packing when $G_R^{q^*}$ has constant treewidth. This naturally leads to the question whether $q^* = (m-1) \cdot p + 2$ is the optimal order for exploiting treewidth. Note that this is a well-posed problem because for every i, the primal graph of G_R^i is a subgraph of the primal graph of G_R^{i+1} and hence $\text{tw}(G_R^i) \leq \text{tw}(G_R^{i+1})$. Formally, the question is whether there exists a polynomial algorithm for Data Packing assuming that the hypergraph $G_R^{q^*-1}$ has constant treewidth. In this section, we show that this problem is NP-hard and hence, unless P=NP, there is no such algorithm and q^* is the optimal order. We then show that for a slightly smaller order, i.e. $q^* - 4 \cdot p - 1 = (m-5) \cdot p + 1$, the problem becomes hard to approximate.

Theorem 5.5 (Hardness of Data Packing in Constant Treewidth)**.** *Given a Data Packing instance $I = (n, m, p, R)$, for any cache size $m \geq 2$ and any packing factor $p \geq 3$, Data Packing is NP-hard even if the underlying access hypergraph $G_R^{q^*-1}$ has constant treewidth.*

Proof. By Theorem 5.1, we know that Data Packing is NP-hard for any $p \geq 3$ and $m = 1$. We use this problem to obtain our reduction. Formally, for every m, we provide a linear-time reduction that transforms the Data Packing instance $I = (n, 1, p, R)$ to a new instance $I' = (n', m, p, R')$ such that the access hypergraph $G_{R'}^{q^*-1}$ is of constant treewidth.

Given a positive integer m and an instance I as above, we introduce $(m+1) \cdot p$ new data elements $d_1, d_2, \ldots, d_{(m+1)\cdot p}$. We then define three sequences X, Y and Z as follows:

$$X := d_1, d_2, \ldots, d_{(m-1)\cdot p}, \qquad Y := d_{(m-1)\cdot p+1}, d_{(m-1)\cdot p+2}, \ldots, d_{m\cdot p}, \qquad Z := d_{m\cdot p+1}, d_{m\cdot p+2}, \ldots, d_{(m+1)\cdot p},$$

and construct the reference sequence R' as:

$$R' := X, R[1], X, R[2], X, \ldots, X, R[N], \underbrace{X, Y, X, Y, \ldots, X, Y}_{a \text{ times}}, \underbrace{X, Z, X, Z, \ldots, X, Z}_{b \text{ times}},$$

i.e. R' is obtained by adding X before every element of R and then appending the result with a copies of X, Y and b copies of X, Z. The instance I is then reduced to $I' = (n + (m+1) \cdot p, m, p, R')$.

Our goal is to set the right values for a and b in a way that forces any optimal data packing scheme σ to put X in exactly $m-1$ blocks. We set $a := N \cdot (m-1) \cdot p + N + 2 \cdot m + 1$ and $b := N \cdot (m-1) \cdot p + N + a \cdot m \cdot p + m + 1$. Using these parameters, every optimal data packing scheme σ has to put $X \cup Z$ in exactly m blocks. This is because using more than m blocks for them leads to at least b misses in the last part of the sequence R', while putting them in exactly m blocks can cause a maximum of $N \cdot (m-1) \cdot p + N + a \cdot m \cdot p + m = b - 1$ misses overall, i.e. even if every access up until the end of the last Y leads to a miss. Given that σ puts $X \cup Z$ in exactly m blocks, we also infer that σ causes at most m misses over the b repetitions of X, Z.

We now prove that σ has to put $X \cup Y$ in exactly m blocks. The reasoning is similar. If σ puts $X \cup Y$ in more than m blocks, it causes at least a cache misses, but if it puts them in exactly m blocks the number of cache misses is at most $N \cdot (m-1) \cdot p + N + 2 \cdot m = a - 1$, i.e. even if every access up until $R[N]$ is missed and σ misses m times over the repetitions of X, Z. Given that $X \cup Z$ and $X \cup Y$ are both put into m blocks, it follows that σ puts X in exactly $m - 1$ blocks.

We claim that the optimal number of cache misses in I' is $m+1$ plus the optimal number of cache misses in I. To see this, we track the behavior of σ over the access sequence R'. First, the $m-1$ blocks of X are loaded into the cache causing $m-1$ cache misses. These remain in the cache forever because of the way X is repeated in R'. Therefore, σ has filled $m-1$ spots of the cache with X and has only 1 spot for handling the $R[i]$'s. This leads to exactly as many cache misses as in the optimal solution to I. Finally, σ causes two more cache misses, one on the first access to Y and the other one on the first access to Z. Therefore, the optimal number of cache misses in I' is equal to the optimal number of cache misses in I plus $m+1$. The reduction is now complete.

It remains to show that $G_{R'}^{q^*-1} = G_{R'}^{(m-1)\cdot p+1}$ has constant treewidth. Figure 5.13 shows a tree decomposition of this graph with width $m\cdot p-1$. Therefore, $\mathrm{tw}(G_{R'}^{q^*-1}) \leq m\cdot p-1 = O(1)$.

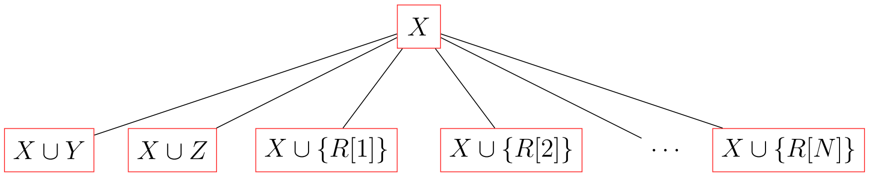


Figure 5.13: A tree decomposition of $G_{R'}^{q^*-1}$ with constant width $m \cdot p - 1$.

□

We now turn to the hardness of approximation. We provide a reduction that follows the same intuition as in the previous theorem.

Theorem 5.6 (Hardness of Approximating Data Packing in Constant Treewidth)**.** *Given a Data Packing instance $I = (n, m, p, R)$, for any cache size $m \geq 6$, any packing factor $p \geq 2$ and any constant $\epsilon > 0$, unless P=NP, Data Packing cannot be approximated within a factor of $O(N^{1-\epsilon})$ even if the underlying access hypergraph $G_R^{q^*-4\cdot p-1} = G_R^{(m-5)\cdot p+1}$ has constant treewidth. Here, N is the length of the reference sequence R.*

Proof. We know from Theorem 5.1 that for $m = 5$, and $p \geq 2$, it is hard to approximate Data Packing within a factor of $O(N^{1-\epsilon})$. We reduce this problem to Data Packing on a constant-treewidth $G_R^{(m-5)\cdot p+1}$. Formally, for every $m \geq 6$, we provide a linear-time reduction from every instance $I = (n, 5, p, R)$ to an instance $I' = (n', m, p, R')$ such that $G_{R'}^{(m-5)\cdot p+1}$ has constant treewidth.

The reduction and the argument for its correctness are similar to those of Theorem 5.5. Given an instance I as above, we introduce $(m+5)\cdot p$ new data elements $d_1, d_2, \ldots, d_{(m+5)\cdot p}$. We then define the following four sequences in a manner similar to Theorem 5.5:

$$X := d_1, d_2, \ldots, d_{(m-5)\cdot p}, \quad Y := d_{(m-5)\cdot p+1}, d_{(m-5)\cdot p+2}, \ldots, d_{m\cdot p}, \quad Z := d_{m\cdot p+1}, d_{m\cdot p+2}, \ldots, d_{(m+5)\cdot p};$$

$$R' := X, R[1], X, R[2], \ldots, X, R[N], \underbrace{X, Y, X, Y, \ldots, X, Y}_{a \text{ times}}, \underbrace{X, Z, X, Z, \ldots, X, Z}_{b \text{ times}},$$

where $a = N \cdot (m-5) \cdot p + N + 2 \cdot m + 1$ and $b = N \cdot (m-5) \cdot p + N + a \cdot m \cdot p + m + 1$. The reduction is then from $I = (n, 5, p, R)$ to $I' = (n + (m+5) \cdot p, m, p, R')$. It is straightforward to check that every optimal data placement scheme σ has to put each of $X \cup Y$ and $X \cup Z$ in exactly m blocks. Hence, it has to put X in exactly $m-5$ blocks. Therefore, the optimal number of cache misses in I' is $m-5$ (for loading the first X) plus the optimal number of cache misses in I plus 10 (5 misses for loading the first Y and 5 for the first Z). Finally, the same tree decomposition as in Figure 5.13, shows that $\text{tw}(G_{R'}^{(m-5)\cdot p+1}) \leq m \cdot p - 1 = O(1)$. $\square$

5.6 Experimental Results

In this section, we report on a prototype implementation of our algorithms.

Implementation and Machine. We implemented our approach (i.e. Algorithms 5.1 and 5.3) in C++. We used LibTW [van Dijk et al., 2006] to obtain the tree decompositions. All results are obtained using an Intel Xeon E5-1650v3 3.5GHz processor running Debian 8.

Benchmarks. We used a variety of classical algorithms to generate the access sequences R for the Data Packing problem. For each classical algorithm, we generated random inputs of various sizes, which in turn led to random access sequences of varying lengths. We included algorithms from the following categories in our benchmark set: (i) linear algebra algorithms, (ii) sorting algorithms, (iii) dynamic programming, (iv) recursive algorithms, (v) string matching algorithms, (vi) computational geometry algorithms, (vii) algorithms on trees and (viii) algorithms on sorted arrays. For a complete list of the classical algorithms we used to generate benchmarks, see our technical report [Chatterjee et al., 2019e]. Moreover, each generated reference sequence R was executed for all $1 \leq m \leq 5$ and $2 \leq p \leq 5$. We did not include $p = 1$, because Data Packing is trivial in this case, i.e. each data element must form its own block.

Treewidth of Benchmarks. We observed that in many cases, by increasing the length of the access sequence R, the treewidth of the access graph G_R and the access hypergraph

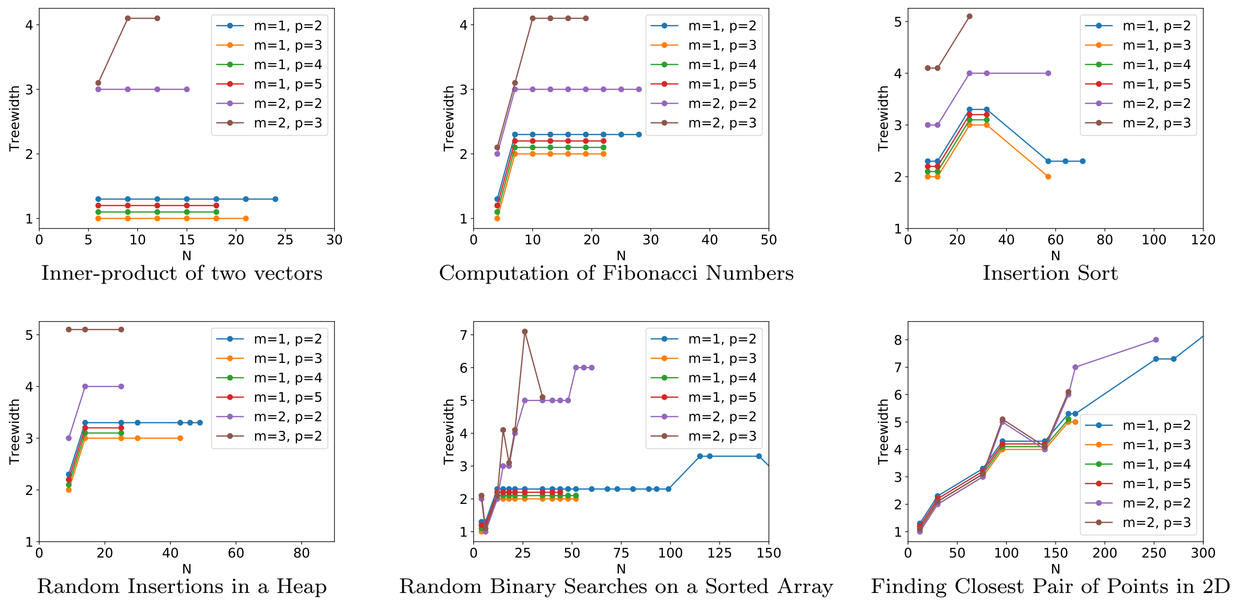


Figure 5.14: Treewidth of the hypergraph $G_R^{q^*}$ wrt the length N of the access sequence R generated from several classical algorithms. Note that the treewidth is always an integer. We have added small noises to the figures in order to make sure all the lines remain visible.

$G_R^{q^*}$ first slowly increase and then stabilize at a small value. Generally, we observe this phenomenon when the underlying data structure has small treewidth, which is the case in many real-world programs and most of our benchmarks. Figure 5.14 shows some of our benchmark algorithms, together with the treewidth of the resulting access (hyper)graphs of order q^*. The benchmark at the bottom-right corner of Figure 5.14, i.e. finding the Closest Pair among a given set of points in the plane, is an example of a real-world algorithm that does not have small treewidth.

Previous Approaches. We implemented several different state-of-the-art heuristic-based algorithms for data placement. These include CCDP [Calder et al., 1998], CPACK [Ding and Kennedy, 199 CPACK+, GPART+, CApRI+[*] [Ding and Kandemir, 2014] and two methods based on affinity hierarchies, namely the k-Distance method of [Zhong et al., 2004] and the Sampling

[*]The + in the names of these algorithms comes from applying the CApRI method which takes a data placement scheme created by a previous heuristic as its input and attempts to optimize it and produce a better data placement scheme. CApRI+ is the result of applying CApRI to an initial data placement scheme with unary blocks. For more information, see [Ding and Kandemir, 2014].

method of [Zhang et al., 2006]. We apply the latter algorithm with a sampling rate of 1, i.e. the highest possible sampling rate, to obtain the minimal number of cache misses it can produce.

Running Time. Note that the Data Packing formulation as considered in the literature [Thabit, 1982, Lavaee, 2016] is an offline problem, and these algorithms run once, but the output data placement schemes can lead to a reduction of cache-miss overheads every time the instance is run. Thus the main goal is to obtain optimal results within reasonable time limit. We set a runtime limit of 5 minutes per instance for each of the algorithms. In cases where only a single heuristic fails to terminate within 5 minutes, we report the result of the best-performing heuristic instead. This ensures we do not unfairly report too many misses for a heuristic. With the time limit above, our algorithm produces results on 2726 instances, and in most of these cases, the previous known optimal algorithm, i.e. an exhaustive search, does not terminate even in a day.

Experimental Results. Our experimental results over all instances are illustrated in Figure 5.15. Each row of Figure 5.15 shows a comparison between our algorithm and a number of heuristics. The x-axis denotes the optimal number of cache misses and the y-axis the number of cache misses incurred by the heuristic algorithm. Therefore, our algorithms' results correspond to the $y = x$ line, and, as expected, all heuristic-based results are above or on this line, i.e., they lead to more cache misses than optimal. To give a better view of the results, each row starts with a full plot of all the instances (on the left) and then zooms into the areas with a high density of points (which correspond to the instances that led to relatively few cache misses). In this figure, we did not include the points corresponding to heuristics that timed out, e.g. Sampling timed out on several larger instances, therefore there are few visible blue points in the leftmost plot of the second row of Figure 5.15.

We found that in total, our algorithm reduces the number of cache misses by between 15% (over Sampling) to 31% (over k-Distance). We also found that our algorithm is effective on every category of benchmarks. These results are illustrated in Figure 5.16.

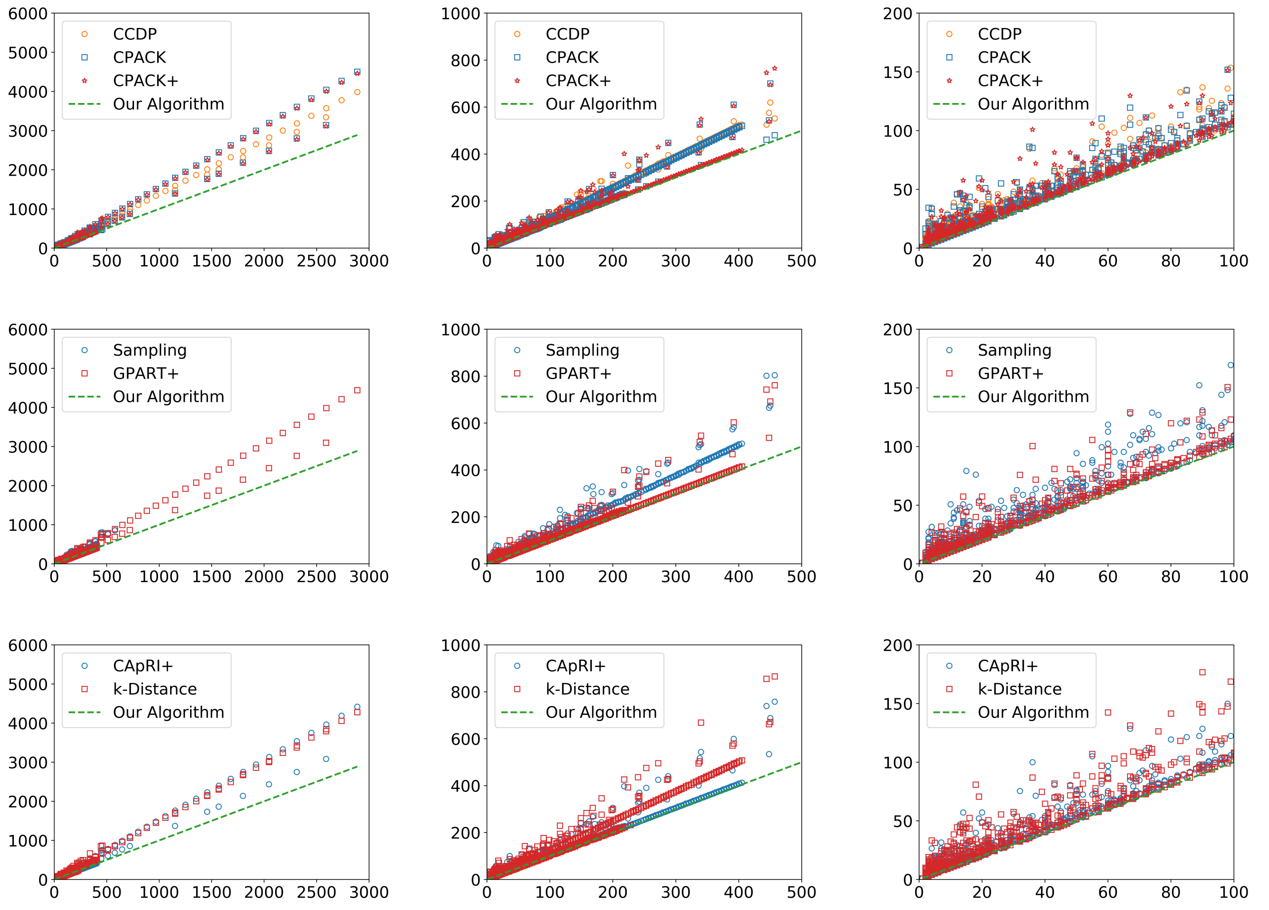
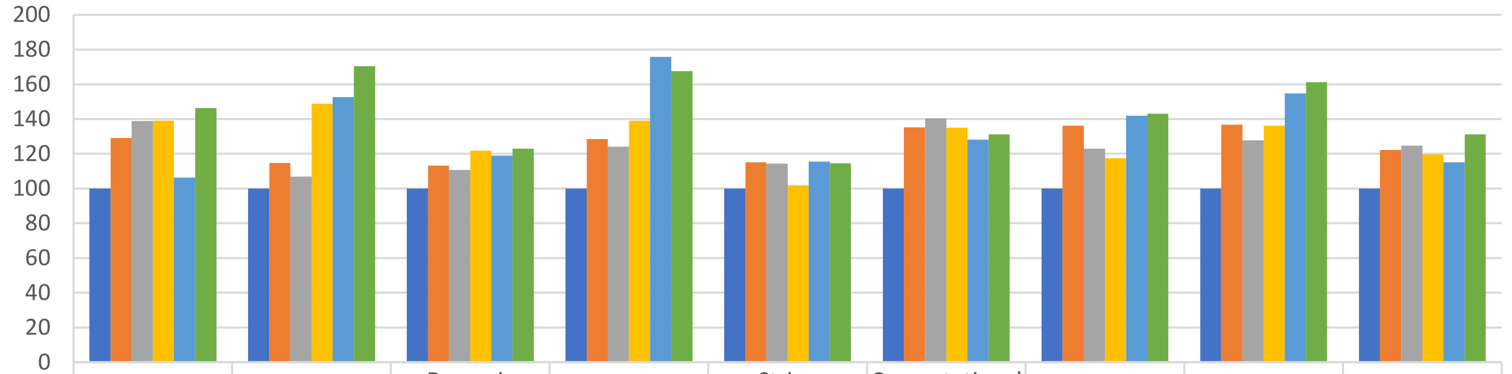


Figure 5.15: Experimental Results over all instances. In each plot, the x-axis is the optimal number of cache misses, i.e. the number of cache misses incurred by our algorithm, and the y-axis is the number of cache misses incurred by the heuristic-based algorithm. Each dot corresponds to a single instance. Each row begins with a plot of all instances at the left and then zooms into areas with a high density of points (center and right).



	Linear Algebra	Sorting	Dynamic Programming	Recursion	String Matching	Computational Geometry	Trees	Sorted Arrays	Total
🟦 Ours	100	100	100	100	100	100	100	100	100
🟧 CCDP	129.12	114.65	113.24	128.57	115.04	135.27	136.16	136.8	122.1
⬜ CPACK	138.77	106.95	110.62	124.02	114.4	140.31	123.04	127.75	124.61
🟨 CPACK+/GPART+/CApRI+	139.07	148.78	121.9	139.13	101.87	135.01	117.38	136.08	119.78
🟦 Sampling	106.28	152.71	118.95	175.78	115.4	128.06	142	154.69	115.2
🟩 k-Distance	146.32	170.54	122.88	167.49	114.55	131.12	143.15	161.18	131.23

Figure 5.16: Summary of Results by Benchmark Category. In each case we report the number of cache misses incurred by an algorithm as a percentage of the optimal number of cache misses. The optimal total number of cache misses over all the instances was 145,544. See [Chatterjee et al., 2019e] for all the numbers. We observe that there is no best heuristic that works better than others in all cases. Each heuristic-based algorithm works best on some specific categories of benchmarks. Our algorithm significantly outperforms the heuristics in all cases, and especially more so in benchmarks from Recursion, Computational Geometry and Sorted Arrays.

6

Invariant Generation for Polynomial Programs

This chapter originally appeared in the following publication:

[•] Chatterjee, K., Fu, H., **Goharshady, A. K.**, and Goharshady, E. K. **Polynomial Invariant Generation for Non-deterministic Recursive Programs**. In *41st ACM Conference on Programming Language Design and Implementation* **(PLDI)**, 2020.

6.1 Introduction

In this chapter, we consider the classical problem of invariant generation for programs with polynomial guards and assignments and focus on synthesizing invariants that are a conjunction of strict polynomial *inequalities*. We present a sound and semi-complete method based on positivstellensätze.

Our Theoretical Results. On the theoretical side, the worst-case complexity of our approach is subexponential, whereas the worst-case complexity of the previous complete method [Kapur, 2006] is doubly-exponential. Even when restricted to linear invariants, the best previous complexity for complete invariant generation is exponential [Colón et al., 2003].

Our Practical Results. On the practical side, we reduce the invariant generation problem to quadratic programming (QP), which is a classical optimization problem with many industrial solvers. We demonstrate the applicability of our approach by providing experimental results on several academic benchmarks. To the best of our knowledge, the only previous invariant generation method that provides completeness guarantees for invariants consisting of polynomial inequalities is [Kapur, 2006], which relies on quantifier elimination and cannot even handle toy programs such as our running example.

Invariants. An assertion at a program location that is always satisfied by the variables whenever the location is reached is called an *invariant*. Invariants are essential for many formal and quantitative analyses [Halbwachs et al., 1997, Henzinger and Ho, 1994, Ngo et al., 2018]. Invariant generation is a classical problem in verification and programming languages, and has been studied for decades, e.g. for safety and liveness analysis [Manna and Pnueli, 1995, Cousot and Halbwachs, 1978, Cousot and Cousot, 1977].

Inductive Invariants. An *inductive* assertion is an assertion that holds at a location for the first visit to it and is preserved under every cyclic execution path from and to the location. Inductive assertions are guaranteed to be invariants, and the well-established method to prove an assertion is an invariant is to find an inductive invariant that strengthens it [Colón et al., 2003, Manna and Pnueli, 1995].

Previous Approaches. Given the importance of invariant generation, the problem has received significant attention. One technique is *abstract interpretation* [Cousot and Halbwachs, 1978], which is primarily a theory of semantic approximations. It has been used for invariant generation by computing least fixed points of abstractions of the collecting semantics, but it guarantees completeness only for rare special cases [Giacobazzi and Ranzato, 1997].

Linear vs Polynomial Invariants. For *linear* invariant generation over programs with *linear* updates, a sound and complete methodology was obtained by [Colón et al., 2003]. We consider programs with *polynomial* updates and the problem of generating *polynomial* invariants, i.e. invariants that are a conjunction of polynomial *inequalities* over program variables. Hence, our setting is more general than [Colón et al., 2003] in terms of the programs we analyze, and also the desired invariants. The only previous approach that provides completeness for this problem is [Kapur, 2006]. However, it has doubly-exponential complexity and is not practically applicable even to toy programs. Efficient incomplete methods were proposed in [Kincaid et al., 2018, Farzan and Kincaid, 2015, Kincaid et al., 2017].

Motivation for Polynomial Invariants. Invariants are used as inputs in a wide variety of program analyses, and their accuracy can directly impact the effectiveness of those analyses. Given that polynomial invariants provide a higher degree of accuracy in comparison with linear invariants, using them improves existing solutions to many classical problems in programming languages, such as the following:

- *Safety Verification.* This is one of the most well-studied model checking problems: Given a program and a set of safety assertions that must hold at specific points of the program, prove that the assertions hold or report that they might be violated by the program. Many existing approaches for safety verification rely on invariants to prove the desired assertions (see [Manna and Pnueli, 1995, Alur et al., 2006, Padon et al., 2016, Albarghouthi et al., 2012b]). In these cases, weak invariants can lead to an increase in false positives, i.e. if the supplied invariants are inaccurate and grossly overestimate the program's behavior, then the verifier might falsely infer that a true assertion can be violated.

- *Termination Analysis.* A principal approach in proving termination of programs is to synthesize ranking functions, i.e. well-founded functions whose value decreases at every step of the program [Floyd, 1993]. Virtually all synthesis algorithms for ranking functions, including our approach in Chapter 8, depend on invariants, e.g. [Colón and Sipma, 2001, Bradley et al., 2005a, Chen et al., 2007]. Having inaccurate invariants, such as linear instead of polynomial, can lead to a failure in the synthesis and hence inability to prove termination. The same point also applies to termination analysis of probabilistic programs [Chakarov and Sankaranarayanan, 2013].

- *Inferring Complexity Bounds.* Another fundamental problem is to find automated algorithms that infer asymptotic complexity bounds on the runtime of (recursive) programs. Current algorithms for tackling this problem, such as [Chatterjee et al., 2017a], rely heavily on invariants and their accuracy. Inaccurate invariants can lead to an over-approximation of complexity or even failure to synthesize any complexity bound.

Completeness Guarantees. The points above not only justify the use of polynomial invariants due to their higher accuracy, but also demonstrate the need for approaches with completeness guarantees. State-of-the-art approaches in polynomial invariant generation either lack completeness guarantees or have doubly-exponential runtime and cannot be applied even to toy programs. In this chapter, we address this gap.

Our Contribution. We consider two variants of the invariant generation problem. Informally, the *weak* variant asks for an optimal invariant with respect to a given objective function, while the *strong* variant asks for a representative set of all invariants. Our contributions are as follows:

- *Soundness and Semi-completeness.* We present a sound and semi-complete method to generate polynomial invariants for programs with polynomial updates and solve the strong invariant generation problem. Our completeness requires a compactness condition that is satisfied by all real-world programs (Remark 6.4). Using the standard notions of pre and post-conditions, our method can be extended to handle recursion as well. See [Chatterjee et al., 2020a] for details.

- *Theoretical Complexity.* We show that the worst-case complexity of our procedure is *subexponential* if we consider polynomial invariants with rational coefficients. In comparison, complexity of the procedure in [Kapur, 2006] is doubly-exponential and the approach of [Colón et al., 2003], which is sound and complete for *linear* invariants, has *exponential* complexity, whereas we show how to generate *polynomial* invariants in *subexponential* time.

- *Practical Approach.* We present a polynomial-time reduction from the weak invariant generation problem to solving a quadratically-constrained linear program (QCLP)*. Solving QCLPs is an active area of research in optimization. Moreover, there are many industrial solvers for handling real-world instances of quadratic programming and, using our algorithm, practical improvements to such solvers carry over to polynomial invariant generation.

Hence, our main contribution is theoretical, i.e. presenting a subexponential sound and *semi-complete* method for generating polynomial invariants. Moreover, we also demonstrate the applicability of our approach by providing experimental results on several academic examples from [Rodríguez-Carbonell, 2018] that require polynomial invariants. Unsurprisingly, we observe that our approach is slower than previous sound but incomplete methods, so there is a trade-off between completeness and efficiency. However, we expect practical improvements in solving QCLPs to narrow the efficiency gap in the future. On the other hand, the only previous complete method, proposed in [Kapur, 2006], is extremely impractical and cannot handle any of our benchmarks, not even our toy running example.

Techniques. While the approaches of [Colón et al., 2003, Kapur, 2006] use Farkas' lemma and quantifier elimination to generate invariants, our technique is based on a positivstellensatz. Our method replaces the quantifier elimination step with either (i) an algorithm of [Grigor'ev and Vorobjov, 1988] for characterizing solutions of systems of polynomial inequalities or (ii) a reduction to QCLP.

*Some of our reductions are to quadratically-constrained quadratic programs (QCQP). However, QCQP is itself reducible to QCLP by adding a new variable that is constrained to be equal to the objective function.

6.2 Related works

Invariant Generation. Automated invariant generation is an important research topic that has received much attention in the past years, and various classes of approaches have been proposed, including the following:

- Recurrence analysis [Kincaid et al., 2018, Farzan and Kincaid, 2015, Humenberger et al., 2017, Kincaid et al., 2017],
- Abstract interpretation [Bagnara et al., 2005, Chakarov and Sankaranarayanan, 2014, Rodríguez-Carbonell and Kapur, 2007, Cousot et al., 2005, Müller-Olm and Seidl, 2004],
- Constraint solving [Kapur, 2006, Katoen et al., 2010, Chen et al., 2015, Feng et al., 2017, Colón et al., 2003, Sankaranarayanan et al., 2004, Rodríguez-Carbonell and Kapur, 2004, de Oliveira et al., 2016, Chatterjee et al., 2017d, Yang et al., 2010, Cousot, 2005, Lin et al., 2014],
- Inference [Gulwani et al., 2009, Dillig et al., 2013, Sharma and Aiken, 2016],
- Interpolation [McMillan, 2008],
- Symbolic execution [Csallner et al., 2008],
- Dynamic analysis [Nguyen et al., 2012], and
- Learning [Garg et al., 2016].

Summary. A summary of the results of the literature with respect to types of assignments, types of generated invariants (polynomial, linear, etc.), programming language features that can be handled (i.e. non-determinism, probability and recursion), soundness, completeness, and whether the approach can handle weak/strong invariant generation is presented in Table 6.1. For approaches that are applicable to weak/strong invariant generation, the respective runtimes are also reported. Most of the previous approaches, which are included in Table 6.1 for the sake of completeness, are indeed incomparable with our approach, given that they handle different problems, e.g. different types of programs.

Approach	Assignments and Guards	Invariants	Nondet	Rec	Prob	Sound	Complete	Weak	Strong
This Work	Polynomial	Polynomial	✓	✓	✗	✓	✓[♦]	✓ QCLP	✓ Subexp
[Colón et al., 2003] CAV'03	Linear[c]	Linear	✓	✗	✗	✓	✓	✓ Exp[†]	✓ Exp[†]
[Kapur, 2006] ACA'04	Polynomial	Polynomial	✓	✓	✗	✓	✓	✓ 2Exp	✓ 2Exp
[Dillig et al., 2013] OOPSLA'13	General	Linear (Presburger)	✓	✓	✗	✓	✗	✗	✗
[Feng et al., 2017] ATVA'17	Polynomial	Polynomial	✗	✗	✓	✓	✓[a]	✓ Poly	✗
[Hrushovski et al., 2018] LICS'18	Linear[‡]	Polynomial *Equalities*	✓	✗	✗	✓[‡]	✓[‡]	✗	✓[‡,b]
[Kincaid et al., 2018] POPL'18	Polynomial, Exponential, Logarithmic	Polynomial, Exponential, Logarithmic	✓	✓	✗	✓	✗	✗	✗
[Rodríguez-Carbonell and Kapur, 2004]* ISSAC'04	Polynomial, Exponential	Polynomial *Equalities*	✓	✗	✗	✓	✓	✓[b]	✓[b]
[Sankaranarayanan et al., 2004] POPL'04	Polynomial[c]	Polynomial *Equalities*	✓	✗	✗	✓	✓[b]	✓[b]	✓[b]
[Farzan and Kincaid, 2015] FMCAD'15	General[d]	General[d]	✓	✗[e]	✗	✓	✗	✗	✗
[Kincaid et al., 2017] PLDI'17	General	General	✓	✓	✗	✓	✗	✗	✗
[de Oliveira et al., 2016] ATVA'16	Polynomial, *Without Conditional Branching*	Polynomial *Equalities*	✓	✗	✗	✓	✓	✓ Poly	✓ Poly
[Humenberger et al., 2017]* ISSAC'17	Polynomial[‡]	Polynomial *Equalities*	✓	✗	✗	✓	✓[‡]	✓[‡,b]	✓[‡,b]
[Adjé et al., 2015]* SAS'15	Polynomial	Polynomial	✗	✗	✗	✓	✗	✗	✗

[♦] Semi-complete, assuming compactness (see Remark 6.4 and Lemma 6.2)
[†] Generates a system of quadratic inequalities, but then applies quantifier elimination, leading to exponential runtime.
[‡] Treats branching conditions as non-determinism.
$*$ Does not support nested loops.
[a] Semi-complete
[b] Uses Gröbner basis computations (super-exponential worst-case runtime in theory).
[c] Considers general transition systems instead of programs.
[d] Handles non-linearity using linearization heuristics.
[e] Can be extended to handle recursion (see [Kincaid et al., 2017]).

Table 6.1: Summary of approaches for invariant generation.

Compared to previous works, we present the first applicable sound and semi-complete approach for polynomial invariant generation. Our complexity (subexponential) is not only better than the previous doubly-exponential complexity for polynomial invariants [Kapur, 2006], it even beats the exponential complexity of complete methods for linear invariants [Colón et al., 2003].

Recurrence Analysis. While approaches based on recurrence analysis can derive exact invariants, they are applicable to a restricted class of programs where closed-form solutions exist. Our approach does not require closed-form solutions.

Abstract Interpretation. This is the oldest and most classical approach to invariant generation [Cousot and Halbwachs, 1978, Cousot and Cousot, 1977] and has also been used for generating quadratic invariants [Adjé et al., 2010]. However, unlike our approach, it cannot provide completeness, except in very special cases [Giacobazzi and Ranzato, 1997]. There are efficient tools and algorithms for invariant generation using abstract interpretation [Singh et al., 2017, Singh et al., 2015], but they focus on generating linear invariants.

Approaches in Dynamical Systems. Similar techniques have also been applied in the context of continuous and hybrid dynamical systems [Oustry et al., 2019, Ben Sassi et al., 2015, Sankaranarayanan, 2011]. However, they ensure neither completeness nor subexponential complexity. A related well-known method is that of "barrier certificates" for safety verification of hybrid systems [Prajna and Jadbabaie, 2004]. A barrier certificate is a function whose initial value is non-positive and each loop iteration will never increase its value. In this sense, barrier certificates are a special type of invariants, while our approach targets invariants in general form.

Constraint Solving. Our approach falls in this category. First, we handle polynomial invariants, thus extending approaches based on linear arithmetics, such as [Katoen et al., 2010, Colón et al., 2003, de Oliveira et al., 2016, Chatterjee et al., 2017d]. Second, we generate invariants consisting of polynomial *inequalities*, whereas several previous approaches synthesize polynomial *equalities* [Sankaranarayanan et al., 2004, Rodríguez-Carbonell and Kapur, 2004]. Third, our approach is semi-complete, thus it is more accurate than approaches with relaxations (e.g. [Cousot, 2005, Lin et al., 2014]). Fourth, compared to previous complete approaches that solve formulas in the first-order theory of reals (e.g. [Chen et al., 2015, Yang et al., 2010, Kapur, 2006]) to generate invariants (often for a more limited set of programs), our approach has lower complexity, i.e. our approach is subexponential, whereas methods based on quantifier elimination and solving first-order formulas take exponential or doubly-exponential time. Another notable approach in this category is [Zhu et al., 2019] that synthesizes barrier certificates for the verification of reinforcement learning methods. Compared to [Zhu et al., 2019], our approach is not restricted to barrier certificates and can handle non-convex invariants, whereas [Zhu et al., 2019] relies on [Andersen and Andersen, 2018] and can only handle convex inequalities.

Comparison with [Feng et al., 2017]. Finally, we compare our approach with the most related work, i.e. [Feng et al., 2017]. A main difference is that our approach can find a representative set of all solutions, but [Feng et al., 2017] might miss some solutions, i.e. it only guarantees to find at least one solution as long as the problem is feasible. In terms of techniques, [Feng et al., 2017] uses Stengle's positivstellensatz, while we use Putinar's positivstellensatz and the algorithm of [Grigor'ev and Vorobjov, 1988]. Moreover, [Feng et al., 2017] considers the class of probabilistic programs without non-determinism and only focuses on single probabilistic while loops, while we consider programs in general form, with non-determinism and recursion, but without probability.

6.3 Invariants and Inductive Assertion Maps

In this section, we formally define the problems considered in this chapter and fix our notation.

Polynomial Programs. In the sequel, we consider polynomial transition systems, i.e. systems $S = (\mathbf{V}, \mathbf{L}, \ell_0, I, \theta)$ in which for every transition $\theta = (\ell, \ell', \varphi, \mu) \in \mathbf{\Theta}$, φ is a *conjunction* of polynomial inequalities over $\mathbf{V}$ and μ assigns a polynomial over $\mathbf{V}$ to each variable.

Example 6.1. *Consider the simple program in Figure 6.1. We will use this program as our running example. It contains a single function* sum *that takes a parameter n and then non-deterministically sums up some of the numbers between* 1 *and n and returns the summation. We assume that we initially have $n \geq 1$. Our goal is to prove that the final value of s is always less than* $0.5 \cdot n^2 + 0.5 \cdot n + 1$.

```
sum(n) {
1:  $i$ := 1;
2:  $s$ := 0;
3:  while $i \le n$ do
4:          □ $s$ := $s + i$
5:          □ skip;
6:          $i$ := $i + 1$
    od;
7:  }
```

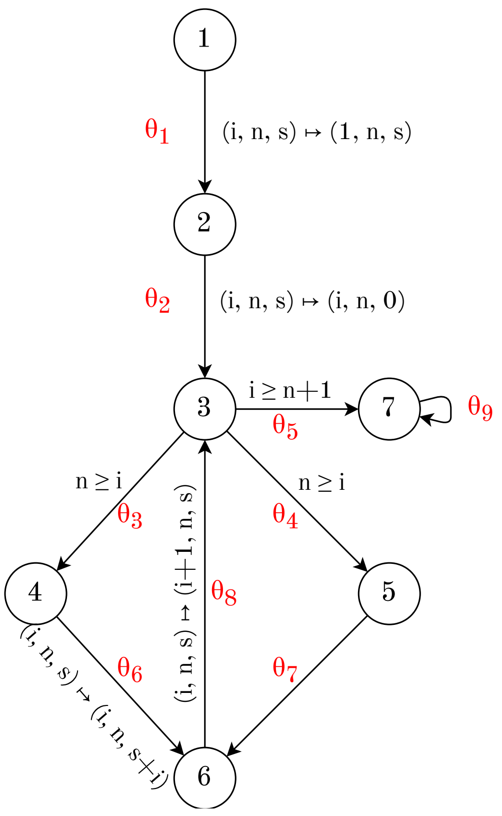


Figure 6.1: A non-deterministic summation program (left) and its representation as a transition system (right).

Pre-conditions. A *pre-condition* is a function Pre mapping each label $\ell \in \mathbf{L}$ of the transition system to a conjunctive propositional formula $\mathsf{Pre}(\ell) := \bigwedge_{i=0}^{m} (\mathfrak{e}_i \geq 0)$, where each $\mathfrak{e}_i$ is an arithmetic expression over the set $\mathbf{V}$ of variables[†] Intuitively, a pre-condition specifies a set of requirements for the runs of the program, i.e. a run that does not satisfy the pre-condition is considered to be invalid or impossible.

Valid Runs. A run $\mathsf{r} = \langle(\ell_i, \mathsf{v}_i), \theta_i\rangle_{i=0}^{\infty}$ is *valid* with respect to a pre-condition Pre if for every i, we have $\mathsf{v}_i \models \mathsf{Pre}(\ell_i)$. A state is *reachable* if it appears in a valid run.

Model of Computation. We consider programs in which variables can have arbitrary real values. However, some of our results only hold if the variable values are bounded. In such cases we explicitly mention that the result holds on "bounded reals". The formal interpretation of this point is that there exists a constant value $c \in \mathbb{R}^{\geq 0}$ such that for every label $\ell \in \mathbf{L}$ and every variable $v \in \mathbf{V}$, the pre-condition $\mathsf{Pre}(\ell)$ contains the inequalities $-c \leq v \leq c$. In other words, in the bounded reals model of computation, a variable overflows if its value becomes more than c (resp. underflows if its value becomes less than $-c$), and any run containing an overflow or underflow is considered invalid. As a direct consequence, for every reachable state $\sigma = (\ell, \mathsf{v})$, we have $\|\mathsf{v}\|_2 \leq c \cdot \sqrt{|\mathbf{V}|}$. Hence, when discussing bounded reals, we assume every pre-condition contains the inequality $\|\mathbf{V}\|_2^2 \leq c^2 \cdot |\mathbf{V}|$, too.[‡] Note that this inequality is entailed by the bounds on values of individual variables. However, we will later need it to satisfy the requirements of our positivstellensatz (Theorem 2.2).

Invariants. Given a transition system $S = (\mathbf{V}, \mathbf{L}, \ell_0, I, \boldsymbol{\Theta})$ and a pre-condition Pre with $\mathsf{Pre}(\ell_0) = I$, an *invariant* is a function Inv mapping each label $\ell \in \mathbf{L}$ of the program to a conjunctive propositional formula $\mathsf{Inv}(\ell) := \bigwedge_{i=0}^{m} (\mathfrak{e}_i > 0)$ over $\mathbf{V}$, such that for every reachable state $\sigma = (\ell, \mathsf{v})$, it holds that $\mathsf{v} \models \mathsf{Inv}(\ell)$.

[†]Classically, pre-conditions are only defined for the first labels. Indeed, our definition of transition systems contains $I = \mathsf{Pre}(\ell_0)$. In this chapter, we allow pre-conditions for every label. This setting is strictly more general, given that one can let $\mathsf{Pre}(\ell) = \textbf{true}$ for every other label.

[‡]More concretely, if $\mathbf{V} = \{v_1, v_2, \dots, v_n\}$, then the pre-condition contains the inequality

$$v_1^2 + v_2^2 + \ldots + v_n^2 \leq c^2 \cdot n.$$

Positivity Witnesses. Let $\mathfrak{e}$ be an arithmetic expression on $\mathbf{V}$ and $\phi = \bigwedge_{i=0}^{m} (\mathfrak{e}_i \bowtie_i 0)$ for $\bowtie_i \in \{>, \geq\}$, such that for every valuation ν, we have $\nu \models \phi \Rightarrow \mathfrak{e}(\nu) > 0$. We say that a constant $\epsilon > 0$ is a *positivity witness* for $\mathfrak{e}$ with respect to ϕ if for every valuation ν, we have $\nu \models \phi \Rightarrow \mathfrak{e}(\nu) > \epsilon$. In the sequel, we limit our focus to inequalities that have positivity witnesses. Intuitively, this means that we consider invariants of the form $\bigwedge_{j=1}^{m} (\mathfrak{e}_j > 0)$ where the values of $\mathfrak{e}_j$'s in the valid runs of the program cannot get arbitrarily close to 0[§].

Inductive Assertion Maps. Given a transition system $S = (\mathbf{V}, \mathbf{L}, \ell_0, I, \mathbf{\Theta})$ and a pre-condition Pre with $\mathsf{Pre}(\ell_0) = I$, an *inductive assertion map* is a function Ind mapping each label $\ell \in \mathbf{L}$ of the program to a conjunctive propositional formula $\mathsf{Ind}(\ell) := \bigwedge_{i=0}^{m} (\mathfrak{e}_i > 0)$ over $\mathbf{V}$, such that the following two conditions hold:

- *Initiation.* For every state $\sigma = (\ell_0, \mathsf{v}_0)$, we have $\mathsf{v}_0 \models \mathsf{Pre}(\ell_0) \Rightarrow \mathsf{v}_0 \models \mathsf{Ind}(\ell_0)$. Intuitively, $\mathsf{Ind}(\ell_0)$ should be entailed by the pre-condition $\mathsf{Pre}(\ell_0) = I$.

- *Consecution.* Let $\sigma' = (\ell', \mathsf{v}')$ be a successor state of $\sigma = (\ell, \mathsf{v})$. We must have

$$(\mathsf{v} \models \mathsf{Ind}(\ell) \ \wedge \ \mathsf{v} \models \mathsf{Pre}(\ell) \ \wedge \ \mathsf{v}' \models \mathsf{Pre}(\ell')) \Rightarrow \mathsf{v}' \models \mathsf{Ind}(\ell').$$

  Intuitively, this condition means that the inductive assertion map cannot be falsified by running a valid step of the execution of the program.

It is well-known that every inductive assertion map is an invariant. So, inductive assertion maps are often called *inductive invariants*, too.

Example 6.2. *Consider the program in Figure 6.1. Assuming that we have the pre-condition $n \geq 0$ at label 1, it is easy to show that for any $\epsilon > 0$, $\mathsf{Ind}(\ell) := (n+\epsilon > 0 \wedge i+\epsilon > 0 \wedge s+\epsilon > 0)$ for all $\ell \in \{1, \ldots, 7\}$ is an inductive assertion map, i.e. it holds at the beginning of the program and no valid execution step falsifies it. Hence, it is also an invariant.*

[§]Note that this is a very minor restriction, in the sense that if $\mathfrak{e} > 0$ is an invariant, then so is $\mathfrak{e} + \epsilon > 0$. We are unable to find invariants $\mathfrak{e} > 0$ where $\mathfrak{e}$ can get arbitrarily close to 0 over all valid runs of the program. However, in such cases, we can synthesize $\mathfrak{e} + \epsilon > 0$ for *any* positive ϵ.

We define our synthesis problem in terms of inductive invariants, because the classical method for finding or verifying invariants is to consider inductive invariants that strengthen them [Colón et al., 2003, Manna and Pnueli, 1995].

The Invariant Synthesis Problem. Given a transition system $S = (\mathbf{V}, \mathbf{L}, \ell_0, I, \theta)$, together with a pre-condition Pre with $\mathsf{Pre}(\ell_0) = I$, the invariant synthesis problem asks for *inductive* invariants of a given form and size (e.g. linear or polynomial of a given degree). The problem can be divided into two variants:

- The *Strong Invariant Synthesis Problem* asks for a characterization or a representative set of all possible invariants.
- The *Weak Invariant Synthesis Problem* provides an objective function over the invariants (e.g. a function over the coefficients of polynomial invariants) and asks for an invariant that maximizes the objective function.

Polynomial Invariants. In the sequel, we consider the synthesis problems for *polynomial* invariants and pre-conditions, i.e. we assume that all arithmetic expressions used in the atomic assertions are polynomials with real coefficients.

Remark 6.1. *In this chapter, we focus on transition systems. However, our approaches are easily extensible to recursive programs, as well. See [Chatterjee et al., 2020a] for details.*

Notation. To avoid defining spurious notation, we use the polynomial $g \in \mathbb{R}[\mathbf{V}]$ and the boolean predicate $g \geq 0$ interchangeably. Similarly, we interchange a set $\{g_1, \ldots, g_n\}$ of polynomials and the formula $\bigwedge_{i=1}^{n} g_i \geq 0$.

6.4 Our Positivstellensatz-based Approach

We first provide a sound and semi-complete reduction from inductive invariants to solutions of a system of quadratic constraints. Our main tool is Putinar's positivstellensatz (Theorem 2.2). Using this reduction, we show that the Strong Invariant Synthesis problem can be solved in subexponential time. We also show that the Weak Invariant Synthesis problem can be reduced to QCLP.

6.4.1 Overview of the Approach

In this section, we provide an overview of our algorithms. The next sections go through all the details. Our algorithms for Strong and Weak Invariant Synthesis are very similar. They each consist of four main steps and differ only in the last step. The steps are as follows:

- *Step 1.* First, the algorithm creates a template for the inductive invariant at each label. More specifically, it creates polynomial templates of the desired size and degree, but with *unknown coefficients*. The goal is to synthesize values for these unknown coefficients so that the template becomes a valid inductive invariant.

- *Step 2.* The algorithm generates a set of constraints that should be satisfied by the template so as to ensure that it becomes an inductive invariant. These constraints encode the initiation and consecution requirements as in the definition of inductive invariants. Moreover, they have a very specific form: each constraint consists of polynomials $g_1, \ldots, g_m$ and g over $\mathbf{V}$ and encodes the requirement that for every valuation ν, if we have $g_1(\nu) \geq 0, g_2(\nu) \geq 0, \ldots, g_m(\nu) \geq 0$, then we must also have $g(\nu) > 0$.

- *Step 3.* Exploiting the structure of the constraints generated in the previous step, the algorithm applies Putinar's positivstellensatz to translate the constraints into quadratic constraints over the unknown coefficients.

- *Step 4.* The algorithm uses an external solver for handling the system of quadratic constraints generated in the previous step. In case of Strong Invariant Synthesis, the external solver would use the algorithm of [Grigor'ev and Vorobjov, 1988] to provide a representative set of all invariants. In contrast, for Weak Invariant Synthesis, the external solver is an optimization suite for quadratic programming (QCLP).

6.4.2 Strong Invariant Synthesis

We now provide a formal description of the input to our algorithm for Strong Invariant Synthesis and then present details of every step.

The StrongInvSynth Algorithm. We present an algorithm StrongInvSynth that gets the following items as its input:

- A polynomial transition system $S = (\mathbf{V}, \mathbf{L}, \ell_0, I, \mathbf{\Theta})$ with finitely many variables and locations,
- A polynomial pre-condition Pre, such that $\mathsf{Pre}(\ell_0) = I$,
- Positive integers d, n and Υ, where d is the degree of polynomials in the desired inductive invariants, n is the desired size of the invariant generated at each label, i.e. number of atomic assertions, and Υ is a technical parameter to ensure semi-completeness, which will be discussed later;

and produces a representative set of all inductive invariants Ind of the transition system S, such that for all $\ell \in \mathbf{L}$, the set $\mathsf{Ind}(\ell)$ consists of n atomic assertions of degree at most d. Our algorithm consists of the following four steps:

Step 1) Setting up templates. Let $\mathbf{V} = \{v_1, v_2, \ldots, v_t\}$ and define $\mathbf{M}_d = \{m_1, m_2, \ldots, m_r\}$ as the set of all monomials of degree at most d over $\mathbf{V}$, i.e. $\mathbf{M}_d := \{\prod_{i=1}^{t} v_i^{\alpha_i} \mid \forall i \;\; \alpha_i \in \mathbb{Z}^{\geq 0} \;\wedge\; \sum_{i=1}^{t} \alpha_i \leq d\}$. At each label $\ell \in \mathbf{L}$ of the program S, the algorithm generates a template $\eta(\ell) := \bigwedge_{i=1}^{n} \varphi_{\ell,i}$ where each $\varphi_{\ell,i}$ is of the form $\varphi_{\ell,i} := \left(\sum_{j=1}^{r} s_{\ell,i,j} \cdot m_j > 0\right)$. Here, the $s_{\ell,i,j}$'s are new unknown variables. For brevity, we call them s-variables. Intuitively, our goal is to synthesize values for s-variables such that η becomes an inductive invariant.

Example 6.3. *Consider the summation program in Figure 6.1. We have* $\mathbf{V} = \{n, i, s\}$*. Suppose that we want to synthesize a single quadratic assertion as the invariant at each label. In Step 1, the algorithm creates the following template for each label* $\ell \in \{1, 2, \ldots, 7\}$*:*

$$\begin{aligned}\eta(\ell) := \;\; & s_{\ell,1,1} + s_{\ell,1,2} \cdot n + s_{\ell,1,3} \cdot i + s_{\ell,1,4} \cdot s + s_{\ell,1,5} \cdot n^2 + s_{\ell,1,6} \cdot n \cdot i + \\ & s_{\ell,1,7} \cdot n \cdot s + s_{\ell,1,8} \cdot i^2 + s_{\ell,1,9} \cdot i \cdot s + s_{\ell,1,10} \cdot s^2\end{aligned}$$

Step 2) Setting up constraint pairs. For each transition $\theta = (\ell, \ell', \varphi, \mu) \in \mathbf{\Theta}$, the algorithm constructs a set Λ_θ of *constraint pairs* of the form $\lambda = (\Gamma, g)$ where $\Gamma = \bigwedge_{i=1}^{m} (g_i \geq 0)$

and $g, g_1, \ldots, g_m$ are polynomials in $\mathbb{R}[\mathbf{V}]$ with unknown coefficients based on the s-variables. Intuitively, a constraint pair (Γ, g) encodes the following condition:

$$\forall \nu \in \mathbb{R}^{\mathbf{V}} \;\; \nu \models \Gamma \Rightarrow g(\nu) > 0 \;\; \equiv \;\; \forall \nu \in \mathbb{R}^{\mathbf{V}} \, (\forall g_i \in \Gamma \;\; g_i(\nu) \geq 0) \Rightarrow g(\nu) > 0.$$

The construction is as follows[¶]:

- For every polynomial g such that $g > 0$ appears in $\eta(\ell')$, the algorithm adds the constraint pair
$$(\eta(\ell) \;\wedge\; \mathsf{Pre}(\ell) \;\wedge\; \varphi \;\wedge\; \mathsf{Pre}(\ell') \circ \mu \;,\; g \circ \mu)$$
to Λ_θ. Here, $\circ$ denotes composition.

Additionally, the algorithm constructs the following set Λ_0:

- For every polynomial g for which $g > 0$ appears in $\eta(\ell_0)$, the algorithm adds the constraint pair $(\mathsf{Pre}(\ell_0), g)$ to Λ_0.

Finally, the algorithm computes $\Lambda = \bigcup_{\theta \in \mathbf{\Theta}} \Lambda_\theta \cup \Lambda_0$.

Example 6.4. *In the summation program of Figure 6.1, suppose that* $I = \mathsf{Pre}(1) := (n \geq 0)$ *and* $\mathsf{Pre}(\ell) := (1 \geq 0) \equiv \mathbf{true}$ *for every* $\ell \neq 1$. *We provide some examples of constraint pairs generated in Step 2 of the algorithm:*

- *We have the transition* $\theta_1 = (1, 2, \textit{\textbf{true}}, (i, n, s) \mapsto (1, n, s))$ *in* $\mathbf{\Theta}$. *So, the algorithm considers the pair*
$$(\eta(1) \;\wedge\; \mathsf{Pre}(1) \;\wedge\; \textit{\textbf{true}} \;\wedge\; \mathsf{Pre}(2)[i \leftarrow 1], \eta(2)[i \leftarrow 1])$$
which is symbolically computed as
$$\left(\begin{array}{c} s_{1,1,1} + s_{1,1,2} \cdot n + \ldots + s_{1,1,10} \cdot s^2 \geq 0 \;\wedge \\ n \geq 0 \end{array} \quad , \quad \begin{array}{c} s_{2,1,1} + s_{2,1,2} \cdot n + s_{2,1,3} + s_{2,1,4} \cdot s + \\ s_{2,1,5} \cdot n^2 + s_{2,1,6} \cdot n + s_{2,1,7} \cdot n \cdot s + \\ s_{2,1,8} + s_{2,1,9} \cdot s + s_{2,1,10} \cdot s^2 \end{array} \right)$$

[¶]Note that all computations are done symbolically.

and added to Λ_{θ_1}*. Note that* $\mathsf{Pre}(2)$ *is* **true** *and is hence ignored.*

- *We have the transition* $\theta_3 = (3, 4, n - i \geq 0, (i, n, s) \mapsto (i, n, s))$ *in* $\mathbf{\Theta}$*. So, the algorithm considers the pair*

$$(\eta(3) \ \wedge \ \mathsf{Pre}(3) \ \wedge \ n - i \geq 0 \ \wedge \ \mathsf{Pre}(4), \eta(4))$$

Note that in this case there is no update, i.e. the update function μ *is the identity function. Moreover,* $\mathsf{Pre}(3)$ *and* $\mathsf{Pre}(4)$ *are both* ***true****. Hence, this constraint pair is symbolically computed as follows*

$$\left(\begin{matrix} s_{3,1,1} + s_{3,1,2} \cdot n + \ldots + s_{3,1,10} \cdot s^2 \geq 0 \ \wedge \\ n - i \geq 0 \end{matrix} \quad , \quad s_{4,1,1} + s_{4,1,2} \cdot n + \ldots + s_{4,1,10} \cdot s^2 \right).$$

and added to Λ_{θ_3}.

- *We have* $I = \mathsf{Pre}(1) := (n \geq 0)$ *so the algorithm computes the following constraint pair and adds it to* Λ_0 *:*

$$\left(n \geq 0 \quad , \quad s_{1,1,1} + s_{1,1,2} \cdot n + \ldots + s_{1,1,10} \cdot s^2\right).$$

Step 3) Translating constraint pairs to quadratic equalities. Let Λ be the set of all constraint pairs from the previous step. For each $\lambda = \left(\bigwedge_{i=1}^{m} (g_i \geq 0), g\right) \in \Lambda$, the algorithm takes the following actions:

(i) Let $\mathbf{V}' = \{v_1, \ldots, v_{t'}\} \subseteq \mathbf{V}$ be the set of all program variables that appear in g or the g_i's. The algorithm computes the set $\mathbf{M}_\Upsilon = \{m'_1, m'_2, \ldots, m'_{r'}\}$ of all monomials of degree at most Υ over $\mathbf{V}'$. Note that Υ is a technical parameter that was supplied as part of the input.

(ii) It symbolically computes an equation of the form (2.2):

$$g = \epsilon + h_0 + \textstyle\sum_{i=1}^{m} h_i \cdot g_i \qquad (\dagger)$$

where ϵ is a new unknown and positive real variable and each polynomial h_i is of the form $\sum_{j=1}^{r} t_{i,j} \cdot m'_j$. Here, the $t_{i,j}$'s are also new unknown variables. Intuitively, we aim to synthesize values for both t-variables and s-variables in order to ensure the polynomial equality (†). Note that both sides of (†) are polynomials in $\mathbb{R}[\mathbf{V}']$ whose coefficients are quadratic expressions over the newly-introduced s-, t- and ϵ-variables.

(iii) The algorithm equates the coefficients of corresponding monomials in the left and right hand sides of (†), leading to a set of quadratic equalities over the unknown variables, i.e. s-, t- and ϵ-variables.

(iv) The algorithm computes a set of quadratic equalities which are equivalent to the assertion that the h_i's can be written as sums of squares (Lemma 2.3).

The algorithm conjunctively compiles all the generated quadratic equalities into a single system of quadratic constraints.

Remark 6.2. *Based on above, the technical parameter Υ is the maximum degree of the sum-of-squares polynomials h_i in (†). More specifically, in Step 3, we are applying a special case of Putinar's positivstellensatz, in which the sum-of-square polynomials can have a degree of at most Υ.*

Example 6.5. *Consider the first constraint pair generated in Example 6.4. The algorithm writes (†), i.e. $g = \epsilon + h_0 + \sum_{i=1}^{2} h_i \cdot g_i$ where $g = s_{2,1,1} + \ldots + s_{2,1,10} \cdot s^2$ is the polynomial in the second component of the constraint pair, and $g_1 = s_{1,1,1} + \ldots + s_{1,1,10} \cdot s^2$ and $g_2 = n$ are the polynomials in the first component of the constraint pair. Moreover, each h_i is a newly generated polynomial containing all possible monomials of degree at most Υ, e.g. if $\Upsilon = 2$, we have $h_i = t_{i,1} + t_{i,2} \cdot n + \ldots + t_{i,10} \cdot s^2$, where each $t_{i,j}$ is a new unknown variable. It then equates the coefficients of corresponding monomials on the two sides of (†). For example, consider the monomial n. Its coefficient in g is $s_{2,1,2} + s_{2,1,6}$. In the RHS of (†), there are a variety of ways to obtain n, hence its coefficient is the sum of the following:*

- *$t_{0,2}$, i.e. the coefficient of n in h_0,*
- *$s_{1,1,1} \cdot t_{1,2} + s_{1,1,2} \cdot t_{1,1}$, i.e. the coefficient of n in $h_1 \cdot g_1$,*

- $t_{2,1}$, *i.e. the coefficient of* n *in* $h_2 \cdot g_2$.

Hence, the algorithm generates the quadratic equality

$$s_{2,1,2} + s_{2,1,6} = t_{0,2} + s_{1,1,1} \cdot t_{1,2} + s_{1,1,2} \cdot t_{1,1} + t_{2,1}$$

over the s*- and* t*-variables. The algorithm computes similar equalities for every other monomial, including* 1.

Step 4) Finding representative solutions. The previous step has created a system of quadratic constraints over s-variables and other new variables. In this step, the algorithm finds a representative set Ξ of solutions to this system by calling an external solver. Then, for each solution $\xi \in \Xi$, it plugs the values synthesized for the s-variables into the template η to obtain an inductive invariant $\eta_\xi := \eta[s_{\ell,i,j} \leftarrow \xi(s_{\ell,i,j})]$. The algorithm outputs $\{\eta_\xi \mid \xi \in \Xi\}$.

Remark 6.3 (Representative Solutions). *In real algebraic geometry, a standard notion for a representative set of solutions to a polynomial system of equalities is to include one solution from each connected component of the set of solutions [Basu et al., 2007]. The classical algorithm for this problem is called* cylindrical algebraic decomposition *and has a doubly-exponential runtime [Basu et al., 2007, Sturmfels, 2002]. However, if the coefficients are limited to rational numbers instead of real numbers, then a subexponentail algorithm is provided in [Grigor'ev and Vorobjov, 1988]*[‖]*. Hence, Step 4 of* StrongInvSynth *has subexponential runtime in theory.*

Lemma 6.1 (Soundness). *Every output of* StrongInvSynth *is an inductive invariant. More generally, for every solution* $\xi \in \Xi$ *obtained in Step 4, the function* $\eta_\xi := \eta[s_{\ell,i,j} \leftarrow \xi(s_{\ell,i,j})]$ *is an inductive invariant.*

Proof. The valuation ξ satisfies the system of quadratic constraints obtained in Step 3. Hence, for every constraint pair $(\Gamma, g) \in \Lambda$, $g[s_{\ell,i,j} \leftarrow \xi(s_{\ell,i,j})]$ can be written in the form (†). Hence, we have $\xi \models (\Gamma, g)$. By definition of Step 2, this is equivalent to η_ξ having the initiation and consecution properties and hence being an inductive invariant. □

[‖]No tight runtime analysis is available for this algorithm, but [Grigor'ev and Vorobjov, 1988] proves that its runtime is subexponential.

We now prove our completeness result. Our approach is semi-complete for bounded reals in the sense of [Chatterjee et al., 2016a]. Concretely, this means that if we assume the bounded reals model of computation (see Section 6.3), then any valid inductive invariant can be found by our approach so long as the technical parameter Υ is large enough. Recall that Υ is a bound on the degree of the sum-of-square polynomials (see Remark 6.2).

Lemma 6.2 (Semi-completeness with Compactness). *If the pre-condition* Pre *satisfies the compactness condition of Theorem 2.2, i.e. if in every label* ℓ, $\mathsf{Pre}(\ell)$ *contains an atomic proposition of the form* $g \geq 0$ *such that the set* $\{\nu \in \mathbb{R}^{\mathbf{V}} \mid g(\nu) \geq 0\}$ *is compact, then for every inductive invariant* Ind *that has the form of the template* η*, there exists a natural number* Υ_{Ind}*, such that for every technical parameter* $\Upsilon \geq \Upsilon_{\mathsf{Ind}}$*, the invariant* Ind *corresponds to a solution of the system of quadratic equalities obtained in Step 3 of* $\mathsf{StrongInvSynth}$.

Proof. Let Ind be an inductive invariant in the form of the template η. We denote the value of $s_{\ell,i,j}$ in Ind by $\xi(s_{\ell,i,j})$. Given that Ind satisfies initiation and consecution, the valuation ξ satisfies every constraint pair (Γ, g) generated in Step 2. Each such Γ contains an assertion $g_i \geq 0$ such that $\{x \in \mathbb{R}^{\mathbf{V}} \mid g_i(x) \geq 0\}$ is compact. Hence, by Corollary 2.3, g can be written in the form (†)[**] and for large enough Υ, there exists a solution to the system that maps each $s_{\ell,i,j}$ to $\xi(s_{\ell,i,j})$. □

Remark 6.4 (Bounded Reals, Compactness and Real-world programs). *Note that in the bounded reals model of computation, every pre-condition enforces that the value of every variable is between* $-c$ *and* c *and also contains the polynomial inequality* $\left\|\mathbf{V}^f\right\|_2^2 \leq c^2 \cdot |\mathbf{V}^f|$ *(see Section 6.3). The set of valuations that satisfy the latter polynomial are points in* $\mathbb{R}^{\mathbf{V}}$ *whose distance from the origin is at most a fixed amount* $c \cdot \sqrt{|\mathbf{V}|}$*. Hence, this set is closed and bounded and therefore compact, and satisfies the requirement of Putinar's positivstellensatz. So, our approach is semi-complete for bounded reals.*

It is worth mentioning that almost all real-world programs have bounded variables. For example, programs that use floating-point variables can at most store a finite number of

[**]Theorem 2.2 requires compactness and so does Corollary 2.3.

values in each variable, hence their variables are always bounded[††]*. Also, note that while the completeness result is dependent on bounded variables, our soundness result holds for general unbounded real variables.*

Remark 6.5 (Non-strict inequalities)**.** *Although we considered invariants consisting of inequalities with positivity witnesses, our algorithm can easily be extended to generate invariants with non-strict inequalities, i.e. invariants of the form $\bigwedge(g(x) \geq 0)$. To do so, it suffices to replace Equation* (†) *in Step 3 of the algorithm with Equation (2.3), i.e. remove the ϵ-variables (positivity witnesses). This results in a sound, but not complete, method for generating non-strict polynomial invariants. An alternative approach, with a more in-depth use of the sätze, can ensure semi-completeness. See Chapter 7 for details.*

Remark 6.6 (Complexity)**.** *It is straightforward to verify that Steps 1–3 of* StrongInvSynth *have polynomial runtime. Hence, our algorithm provides a polynomial reduction from the Strong Invariant Synthesis problem to the problem of finding representative solutions of a system of quadratic equalities. As mentioned earlier, this problem is solvable in subexponential time [Grigor'ev and Vorobjov, 1988]. Hence, the runtime of our approach is subexponential, too. Note that we consider d and Υ to be fixed constants.*

Theorem 6.1 (Strong Invariant Synthesis)**.** *Given a transition system $S = (\mathbf{V}, \mathbf{L}, \ell_0, I, \theta)$ and a pre-condition* Pre *that satisfies the compactness condition, the* StrongInvSynth *algorithm solves the Strong Invariant Synthesis problem in subexponential time. This solution is sound and semi-complete with respect to the technical parameter Υ.*

Remark 6.7 (Inefficiency)**.** *Despite its subexponential runtime, the algorithm of [Grigor'ev and Vorobjov, 1 has a poor performance in practice [Hong, 1991]. Hence, Theorem 6.1 can only be considered as a theoretical contribution and is not applicable to real-world programs.*

6.4.3 Weak Invariant Synthesis and Practical Method

Due to the practical inefficiency mentioned in Remark 6.7, in this section we focus on using a very similar approach to reduce the Weak Invariant Synthesis problem to QCLP. Given

[††]To obtain a more realistic model of floating-point variables, one should also introduce constraints that ensure a variable cannot hold an arbitrarily small non-zero value. However, these details are beyond the scope of the current thesis, which focuses on real variables.

that there are many industrial solvers capable of handling real-world instances of QCLP, this reduction will provide a practical sound and semi-complete method for polynomial invariant generation. We now provide an algorithm for the Weak Invariant Synthesis problem. This is very similar to StrongInvSynth, so we only describe the differences.

The WeakInvSynth Algorithm. Our algorithm WeakInvSynth takes the same set of inputs as StrongInvSynth, as well as an objective function obj over the resulting inductive invariants. We assume that obj is a linear or quadratic polynomial over the s-variables in the template. Intuitively, obj serves as a measure of desirability of a synthesized invariant and the goal is to find the most desirable invariant.

The first three steps of the algorithm are the same as StrongInvSynth. The only difference is in Step 4, where WeakInvSynth needs to find only one solution for the computed system of quadratic equalities, i.e. the solution that maximizes obj. Hence, Step 4 is changed as follows:

Step 4) Finding the optimal solution. Step 3 has generated a system of quadratic equalities. In this step, the algorithm uses a QCLP-solver to find a solution ξ of this system that maximizes the objective function obj. It then outputs the inductive invariant $\eta_\xi := \eta[s_{\ell,i,j} \leftarrow \xi(s_{\ell,i,j})]$.

Example 6.6. *In Example 6.1, we mentioned that our goal is to prove that the value of s at the end of the function is less than $0.5 \cdot n^2 + 0.5 \cdot n + 1$, i.e. we want to obtain*

$$0.5 \cdot n^2 + 0.5 \cdot n + 1 - r > 0 \qquad (*)$$

at the endpoint label 7 of sum. To do so, our algorithm calls a QCLP-solver over the system of quadratic constraints obtained in Example 7.9, with the objective of minimizing the Euclidean distance between the coefficients synthesized for $\eta(7)$ and those of $(*)$. The QCLP-solver obtains a solution ξ (i.e. a valuation to the new unknown $s-$, $t-$ and $\epsilon-$variables), such that $\eta(7)[s_{7,i,j} \leftarrow \xi(s_{7,i,j})] = 0.5 \cdot n^2 + 0.5 \cdot n + 1 - r > 0$, hence proving the desired invariant. The complete solution is provided in [Chatterjee et al., 2020a].

Remark 6.8 (Form of the Objective Function)**.** *At first sight, the objective functions considered above might seem rather bizarre, given that they are functions of the unknown s-variables,* *i.e. the coefficients of the invariant which should be synthesized by the algorithm. We remark two points:*

- *In our view, this is the most useful formulation. Note that in many cases, such as the example above and our experimental results in Section 6.5, the goal of a verification process is to prove that a certain desired invariant* $\mathsf{Inv}(\ell)$ *holds at a specific point* ℓ *of the program. This goal can be specified as an objective function over the s-variables. However, it does not simplify the invariant generation problem, because although* $\mathsf{Inv}(\ell)$ *is given, in order to prove that it is really an invariant, the algorithm has to generate an inductive invariant at every other point of the program, too.*

- *Our approach does not depend on the form of objective functions, hence the objective can be any other linear or quadratic function (possibly depending on other variables) and our results will remain intact. It can even be a non-quadratic function, in which case the reduction would be to general quadratically-constrained optimization.*

Remark 6.9 (QCLP). *QCLP is a well-studied optimization problem [Chen et al., 2017, Linderoth, 2005]. It is NP-hard, but there are many efficient solvers for handling its real-world instances [Andersen and Andersen, 2018, Rothberg et al., 2018, IBM, 2019, Vanderbei, 2006]. These scalable solvers have been successfully applied to real-world verification problems, such as solving large POMDPs [Amato et al., 2007].*

Note that Lemmas 6.1 and 6.2 (Soundness and Completeness) carry over to this case without any modification, so we have the following theorem:

Theorem 6.2 (Weak Invariant Synthesis). *Given a transition system* $S = (\mathbf{V}, \mathbf{L}, \ell_0, I, \mathbf{\Theta})$*, a pre-condition* Pre *that satisfies the compactness condition and a linear/quadratic objective function* obj*, the* $\mathsf{WeakInvSynth}$ *algorithm reduces the Weak Invariant Synthesis problem to QCLP in* polynomial time*. This reduction is sound and semi-complete with respect to the technical parameter* Υ*.*

Benchmark	**n**	**d**	**|V|**	**|Sys|**	**Runtime**
cohendiv	1	1	6	622	15.236s
divbin	1	1	5	738	5.399s
hard	1	2	6	8324	27.952s
mannadiv	1	2	5	2561	18.222s
wensely	1	2	7	9422	20.051s
sqrt	1	2	4	2030	5.808s
dijkstra	1	2	5	5072	12.776s
z3sqrt	1	2	6	4692	12.944s
freire1	1	2	3	1210	26.474s
freire2	1	2	4	1016	10.670s
euclidex1	1	2	11	11191	1m37.493s
euclidex2	1	2	8	11156	39.323s
euclidex3	1	2	13	36228	3m23.110s
lcm1	1	2	6	6589	17.851s
lcm2	1	2	6	6176	18.714s
prodbin	1	2	5	5038	12.125s
prod4br	1	2	6	10522	43.205s
cohencu	1	2	5	3424	11.778s
petter	1	2	3	1080	20.390s

Table 6.2: Experimental results over the benchmarks of [Rodríguez-Carbonell, 2018]. $|\mathbf{V}|$ is the number of program variables and $|\mathbf{Sys}|$ is the size of the quadratic system, i.e. number of (in)equalities.

6.5 Experimental Results

Implementation. We implemented our algorithms for weak invariant generation in Java and used the LOQO optimizer [Vanderbei, 2006] for solving the QCLPs. All results were obtained on an Intel Core i5-7200U machine with 6 GB of RAM, running Ubuntu 18.04. See our technical report [Chatterjee et al., 2020a] for more details.

Results. We used the benchmarks in [Rodríguez-Carbonell, 2018], which contain programs, pre-conditions, and the desired partial invariants (invariants at a few points of the program) that are needed for their verification. We ignored benchmarks that contained non-polynomial assignments/pre-conditions. The results are summarized in Table 6.2. Our algorithm is not complete for *non-strict* invariants (Remark 6.5), but it could successfully and accurately generate all the desired invariants for these benchmarks.

Runtimes. Our runtimes over these benchmarks are typically under a minute, while the maximum runtime is close to 3.5 minutes. This shows that our approach is applicable in practice and does not suffer from the same impracticalities as [Grigor'ev and Vorobjov, 1988], which would take years on problems of this size [Hong, 1991].

Comparison with Complete Approaches. Almost none of the previous complete approaches are applicable to our benchmarks due to the existence of non-linear assignments and also because the desired invariants are polynomial *inequalities* (See Table 6.1). The only previous complete approach that handles polynomial programs and polynomial inequalities in invariants is [Kapur, 2006]. However, it relies on quantifier elimination and is extremely inefficient. We confirmed this point experimentally. We manually created the constraints of [Kapur, 2006] for our benchmarks [‡‡] and used state-of-the-art quantifier elimination / SMT solver tools (Mathematica [Wolfram Research, 2020], QEPCAD [Brown, 2010] and Z3 [De Moura and Bjørner, 2008]) to solve them. In all cases, the solver either did not terminate in 12 hours or returned with failure. This was the case even for our simple running example (Figure 6.1).

Comparison with Incomplete Approaches. Our approach is much slower than previous sound approaches that do not provide any completeness guarantee, e.g. [Farzan and Kincaid, 2015]. Hence, there is currently a trade-off between accuracy (completeness guarantees) and efficiency. While the semi-completeness guarantee is a key novelty of our approach, we expect that advancements in quadratic programming, which is an active research topic in optimization, will narrow the runtime gap.

Given that our approach has semi-completeness guarantees (over bounded reals), it is no surprise that it can generate desired polynomial invariants for inputs which no previous incomplete approach could handle. In our technical report [Chatterjee et al., 2020a], we present a classical example of a program that approximates $\sqrt{2}$ using its continued fraction representation. Our implementation generates required invariants of degree 5, which, to the best of our knowledge, is beyond the reach of all previous methods. Specifically, we tried all the incomplete approaches in Table 6.1. They are either not applicable to this example or

[‡‡]Following [Kapur, 2006], we only created templates at cutpoints and endpoints.

fail to synthesize the desired invariant. Our approach can also handle recursive benchmarks that are beyond the reach of all previous methods. See [Chatterjee et al., 2020a] for more details and examples.

7

Reachability Analysis for Polynomial Programs

This chapter is based on the following unpublished work:

[•] Asadi, A., Chatterjee, K., Fu, H., **Goharshady, A. K.**, and Mahdavi, M. **Inductive Reachability Witnesses**. *arXiv preprint arXiv:2007.14259*, 2020.

7.1 Introduction

Outline. In this chapter, we consider the fundamental problem of reachability analysis over polynomial imperative programs with real variables. The reachability property requires that a program can reach certain target states during its execution. Previous works that tackle reachability analysis are either unable to handle programs consisting of general loops (e.g. symbolic execution), or lack completeness guarantees (e.g. abstract interpretation), or are not automated (e.g. incorrectness logic/reverse Hoare logic). In contrast, we propose a novel approach for reachability analysis that can handle general programs, is (semi-)complete, and can be entirely automated for a wide family of programs. Our approach extends techniques from both invariant generation and ranking-function synthesis to reachability analysis through the notion of *(Universal) Inductive Reachability Witnesses* (IRWs/UIRWs). While traditional invariant generation uses over-approximations of reachable states, we consider the natural dual problem of under-approximating the set of program states that can reach a target state. We then apply an argument similar to ranking functions to ensure that all states in our under-approximation can indeed reach the target set in finitely many steps.

Our Results. On the theoretical level, we first show that our IRW/UIRW-based approach is sound and complete for reachability analysis of imperative programs. Then, we focus on linear and polynomial programs and develop automated methods for synthesizing linear and polynomial IRWs/UIRWs. In the linear case, our techniques are based on Farkas' lemma. For the polynomial case, our approach utilizes Handelman's Theorem, Hilbert's Nullstellensatz and Putinar's Positivstellensatz. To the best of our knowledge, such a combination of these theorems to obtain algorithms for program analysis is a novel contribution. On the practical side, our experimental results show that our automated approaches can efficiently prove complex reachability objectives over standard benchmarks.

Reachability. *Reachability analysis* is a basic and fundamental problem in computer science, starting from the halting problem of Turing machines [Turing, 1936]. It is a core problem in program verification as it aims at checking whether states with certain properties can be reached during the execution of a program. It also constitutes the most basic liveness property and has been widely studied as a fundamental problem in program analysis and model checking [Pnueli, 1977, Manna and Pnueli, 2012, Floyd, 1993, Hoare, 1969, Clarke et al., 2018]. The target states considered in reachablity analysis can be either *desirable* so that reachability to these states should be guaranteed, or *undesirable* so that the goal is to find an execution path leading to an unwanted behavior, hence proving incorrectness of the system. As mentioned, reachability to desirable states encodes the most basic type of liveness property. Reachability to undesirable states is also ubiquitous in verification problems and useful when one needs to identify realistic bugs in software implementations (see e.g. [O'Hearn, 2020]). Indeed, in real-world software development, most bugs are identified by finding an execution path that leads to a specific error [Distefano et al., 2019, Godefroid, 2007, Majumdar and Sen, 2007]. This is the idea that led to developments such as incorrectness logic [O'Hearn, 2020].

Previous Works on Formal Models. A large body of research on reachability analysis is conducted over formal models [Clarke et al., 2018], such as finite-state systems [Baier and Katoen, 2008, Chapter 3–6], pushdown automata [Walukiewicz, 2001], Petri nets [Mayr, 1981, Czerwinski et al., 2019, Atig and Ganty, 2011, Darondeau et al., 2012] and timed automata [Alur and Dill, 1990]. For these models, precise decidability and complexity results are attained. Moreover, numerous efficient algorithms have been developed to automate reachability analysis over these models (See [Baier and Katoen, 2008] for a comprehensive overview). Although the formal models above serve as an important abstraction mechanism for realistic systems, the techniques for reachability analysis over thems cannot be applied directly to imperative programs, in particular with real-valued variables. This is because the values taken by the variables in a program typically come from an infinite, even uncountable, domain and the underlying program structure might be irregular, i.e. in many cases a given piece of program code cannot be directly translated into any of the formal models above.

Reachability in Software Model Checkers. Many of the most successful software model checkers rely heavily on reachability analysis [Beyer and Keremoglu, 2011, Beyer et al., 2007, Ball and Rajamani, 2002, Holzmann, 1997]. Notably, the BLAST project [Beyer et al., 2007] describes itself as "a verification tool for the C language that solves the reachability problem".

Moreover, even when considering the verification of safety properties, all approaches and tools based on Counterexample-Guided Abstraction Refinement (CEGAR) [Clarke et al., 2000, Alur et al., 1995, Balarin and Sangiovanni-Vincentelli, 1993, Gurfinkel et al., 2006], including SLAM [Ball et al., 2011, Ball and Rajamani, 2002] and BLAST [Beyer et al., 2007], need to constantly perform reachability analyses to obtain their counterexamples. These model checkers rely on predicate abstraction refinement and, assuming that we require the analysis to terminate in finite time, can guarantee completeness when the variables have finite domains [Henzinger et al., 2002], but do not provide such guarantees for programs with real-valued variables.

Previous Works on (Imperative) Programs. When considering imperative programs, the reachability problem, and in particular the special case of termination analysis, has been widely studied over the past decades. There are several relevant categories of previous work, including symbolic execution [Cadar and Sen, 2013], termination analysis [Floyd, 1993], abstract interpretation [Cousot and Cousot, 1977] and recent results on incorrectness logic/reverse Hoare logic [O'Hearn, 2020, de Vries and Koutavas, 2011].

- *Symbolic execution* runs program codes statically in a symbolic fashion, and is thus effective for programs without general unbounded loops. For programs with loops, symbolic execution can only unfold the loop up to a bounded depth, and hence cannot handle general loops with an unbounded number of iterations. This point is also applicable to other approaches that rely on loop unrolling, such as [Albarghouthi et al., 2012a].

- *Termination analysis* is a special kind of reachability that requires the program to reach the terminal program counter, which is usually guaranteed by well-foundedness reasoning such as (lexicographic) ranking functions (See Chapter 8 for an overview). Termination analyses do not consider reachability to target program states defined through numerical constraints over program variables.

- *Abstract interpretation* is mainly used to generate over-approximations of reachable states (i.e. certain states *may* be reached), but there are also several abstraction-based approaches that compute under-approximations [Giacobazzi et al., 2000, Ranzato, 2013, Rival, 2005, Albarghouthi et al., 2012a]. However, they cannot provide guarantees of completeness except in specific special cases [Giacobazzi and Ranzato, 1997].

- Finally, *incorrectness logic* [O'Hearn, 2020] is a sound and complete logic that is similar to Hoare logic but performs under-approximation for reachable program states. A disadvantage of incorrectness logic, much like Hoare logic, is that it requires a considerable amount of manual effort for writing assertions, and cannot be directly automated.

Previous Works on Invariants. It is noteworthy that an *invariant* is, in a sense, a dual notion of reachability, and invariant generation is also prominent in the PL literature. Informally, an invariant is an over-approximation of the set of reachable states that can be used to prove safety properties over programs. Invariant generation has been a central research area in program analysis and verification, and many efficient approaches are present. See Chapter 6 for a more in-depth treatment of invariant generation.

Our Focus. In this chapter, we consider reachability analysis over imperative programs. We study the problem of automatically verifying that a set of target program states can be reached in program execution. While invariants provide an over-approximation of the set of reachable states, we consider their natural dual, i.e. under-approximations of the set of states that can reach a target. We consider programs with non-determinism and distinguish between *existential* and *universal* reachability. *Existential* reachability is the more classical and useful notion and, intuitively speaking, requires that target states are reachable under *some* resolution of the non-deterministic choices in the program. In contrast, *universal* reachability requries the program to reach the target states no matter how the non-determinism is resolved. Our main focus is on existential reachability, but our results generalize to the universal case, as well.

Our approach. Our methods are based on constraint solving, and extend ideas from both ranking functions and inductive invariant generation to cover the reachability problem. Informally, we use techniques from inductive invariant generation to capture a subset $\mathfrak{T}^{\diamond}$ of program states from which the execution steps of the program will either reach our target states or stay in $\mathfrak{T}^{\diamond}$ itself. Simultaneously, we use arguments similar to ranking functions to ensure that every state in $\mathfrak{T}^\diamond$ can reach a target state in finitely many steps. As mentioned above, the key distinction between our method and invariant generation approaches is that our set $\mathfrak{T}^\diamond$ is an *under-approximation* of the set of states that can eventually reach a target state, whereas invariants are, by definition, *over-approximations* of reachable states.

Our Contributions. We propose a novel approach for reachability analysis over programs. In detail, we have the following contributions:

- We propose the novel notion of Inductive Reachability Witnesses (IRWs) for existential reachability, which consists of a state set $\mathfrak{T}^\diamond$ of program states and a ranking function f over $\mathfrak{T}^\diamond$. The state set $\mathfrak{T}^\diamond$ satisfies certain invariant-like conditions. The ranking function f serves as a proof that every state in $\mathfrak{T}^\diamond$ can indeed reach a target. We also propose the notion of Universal Inductive Reachability Witnesses (UIRWs), the counterpart of IRWs for the universal case.
- From a theoretical point-of-view, we show that IRWs and UIRWs are sound and complete for proving existential and universal reachability, respectively.
- We follow use Farkas' Lemma, Putinar's Positivstellensatz, and Handelman's Theorem for automatically synthesizing linear and polynomial IRWs/UIRWs. However, we face new challenges regarding satisfiability in the polynomial case and address them with methods based on Hilbert's Strong Nullstellensatz. To the best of our knowledge, this combination (especially Section 7.4.3 and Theorem 2.5) is a novel contribution to constraint-based analysis of polynomial programs. Moreover, it is noteworthy that our synthesis method is complete in the linear case and semi-complete in the polynomial case.
- We show that our completeness results also pay off in practice. We provide experimental results over standard linear benchmarks from SV-COMP 2020 [Beyer, 2020]. The results show that in reachability analysis of linear programs, our approach beats every model checker that participated in the competition. Moreover, we present several examples of polynomial programs for which the champions of SV-COMP 2020 fail to prove reachability. In contrast, our approach can successfully handle these programs.

Novelty. Our technical novelty is two-fold. First, we define the sound and complete notions of IRW/UIRW for proving reachability. Second, to capture a large class of imperative programs, we build an automated approach over linear/polynomial transition systems, or equivalently flowchart programs [Alias et al., 2010], where transitions between states with affine/polynomial updates are allowed. We provide (semi-)complete synthesis algorithms based on Farkas' Lemma, Putinar's Positivstellensatz, Handelman's Theorem and Hilbert's Strong Nullstellensatz. While these theorems have previously been used for termination analysis and invariant generation [Colón et al., 2003], their application in the context of reachability analysis is novel. Moreover, our combination of Nullstellensätze and Positivstellensätze to obtain program analysis algorithms (see Section 7.4.3 and Theorem 2.5) is entirely novel and had not previously been considered even in termination analysis or invariant generation.

7.2 Inductive Reachability Witnesses

In this section, we provide the basic definitions needed for reachability analysis, formalize our problems, and introduce the concept of Inductive Reachability Witnesses (IRWs/UIRWs). Finally, we show that IRWs/UIRWs are sound and complete for proving reachability. In the sequel, we use transition systems with real variables, as defined in Section 2.5 to model the programs we are studying. Let $S = (\mathbf{V}, \mathbf{L}, \ell_0, I, \mathbf{\Theta})$ be a transition system and Σ its set of states. We consider two types of reachability: existential and universal.

Existential Reachability. A set $\mathfrak{T} \subseteq \Sigma$ is called *existentially reachable* or simply *reachable* if there exists an integer n and a run $\mathsf{r} = \{\sigma_i, \theta_i\}_{i=0}^{\infty}$ such that $\sigma_n \in \mathfrak{T}$. In other words, $\mathfrak{T}$ is reachable if there exists a run that visits $\mathfrak{T}$.

Informally, assuming that $\mathfrak{T}$ is a set of undesirable states, e.g. states that lead to a certain error that we would like to avoid, the definition above models the cases when the non-determinism is *demonic* [Back and Wright, 2012], i.e. whenever it is possible to choose among multiple transitions, the choice is made in favor of reaching the undesirable set $\mathfrak{T}$. However, we can also consider reachability in presence of *angelic* non-determinism [Bodik et al., 2010], i.e. when the choices are made in favor of *not* reaching the undesirable set $\mathfrak{T}$. The words "angelic" and "demonic" can have the opposite meaning when considering cases in which $\mathfrak{T}$ is a set of desirable states. Therefore, to prevent confusion, we use the terms "existential" and "universal".

Universal Reachability. A set $\mathfrak{T} \subseteq \Sigma$ is called *universally reachable* if there exists a valuation $\nu_0 \in \mathbb{R}^{\mathbf{V}}$ and an integer n, such that (i) $\nu_0 \models I$, and (ii) every run $\mathsf{r} = \{(\ell_i, \nu_i), \theta_i\}_{i=0}^{\infty}$ visits $\mathfrak{T}$ in its first n steps. In other words, for each such r, there exists an index $i \leq n$ such that $(\ell_i, \nu_i) \in \mathfrak{T}$.

Intuitively, the definition above requires that we can fix an initial valuation for the program such that no matter how the non-determinism is resolved, the execution is forced to visit $\mathfrak{T}$ after at most n steps. In this work, our primary focus is on existential reachability. However, our results extend to universal reachability as well.

Example 7.1. *Consider the system in Figure 2.6 (right), and let $\mathfrak{T} = \{(d, \nu) \mid \nu \in \mathbb{R}^{\mathbf{V}}\}$. In this case, reaching $\mathfrak{T}$ is equivalent to the termination of the program in Figure 2.6 (left). Note that $\mathfrak{T}$ is existentially reachable, i.e. there are runs of the system that reach label d, for example the following:*

$$(a, 0, 0, 0) \xrightarrow{\theta_1} (b, 0, 0, 0) \xrightarrow{\theta_4} (a, 1, 2, 0) \xrightarrow{\theta_3} (d, 1, 2, 0) \to \dots .$$

It is also universally reachable, because every execution starting from $(a, 1, 2, 3)$ will reach d in a single step. As another example, consider the target set $\mathfrak{T}' = \{(d, \nu) \mid \nu(x) < 0\}$. This corresponds to reaching d (ending the program) with a negative value for x. This time, the set $\mathfrak{T}'$ is existentially reachable, for example through the following run:

$$(a, 0, 0, 0) \xrightarrow{\theta_2} (c, 0, 0, 0) \xrightarrow{\theta_5} (a, 0, 0, -1) \xrightarrow{\theta_2} (c, 0, 0, -1) \xrightarrow{\theta_5} (a, -1, -1, -2) \xrightarrow{\theta_2} (c, -1, -1, -2) \xrightarrow{\theta_5}$$
$$(a, -3, -3, -3) \xrightarrow{\theta_1} (b, -3, -3, -3) \xrightarrow{\theta_4} (a, -2, -1, -3) \xrightarrow{\theta_3} (d, -2, -1, -3) \to \dots,$$

but it is not universally reachable. To see this, note that if an initial value satisfies $x < y$, then it does not enter the while loop at all, and hence when it reaches d it satisfies $x \geq 0$ (the initial condition). On the other hand, if an initial value satisfies $x \geq y$, there is a run that always chooses the transition θ_2 when at a, and hence never reaches $\mathfrak{T}'$.

We now look into proof concepts for universal and existential reachability. The constructs we define for proving reachability are a mixture of inductive sets, which are often used for proving invariants [Colón et al., 2003, Manna and Pnueli, 2012], and ranking functions [Floyd, 1993], which are the classical method for proving termination.

$\mathfrak{T}$-inductive Sets. Given a set $\mathfrak{T} \subseteq \Sigma$ of target states, a set $\mathfrak{T}^\diamond \subseteq \Sigma$ is called $\mathfrak{T}$-*inductive* if for every $\sigma \in \mathfrak{T}^\diamond \setminus \mathfrak{T}$, *there exists* a successor σ' of σ such that $\sigma' \in \mathfrak{T}^\diamond$.

Intuitively, if $\mathfrak{T}^\diamond$ is $\mathfrak{T}$-inductive, then if we start the execution of the program from a state in $\mathfrak{T}^\diamond$, there exists a way for resolving the non-determinism so that we either reach $\mathfrak{T}$ or can inductively prove that we will never leave $\mathfrak{T}^\diamond$.

Example 7.2. *Consider the system in Figure 2.6 and let* $\mathfrak{T} = \{(d, \nu) \mid \nu(x) < 0\}$, *i.e. the target is reaching d with x having a negative value. Let* $\mathfrak{T}^\diamond := \{(\ell, \nu) \mid \ell \in \mathbf{L}, \nu \in \mathbb{R}^{\mathbf{V}}, \nu \models A_\ell\}$ *be the set of states satisfying the following assertions:*

ℓ	A_ℓ
a	$x, y, z \leq 0 \ \wedge\ (x - y) \cdot (x - y + 1) = 0$
b	$x \leq -2 \ \wedge\ y, z \leq 0 \ \wedge\ x = y$
c	$x, y, z \leq 0 \ \wedge\ x = y$
d	$x < 0$

Then, we can verify that $\mathfrak{T}^\diamond$ *is a* $\mathfrak{T}$*-inductive set. Concretely, consider a state* $(a, \nu_a) \in \mathfrak{T}^\diamond$. *In other words,* $\nu_a \models A_a$. *In such a state, we have* $(x - y) \cdot (x - y + 1) = 0$. *Therefore, either* $\nu_a(x) = \nu_a(y)$ *or* $\nu_a(x) = \nu_a(y) - 1$. *In the former case, we can take transition* θ_2, *and it is easy to verify that the new state satisfies* A_c, *hence there is a successor that is also in* $\mathfrak{T}^\diamond$. *In the latter case, we can take* θ_3 *and reach d with a valuation that satisfies* $x < 0$, *because* $\nu_a \models (y \leq 0 \ \wedge\ x = y - 1)$. *Similarly, if* $(b, \nu_b) \in \mathfrak{T}^\diamond$, *we know that* $\nu_b \models (x \leq -2 \ \wedge y, z \leq 0 \ \wedge\ x = y)$. *Therefore, taking the transition* θ_4, *corresponding to the update* $(x, y) := (x+1, y+2)$, *leads to a state in a that satisfies* $(x, y, z \leq 0 \wedge x = y-1)$. *Note that* $x = y - 1 \Rightarrow (x - y) \cdot (x - y + 1) = 0$, *therefore* A_a *is satisfied and we have a successor in* $\mathfrak{T}^\diamond$. *It is easy to verify the same property at c. Finally, if we have a state* $(d, \nu_d) \in \mathfrak{T}^\diamond$, *by definition of* $\mathfrak{T}$ *and* A_d, *we know that* $(d, \nu_d) \in \mathfrak{T}$, *and hence we do not need to find any successor for this state.*

In this example, if we start at an initial state that satisfies A_a, we can find a run of the system that either reaches $\mathfrak{T}$ or stays inside $\mathfrak{T}^\diamondsuit$. However, this is not enough for reachability to $\mathfrak{T}$. Such a run might stay inside $\mathfrak{T}^\diamondsuit$ forever without visiting $\mathfrak{T}$. For example, we can keep taking the transition θ_2 when at a, and hence never reach d. To avoid such a scenario, we need a $\mathfrak{T}$-ranking function.

$\mathfrak{T}$-ranking Functions. Given a $\mathfrak{T}$-inductive set $\mathfrak{T}^\diamondsuit$, a function $f : \mathfrak{T}^\diamondsuit \to [0, \infty)$ is called a $\mathfrak{T}$-*ranking function* with parameter $\epsilon > 0$, if for every $\sigma \in \mathfrak{T}^\diamondsuit \setminus \mathfrak{T}$, *there exists* a successor $\sigma' \in \mathfrak{T}^\diamondsuit$ of σ, for which we have $f(\sigma') \leq f(\sigma) - \epsilon$.

Inductive Reachability Witnesses (IRWs). Given a set $\mathfrak{T}$ of target states in a system $S = (\mathbf{V}, \mathbf{L}, \ell_0, I, \mathbf{\Theta})$, an *Inductive Reachability Witness* for $\mathfrak{T}$ is a tuple $(\mathfrak{T}^\diamondsuit, f, \epsilon)$ such that:

- $\mathfrak{T}^\diamondsuit$ is a $\mathfrak{T}$-inductive set;
- $\epsilon \in (0, \infty)$;
- $f : \mathfrak{T}^\diamondsuit \to [0, \infty)$ is a $\mathfrak{T}$-ranking function with parameter ϵ;
- There exists a valuation $\nu \in \mathbb{R}^{\mathbf{V}}$ such that $(\ell_0, \nu) \in \mathfrak{T}^\diamondsuit$ and $\nu \models I$.

Informally, an IRW serves as a proof of existential reachability for a target set $\mathfrak{T}$. The inductivity of $\mathfrak{T}^\diamondsuit$ ensures that starting from the initial state $(\ell_0, \nu) \in \mathfrak{T}^\diamondsuit$, we will never be forced to leave $\mathfrak{T}^\diamondsuit$ unless we reach $\mathfrak{T}$, while the existence of the $\mathfrak{T}$-ranking function f proves that we cannot avoid $\mathfrak{T}$ forever. It is also noteworthy that the $\mathfrak{T}$-inductive set $\mathfrak{T}^\diamondsuit$ is similar to an inductive invariant, but the main difference is that while an invariant is by definition a *superset* of all reachable states, a $\mathfrak{T}$-inductive set $\mathfrak{T}^\diamondsuit$ is a *subset* of those states from which we can reach the target set $\mathfrak{T}$. An IRW $(\mathfrak{T}^\diamondsuit, f, \epsilon)$ is called bounded if $\mathfrak{T}^\diamondsuit$ is bounded.

Example 7.3. *Consider the system in Figure 2.6, with the same target set as in Example 7.2, i.e.* $\mathfrak{T} = \{(d, \nu) \mid \nu(x) < 0\}$. *Let* $\mathfrak{T}^\diamondsuit := \{(\ell, \nu) \mid \nu \models A_\ell\}$ *and* $f(\ell, \nu) := f_\ell(\nu)$ *be defined as follows:*

ℓ	A_ℓ	f_ℓ
a	$-10 \leq x, y, z \leq 0 \ \wedge \ \left(x = y - 1 \vee x = y = \frac{-z\cdot(z+1)}{2}\right)$	$100 + x - y + z$
b	$-10 \leq x \leq -2 \ \wedge \ z \leq 0 \ \wedge \ x = y = \frac{-z\cdot(z+1)}{2}$	$99.5 + z$
c	$-2 \leq x \leq 0 \ \wedge \ z \leq 0 \ \wedge \ x = y = \frac{-z\cdot(z+1)}{2}$	$99.5 + z$
d	$x \leq -0.5$	*0*

Note that the A_ℓ's are more restrictive than in Example 7.2. We can verify that $\mathfrak{T}^\diamond$ is a $\mathfrak{T}$-inductive set in the same manner as in Example 7.2. We should also verify that f is a valid $\mathfrak{T}$-ranking function. Whenever we take either transition θ_1 or θ_2 (from a to b or c), we are assured that $x = y$, hence the value of f goes from $100+z$ to $99.5+z$ and decreases by 0.5. Also, because in A_a we have $-10 \leq x, y, z \leq 0$, the value of f at a is at least 80, and hence transition θ_3 (from a to d) decreases f by more than 0.5. Now consider transition θ_4 (from b to a). This transition does not change the value of z, but makes it so that $y = x + 1$. So it changes the value of f from $99.5+z$ to $99+z$. Note that transition θ_5 (from c to a), decreases z by 1 while keeping $x = y$. Hence, it decreases f by 0.5. Also, θ_6 (the self-transition from d to d) is irrelevant in this case, because our A_d entails inclusion in $\mathfrak{T}$. Finally, $(a, 0, 0, 0)$ is a state that satisfies both the initial condition I and A_a. Hence, we conclude that $(\mathfrak{T}^\diamond, f, 0.5)$ is an IRW for $\mathfrak{T}$.

We now define the counterpart of IRWs for universal reachability.

Universal $\mathfrak{T}$-inductive Sets. Given a set $\mathfrak{T} \subseteq \Sigma$ of target states, a set $\mathfrak{T}^\diamond \subseteq \Sigma$ is called *universally* $\mathfrak{T}$*-inductive* if for every $\sigma \in \mathfrak{T}^\diamond \setminus \mathfrak{T}$ and *every* successor σ' of σ, we also have $\sigma' \in \mathfrak{T}^\diamond$.

The idea behind universal $\mathfrak{T}$-inductive sets is that any execution of the program that starts in such a set $\mathfrak{T}^\diamond$ will either reach $\mathfrak{T}$ or one can prove using induction that it will never leave $\mathfrak{T}^\diamond$, no matter how the non-determinism is resolved.

Universal $\mathfrak{T}$-ranking Functions. Given a universal $\mathfrak{T}$-inductive set $\mathfrak{T}^\diamond$, a function $f : \mathfrak{T}^\diamond \to [0, \infty)$ is called a *universal* $\mathfrak{T}$*-ranking function* with parameter $\epsilon > 0$, if for every $\sigma \in \mathfrak{T}^\diamond \setminus \mathfrak{T}$ and *every successor* σ' of σ, we have $f(\sigma') \leq f(\sigma) - \epsilon$.

Universal Inductive Reachability Witnesses (UIRWs). Given a set $\mathfrak{T}$ of target states in a system $S = (\mathbf{V}, \mathbf{L}, \ell_0, I, \mathbf{\Theta})$, a *Universal Inductive Reachability Witness* for $\mathfrak{T}$ is a tuple $(\mathfrak{T}^\diamond, f, \epsilon)$ such that:

- $\mathfrak{T}^\diamond$ is a universal $\mathfrak{T}$-inductive set;
- $\epsilon \in (0, \infty)$;
- $f : \mathfrak{T}^\diamond \to [0, \infty)$ is a universal $\mathfrak{T}$-ranking function with parameter ϵ;
- There exists a valuation $\nu \in \mathbb{R}^\mathbf{V}$ such that $(\ell_0, \nu) \in \mathfrak{T}^\diamond$ and $\nu \models I$.

```
$I : i = s = 0 \ \wedge \ n \geq 0$

$a$:  while  $i \leq n$:
$b$:     $(s, i) := (s + 1, i + 1)$
$c$:     $\square$  $(s, i) := (s + 2, i + 1)$
$d$:
```

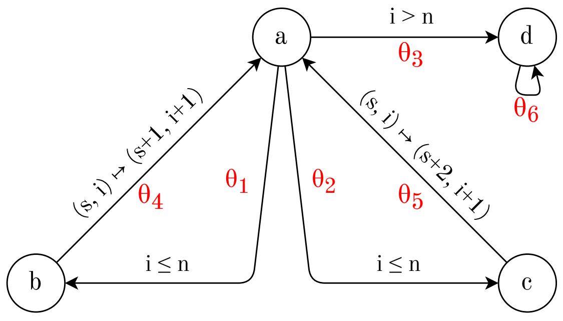


Figure 7.1: A Non-deterministic Program (left) and its Representation as a Transition System (right)

Example 7.4. *Figure 7.1 shows a simple program together with its representation as a transition system. Let* $\mathfrak{T} = \{(d, \nu) \mid \nu(s) \geq 20\}$*, i.e. the target is reaching point* d *with an* s *value of more than* 20*. Let* $\mathfrak{T}^\diamond := \{(\ell, \nu) \mid \nu \models A_\ell\}$ *and* $f(\ell, \nu) := f_\ell(\nu)$ *be defined as follows:*

ℓ	A_ℓ	f_ℓ
a	$n \geq 50 \ \wedge \ s \geq i \geq 0 \ \wedge \ n + 1 \geq i$	$n + 1.5 - i$
b	$n \geq 50 \ \wedge \ s, n \geq i \geq 0$	$n + 1 - i$
c	$n \geq 50 \ \wedge \ s, n \geq i \geq 0$	$n + 1 - i$
d	$s \geq 50$	0

It is easy to check that $(\mathfrak{T}^\diamond, f, 0.5)$ *is a UIRW for* $\mathfrak{T}$*. Intuitively, this guarantees that if a run starts with an initial valuation that satisfies* A_a*, it will definitely reach a target state.*

Remark 7.1. *Note that, as mentioned in Section 7.1, the inductive set $\mathfrak{T}^\diamond$ in Example 7.3 is an under-approximation of the desired states. In* existential *IRWs (such as Example 7.3), the set $\mathfrak{T}^\diamond$ is an under-approximation of the states from which there* exists *a way of resolving the non-determinism so that we eventually reach $\mathfrak{T}$. Similarly, in UIRWs, $\mathfrak{T}^\diamond$ under-approximates the set of states from which every execution of the program is forced to visit $\mathfrak{T}$. Hence, our $\mathfrak{T}$-inductive sets $\mathfrak{T}^\diamond$ are essentially natural duals of the notion of inductive invariants (Chapter 6).*

7.3 Basic Results and Linear/Polynomial Witnesses

Our approach for proving existential (resp. universal) reachability is based on synthesizing an IRW (resp. a UIRW). The reduction from reachability to witness synthesis is both sound and complete.

Theorem 7.1 (Soundness)**.** *Let $\mathfrak{T} \subseteq \Sigma$ be a set of states in the system S.*

(i) If there exists an IRW $(\mathfrak{T}^\diamond, f, \epsilon)$ for $\mathfrak{T}$, then $\mathfrak{T}$ is existentially reachable.

(ii) If there exists a UIRW $(\mathfrak{T}^\diamond, f, \epsilon)$ for $\mathfrak{T}$, then $\mathfrak{T}$ is universally reachable.

Proof. We handle each case separately.

(i) We construct a run of S that visits $\mathfrak{T}$. By definition of IRW, there exists a state $\sigma_0 = (\ell_0, \nu_0) \in \mathfrak{T}^\diamond$ such that $\nu_0 \models I$. We start our run with σ_0 and inductively find the next transitions and states as follows: when we are in a state $\sigma_i \in \mathfrak{T}^\diamond$, either (a) $\sigma_i \in \mathfrak{T}$ in which case the path until this point has already reached $\mathfrak{T}$ and we can extend it to an arbitrary run, or (b) $\sigma_i \in \mathfrak{T}^\diamond \setminus \mathfrak{T}$, in which case there exists a successor $\sigma_{i+1} \in \mathfrak{T}^\diamond$ of σ_i such that $f(\sigma_{i+1}) \leq f(\sigma_i) - \epsilon$, and we transition to σ_{i+1}. Using this procedure, it is not possible to avoid case (a) forever, because each application of (b) decreases the value of f by at least ϵ and f is bounded from below. Hence, the constructed run will reach $\mathfrak{T}$.

(ii) We choose $\sigma_0 = (\ell_0, \nu_0)$ as in the previous case. We now prove that every path of length $n := 1 + \lceil f(\sigma_0)/\epsilon \rceil$ starting from σ_0 will reach $\mathfrak{T}$. Let $\mathsf{r} = \sigma_0, \theta_0, \sigma_1, \theta_1, \ldots, \sigma_n$ be such a path. If no σ_i is in $\mathfrak{T}$, then by definition of universal $\mathfrak{T}$-inductiveness, every σ_i is in $\mathfrak{T}^\diamondsuit \setminus \mathfrak{T}$. So, for each i, we have $f(\sigma_{i+1}) \leq f(\sigma_i) - \epsilon$. Therefore, $f(\sigma_n) \leq f(\sigma_0) - n \cdot \epsilon = f(\sigma_0) - \epsilon - \lceil f(\sigma_0)/\epsilon \rceil \cdot \epsilon < 0$ which is a contradiction because f can only take non-negative values.

□

Theorem 7.2 (Completeness). *Let $\mathfrak{T} \subseteq \Sigma$ be a set of states in the system S.*

(i) If $\mathfrak{T}$ is existentially reachable, then there exists an IRW $(\mathfrak{T}^\diamondsuit, f, \epsilon)$ for $\mathfrak{T}$.

(ii) If $\mathfrak{T}$ is universally reachable, then there exists a UIRW $(\mathfrak{T}^\diamondsuit, f, \epsilon)$ for $\mathfrak{T}$.

Proof. In each case, we construct the required IRW/UIRW.

(i) Given that $\mathfrak{T}$ is reachable, by definition there exists a path $\pi = (\ell_0, \nu_0), \theta_0, \ldots, (\ell_n, \nu_n)$ such that $(\ell_n, \nu_n) \in \mathfrak{T}$ and $\nu_0 \models I$. Without loss of generality, we choose such a π that is prefix-minimal, i.e. that no prefix of π has the same properties. Let $\mathfrak{T}^\diamondsuit = \{(\ell_i, \nu_i) | 0 \leq i \leq n\}$, then $\mathfrak{T}^\diamondsuit$ is $\mathfrak{T}$-inductive, because $(\ell_n, \nu_n) \in \mathfrak{T}$ and for every $i \neq n$, the state (ℓ_i, ν_i) can be succeeded by (ℓ_{i+1}, ν_{i+1}). Let $f : \mathfrak{T}^\diamondsuit \to [0, \infty)$ be defined as follows: $f(\ell_i, \nu_i) := n - i$. It is easy to verify that $(\mathfrak{T}^\diamondsuit, f, 1)$ is an IRW for $\mathfrak{T}$.

(ii) We define $\Sigma_k \subseteq \Sigma$ as the set of all states such that every semi-path of length k starting in these states is guaranteed to visit $\mathfrak{T}$. Note that $\Sigma_0 = \mathfrak{T}$ and if $\sigma \in \Sigma_k \setminus \mathfrak{T}$, then by definition every successor σ' of σ must be in Σ_{k-1}. Let $\mathfrak{T}^\diamondsuit = \bigcup_{i=0}^{\infty} \Sigma_k$, and for every $\sigma \in \mathfrak{T}^\diamondsuit$, define $f(\sigma) := \min\{k \mid \sigma \in \Sigma_k\}$. It is easy to prove by definition-chasing that $(\mathfrak{T}^\diamondsuit, f, 1)$ is a UIRW.

□

Undecidability. Based on the two theorems above, synthesis of IRWs (UIRWs) is equivalent to proving existential (universal) reachability, which are undecidable problems according to Rice's theorem. Hence, whether an arbitrary input system S and target set $\mathfrak{T}$ have an IRW or a UIRW are undecidable problems, too. As such, in this chapter we consider linear or polynomial systems, with target sets that are defined by linear or polynomial inequalities, and focus on the problem of synthesizing linear or polynomial IRWs and UIRWs[*].

Linear/Polynomial Systems. A transition system $S = (\mathbf{V}, \mathbf{L}, \ell_0, I, \mathbf{\Theta})$ is called (d, k)-*polynomial* if

- I is a conjunction of at most k polynomial inequalities of degree at most d over $\mathbf{V}$, and
- for every $\theta = (\ell, \ell', \varphi, \mu) \in \mathbf{\Theta}$, the transition condition φ is a conjunction of at most k polynomial inequalities of degree at most d over $\mathbf{V}$, and
- for every $\theta = (\ell, \ell', \varphi, \mu) \in \mathbf{\Theta}$ and variable $v \in \mathbf{V}$, we have $\mu(v) \in \mathbb{R}[\mathbf{V}]$ and $\deg(\mu(v)) \leq d$, i.e. $\mu(v)$ is a polynomial of degree at most d over $\mathbf{V}$.

A $(1, k)-$polynomial system is also called $k-$linear.

Linear IRWs/UIRWs. An IRW/UIRW $(\mathfrak{T}^\diamond, f, \epsilon)$ is called $k-$linear if for every location $\ell \in \mathbf{L}:$

- The set $\mathfrak{T}^\diamond_\ell := \mathfrak{T}^\diamond \cap (\{\ell\} \times \mathbb{R}^\mathbf{V})$ is a closed polyhedron which is an intersection of at most k half-spaces. In other words, there exists a set A_ℓ of at most k non-strict linear inequalities over $\mathbf{V}$ such that a valuation ν satisfies A_ℓ iff $(\ell, \nu) \in \mathfrak{T}^\diamond$.
- The function $f_\ell : \textsc{Sat}(A_\ell) \to [0, \infty)$, defined as $f_\ell(\nu) = f(\ell, \nu)$, is a linear function over $\mathbf{V}$. Here, $\textsc{Sat}(A_\ell)$ is the set of all valuations that satisfy A_ℓ.

Polynomial IRWs/UIRWs. An IRW/UIRW $(\mathfrak{T}^\diamond, f, \epsilon)$ is called $(d, k)-$polynomial if for every $\ell \in \mathbf{L}:$

- The set $\mathfrak{T}^\diamond_\ell := \mathfrak{T}^\diamond \cap (\{\ell\} \times \mathbb{R}^\mathbf{V})$ is a closed semi-algebraic set defined by at most k non-strict polynomial inequalities of degree d or less. Equivalently, there exists a set A_ℓ of at most k non-strict polynomial inequalities of degree at most d over $\mathbf{V}$ such that $\nu \models A_\ell$ iff $(\ell, \nu) \in \mathfrak{T}^\diamond$.

[*]All these restrictions are necessary, e.g. termination is undecidable for polynomial programs [Bradley et al., 2005b].

- The function f_ℓ, defined as $f_\ell(\mathbf{v}) = f(\ell, \mathbf{v})$, is a polynomial of degree at most d over $\mathbf{V}$.

A $(d,k)-$polynomial IRW/UIRW is *explicitly bounded* if each set A_ℓ contains a polynomial inequality $g \geq 0$ such that $\textsc{Sat}(g \geq 0)$ is bounded.

7.4 Synthesis of Inductive Reachability Witnesses

We now provide sound and (semi-)complete algorithms for synthesizing linear or polynomial IRWs and UIRWs for linear and polynomial systems. We consider three variants of this problem: (i) when the system, the target set, and the desired IRW/UIRW are all $k-$linear (Section 7.4.1), (ii) when the system and the desired IRW/UIRW are $k-$linear, but the target set is $(d,k)-$polynomial (Section 7.4.2), and finally the most general case: (iii) when the system, the target set, and the IRW/UIRW to be synthesized are $(d,k)-$polynomial.

7.4.1 Linear IRWs/UIRWs for Linear Systems with Linear Target Sets

Problem Definition. In this section, we consider the following problem: Given a $k-$linear system $S = (\mathbf{V}, \mathbf{L}, \ell_0, I, \mathbf{\Theta})$, together with a set τ_ℓ of at most k non-strict linear inequalities at every location $\ell \in \mathbf{L}$, synthesize a $k-$linear IRW/UIRW for the target set $\mathfrak{T} := \cup_{\ell \in \mathbf{L}} \{\ell\} \times \textsc{Sat}(\tau_\ell)$ or report that no such IRW/UIRW exists. In the sequel, we assume $\mathbf{V} = \{v_1, \ldots, v_r\}$, and $\mathbf{L} = \{\ell_0, \ldots, \ell_n\}$.

Mathematical Tool. Our approach in this section is based on Farkas' Lemma (Lemma 2.2).

Overview of the Approach. Before presenting our algorithm in detail, we provide a high-level overview of its steps. Our algorithm consists of five steps:

- **Step 1.** The algorithm creates a template for the desired IRW/UIRW. Basically, it considers every expression that should be synthesized as part of an IRW/UIRW, i.e. the descriptions of $\mathfrak{T}^\diamond$ and f, and creates a template for it in which the coefficients are unknown variables whose value should be synthesized.

- **Step 2.** The algorithm generates a series of so-called "constraint pairs". These constraint pairs are of a specific form that is amenable to Farkas' Lemma. They encode the requirements that $\mathfrak{T}^\diamond$ should be a $\mathfrak{T}-$inductive set and that f should be a valid $\mathfrak{T}$-ranking function.

- **Step 3.** In this step, the algorithm applies Farkas' lemma to the constraints generated in Step 2 and translates them to an equivalent system of quadratic (in)equalities over the unknown *template* variables. It is noteworthy that after this step, no program variable appears in the quadratic constraints.

- **Step 4.** The algorithm adds a few additional constraints that ensure the existence of a valid initial valuation for the IRW/UIRW.

- **Step 5.** Finally, the algorithm solves the constraints by calling an off-the-shelf Quadratic Programming (QP) solver. It then plugs back the solution values reported for template variables into the templates generated in Step 1 to obtain the desired IRW/UIRW.

We now dive into the details of each step.

The Synthesis Algorithm. Our algorithm consists of the following five steps:

Step 1. Setting up a template. Consider a $k-$linear IRW/UIRW for reaching $\mathfrak{T}$ in S. It consists of a $k-$linear set $\mathfrak{T}^\diamond$, defined by a set A_ℓ of k linear inequalities at every location ℓ, and a linear function f, similarly defined by a linear expression f_ℓ at every location ℓ.

In this step, the algorithm sets up a symbolic *template* for each A_ℓ and f_ℓ. Concretely, it symbolically computes the following expressions, in which the $\widehat{c_{\ell,i,j}}$'s and $\widehat{d_{\ell,j}}$'s are unknown reals[†]:

$$\widehat{A_\ell} : \begin{cases} \widehat{c_{\ell,1,0}} + \widehat{c_{\ell,1,1}} \cdot v_1 + \ldots + \widehat{c_{\ell,1,r}} \cdot v_r \geq 0 \\ \vdots \\ \widehat{c_{\ell,k,0}} + \widehat{c_{\ell,k,1}} \cdot v_1 + \ldots + \widehat{c_{\ell,k,r}} \cdot v_r \geq 0 \end{cases}$$

[†]Throughout this chapter, we use the notation $\widehat{\cdot}$ to denote variables/expressions whose values should be synthesized by the algorithm.

$$\widehat{f_\ell} = \widehat{d_{\ell,0}} + \widehat{d_{\ell,1}} \cdot v_1 + \ldots + \widehat{d_{\ell,r}} \cdot v_r$$

Intuitively, the goal of the algorithm is to find suitable real values for the unknown coefficients (i.e. $\widehat{c_{\ell,i,j}}$'s and $\widehat{d_{\ell,j}}$'s) so that when we plug them into $\widehat{f_\ell}$'s and $\widehat{A_\ell}$'s, they yield a valid IRW/UIRW. Moreover, the algorithm defines a new unknown $\widehat{\epsilon}$, whose synthesized value will serve as the decrease parameter for f.

Example 7.5. *Consider the system in Figure 7.2. We will use this system as our running example and aim to synthesize a 2-linear IRW and a 2-linear UIRW for it. For the IRW case, suppose that the target set is $\mathfrak{T} = \{(4, \nu) \mid \nu \models (x \geq y + 8)\}$. For the UIRW case, we let $\mathfrak{T}' = \{(4, \nu) \mid \nu \models (x \geq y + 4)\}$. In this step, the algorithm generates a variable $\widehat{\epsilon}$ and the following templates:*

$$\widehat{A_1} : \begin{cases} \widehat{c_0} + \widehat{c_1} \cdot x + \widehat{c_2} \cdot y \geq 0 \\ \widehat{c_3} + \widehat{c_4} \cdot x + \widehat{c_5} \cdot y \geq 0 \end{cases} \quad \widehat{A_2} : \begin{cases} \widehat{c_6} + \widehat{c_7} \cdot x + \widehat{c_8} \cdot y \geq 0 \\ \widehat{c_9} + \widehat{c_{10}} \cdot x + \widehat{c_{11}} \cdot y \geq 0 \end{cases} \quad \widehat{A_3} : \begin{cases} \widehat{c_{12}} + \widehat{c_{13}} \cdot x + \widehat{c_{14}} \cdot y \geq 0 \\ \widehat{c_{15}} + \widehat{c_{16}} \cdot x + \widehat{c_{17}} \cdot y \geq 0 \end{cases}$$

$$\widehat{A_4} : \begin{cases} \widehat{c_{18}} + \widehat{c_{19}} \cdot x + \widehat{c_{20}} \cdot y \geq 0 \\ \widehat{c_{21}} + \widehat{c_{22}} \cdot x + \widehat{c_{23}} \cdot y \geq 0 \end{cases} \quad \begin{array}{l} \widehat{f_1} = \widehat{d_0} + \widehat{d_1} \cdot x + \widehat{d_2} \cdot y \\ \widehat{f_2} = \widehat{d_3} + \widehat{d_4} \cdot x + \widehat{d_5} \cdot y \end{array} \quad \begin{array}{l} \widehat{f_3} = \widehat{d_6} + \widehat{d_7} \cdot x + \widehat{d_8} \cdot y \\ \widehat{f_4} = \widehat{d_9} + \widehat{d_{10}} \cdot x + \widehat{d_{11}} \cdot y \end{array}$$

The goal is to synthesize real values for each of the variables $\widehat{\epsilon}, \widehat{c_0}, \ldots, \widehat{c_{23}}$ and $\widehat{d_0}, \ldots, \widehat{d_{11}}$, so that when we plug them back into the templates above, we get a valid IRW/UIRW.

```
I : $x, y \geq 10$

1:  if  $x < y$:
2:     □  $x := x + 10$
3:     □  $x := x + 5$
4:
```

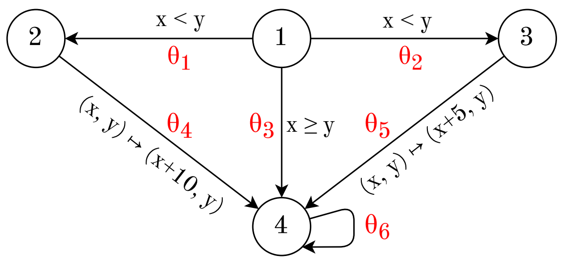


Figure 7.2: Our Running Example as a Program (left) and a Transition System (right)

Step 2a. Computing IRW Constraint Pairs. This step is only performed when we want to synthesize an IRW. In an IRW, the existential inductive set $\mathfrak{T}^\diamond$ should satisfy the condition that for every state $\sigma \in \mathfrak{T}^\diamond \setminus \mathfrak{T}$, there exists a successor $\sigma' \in \mathfrak{T}^\diamond$ of σ. Moreover, there should be at least one such successor for which we have $f(\sigma') \leq f(\sigma) - \epsilon$.

Let $\ell \in \mathbf{L}$ be a location and $\mathbf{\Theta}_\ell$ be the set of transitions out of ℓ, i.e. transitions whose pre-location is ℓ. The IRW properties at ℓ are equivalent to:

$$\forall \nu \in \mathbb{R}^{\mathbf{V}},\ \nu \models \widehat{A_\ell} \Rightarrow \Big(\nu \models \tau_\ell \quad \vee \quad \bigvee_{\theta=(\ell,\ell',\varphi,\mu)\in\mathbf{\Theta}_\ell} \xi(\theta)\Big) \tag{7.1}$$

where $\xi(\theta) = \xi(\ell, \ell', \varphi, \mu)$ is defined as:

$$\xi(\theta) := \Big(\nu \models \varphi \wedge \mu(\nu) \models \widehat{A_{\ell'}} \wedge \widehat{f_{\ell'}}(\mu(\nu)) \leq \widehat{f_\ell}(\nu) - \widehat{\epsilon}\Big) \tag{7.2}$$

Intuitively, the constraint in (7.1) says that if $\nu \models A_\ell$ or equivalently $(\ell, \nu) \in \mathfrak{T}^\diamond$, then either $(\ell, \nu) \in \mathfrak{T}$ which is equivalent to $\nu \models \tau_\ell$, or there exists a transition $\theta \in \mathbf{\Theta}_\ell$, using which we can obtain a successor $(\ell', \mu(\nu)) \in \mathfrak{T}^\diamond$ such that $f(\ell', \mu(\nu)) \leq f(\ell, \nu) - \epsilon$. The latter is formalized by $\xi(\theta)$. In this step, the algorithm symbolically computes (7.1) and writes it in the following equivalent format:

$$\forall \nu \in \mathbb{R}^{\mathbf{V}},\ \Big(\nu \models \widehat{A_\ell} \wedge \bigwedge_{\theta=(\ell,\ell',\varphi,\mu)} \neg\xi(\theta)\Big) \Rightarrow \nu \models \tau_\ell \tag{7.3}$$

Let P_ℓ be the LHS assertion in (7.3) above. Then P_ℓ is constructed from logical operations and atomic strict/non-strict linear inequalities over $\mathbf{V}$. Note that the coefficients in these linear inequalities contain the unknown variables $\widehat{c_{\ell,i,j}}$'s and $\widehat{d_{\ell,j}}$'s defined in the previous step.

The algorithm writes P_ℓ in disjunctive normal form, obtaining $P_\ell = P_{\ell,1} \vee P_{\ell,2} \vee \ldots \vee P_{\ell,p}$, where each $P_{\ell,i}$ is a conjunction of strict/non-strict linear inequalities over $\mathbf{V}$. It then symbolically computes the following "constraint pair" for every $P_{\ell,i}$:

$$\gamma_{\ell,i} := (P_{\ell,i}, \tau_\ell) \tag{7.4}$$

The algorithm computes these constraint pairs for every $\ell \in \mathbf{L}$ and stores them in a set Γ. Note that all computations are symbolic. Every constraint pair $\gamma = (\lambda, \varrho) \in \Gamma$ consists of two parts. λ is a set of strict/non-strict linear inequalities, while ϱ is a set of only *non-strict* linear inequalities. Informally, γ encodes the requirement that every inequality in ϱ be entailed by inequalities in λ.

Example 7.6. *Consider the system in Figure 7.2 together with the templates generated in Example 7.5. In this step, the algorithm considers location* $1 \in \mathbf{L}$, *and writes the constraint in* (7.3)*:*

$$\widehat{A_1} \ \wedge \ \neg\xi(\theta_1) \ \wedge \ \neg\xi(\theta_2) \ \wedge \ \neg\xi(\theta_3) \Rightarrow \tau_1 \tag{7.5}$$

Intuitively, the constraint above says that if we are at a $\mathfrak{T}^\diamond$ *state in location* 1 *(satisfy* A_1*), and cannot transition to another* $\mathfrak{T}^\diamond$ *with smaller* $\widehat{f}$*-value using any of the available transitions, in other words* $\neg\xi(\theta_1) \ \wedge \ \neg\xi(\theta_2) \ \wedge \ \neg\xi(\theta_3)$*, then we must already be in a target state (satisfy* τ_1*). There is no target state at location* 1*, so we can assume* $\tau_1 \equiv (-1 \geq 0)$*. The algorithm computes* (7.5) *symbolically:*

$$\widehat{c_0} + \widehat{c_1} \cdot x + \widehat{c_2} \cdot y \geq 0 \ \wedge \ \widehat{c_3} + \widehat{c_4} \cdot x + \widehat{c_5} \cdot y \geq 0 \ \wedge$$

$$\neg\left(x < y \ \wedge \ \widehat{c_6} + \widehat{c_7} \cdot x + \widehat{c_8} \cdot y \geq 0 \ \wedge \widehat{c_9} + \widehat{c_{10}} \cdot x + \widehat{c_{11}} \cdot y \geq 0 \wedge \widehat{d_3} + \widehat{d_4} \cdot x + \widehat{d_5} \cdot y \leq \widehat{d_0} + \widehat{d_1} \cdot x + \widehat{d_2} \cdot y - \widehat{\epsilon}\right) \ \wedge$$

$$\neg\left(x < y \ \wedge \ \widehat{c_{12}} + \widehat{c_{13}} \cdot x + \widehat{c_{14}} \cdot y \geq 0 \ \wedge \ \widehat{c_{15}} + \widehat{c_{16}} \cdot x + \widehat{c_{17}} \cdot y \geq 0 \ \wedge \ \widehat{d_6} + \widehat{d_7} \cdot x + \widehat{d_8} \cdot y \leq \widehat{d_0} + \widehat{d_1} \cdot x + \widehat{d_2} \cdot y - \widehat{\epsilon}\right) \ \wedge$$

$$\neg\left(x \geq y \ \wedge \ \widehat{c_{18}} + \widehat{c_{19}} \cdot x + \widehat{c_{20}} \cdot y \geq 0 \ \wedge \ \widehat{c_{21}} + \widehat{c_{22}} \cdot x + \widehat{c_{23}} \cdot y \geq 0 \ \wedge \ \widehat{d_9} + \widehat{d_{10}} \cdot x + \widehat{d_{11}} \cdot y \leq \widehat{d_0} + \widehat{d_1} \cdot x + \widehat{d_2} \cdot y - \widehat{\epsilon}\right)$$

$$\Rightarrow (-1 \geq 0)$$

Intuitively, the first line of the constraint above models a state in $\mathfrak{T}^\diamond$ *at location* 1*, i.e. it is the same as* $\widehat{A_1}$*. The second line models the fact that it is not possible to take transition* θ_1 *and reach another state in* $\mathfrak{T}^\diamond$ *at location* 2 *such that the* $\widehat{f}$*-value decreases by at least* $\widehat{\epsilon}$*. The next two lines model similar constraints for* θ_2 *and* θ_3*. Finally, the last line says that if no suitable transition is possible, then the current state must itself be a target, which is impossible in this case because there are no target states at location* 1*. Next, the algorithm writes the constraint above in disjunctive normal form as:*

$$P_{1,1} \ \vee \ P_{1,2} \ \vee \ \ldots \ \vee P_{1,p} \Rightarrow (-1 \geq 0)$$

Just as before, the algorithm computes each of $P_{1,1}, \ldots, P_{1,p}$ *concretely in terms of* $x, y, \widehat{\epsilon}, \widehat{c_i}$*'s and* $\widehat{d_i}$*'s, but to save space, we omit the full expansion here. For example, we can assume*

$P_{1,1}$ *is:*

$$\widehat{c_0}+\widehat{c_1}\cdot x+\widehat{c_2}\cdot y \geq 0 \wedge \widehat{c_3}+\widehat{c_4}\cdot x+\widehat{c_5}\cdot y \geq 0 \wedge x \geq y \wedge \widehat{d_9}+\widehat{d_{10}}\cdot x+\widehat{d_{11}}\cdot y > \widehat{d_0}+\widehat{d_1}\cdot x+\widehat{d_2}\cdot y-\widehat{\epsilon}$$

This corresponds to the case where we cannot use either transition θ_1 or θ_2 because $x \geq y$, and also taking transition θ_3 will lead to a state whose $\widehat{f}$-value is not small enough. For each such $P_{1,i}$ the algorithm generates a constraint pair $(P_{1,i}, \tau_1) = (P_{1,i}, -1 \geq 0)$.The algorithm handles other locations similarly, and adds all the resulting constraint pairs to a set Γ.

Step 2b. Computing UIRW Constraint Pairs. This step is only performed when synthesizing a UIRW and is similar to its IRW variant in Step 2a above. In a UIRW, the universal $\mathfrak{T}$-inductive set $\mathfrak{T}^\diamond$ should satisfy the condition that for every state $\sigma \in \mathfrak{T}^\diamond \backslash \mathfrak{T}$, *every* successor σ' of σ is also in $\mathfrak{T}^\diamond$. Moreover, given that f is a universal $\mathfrak{T}-$ranking function, we must have $f(\sigma') \leq f(\sigma) - \epsilon$ for every such σ'.

Let $\ell \in \mathbf{L}$ be a location. The UIRW properties at ℓ are equivalent to:

$$\forall \nu \in \mathbb{R}^{\mathbf{V}},\ \nu \models \widehat{A_\ell} \Rightarrow \left(\nu \models \tau_\ell \vee \bigwedge\nolimits_{\theta=(\ell,\ell',\varphi,\mu)} \zeta(\theta)\right) \tag{7.6}$$

where $\zeta(\theta) = \zeta(\ell, \ell', \varphi, \mu)$ is defined as:

$$\zeta(\theta) := \left(\nu \models \varphi \Rightarrow \left(\mu(\nu) \models \widehat{A_{\ell'}} \wedge \widehat{f_{\ell'}}(\mu(\nu)) \leq \widehat{f_\ell}(\nu) - \widehat{\epsilon}\right)\right) \tag{7.7}$$

Informally, the constraint in (7.6) says that if $\nu \models A_\ell$ or equivalently $(\ell, \nu) \in \mathfrak{T}^\diamond$, then either $(\ell, \nu) \in \mathfrak{T}$, i.e. $\nu \models \tau_\ell$, or *for every transition* θ from ℓ the assertion $\xi(\theta)$ holds, i.e. if the transition is possible ($\nu \models \varphi$), then the successor state $(\ell', \mu(\nu))$ is also in $\mathfrak{T}^\diamond$, and the f value decreases by at least ϵ when going to this successor. As in the previous case, the algorithm computes (7.6) symbolically and writes it in the following equivalent format:

$$\forall \nu \in \mathbb{R}^{\mathbf{V}}, \left(\nu \models \widehat{A_\ell} \wedge \bigvee\nolimits_{\theta=(\ell,\ell',\varphi,\mu)} \neg\zeta(\theta)\right) \Rightarrow \nu \models \tau_\ell \tag{7.8}$$

Let Q_ℓ be the LHS assertion above. Similar to Step 2a, Q_ℓ is constructed form logical operations and atomic strict/non-strict linear inequalities over $\mathbf{V}$, and its coefficients include the unknown template variables $\widehat{c_{\ell,i,j}}$'s and $\widehat{d_{\ell,j}}$'s defined in Step 1. The algorithm writes Q_ℓ in disjunctive normal form, hence obtaining $Q_\ell = Q_{\ell,1} \vee Q_{\ell,2} \vee \ldots \vee Q_{\ell,q}$ in which each $Q_{\ell,i}$ is a conjunction of strict/non-strict linear inequalities over $\mathbf{V}$. It then computes the following constraint pair symbolically:

$$\gamma'_{\ell,i} := (Q_{\ell,i}, \tau_\ell)$$

The algorithm performs these operations for every location $\ell \in \mathbf{L}$ and stores all the resulting $\gamma'_{\ell,i}$ constraint pairs in a set Γ.

Example 7.7. *In our running example (Figure 7.2), we are looking for a linear UIRW for the target set $\mathfrak{T}' = \{(4, \nu) \mid \nu \models (x \geq y + 4)\}$. In this step, the algorithm creates constraints at every location. We now demonstrate how the process works for location 3. At location 3, the algorithm considers*

$$\widehat{A_3} \ \wedge \ \neg\zeta(\theta_5) \Rightarrow \tau_3$$

and symbolically computes it as:

$$\widehat{c_{12}} + \widehat{c_{13}} \cdot x + \widehat{c_{14}} \cdot y \geq 0 \ \wedge \ \widehat{c_{15}} + \widehat{c_{16}} \cdot x + \widehat{c_{17}} \cdot y \geq 0 \ \wedge$$

$$\neg(1 \geq 0 \Rightarrow (\widehat{c_{18}} + 5 \cdot \widehat{c_{19}} + \widehat{c_{19}} \cdot x + \widehat{c_{20}} \cdot y \geq 0 \ \wedge \ \widehat{c_{21}} + 5 \cdot \widehat{c_{22}} + \widehat{c_{22}} \cdot x + \widehat{c_{23}} \cdot y \geq 0 \ \wedge$$

$$\widehat{d_9} + 5 \cdot \widehat{d_{10}} + \widehat{d_{10}} \cdot x + \widehat{d_{11}} \cdot y \leq \widehat{d_6} + \widehat{d_7} \cdot x + \widehat{d_8} \cdot y - \widehat{\epsilon}))$$

$$\Rightarrow (-1 \geq 0)$$

Note that the transition θ_5 is unconditional, as such we can assume that its condition is simply $1 \geq 0$. Similarly, because there is no target state at location 3, we assume $\tau_3 \equiv (-1 \geq 0)$. Moreover, the transition θ_5 updates the value of x to $x + 5$. This is taken into account when generating the constraint above. The algorithm writes the LHS of the constraint in DNF and handles it exactly as in Example 7.6.

Step 2c. Computing Non-negativity Constraints. Note than in an IRW/UIRW, the ranking function f should have *non-negative* value over $\mathfrak{T}^\diamond$. Let $\ell \in \mathbf{L}$ be a location and $\mathbf{\Theta}_\ell$

the set of transitions out of ℓ. The non-negativity condition at ℓ is equivalent to:

$$\forall \nu \in \mathbb{R}^{\mathbf{V}}, \quad \nu \models \widehat{A_\ell} \Rightarrow \widehat{f_\ell}(\nu) \geq 0$$

To ensure this constraint, for every $\ell \in \mathbf{L}$, the algorithm adds the constraint pair $(\widehat{A_\ell}, \widehat{f_\ell} \geq 0)$ to Γ.

Example 7.8. *In the running example, based the templates generated at Example 7.5, the algorithm creates the following non-negativity constraint pair* $\gamma = (\lambda, \varrho)$*, encoding* $\lambda \Rightarrow \varrho$*, at location* $1 \in \mathbf{L}$*:*

$$\lambda : \begin{cases} \widehat{c_0} + \widehat{c_1} \cdot x + \widehat{c_2} \cdot y \geq 0 \\ \widehat{c_3} + \widehat{c_4} \cdot x + \widehat{c_5} \cdot y \geq 0 \end{cases} \qquad \varrho : (\widehat{d_0} + \widehat{d_1} \cdot x + \widehat{d_2} \cdot y \geq 0)$$

Step 3. Applying Farkas' Lemma. The algorithm applies Corollary 2.1 to every constraint pair generated in the previous step to obtain a non-linear constraint system based on the template variables (i.e. $\widehat{c_{\ell,i,j}}$'s and $\widehat{d_{\ell,j}}$'s), the ranking parameter $\widehat{\epsilon}$, and new variables defined in this step. Crucially, this non-linear constraint system does not include any of the variables in $\mathbf{V}$. We now explain the operations in this step more concretely.

For every constraint pair $\gamma = (\lambda, \varrho) \in \Gamma$, we know that λ is a set of strict/non-strict linear inequalities $\{\lambda_{i,0} + \vec{\lambda_i} \cdot \vec{\mathbf{V}} \bowtie_i 0\}_{i=1}^m$, in which $\bowtie_i \in \{>, \geq\}$. Moreover, ϱ is a set of *non-strict* inequalities and every inequality in ϱ should be entailed by λ. Let $\alpha_0 + \alpha_1 \cdot v_1 + \ldots + \alpha_r \cdot v_r \geq 0 \equiv \alpha_0 + \vec{\alpha} \cdot \vec{\mathbf{V}} \geq 0$ be an inequality in ϱ. According to Corollary 2.1, there are three cases in which $\{\lambda_{i,0} + \lambda_i \cdot \mathbf{V} \bowtie_i 0\}_{i=1}^m$ entails $\alpha_0 + \alpha \cdot \mathbf{V} \geq 0$:

(i) $\alpha_0 + \alpha \cdot \mathbf{V} \geq 0$ is a non-negative combination of $1 \geq 0$ and $\{\lambda_{i,0} + \lambda_i \cdot \mathbf{V} \bowtie_i 0\}_{i=1}^m$, or

(ii) $-1 \geq 0$ can be derived as above, or

(iii) $0 > 0$ can be derived as above.

The algorithm writes constraints that model each of the three cases above and then combines them disjunctively. Given that the three cases are similar, we only explain how (i) is handled:

The algorithm creates $m+1$ new variables $\widehat{y_0}, \widehat{y_1}, \dots, \widehat{y_m}$ and generates the constraints $\widehat{y_i} \geq 0$ for each one of them. As in Corollary 2.1, the algorithm computes the following equality symbolically:

$$\alpha_0 + \alpha \cdot \mathbf{V} = \widehat{y_0} + \textstyle\sum_{i=1}^{m} \widehat{y_i} \cdot (\lambda_{i,0} + \lambda_i \cdot \mathbf{V}) \tag{7.9}$$

Note that the two sides of the equation above are linear expressions over $\mathbf{V}$. As such, they are equal if and only if they agree on the coefficient of every term. The algorithm equates the corresponding coefficients, and adds the following equalities to the constraint system:

$$\begin{array}{lrl} & \alpha_0 = \widehat{y_0} + \sum_{i=1}^{m} \widehat{y_i} \cdot \lambda_{i,0} & \text{i.e. the constant factor should be equal on both sides} \\ \forall j \neq 0 & \alpha_j = \sum_{i=1}^{m} \widehat{y_i} \cdot \lambda_{i,j} & \text{i.e. coefficient of every } v_j \in \mathbf{V} \text{ should be equal on both sides} \end{array}$$

The algorithm handles (ii) and (iii) similarly, except that in (iii) we should ensure that at least one strict inequality is used when trying to obtain $0 > 0$. Hence, in this case, the algorithm also adds the extra constraint $\sum_{\bowtie_i \in \{>\}} \widehat{y_i} > 0$ to the non-linear constraint system. The algorithm performs the same operations for every constraint pair $\gamma = (\lambda, \varrho)$ and every linear inequality in ϱ and combines the resulting non-linear constraint systems conjunctively.

Example 7.9. *Consider the constraint pair $\gamma = (\lambda, \varrho)$ below, which was obtained in Example 7.6:*

$$\lambda : \begin{cases} \widehat{c_0} + \widehat{c_1} \cdot x + \widehat{c_2} \cdot y \geq 0 \\ \widehat{c_3} + \widehat{c_4} \cdot x + \widehat{c_5} \cdot y \geq 0 \\ x - y \geq 0 \\ \widehat{d_9} + \widehat{d_{10}} \cdot x + \widehat{d_{11}} \cdot y - \widehat{d_0} - \widehat{d_1} \cdot x - \widehat{d_2} \cdot y + \widehat{\epsilon} > 0 \end{cases} \qquad \varrho : (-1 \geq 0)$$

We want to make sure that λ entails ϱ. Based on Corollary 2.1, either ϱ or $-1 \geq 0$ or $0 > 0$ should be a non-negative combination of inequalities in λ. Here, ϱ is itself $-1 \geq 0$, so we only consider two cases:

- $-1 \geq 0$ is obtainable from λ: *The algorithm creates* 5 *new variables* $\widehat{y_0}, \widehat{y_1}, \dots, \widehat{y_4}$ *and adds the constraints* $\widehat{y_0}, \dots, \widehat{y_4} \geq 0$. *It then computes the following equality:*

$$\widehat{y_0} + \widehat{y_1} \cdot (\widehat{c_0} + \widehat{c_1} \cdot x + \widehat{c_2} \cdot y) + \widehat{y_2} \cdot (\widehat{c_3} + \widehat{c_4} \cdot x + \widehat{c_5} \cdot y) + \widehat{y_3} \cdot (x - y) +$$

$$\widehat{y_4} \cdot (\widehat{d_9} + \widehat{d_{10}} \cdot x + \widehat{d_{11}} \cdot y - \widehat{d_0} - \widehat{d_1} \cdot x - \widehat{d_2} \cdot y + \widehat{\epsilon}) = -1.$$

  Our program variables are x *and* y. *All other variables are created by the algorithm and we need to synthesize a value for them. The above is an equality between two polynomials in* $\mathbb{R}[x, y]$ *that has to hold for all values of* x *and* y. *Hence, the algorithm equates its corresponding coefficients:*

  - $\widehat{y_1} \cdot \widehat{c_1} + \widehat{y_2} \cdot \widehat{c_4} + \widehat{y_3} + \widehat{y_4} \cdot \widehat{d_{10}} - \widehat{y_4} \cdot \widehat{d_1} = 0$ *(the coefficient of* x *is equal on both sides),*
  - $\widehat{y_1} \cdot \widehat{c_2} + \widehat{y_2} \cdot \widehat{c_5} - \widehat{y_3} + \widehat{y_4} \cdot \widehat{d_{11}} - \widehat{y_4} \cdot \widehat{d_2} = 0$ *(the coefficient of* y *is equal on both sides),*
  - $\widehat{y_0} + \widehat{y_1} \cdot \widehat{c_0} + \widehat{y_2} \cdot \widehat{c_3} + \widehat{y_4} \cdot \widehat{d_9} - \widehat{y_4} \cdot \widehat{d_0} + \widehat{y_4} \cdot \widehat{\epsilon} = -1$ *(the constant factor is equal on both sides).*

- $0 > 0$ is obtainable from λ: *The algorithm creates* 5 *new variables* $\widehat{y_5}, \dots, \widehat{y_9}$ *and proceeds to obtain equalities over non-program variables in the exact same manner as in the previous case, except that it also adds the condition* $\widehat{y_9} > 0$.

Step 4. Computing Initialization Constraints. By definition, in addition to the inductivity, non-negativity and ranking conditions, an IRW/UIRW should also contain at least one initial state (ℓ_0, ν) such that $\nu \models I$. In other words,

$$\exists \nu_0 = (\nu_{0,1}, \dots, \nu_{0,r}) \in \mathbb{R}^{\mathbf{V}}, \nu \models \widehat{A_{\ell_0}} \wedge I. \tag{7.10}$$

By $k-$linearity of the system S, we know that the initial assertion I is a conjunction of at most k linear inequalities. Thus, the assertion above is a conjunction of at most $2k$ linear inequalities, and is equivalent to $\text{SAT}(\widehat{A_{\ell_0}} \wedge I) \neq \emptyset$.

In this step, the algorithm creates r new variables $\widehat{\nu_{0,1}}, \dots, \widehat{\nu_{0,r}}$, and symbolically computes the linear inequalities in (7.10), and adds them (conjunctively) to the non-linear constraint system.

Example 7.10. *For our running example (Figure 7.2), the algorithm creates two new variables $\widehat{\nu_{0,x}}$ and $\widehat{\nu_{0,y}}$ and computes the following:*

$$\begin{aligned} \widehat{c_0} + \widehat{c_1} \cdot \widehat{\nu_{0,x}} + \widehat{c_2} \cdot \widehat{\nu_{0,y}} \geq 0 \qquad & \widehat{\nu_{0,x}} \geq 10 \\ \widehat{c_3} + \widehat{c_4} \cdot \widehat{\nu_{0,x}} + \widehat{c_5} \cdot \widehat{\nu_{0,y}} \geq 0 \qquad & \widehat{\nu_{0,y}} \geq 10 \end{aligned}$$

The first two constraints ensure that the valuation $\widehat{\nu_0} = (\widehat{\nu_{0,x}}, \widehat{\nu_{0,y}})$ satisfies $\widehat{A_1}$ and the last two constraints ensure that it satisfies the initial condition I. The algorithm conjunctively adds these constraints to those generated in previous steps.

Step 5. Solving the Resulting Constraint System. Finally, the algorithm uses an off-the-shelf solver to solve the resulting non-linear constraint system. If the system is unsatisfiable, it reports that no $k-$linear IRW/UIRW exists. Otherwise, it obtains a solution $\mathfrak{s}$ of the non-linear constraint system. Let $\mathfrak{s}(\widehat{x})$ denote the value assigned by $\mathfrak{s}$ to variable $\widehat{x}$, and extend this definition in the natural way so to any expression e. The algorithm outputs $A_\ell := \mathfrak{s}(\widehat{A_\ell})$ and $f_\ell := \mathfrak{s}(\widehat{f_\ell})$, for all $\ell \in \mathbf{L}$, as the IRW/UIRW. Moreover, $\mathfrak{s}(\widehat{\nu_{0,1}}, \dots, \widehat{\nu_{0,r}})$ is the corresponding initial state, and $\mathfrak{s}(\widehat{\epsilon})$ is the decrease parameter for f.

Example 7.11. *When the algorithm solves the non-linear (in)equalities obtainted in the previous steps, it successfully synthesizes the following IRW*[‡] *(left table) for $\mathfrak{T} = \{(4, \nu) \mid \nu \models (x \geq y + 8)\}$ and the following UIRW (right table) for $\mathfrak{T}' = \{(4, \nu) \mid \nu \models (x \geq y + 4)\}$:*

ℓ	A_ℓ	f_ℓ
1	$y-2 \leq x \leq y-1$	2
2	$y-2 \leq x \leq y-1$	1
3	$-1 \geq 0$	-1
4	$x \geq y+8$	0

$\epsilon = 1, \quad \nu_0 = (11, 12)$

ℓ	A_ℓ	f_ℓ
1	$y-0.6 \leq x \leq y-0.5$	2
2	$y-0.6 \leq x \leq y-0.5$	1
3	$y-0.6 \leq x \leq y-0.5$	1
4	$x \geq y+4.4$	0

$\epsilon = 1, \quad \nu_0 = (11, 11.55)$

Theorem 7.3 (Soundness)**.** *Given a $k-$linear system $S = (\mathbf{V}, \mathbf{L}, \ell_0, I, \mathbf{\Theta})$, and a $k-$linear set $\mathfrak{T}$ of target states, every solution of the non-linear constraint system solved in Step 5 of the algorithm above produces a valid $k-$linear IRW/UIRW for $\mathfrak{T}$ in S.*

[‡]Every solution of the system of non-linear (in)equalities corresponds to a valid IRW. The concrete solution obtained in practice depends on the solver.

Proof. Every solution $\mathfrak{s}$ satisfies the constraints generated in Step 3. Therefore, for every constraint pair $\gamma = (\lambda, \varrho) \in \Gamma$ generated in Step 2 and inequality $\alpha_0 + \alpha \cdot \mathbf{V} \geq 0$ in ϱ, either $\mathfrak{s}(\lambda)$ is unsatisfiable, i.e. a non-negative linear combination of its inequalities sums up to $0 \geq 1$ or $0 > 0$, or there is such a linear combination that sums up to $\alpha_0 + \alpha \cdot \mathbf{V} \geq 0$. In each case, the coefficients of the combination are given by $\mathfrak{s}(\widehat{y_i})$ for the corresponding $\widehat{y_i}$ variables. Moreover, no matter which case happens, the inequalities in ϱ are entailed by λ. By definition, the constraint pairs generated in Step 2 modeled inductivity, non-negativity and ranking conditions and hence $\mathfrak{s}$ satisfies these properties. Finally, $\mathfrak{s}$ satisfies the constraints generated in Step 4. Therefore, we have $\mathfrak{s}(\widehat{\mathtt{v}_{0,1}}, \dots, \widehat{\mathtt{v}_{0,r}}) \models \mathfrak{s}(\widehat{A_{\ell_0}}) \wedge I$. So, all the requirements for IRW/UIRW are met. □

Theorem 7.4 (Completeness)**.** *Given a $k-$linear system $S = (\mathbf{V}, \mathbf{L}, \ell_0, I, \mathbf{\Theta})$, and a $k-$linear set $\mathfrak{T}$ of target states, every $k-$linear IRW/UIRW for $\mathfrak{T}$ in S is produced by some solution to the non-linear constraint system solved in Step 5 of the algorithm above.*

Proof. We construct the required solution. Let $(\mathfrak{T}^\diamondsuit, f, \epsilon)$ be a $k-$linear IRW/UIRW for $\mathfrak{T}$ in S. Let A_ℓ be the set of inequalities defining $\mathfrak{T}^\diamondsuit \cap \left(\ell \times \mathbb{R}^{\mathbf{V}}\right)$, and f_ℓ the linear expression defining f at ℓ. We use the coefficients in A_ℓ's and f_ℓ's as the corresponding values for $\mathfrak{s}(\widehat{c_{\ell,i,j}})$'s and $\mathfrak{s}(\widehat{d_{\ell,j}})$'s. Moreover, we let $\mathfrak{s}(\widehat{\epsilon}) = \epsilon$.

By definition, $\mathfrak{T}^\diamondsuit$ is an existential/universal $\mathfrak{T}-$inductive set, and f is an existential/universal $\mathfrak{T}-$ranking function with parameter ϵ. Therefore, A_ℓ's and f_ℓ's satisfy the constraint pairs generated at Step 2 of the algorithm. By Corollary 2.1, there are suitable values for each variable $\widehat{y_i}$ such that the constraints in Step 3 are satisfied. We use these values as $\mathfrak{s}(\widehat{y_i})$. Finally, by definition of IRW/UIRW, there exists a valuation $\overline{\mathtt{v}} \in \mathbb{R}^{\mathbf{V}}$ such that $\overline{\mathtt{v}} \models A_{\ell_0} \wedge I = \mathfrak{s}(\widehat{A_{\ell_0}}) \wedge I$. We let $\mathfrak{s}(\widehat{\mathtt{v}_{0,i}}) = \overline{\mathtt{v}}_i$. It is easy to verify that $\mathfrak{s}$ is a solution to the system of non-linear constraints solved in Step 5. □

Theorem 7.5 (Complexity)**.** *For fixed constants k and β, given a $k-$linear $\beta-$branching system $S = (\mathbf{V}, \mathbf{L}, \ell_0, I, \mathbf{\Theta})$, and a $k-$linear set $\mathfrak{T}$ of target states, Steps 1–4 of the algorithm above lead to a* polynomial-time *reduction from the problem of generating a $k-$linear IRW/UIRW to solving a Quadratic Programming (QP) instance.*

Proof. It is easy to verify that all steps of the algorithm run in polynomial time[§], and that all the generated (in)equalities over non-program variables are quadratic. However, these (in)equalities are not always combined conjunctively. Specifically, in Step 3, the constraints corresponding to cases (i)–(iii) are combined disjunctively. This being said, we can perform the following actions to obtain a QP instance in polynomial time:

- We first convert every inequality of the form $\mathfrak{e} \bowtie 0$ to $\mathfrak{e} - \widehat{x_\mathfrak{e}} = 0$ by introducing a new variable $\widehat{x_\mathfrak{e}} \bowtie 0$.
- We rewrite every disjunction $\mathfrak{e}_1 = 0 \lor \mathfrak{e}_2 = 0$ as $\mathfrak{e}_1 \cdot \mathfrak{e}_2 = 0$. Note that this might create polynomial equalities of higher degree.
- We eliminate terms of degree more than 2 by defining new variables that are equal to their proper divisors, e.g. we rewrite $\widehat{c_1} \cdot \widehat{c_2} \cdot \widehat{c_3}^2$ as $\widehat{v_1} \cdot \widehat{v_2}$ where $\widehat{v_1}, \widehat{v_2}$ are new variables, and add the equalities $\widehat{v_1} = \widehat{c_1} \cdot \widehat{c_2}$ and $\widehat{v_2} = \widehat{c_3}^2$.

The steps above lead to a polynomial blow-up in the size of the system, given that in Step 3 of the algorithm we have disjunctions of at most 3 different boolean formulas. $\square$

7.4.2 Linear IRWs/UIRWs for Linear Systems with Polynomial Target Sets

In this section, we take the first step towards generalizing our results from the linear case to the polynomial. For technical reasons, we need the concept of strong positivity as defined in Section 2.6.1. We recall this definition:

Strong Positivity. Let $X \subseteq \mathbb{R}^{\mathbf{V}}$ be a set of valuations and $g \in \mathbb{R}[\mathbf{V}]$ a polynomial over $\mathbf{V}$. We say that g is *strongly* positive over X, and write $X \models g \gg 0$ (or simply $g \gg 0$ when X is clear from context), if $\inf_{x \in X} g(x) > 0$. The real value $\delta := \inf_{x \in X} g(x)$ is called the *positivity gap* or *positivity witness* of g over X. Moreover, g is strongly greater than h, denoted $g \gg h$, iff $g - h \gg 0$.

[§]The reason for fixing k and β is to avoid exponential blow-up when rewriting boolean expressions in DNF.

Problem Definition. In this section, we consider the following problem: Given a $k-$linear system $S = (\mathbf{V}, \mathbf{L}, \ell_0, I, \mathbf{\Theta})$ together with a set τ_ℓ of at most k *strong* polynomial inequalities of degree at most d at every location $\ell \in \mathbf{L}$, synthesize a $k-$linear IRW/UIRW for a target set $\mathfrak{T}$ that satisfies τ_ℓ at every $\ell \in \mathbf{L}$, or report that no such IRW/UIRW exists.

Mathematical Tool. Our main mathematical tool in this section is Handelman's Theorem (Theorem 2.1) but we also rely on Farkas' Lemma (Lemma 2.2).

The Synthesis Algorithm. Our synthesis algorithm is similar to the one in Section 7.4.1 and consists of five steps. The main difference is in Step 3, in which constraint pairs are translated to non-linear constraints over template variables. In the previous section, our main tool for this translation was Farkas' Lemma. In this section, due to the more complicated nature of our target sets, we now supplement Farkas' Lemma with Handelman's theorem. For brevity, we do not repeat the presentation of other steps, which are the same as our previous algorithm.

Recall that Step 2 (either Steps 2a and 2c for IRWs, or Steps 2b and 2c for UIRWs) has already generated a set Γ of constraint pairs. Each constraint pair $\gamma \in \Gamma$ is of the form $\gamma = (\lambda, \varrho)$ and encodes the requirement that every inequality in ϱ should be entailed by λ. Moreover, λ is a set of strict/non-strict linear inequalities over $\mathbf{V}$, whereas ϱ is a set of *strong* polynomial inequalities of degree at most d. Let $g \gg 0$ be a strong inequality in ϱ. Either λ is satisfiable and g should be represented in the form of Equation 2.1 (cf. Corollary 2.2) or λ is unsatisfiable, in which case $-1 \geq 0$ or $0 > 0$ can be derived as non-negative combinations of inequalities in λ and $1 \geq 0$ (cf. Corollary 2.1).

Step 3. Applying Handelman's Theorem and Farkas' Lemma. For every $\gamma = (\lambda, \varrho) \in \Gamma$ and strong polynomial inequality $g \gg 0$ in ϱ, the algorithm performs the following operations:

- Let $\textsc{Monoid}_d(\lambda) = \{h_1, h_2, \dots, h_u\}$ be the set of all polynomials in $\textsc{Monoid}(\lambda)$ whose degree is at most d. The algorithm symbolically computes $\textsc{Monoid}_d(\lambda)$ and all of it elements.

- The algorithm considers the following three cases, writes constraints that model each of them, and then combines them disjunctively:

  (i) *Writing* g *as in Equation 2.1.* The algorithm creates $u+1$ new variables $\widehat{y_0}, \widehat{y_1}, \dots, \widehat{y_u}$ with the constraints $\widehat{y_0} > 0$ and $\widehat{y_1}, \dots, \widehat{y_u} \geq 0$, and symbolically computes the equation
  $$g = \widehat{y_0} + \textstyle\sum_{i=1}^{u} \widehat{y_i} \cdot h_i.$$
  Note that both sides of this equation are polynomials of degree d over $\mathbf{V}$. Hence, they are equal iff they agree on the coefficient of every monomial. The algorithm equates the coefficients of corresponding monomials in the LHS and RHS of the equation above, hence obtaining a set of equalities over template variables.

  (ii) *Obtaining* $-1 \geq 0$ *as a non-negative combination of* λ *and* $1 \geq 0$.

  (iii) *Obtaining* $0 > 0$ *as a non-negative combination of* λ *and* $1 \geq 0$.

  Cases (ii) and (iii) are handled using Farkas' Lemma in the exact same manner as in our previous algorithm (Section 7.4.1).

- The algorithm adds the resulting constraints to the non-linear constraint system

Example 7.12. *Consider our running example (Figure 7.2) together with the templates generated in Example 7.5. Moreover, assume that we aim to synthesize an IRW for* $\tau_3 := (x^2 - x - 100 \gg 0)$, *and no target sets in other locations. When Step 2 of the algorithm is applied to location* 3 *(in exactly the same manner as in Section 7.4.1) it creates several constraint pairs, including the following:*

$$\lambda : \begin{cases} \widehat{c_{12}} + \widehat{c_{13}} \cdot x + \widehat{c_{14}} \cdot y \geq 0 \\ \widehat{c_{15}} + \widehat{c_{16}} \cdot x + \widehat{c_{17}} \cdot y \geq 0 \\ -\widehat{c_3} - 5 \cdot \widehat{c_4} - \widehat{c_4} \cdot x - \widehat{c_5} \cdot y > 0 \end{cases} \qquad \varrho : (x^2 - x - 100 \gg 0)$$

In Step 3 of the algorithm, the constraint pair $\gamma = (\lambda, \varrho)$ *is handled as follows:*

- *The algorithm computes* $\text{MONOID}_2(\lambda)$ *which consists of all products of polynomials in* λ *up to degree* 2. *Explicitly, it computes an expanded version of the following polynomials:*

$$h_1 := 1 \qquad h_2 := \widehat{c_{12}} + \widehat{c_{13}} \cdot x + \widehat{c_{14}} \cdot y$$
$$h_3 := \widehat{c_{15}} + \widehat{c_{16}} \cdot x + \widehat{c_{17}} \cdot y \qquad h_4 := -\widehat{c_3} - 5 \cdot \widehat{c_4} - \widehat{c_4} \cdot x - \widehat{c_5} \cdot y$$
$$h_5 := h_2^2 \qquad h_6 := h_2 \cdot h_3$$
$$h_7 := h_2 \cdot h_4 \qquad h_8 := h_3^2$$
$$h_9 := h_3 \cdot h_4 \qquad h_{10} := h_4^2$$

- *The algorithm considers cases (i)-(iii) as above. Cases (ii) and (iii) are similar to Section 7.4.1, so we focus on (i). The algorithm introduces 11 new variables $\widehat{y_0}, \ldots, \widehat{y_{10}}$, adds the constraints $\widehat{y_0} > 0$ and $\widehat{y_1} \ldots \widehat{y_{10}} \geq 0$ and symbolically computes the following equality:*

  $$x^2 - x - 100 = \widehat{y_0} + \textstyle\sum_{i=1}^{10} \widehat{y_i} \cdot h_i$$

  As before, this is a polynomial equality in $\mathbb{R}[x, y]$, and must hold for all values of x, y. So, the corresponding coefficients of the two sides should be equal. The algorithm generates these equalities. For example, given that the constant factor must be the same in the LHS and RHS, the algorithm generates this equality:

  $$-100 = \widehat{y_0} + \widehat{y_1} + \widehat{y_2} \cdot \widehat{c_{12}} + \widehat{y_3} \cdot \widehat{c_{15}} - \widehat{y_4} \cdot \widehat{c_3} - 5 \cdot \widehat{y_4} \cdot \widehat{c_4} + \widehat{y_5} \cdot \widehat{c_{12}}^2 + \widehat{y_6} \cdot \widehat{c_{12}} \cdot \widehat{c_{15}} - \widehat{y_7} \cdot \widehat{c_{12}} \cdot \widehat{c_3} - 5 \cdot \widehat{y_7} \cdot \widehat{c_{12}} \cdot \widehat{c_4} + \widehat{y_8} \cdot \widehat{c_{15}}^2 - \widehat{y_9} \cdot \widehat{c_{15}} \cdot \widehat{c_3} - 5 \cdot \widehat{y_9} \cdot \widehat{c_{15}} \cdot \widehat{c_4} + \widehat{y_{10}} \cdot \widehat{c_3}^2 + 10 \cdot \widehat{y_{10}} \cdot \widehat{c_3} \cdot \widehat{c_4} + 25 \cdot \widehat{y_{10}} \cdot \widehat{c_4}^2.$$

  The algorithm generates similar equalities for the coefficients of $x, y, x^2, x \cdot y$, and y^2.

Note that Steps 4 and 5 are also exactly the same as in our previous algorithm and are omitted here. This being said, we have the following theorems, whose proofs are similar to Section 7.4.1:

Theorem 7.6 (Soundness)**.** *Given a $k-$linear system $S = (\mathbf{V}, \mathbf{L}, \ell_0, I, \mathbf{\Theta})$, and a set τ_ℓ of at most k polynomial inequalities of degree d or less at every $\ell \in \mathbf{L}$, every solution of the non-linear constraint system solved in Step 5 of the algorithm above produces a valid $k-$linear IRW/UIRW for a target set $\mathfrak{T}$ that satisfies τ_ℓ at every $\ell \in \mathbf{L}$.*

Theorem 7.7 (Completeness)**.** *Given a $k-$linear system $S = (\mathbf{V}, \mathbf{L}, \ell_0, I, \mathbf{\Theta})$, and a set τ_ℓ of at most k strong polynomial inequalities of degree d or less at every $\ell \in \mathbf{L}$, every bounded $k-$linear IRW/UIRW for a target set $\mathfrak{T}$ that satisfies τ_ℓ at every $\ell \in \mathbf{L}$, is produced by some solution of the non-linear constraint system solved in Step 5 of the algorithm above.*

Theorem 7.8 (Complexity)**.** *For fixed constants k, d and β, given a $k-$linear β-branching system $S = (\mathbf{V}, \mathbf{L}, \ell_0, I, \mathbf{\Theta})$, and a set τ_ℓ of at most k polynomial inequalities of degree d or less at every $\ell \in \mathbf{L}$, Steps 1–4 of the algorithm above lead to a* polynomial-time *reduction from the problem of generating a k-linear IRW/UIRW to solving a QP instance.*

Remark 7.2. *Unlike the linear case, our completeness result in Theorem 7.7 requires* strong *inequalities and* boundedness*. This is because Handelman's theorem is only applicable when* $\textsc{Sat}(\Phi)$ *is compact, and hence Corollary 2.2 can only handle strong inequalities over bounded polyhedra. These requirements do not apply to our soundness result, and although they are theoretically necessary, they have very little impact in practice. If there is an IRW/UIRW for a target set $\mathfrak{T}$ that ensures reachability within n steps, it is easy to verify that there is also a bounded IRW/UIRW with the same property, i.e. the semi-runs starting at $\mathbf{v}_0$ and taking n transitions cannot visit an unbounded set of valuations. Moreover, if the target set contains a non-strong inequality such as $g \geq 0$ or $g > 0$, one can replace this inequality with $g + \bar{\epsilon} \gg 0$ for a new variable $\bar{\epsilon} \geq 0$ and solve a quadratic programming instance with the goal of minimizing $\bar{\epsilon}$. This trick will slightly change the problem, but it rarely has practical significance.*

7.4.3 Polynomial IRWs/UIRWs for Polynomial Systems with Polynomial Target Sets

We now provide the most general extension of our algorithm to the case where the system, the target set, and the IRW/UIRW are all polynomial.

Problem Definition. We consider the following problem: Given four technical constants $\Upsilon_1, \ldots, \Upsilon_4 \in \mathbb{N}$, a $(d, k)-$polynomial system $S = (\mathbf{V}, \mathbf{L}, \ell_0, I, \mathbf{\Theta})$, together with a set τ_ℓ of at most k *strong* polynomial inequalities of degree at most d at every location $\ell \in \mathbf{L}$, synthesize a $(d, k)-$polynomial IRW/UIRW for a target set $\mathfrak{T}$ that satisfies τ_ℓ at every $\ell \in \mathbf{L}$, i.e. $\mathfrak{T} \cap \left(\{\ell\} \times \mathbb{R}^{\mathbf{V}}\right) \models \tau_\ell$, or report that no such IRW/UIRW exists. The technical constants Υ_i are bounds on the degrees of various polynomials we construct as part of our algorithm. We will soon discuss them more concretely.

Mathematical Tools. We rely on Putinar's Positivstellensatz (Theorem 2.2) and Hilbert's Strong Nullstellensatz (Theorem 2.4).

The Synthesis Algorithm. We are now ready to provide our most general synthesis algorithm for polynomial IRWs/UIRWs over polynomial transition systems. As in the previous cases, our algorithm consists of 5 steps. The main differences are in Steps 1 and 3. In Step 1, our algorithm should now generate a polynomial template. Moreover, in Step 3, it employs Corollary 2.4 and Theorem 2.5 for characterizing entailment. The other steps are exactly like our previous algorithms.

Step 1. Setting up a template. The algorithm symbolically computes the set of monomials of degree at most d over the variables in $\mathbf{V}$:

$$M_d(\mathbf{V}) := \{\mathfrak{m}_1, \mathfrak{m}_2, \ldots, \mathfrak{m}_u\} := \{v_1^{\alpha_1} \cdot v_2^{\alpha_2} \cdot \ldots \cdot v_r^{\alpha_r} \mid \alpha_1, \ldots, \alpha_r \in \mathbb{N} \cup \{0\} \wedge \alpha_1 + \ldots + \alpha_r \leq d\}.$$

It then sets up the following templates for A_ℓ and f_ℓ at every location $\ell \in \mathbf{L}$:

$$\widehat{A_\ell} : \begin{cases} \widehat{c_{\ell,1,1}} \cdot \mathfrak{m}_1 + \ldots + \widehat{c_{\ell,1,u}} \cdot \mathfrak{m}_u \geq 0 \\ \vdots \\ \widehat{c_{\ell,k,1}} \cdot \mathfrak{m}_1 + \ldots + \widehat{c_{\ell,k,u}} \cdot \mathfrak{m}_u \geq 0 \end{cases}$$

$$\widehat{f_\ell} = \widehat{d_{\ell,1}} \cdot \mathfrak{m}_1 + \ldots + \widehat{d_{\ell,u}} \cdot \mathfrak{m}_u$$

As usual, the $\widehat{c_{\ell,i,j}}$'s and $\widehat{d_{\ell,j}}$'s are unknown variables for which we should synthesize a value such that the $\widehat{A_\ell}$'s and $\widehat{f_\ell}$'s form an IRW or a UIRW. Note that we do not need to add a separate constant factor to our templates because $1 \in M_d(\mathbf{V})$.

Step 2. Computing Constraint Pairs. Steps 2a–2c are the same as in Section 7.4.1. However, note that the resulting constraint pairs $\gamma = (\lambda, \varrho) \in \Gamma$ are now polynomial. Concretely, λ is a set of strict or non-strict polynomial inequalities over $\mathbf{V}$ and ϱ is a set of strong polynomial inequalities over $\mathbf{V}$.

Step 3. Applying Putinar's Positivstellensatz and Hilbert's Nullstellensatz. The algorithm applies Corollary 2.4 and Theorem 2.5 to every constraint pair generated in the previous step to obtain a non-linear constraint system based on the template variables, $\widehat{\epsilon}$, and new variables defined in this step. Let $\gamma = (\lambda, \varrho) \in \Gamma$ be a constraint pair. λ is a set of polynomial inequalities of the form $\{g_i \bowtie_i 0\}_{i=1}^m$. Let $g \gg 0$ be a strong polynomial inequality in ϱ. We have to make sure that λ entails $g \gg 0$. The algorithm considers three cases:

(i) λ *is unsatisfiable due to case (i) in Theorem 2.5*: The algorithm considers the set $M_{\Upsilon_1}(\mathbf{V}) := \{\mathfrak{m}_1, \mathfrak{m}_2, \dots, \mathfrak{m}_\mathfrak{n}\}$ of all monomials of degree at most Υ_1 over $\mathbf{V}$. Recall that Υ_1 is the first technical parameter given in input. It then generates the following templates $\widehat{h_i}$ for $0 \leq i \leq m$:

$$\widehat{h_i} := \widehat{\eta_{i,1}} \cdot \mathfrak{m}_1 + \ldots + \widehat{\eta_{i,\mathfrak{n}}} \cdot \mathfrak{m}_\mathfrak{n}$$

by introducing new variables $\widehat{\eta_{i,j}}$. It also adds certain constraints on $\widehat{\eta_{i,j}}$'s that ensure every $\widehat{h_i}$ is a sum-of-squares. See Section 2.7 for more details. Then, the algorithm introduces a new variable $\widehat{y_0}$ constrained with $\widehat{y_0} > 0$ and symbolically computes the following equality:

$$-1 = \widehat{y_0} + \widehat{h_0} + \textstyle\sum_{i=1}^m \widehat{h_i} \cdot g_i$$

Finally, the algorithm equates the corresponding coefficients on the two sides of the equality above, and obtains quadratic equalities over the unknown variables. As before, no program variable appears in these quadratic equalities.

(ii) λ *is unsatisfiable due to case (ii) in Theorem 2.5*: The algorithm considers the set $M^*_{\Upsilon_2} := \{\mathfrak{m}^*_1, \dots, \mathfrak{m}^*_{\mathfrak{n}^*}\}$ of all monomials of degree at most Υ_2 (our second technical parameter) over the extended variable set $\mathbf{V}^* = \mathbf{V} \cup \{w_1, \dots, w_m\}$. It generates the following templates $\widehat{h_i}$ for $1 \leq i \leq m$:

$$\widehat{h_i} := \widehat{\eta_{i,1}} \cdot \mathfrak{m}^*_1 + \ldots + \widehat{\eta_{i,\mathfrak{n}}} \cdot \mathfrak{m}^*_{\mathfrak{n}^*}$$

and symbolically computes the following equality for every index j that corresponds to a *strict* inequality $g_j > 0$ in λ:

$$w_j^{2\cdot\Upsilon_3} = \textstyle\sum_{i=1}^m \widehat{h_i} \cdot (g_i - w_i^2).$$

Here Υ_3 is our third technical parameter and both sides are polynomials in $\mathbb{R}[\mathbf{V}^*]$. As in the previous case, the algorithm equates the corresponding coefficients on the LHS and RHS and obtains quadratic equalities over unknown variables, i.e. no element of $\mathbf{V}^*$ appears in these equalities. The systems of quadratic equalities generated for each index j are then combined together *disjunctively*.

(iii) *g is a combination of g_i's as in Corollary 2.4*: The algorithm considers the set $M_{\Upsilon_4} := \{\mathfrak{m}_1, \ldots, \mathfrak{m}_\mathfrak{n}\}$ of monomials of degree at most Υ_4 over $\mathbf{V}$, and generates the following templates $\widehat{h_i}$ for $0 \leq i \leq m$:

$$\widehat{h_i} := \widehat{h_i} := \widehat{\eta_{i,1}} \cdot \mathfrak{m}_1 + \ldots + \widehat{\eta_{i,\mathfrak{n}}} \cdot \mathfrak{m}_\mathfrak{n}$$

by introducing new variables $\widehat{\eta_{i,j}}$ and adding constraints that ensure every $\widehat{h_i}$ is a sum-of-squares polynomial (Section 2.7). It then introduces a new variable $\widehat{y_0}$ constrained with $\widehat{y_0} > 0$ and symbolically computes this equality:

$$g = \widehat{y_0} + \widehat{h_0} + \textstyle\sum_{i=1}^{m} \widehat{h_i} \cdot g_i.$$

Finally, the algorithm translates this equality to quadratic equalities over template variables in exactly the same manner as in previous cases.

The systems of quadratic equalities generated in (i)–(iii) above are combined disjunctively.

Steps 4 and 5. These steps are exactly the same as those in Section 7.4.1.

Example 7.13. *Suppose that $\Upsilon_1 = \Upsilon_2 = \Upsilon_3 = \Upsilon_4 = 1$, and the algorithm is in Step 3, handling the following constraint pair:*

$$\lambda : \begin{cases} \widehat{c_1} \cdot x > 0 \\ \widehat{c_2} \cdot y \geq 0 \end{cases} \qquad\qquad \varrho : (\widehat{c_3} \cdot x \cdot y + c_4 \gg 0)$$

The algorithm considers the following cases:

(i) It generates three new template polynomials

$$\widehat{h_0} = \widehat{\eta_{0,1}} + \widehat{\eta_{0,2}} \cdot x + \widehat{\eta_{0,3}} \cdot y$$
$$\widehat{h_1} = \widehat{\eta_{1,1}} + \widehat{\eta_{1,2}} \cdot x + \widehat{\eta_{1,3}} \cdot y$$
$$\widehat{h_2} = \widehat{\eta_{2,1}} + \widehat{\eta_{2,2}} \cdot x + \widehat{\eta_{2,3}} \cdot y$$

and computes a quadratic system of (in)equalities over the $\widehat{\eta_{i,j}}$'s that ensures every $\widehat{h_i}$ is a sum-of-squares (See Section 2.7 for details). The algorithm then computes the following equality symbolically (with $\widehat{y_0} > 0$):

$$-1 = \widehat{y_0} + \widehat{h_0} + \widehat{h_1} \cdot \widehat{c_1} \cdot x + \widehat{h_2} \cdot \widehat{c_2} \cdot y$$

and rewrites it as quadratic equalities between the unknown variables in the usual way, i.e. by equating the coefficients of corresponding terms on the two sides of the polynomial equality. Intuitively, if there is a valuation for the unknown variables that satisfies these constraints, then -1 is a combination of $\widehat{c_1} \cdot x, \widehat{c_2} \cdot y$ and sum-of-square polynomials. Hence, λ is unsatisfiable.

(ii) The algorithm creates two new program variables w_1, w_2 and sets up the following templates:

$$\begin{aligned}\widehat{h_3} &= \widehat{\eta_{3,1}} + \widehat{\eta_{3,2}} \cdot x + \widehat{\eta_{3,3}} \cdot y + \widehat{\eta_{3,4}} \cdot w_1 + \widehat{\eta_{3,5}} \cdot w_2 \\ \widehat{h_4} &= \widehat{\eta_{4,1}} + \widehat{\eta_{4,2}} \cdot x + \widehat{\eta_{4,3}} \cdot y + \widehat{\eta_{4,4}} \cdot w_1 + \widehat{\eta_{4,5}} \cdot w_2\end{aligned}.$$

Unlike the previous case, $\widehat{h_3}$ and $\widehat{h_4}$ need not be sum-of-squares. It then writes the equality:

$$w_1^2 = \widehat{h_3} \cdot (\widehat{c_1} \cdot x - w_1^2) + \widehat{h_4} \cdot (\widehat{c_2} \cdot y - w_2^2),$$

and converts this polynomial equality to quadratic equalities over the unknown variables by equating the corresponding coefficients. However, note that the LHS and RHS of the polynomial equality above are in $\mathbb{R}[x, y, w_1, w_2]$. According to Theorem 2.5, any solution to the constraints generated here can serve as a proof for unsatisfiability of λ.

(iii) The algorithm generates the following template polynomials[¶]*:*

$$\widehat{h_5} = \widehat{\eta_{5,1}} + \widehat{\eta_{5,2}} \cdot x + \widehat{\eta_{5,3}} \cdot y$$
$$\widehat{h_6} = \widehat{\eta_{6,1}} + \widehat{\eta_{6,2}} \cdot x + \widehat{\eta_{6,3}} \cdot y,$$
$$\widehat{h_7} = \widehat{\eta_{7,1}} + \widehat{\eta_{7,2}} \cdot x + \widehat{\eta_{7,3}} \cdot y$$

enforces them to be sum-of-squares just as in case (i) above (Section 2.7) and writes the polynomial equality:

$$\widehat{c_3} \cdot x \cdot y + \widehat{c_4} = \widehat{y_1} + \widehat{h_5} + \widehat{h_6} \cdot \widehat{c_1} \cdot x + \widehat{h_7} \cdot \widehat{c_2} \cdot y$$

in which $\widehat{y_1} > 0$. It handles it similarly to the previous cases. Note that this is again a polynomial equality in $\mathbb{R}[x, y]$.

The algorithm combines the systems of quadratic inequality in (i)–(iii) above disjunctively.

It is now easy to obtain the following theorems, whose proofs are similar to previous cases:

Theorem 7.9 (Soundness)**.** *Given a $(d, k)-$polynomial system $S = (\mathbf{V}, \mathbf{L}, \ell_0, I, \mathbf{\Theta})$, and a set τ_ℓ of at most k polynomial inequalities of degree d or less at every $\ell \in \mathbf{L}$, every solution of the non-linear constraint system solved in Step 5 of the algorithm above produces a valid (d, k)-polynomial IRW/UIRW for a target set $\mathfrak{T}$ that satisfies τ_ℓ at every $\ell \in \mathbf{L}$.*

Theorem 7.10 (Semi-completeness)**.** *Consider a $(d, k)-$polynomial system $S = (\mathbf{V}, \mathbf{L}, \ell_0, I, \mathbf{\Theta})$ and a set τ_ℓ of at most k strong polynomial inequalities of degree d or less at every $\ell \in \mathbf{L}$. Let $W = (\mathfrak{T}^\diamond, f, \epsilon)$ be an explicitly bounded $(d, k)-$polynomial IRW/UIRW for a target set $\mathfrak{T}$ that satisfies τ_ℓ at every $\ell \in \mathbf{L}$. If large enough values are assigned to technical constants $\Upsilon_1, \ldots, \Upsilon_4$, the witness W is produced by some solution of the non-linear constraint system solved in Step 5 of the algorithm above.*

Theorem 7.11 (Complexity)**.** *For fixed constants k, d and β, and technical constants $\Upsilon_1, \ldots, \Upsilon_4$, given a $(k, d)-$polynomial $\beta-$branching system $S = (\mathbf{V}, \mathbf{L}, \ell_0, I, \mathbf{\Theta})$, and a set τ_ℓ of at most*

[¶]In practice, the templates in part (i) can be re-used for part (iii). This is a simple heuristic that we applied in our implementation and helped decrease the size of the resulting QP.

k polynomial inequalities of degree d or less at every $\ell \in \mathbf{L}$, Steps 1–4 of the algorithm above lead to a polynomial-time *reduction from the problem of generating a $(k, d)-$polynomial IRW/UIRW to solving a QP instance.*

Remark 7.3. *Note that Theorem 7.10 provides* semi-*completeness, i.e. completeness when the chosen technical constants are large enough. This is because Putinar's Positivstellensatz and Hilbert's Nullstellensatz do not establish a bound on the degree of polynomials that appear in their respective characterizations. Nevertheless, we have to fix a degree in our algorithm when we are generating templates for such polynomials. We use the technical constants $\Upsilon_1, \ldots, \Upsilon_4$ for this purpose. Such semi-completeness results arise routinely in constraint-based termination analysis (e.g. Chapter 8) and invariant generation (e.g. [Feng et al., 2017] and Chapter 6). In practice, solutions are often found with small technical constants (see Section 7.5 for examples).*

7.5 Experimental Results

Implementation. We implemented our algorithms for IRW synthesis in Python using SymPy [Meurer et al., 2017] for symbolic computations. The implementation also contains several heuristics for improving performance. Notably, we used Z3 [De Moura and Bjørner, 2008] to identify and discard unsatisfiable or tautological constraint pairs, hence reducing the sizes of our QP instances. The QPs were solved by LOQO [Vanderbei, 2006]. All results were obtained on an Intel Core i5-2540M (2.6 GHz) machine with 8 GB of RAM running Ubuntu 20.04 LTS. We enforced a time limit of 1800 seconds per verification task.

Benchmarks. For the linear case, we used benchmarks from SV-COMP 2020 [Beyer, 2020]. We considered all the tasks in the "Reachability/Safety" category of the competition and removed any benchmarks that asked for safety instead of reachability, or that could not be modeled as transition systems (e.g. due to the presence of pointers or arrays). This left us with 25 benchmarks. For the polynomial case, all standard benchmarks focused on safety. Hence, we created 6 simple programs with complex reachability structure to showcase the strengths of our approach. See our technical report [Asadi et al., 2020a] for details of these benchmarks. Specifically, it is noteworthy that these benchmarks demonstrate the fact that our algorithm's success is not dependent on the length or proportion of paths that reach the target set $\mathfrak{T}$.

Previous Tools. We compare our approach against the two best-performing tools in the Reachability/Safety category of SV-COMP 2020, namely VeriAbs [Afzal et al., 2020] and CPAchecker [Beyer and Keremoglu, 2011].

Linear Results. The results over linear benchmarks are summarized in Table 7.1. Our approach could handle every linear reachability benchmark in SV-COMP 2020. It is noteworthy that according to the SV-COMP results, none of the participating model checkers could handle all 25 benchmarks of Table 7.1. CPAchecker times out on 9 of the instances, whereas VeriAbs fails on only 1 instance.

After a manual inspection of the benchmarks, we realized that CPAChecker and VeriAbs are faster than our approach when reachability can be attained using liberal abstractions and a relatively short path (benchmarks towards the top of Table 7.1). This is not surprising, given that in these situations, abstract interpretation and symbolic execution are considerably faster than quadratic constraint solving. However, as the paths to target states become longer and sparser (benchmarks towards the the bottom of Table 7.1), the advantages of our approach begin to show. When the paths are long, e.g. thousands of steps of program execution, CPAchecker always fails to verify the instance. VeriAbs manages to handle these instances by a clever combination of ideas from loop pruning, loop summarization, abstract interpretation and bounded model checking. However, this comes with a considerable runtime overhead, leading to a much worse performance in comparison with our approach.

Benchmark	$\lvert\mathbf{L}\rvert$	$\lvert\mathbf{\Theta}\rvert$	$\lvert\mathbf{V}\rvert$	k	$\lvert$QP$\rvert$	Gen	Solve	**Ours**	**CPAchecker**	**VeriAbs**
`gcnr2008`	8	14	4	2	1838	14.8	81.4	96.2	**1.8**	17.6
`count_up_down-2`	3	4	3	2	244	1.6	1.9	**3.5**	4.4	5.9
`while_infinite_loop_4`	10	14	1	2	1223	5.1	7.9	13.0	**4.1**	15.3
`nec11`	4	8	3	2	2871	13.2	45.7	58.9	**4.2**	10.8
`terminator_02-1`	5	8	3	3	1962	12.0	19.4	31.4	**4.2**	17.2
`trex02-2`	5	7	1	2	260	1.8	3.4	5.3	**4.3**	16.6
`multivar_1-2`	3	6	2	2	900	5.8	19.8	25.6	**4.4**	9.0
`trex01-1`	14	27	6	3	9491	69.7	228.2	297.8	**4.5**	17.3
`sum04-1`	6	10	2	2	1082	5.4	8.0	13.4	**5.1**	17.0
`terminator_03-1`	6	11	2	3	1740	10.0	25.7	35.6	**5.1**	9.7
`trex03-1`	4	12	6	2	8500	49.2	197.9	247.0	**5.2**	9.1
`for_bounded_loop1`	10	13	5	2	1579	9.8	30.1	39.9	**5.6**	16.8
`Mono1_1-1`	3	5	1	2	262	1.3	4.4	**5.7**	**T/O**	377.2
`sum01_bug02_base`	7	13	3	2	7972	38.0	133.4	171.4	**6.0**	17.3
`sum03-1`	9	14	2	2	20963	77.9	413.1	491.0	**6.1**	16.3
`id_trans`	5	11	5	2	11192	68.7	171.2	239.8	**6.4**	19.8
`sum01_bug02`	7	12	4	3	17632	60.0	218.6	278.6	**6.5**	17.3
`sum01-1`	7	12	3	2	7316	36.9	55.1	92.0	**7.6**	16.7
`nested_1-2`	4	6	2	2	329	2.9	8.0	**10.9**	**T/O**	86.0
`const_1-2`	3	6	2	2	901	4.6	17.6	**22.2**	**T/O**	49.6
`Mono3_1`	6	8	2	3	660	4.0	20.0	**24.0**	**T/O**	369.9
`Mono4_1`	5	7	2	4	949	5.3	22.1	**27.3**	**T/O**	635.8
`Mono5_1`	5	7	3	4	1048	8.1	31.8	**39.8**	**T/O**	332.4
`Mono6_1`	5	7	3	5	1502	11.6	48.5	**60.1**	**T/O**	382.2
`deep_nested`	7	17	5	5	3686	28.6	69.6	**98.2**	**T/O**	**T/O**

Table 7.1: Experimental Results over Linear Reachability Benchmarks from SV-COMP. All times are reported in seconds. "|QP|" is the size of the generated QP instance. "Gen" is the time spent for generating the QP instance and "Solve" is the time spent for solving it. "Ours" is the total runtime of our approach over the instance. The last two columns are the runtimes of CPAchecker and VeriAbs. "T/O" denotes a timeout. The time limit was 1800 seconds per instance. Instances are ordered by the minimum time it took for an approach to solve them.

Benchmark	$\lvert\mathbf{L}\rvert$	$\lvert\mathbf{\Theta}\rvert$	$\lvert\mathbf{V}\rvert$	k	d	$\lvert$QP$\rvert$	Gen	Solve	**Ours**	**CPAchecker**	**VeriAbs**
`sqrt2`	5	7	2	5	2	2494	24.1	22.4	46.4	**10.5**	19.7
`sqrt1`	3	4	2	4	2	920	10.7	30.1	**40.8**	**T/O**	207.3
`sum`	3	4	3	5	2	1826	20.4	59.7	**80.1**	**T/O**	**F**
`sum2`	3	4	3	5	3	2476	36.8	167.5	**204.2**	**T/O**	**T/O**
`robot2`	5	8	4	5	2	5537	71.1	681.7	**752.9**	**T/O**	**F**
`robot1`	5	8	4	5	2	5537	69.8	724.2	**794.0**	**T/O**	**F**

Table 7.2: Experimental Results over Polynomial Programs. 'F' denotes that the tool terminated but failed to prove reachability. In all cases, we set our Υ variables equal to d.

```
for ($a := 0 ; a < M - 1 ; a := a + 1$):
    for ($b := 0 ; b < M - 1 ; b := b + 1$):
        for ($c := 0 ; c < M - 1 ; c := c + 1$):
            for ($d := 0 ; d < M - 1 ; d := d + 1$):
                for ($e := 0 ; e < M - 1 ; e := e + 1$):
                    if $M - 2 \leq a, b, c, d, e$:
                        print("target reached")
```

Figure 7.3: A simplified version of the deep-nested benchmark. $M > 10^9$ is a very large integer.

Nested Loop Benchmark. Figure 7.3 shows a simple illustration of the main part of deep-nested, the only linear reachability benchmark that could be handled by neither VeriAbs nor CPAchecker. We also ran these tools over this benchmark with an extended time limit of 12 hours, but they timed out. Moreover, according to SV-COMP results, no other participating model checker could handle this example, either. We believe this is because the target state can only be reached after an enormously-long path. Moreover, the target set is quite thin and even the smallest loss of precision in abstraction can cause a failure to prove reachability. However, this particular benchmark is not at all challenging for our method. The runtime of our method does not depend on the length of the paths, and we do not perform abstraction. Moreover, our approach is complete for linear IRWs. As such, it can easily prove reachability in Figure 7.3.

Polynomial Results. Table 7.2 shows our experimental results over 6 polynomial instances. Informally, sqrt1 is a simple program that given an input integer $n \geq 1$ computes $s = \lfloor \sqrt{n} \rfloor$ by trying every possible integer starting from 1. The goal is to (choose a value for n so as to) reach a state with $n - s > 10^5$. sqrt2 is a more clever variant of the same program that doubles the current value in a single step when the doubled value does not exceed $\lfloor \sqrt{n} \rfloor$. sum is a program that sums up all the integers from 1 to n. The goal is to synthesize a value for n such that the sum falls in a specific interval. sum2 is a similar benchmark in which the program sums squares of all integers from 1 to n. In robot1, two robots are put in the same position in a 2d plane. At each step, each robot non-deterministically chooses to move one unit either upwards or to the right. The goal is to reach a state where the square of the distance between the two robots is more than 10^5. In robot2, the same two robots are placed on the lower-right and upper-left corners of a square of side length 10^4. The goal is to show that they can reach a distance of less than 10 from each other. See [Asadi et al., 2020a] for details.

Similarly to the linear case, we observe that CPAchecker and VeriAbs can handle cases where the path reaching the targets is quite short, and when there is no combinatorial explosion in the number of paths due to repeated non-deterministic choice. Notably, CPAchecker can handle `sqrt2` but not `sqrt1`. The only difference between these two programs is that `sqrt2` is more efficient and hence the path to targets is shorter. Moreover, we observe that the various other techniques used by VeriAbs, which made it more successful in the linear case, do not extend well to the polynomial case. In several of the instances, VeriAbs terminates without producing an answer, i.e. reporting `unknown` as the output. In contrast, our approach is able to handle these examples, given its semi-completeness over polynomial IRWs.

8

Termination Analysis for Polynomial Programs

This chapter originally appeared in the following publications:

[•] Chatterjee, K., Fu, H., and **Goharshady, A. K.** **Termination Analysis of Probabilistic Programs through Positivstellensatz's**. In *28th International Conference on Computer Aided Verification* **(CAV)**, 2016.

[•] Huang, M., Fu, H., Chatterjee, K., and **Goharshady, A. K.** **Modular Verification for Almost-Sure Termination of Probabilistic Programs**. In *34th ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications* **(OOPSLA)**, 2019.

8.1 Introduction

Outline. In this chapter, we consider polynomial probabilistic transition systems with the most basic liveness property of termination. We present efficient methods that reduce the termination analysis to linear or semi-definite programming. This is in contrast to Chapters 6 and 7 in which the analysis was reduced to the much more general and costlier problem of quadratic programming. Our approach synthesizes polynomial ranking supermartingales (PRSMs). On one hand, PRSMs significantly generalize linear ranking supermartingales (LRSMs) and on the other hand, they are a counterpart of polynomial ranking functions for proving termination of non-probabilistic programs. Our approach synthesizes PRSMs using the positivstellensätze of Schmüdgen and Putinar, yielding an efficient method which is not only sound, but also semi-complete. We show experimental results to demonstrate that our approach can efficiently handle several classical programs with complex polynomial guards and assignments.

Probabilistic Programs. Classic imperative programs extended with *random-value generators* gives rise to probabilistic programs. Probabilistic programs provide the appropriate framework to model applications ranging from randomized algorithms [Motwani and Raghavan, 1995, Dubhashi and Panconesi, 2009], to stochastic network protocols [Baier and Katoen, 2008, Kwiatkowska et al., 2011], to robot planning [Kress-Gazit et al., 2009, Kaelbling et al., 1998]. Non-determinism plays a crucial role in modeling, for example to model behaviors over which there is no control, or for abstraction. Thus non-deterministic probabilistic programs are crucial in a huge range of problems, and hence their formal analysis has been studied across disciplines, such as probability theory and statistics [Durrett, 2019, Howard, 1960], formal methods [Baier and Katoen, 2008, Kwiatkowska et al., 2011], artificial intelligence [Kaelbling et al., 1996, Kaelbling et al., 1998], and programming languages [Colón et al., 2003, Fioriti and Hermanns, 2015, Sankaranarayanan et al., 2013, Esparza et al., 2012].

Fundamental Termination Problems. In absence of probability, i.e. for non-probabilistic programs, the synthesis of *ranking functions* and proof of termination are equivalent [Floyd, 1993]. Numerous approaches exist for synthesizing linear ranking functions in non-probabilistic programs [Bradley et al., 2005a, Colón and Sipma, 2001, Podelski and Rybalchenko, 2004].

These approaches use a Farkas-based method similar to Chapter 7 to synthesize linear ranking functions. The most basic extension of the termination question for probabilistic programs is the *almost-sure termination* question which asks whether a program terminates with probability 1. Another fundamental question is *finite termination* (also known as positive almost-sure termination [Fioriti and Hermanns, 2015]) which asks whether the expected termination time of the program is finite. Yet another interesting question is the *concentration* bound computation problem that asks to compute a bound M such that the probability that the termination time is below M is concentrated, or in other words, the probability that the termination time exceeds $M + n$ decreases exponentially with respect to n.

Previous Results. We now discuss several previous results:

- *Probabilistic Programs.* Quantitative invariants were introduced to establish termination of discrete probabilistic programs with demonic non-determinism [McIver and Morgan, 2004, McIver et al., 2005]. This was extended in [Chakarov and Sankaranarayanan, 2013] to *ranking supermartingales* resulting in a sound, but not complete, approach to prove almost-sure termination of probabilistic programs without non-determinism. For probabilistic programs with countable state space and without non-determinism, the Lyapunov ranking functions provide a sound and complete method for proving finite termination [Bournez and Garnier, 2005, Foster et al., 1953]. Another sound method is to explore bounded termination with exponential decrease of probabilities [Monniaux, 2001] through abstract interpretation [Cousot and Cousot, 1977]. For probabilistic programs with non-determinism, a sound and complete characterization for finite termination through ranking supermartingales is obtained in [Fioriti and Hermanns, 2015]. Ranking supermartingales thus provide a very powerful approach for termination analysis of probabilistic programs.

- *Synthesizing Ranking Functions/Supermartingales.* Synthesis of linear ranking functions/supermartingales has been studied extensively in [Podelski and Rybalchenko, 2004, Chakarov and Sankaranarayanan, 2013]. In the context of probabilistic programs, algorithmic study of synthesis of linear ranking supermartingales for probabilistic programs [Chakarov and Sankaranarayanan, 2013] and probabilistic programs has been studied. The major technique adopted in these results is Farkas' Lemma (Lemma 2.2) which serves as a complete reasoning method for linear inequalities. Beyond linear ranking functions, polynomial ranking functions have also been considered. Heuristic synthesis method of polynomial ranking functions is studied in the following works:

- [Babić et al., 2013] checked termination of deterministic polynomial programs by detecting divergence on program variables.
- [Bradley et al., 2005a] extended to non-deterministic programs through an analysis on finite differences over transitions.
- [Cousot, 2005] uses Lagrangian Relaxation.
- [Shen et al., 2013] uses Putinar's Positivstellensatz (Theorem 2.2).
- Complete methods of synthesizing polynomial ranking functions for non-deterministic programs are studied in [Yang et al., 2010], where a complete method through root classification/real root isolation of semi-algbebraic systems and quantifier elimination is proposed.

To summarize, while many different approaches have been studied, the algorithmic study of the synthesis of ranking supermartingales for probabilistic programs has only been limited to linear ranking supermartingales. For example, [Chakarov and Sankaranarayanan, 2013] presents a method of synthesis of linear ranking supermartingales for probabilistic programs without non-determinism, and identifies synthesis of more general nonlinear supermartingales, or extension to probabilistic programs with non-determinism as important challenges. While the approach of [Chakarov and Sankaranarayanan, 2013] has been extended to probabilistic programs with non-determinism in [Chatterjee et al., 2016c], it is still restricted to linear ranking supermartingales. Hence, there is no previous algorithmic approach to handle non-linear ranking supermartingales even for probabilistic programs without non-determinism.

Our Contributions. Our contributions are as follows:

- *Polynomial Ranking Supermartingales.* We extend the notion of linear ranking supermartingales (LRSM) to polynomial ranking supermartingales (PRSM). We show by a straightforward extension of LRSM that the existence of a PRSM implies both almost-sure and finite termination.

- *Positivstellensätze.* We apply the positivstellensätze of Putinar (Theorem 2.2) and Schmüdgen (Theorem 2.3) to synthesize PRSMs. In case of linear programs, we also make use of Handelman's Theorem (Theorem 2.1).

- *New Approach for Non-probabilistic Programs.* Our results also extend existing results for non-probabilistic programs. We present the first result that uses Schmüdgen's Positivstellensatz and Handelman's Theorem to synthesize polynomial ranking functions for non-probabilistic programs.

- *Efficiency.* The previous complete method [Yang et al., 2010] suffers from high computational complexity due to the use of quantifier elimination. In contrast, our sound and semi-complete approach is efficient because the synthesis is reduced to linear or semi-definite programming, which are solvable in polynomial time [Grötschel et al., 2012]. In particular, our approach does not require quantifier elimination, and works for non-deterministic probabilistic programs.

- *Experimental Results.* We demonstrate the effectiveness of our approach on several classical examples. We show that on classical examples, such as Gambler's Ruin and Random Walk, our approach can synthesize a PRSM efficiently. For these examples, LRSMs do not exist, and they cannot be analyzed efficiently by previous approaches.

8.2 Termination of Probabilistic Programs

In this section, we first formally define the notions of finite and almost-sure termination, then formalize the problems considered in this chapter. We fix a probabilistic transition system $S = (\mathbf{V}, \mathbf{L}, \mathbf{L}_p, \ell_0, I, \theta)$, and assume that a location $\bot \in \mathbf{L}$ is set as the *end* location. Moreover, we assume that $\bot$ is a non-probabilistic location and the only transition out of $\bot$ is the trivial transition (with no condition or update) to itself.

Termination [Bournez and Garnier, 2005, Fioriti and Hermanns, 2015]**.** A run $\mathsf{r} = \langle (\ell_i, \nu_i), \theta_i \rangle_{i=0}^{\infty}$ is *terminating* if $\ell_n = \perp$ for some $n \in \mathbb{Z}^{\geq 0}$. The *termination time* of S is a random variable T_S such that for every terminating run r, $T_S(\mathsf{r})$ is the smallest n with $\ell_n = \perp$. If r is non-terminating, then $T_S(\mathsf{r}) = +\infty$. The transition system S is said to be *almost-surely* terminating (resp. finitely terminating) if for every initial valuation $\nu_0 \models I$ and every scheduler s, we have $\mathbb{P}^{\mathsf{s}}_{\nu_0}(T_S < \infty) = 1$ (resp. $\mathbb{E}^{\mathsf{s}}_{\nu_0}(T_S) < \infty$). Here, $\mathbb{P}^{\mathsf{s}}_{\nu_0}$ is the unique probability measure induced by s over runs that start at (ℓ_0, ν_0) and $\mathbb{E}^{\mathsf{s}}_{\nu_0}$ is the corresponding expectation. Additionally, we define $\mathsf{ET}(S, \nu_0) := \sup_{\mathsf{s}} \mathbb{E}^{\mathsf{s}}_{\nu_0}(T_S)$.

Note that almost-sure termination implies finite termination, but the converse does not necessarily hold.

Concentration on Termination Time [Monniaux, 2001]**.** A concentration bound for S given an initial valuation ν_0 is a non-negative integer M such that there exist real constants $c_1 \geq 0$ and $c_2 > 0$ so that for all $N \geq M$ and all schedulers s, we have

$$\mathbb{P}^{\mathsf{s}}_{\nu_0}(T_S > N) \leq c_1 \cdot e^{-c_2 \cdot N}.$$

Informally, a concentration bound shows exponential decrease of probabilities of non-termination beyond the bound. On the one hand, it can be used to give an upper bound on probability of non-termination beyond a large step; and on the other hand, it can be used to approximate $\mathsf{ET}(S, \nu_0)$ [Chatterjee et al., 2016c].

Problem Definition. We consider the following termination analysis problem:

- *Input:* A polynomial probabilistic transition system $S = (\mathbf{V}, \mathbf{L}, \mathbf{L}_p, \ell_0, I, \theta)$, an initial valuation $\nu_0 \models I$, and a polynomial invariant Inv for S.
- *Output 1 (Termination):* "yes" if the algorithm finds that S almost-surely/finitely terminates using any scheduler and starting from (ℓ_0, ν_0), or "fail" otherwise.
- *Output 2 (Concentration on Termination):* A concentration bound as above if the algorithm finds one and "fail" otherwise.

Note that we are assuming a polynomial invariant is provided as part of the input. Such an invariant can be generated using our approach in Chapter 6.

8.3 Ranking Supermartingales

In this section, we introduce Polynomial Ranking Supermartingales (PRSMs). PRSMs are a natural but significant generalization of the notion of Linear Ranking Supermartingales (LRSMs) as studied in [Chakarov and Sankaranarayanan, 2013, Chatterjee et al., 2016c]. We first define the general notion of Ranking Supermartingales (RSMs).

RSMs [Fioriti and Hermanns, 2015]**.** A discrete-time stochastic process $\{X_n\}_{n=0}^{\infty}$ with respect to a filtration $\{\mathcal{F}_n\}_{n=0}^{\infty}$ is a *Ranking Supermartingale* (RSM) if there exists $K < 0$ and $\epsilon > 0$ such that for all $n \in \mathbb{Z}^{\geq 0}$: (i) we have $\mathbb{E}(|X_n|) < \infty$, (ii) it holds almost surely that $X_n \geq K$, and (iii) $\mathbb{E}(X_{n+1}|\mathcal{F}_n) \leq X_n - \epsilon$ if $X_n \geq 0$, otherwise $\mathbb{E}(X_{n+1}|\mathcal{F}_n) \leq X_n$.

Lemma 8.1 ([Fioriti and Hermanns, 2015, Chatterjee et al., 2016c])**.** *Let $\{X_n\}_{n=0}^{\infty}$ be an RSM adapted to a filtration $\{\mathcal{F}_n\}_{n=0}^{\infty}$ and parameters K, ϵ as above. Let Z be the random variable defined by $Z := \min\{n \in \mathbb{Z}^{\geq 0} \mid X_n < 0\}$ with $\min \emptyset := \infty$, denoting the first time n that the RSM drops below 0. Then, we have $\mathbb{P}(Z < \infty) = 1$ and $\mathbb{E}(Z) \leq \frac{\mathbb{E}(X_0) - K}{\epsilon}$.*

Based on the lemma above, an RSM terminates, i.e. reaches a negative value, almost-surely and in finite expected time. The main idea behind using RSMs in termination analysis is to embed an RSM inside the probabilistic transition system S such that whenever the value of the RSM becomes zero or negative, we are ensured that S has also terminated (reached $\perp$). To do this, we first need to define the classical concept of pre-expectation.

Pre-expectation [Chakarov and Sankaranarayanan, 2013, Chatterjee et al., 2016c]**.** Consider a probabilistic transition system $S = (\mathbf{V}, \mathbf{L}, \mathbf{L}_p, \ell_0, I, \mathbf{\Theta})$. Recall that Σ is the set of states of S. Let $\eta : \Sigma \to \mathbb{R}$ be a function. The pre-expectation $\mathsf{pre}_\eta : \Sigma \to \mathbb{R}$ is defined as

ℓ	$\eta(\ell, x)$	$\mathrm{pre}_\eta(\ell, x)$	ℓ	$\eta(\ell, x)$	$\mathrm{pre}_\eta(\ell, x)$
1	$g(x) + 10$	$\mathbf{1}_{1\le x\le 10} \cdot (g(x) + 9.8) + \mathbf{1}_{x<1 \vee x>10} \cdot (-0.2)$	5	$g(x) + 2x - 1.8$	$g(x) + 2x - 2$
2	$g(x) + 9.8$	$g(x) + 9.6$	6	$g(x) - 2x + 20.2$	$g(x) - 2x + 20$
3	$g(x) + 9.6$	$g(x) + 9$	$\perp$	-0.2	-0.2
4	$g(x) + 9.6$	$g(x) + 0.04x + 8.98$			

Table 8.1: An Example η and pre_η for the Program in Figure 8.1.

follows:

$$\mathsf{pre}_\eta(\ell, \nu) := \begin{cases} \eta(\ell, \nu) & \ell = \perp \\ \sum_{\theta=(\ell,\ell',p,\mathbf{true},\mu)\in\Theta} \mathbb{E}\left[\eta(\ell', \mu(\nu))\right] \cdot p & \ell \in \mathbf{L}_p \ \wedge \ \ell \neq \perp \\ \max_{\theta=(\ell,\ell',\star,\varphi,\mu)\in\Theta, \nu\models\varphi} \mathbb{E}\left[\eta(\ell', \mu(\nu))\right] & \ell \in \mathbf{L} \setminus \mathbf{L}_p \ \wedge \ \ell \neq \perp \end{cases}$$

In the sequel, we focus on polynomial η's, i.e. η's for which each $\eta(\ell, \cdot)$ for $\ell \in \mathbf{L}$ is a polynomial over $\mathbf{V}$. We also assume that the update functions μ are such that pre_η is polynomial and can be computed symbolically. The following lemma can be proven by a simple definition-chasing:

Lemma 8.2. *Let $\sigma_i = (\ell_i, \nu_i)$ denote the state of S that is reached after i steps of execution from $\sigma_0 = (\ell_0, \nu_0)$, and define the stochastic process $\{X_n\}_{n=0}^\infty$ adapted to $\{\mathcal{F}_n\}_{n=0}^\infty$ as $X_n := \eta(\sigma_n) = \eta(\ell_n, \nu_n)$. For every scheduler s, we have:*

$$\mathbb{E}^{\mathsf{s}}(X_{n+1} \mid \mathcal{F}_n) \le \mathsf{pre}_\eta(\ell_n, \nu_n).$$

Example 8.1. *Consider the program in Figure 8.1. We will use this program as our running example. Let η be the function specified in Table 8.1, where $g(x) := (x-1)\cdot(10-x)$. The values of pre_η are also given in Table 8.1. Note that the case for $i = 2$ is obtained from $\mathsf{pre}_\eta(2, x) = \max\{\eta(3, x), \eta(4, x)\} = \max\{g(x) + 9.6, g(x) + 9.6\}$, and for $i = 3$ we have $\mathsf{pre}_\eta(3, x) = 0.5 \cdot \eta(1, x+1) + 0.5 \cdot \eta(1, x-1)$.*

We now define the notion of Polynomial Ranking Supermartingale (PRSM). The intuition is that we encode the RSM difference condition as a logical formula, treat zero as the threshold

```
1:  while $1 \le x \le 10$:
2:     if $\star$:
3:        $x := x + r$
       else:
4:        if prob(0.51):
5:           $x := x - 1$
          else:
6:           $x := x + 1$
⊥:
```

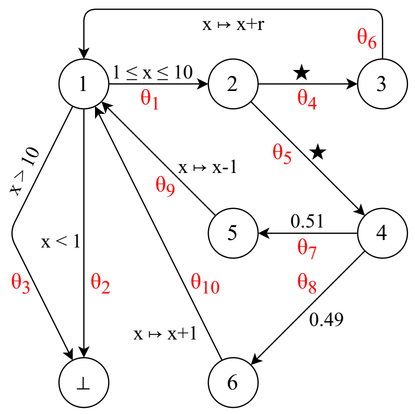


Figure 8.1: A Simple Program (Gambler's Ruin) and its Representation as a Probabilistic Transition System. In this figure, $\star$ denotes non-determinism and r is a random variable with $\mathbb{P}[r = 1] = \mathbb{P}[r = -1] = 0.5$.

between terminal and non-terminal states, and use the invariant Inv to over-approximate the set of reachable valuations at each label.

Polynomial Ranking Supermartingales. A *Polynomial Ranking Supermartingale* (PRSM) of degree d is a function $\eta : \Sigma \to \mathbb{R}$ for which there exist $\epsilon > 0$ and $K \le -\epsilon$ such that for all $\sigma = (\ell, \nu) \in \Sigma$, the following conditions hold:

(C1): The function $\eta(\ell, \cdot) : \mathbb{R}^{\mathbf{V}} \to \mathbb{R}$ is a polynomial of degree at most d over $\mathbf{V}$;

(C2): If $\ell \ne \perp$ and $\nu \models \mathsf{Inv}(\ell)$, then $\eta(\ell, \nu) \ge 0$;

(C3): If $\ell = \perp$, then $\eta(\ell, \nu) = K$;

(C4): If $\ell \ne \perp$ and $\nu \models \mathsf{Inv}(\ell)$, then $\mathsf{pre}_\eta(\ell, \nu) \le \eta(\ell, \nu) - \epsilon$.

Note that (C2) and (C3) together separate non-termination and termination by the threshold 0, and (C4) is the RSM difference condition which is intuitively related to the ϵ difference in the definition of an RSM.

Theorem 8.1. *Given a probabilistic transition system $S = (\mathbf{V}, \mathbf{L}, \mathbf{L}_p, \ell_0, I, \mathbf{\Theta})$ and an invariant* Inv, *if there is a PRSM η of degree d for S with parameters ϵ and K, then S is almost-surely terminating for every initial valuation $\nu_0 \models I \;\wedge\; \mathsf{Inv}(\ell_0)$ and every scheduler* s. *Moreover, we have* $\mathsf{ET}(S, \nu_0) \le \frac{\eta(\ell_0, \nu_0) - K}{\epsilon}$.

Proof. Define $\mathsf{UB}(S, \nu_0) := \frac{\eta(\ell_0,\nu_0)-K}{\epsilon}$ and let $\{X_n\}_{n=0}^{\infty}$ be the stochastic process defined in Lemma 8.2. By Lemma 8.2, (C4) and the fact that $K \leq -\epsilon$, $\{X_n\}_{n=0}^{\infty}$ is an RSM adapted to $\{\mathcal{F}_n\}_{n=0}^{\infty}$. Then by (C2), (C3) and Lemma 8.1 we have:

$$\mathsf{ET}(S, \nu_0) = \sup_{\mathsf{s}} \mathbb{E}_{\nu_0}^{\mathsf{s}}(T_S) \leq \frac{\eta(\ell_0, \nu_0) - K}{\epsilon}.$$

□

Example 8.2. *Consider our running example (Figure 8.1) and the function η given in Example 8.1. Assuming that the initial valuation satisfies $1 \leq x \leq 10$, we use the trivial invariant* Inv *such that* $\mathsf{Inv}(1) = 0 \leq x \wedge x \leq 11$, $\mathsf{Inv}(j) = 1 \leq x \wedge x \leq 10$ *for* $2 \leq j \leq 6$ *and* $\mathsf{Inv}(\bot) = x < 1 \vee x > 10$*. It is straightforward to verify that η is a PRSM of degree 2 with $\epsilon = 0.2$ and $K = -0.2$. Hence, by Theorem 8.1, the program terminates almost-surely under any scheduler and its expected termination time is at most $5 \cdot (x_0 - 1) \cdot (10 - x_0) + 51$, given the initial value x_0.*

Remark 8.1. *Our running example (Figure 8.1) does not admit a linear RSM (a PRSM of degree 1). This indicates that LRSMs may not exist even over simple affine programs and thus motivates the study of PRSMs even for affine programs.*

Remark 8.2. *The non-strict inequalities of (C2) and (C4) can be replaced by their strict counterparts. This does not change the notion of PRSM as we can simply add constant factors to η or scale it. On the same note, we can assume $K = -1$ and $\epsilon = 1$ without losing generality.*

Theorem 8.1 provides a tool for answering the problems of almost-sure and finite termination in a unified fashion. Generalizing the approach of [Chatterjee et al., 2016c], we now show that by restricting a PRSM to have *bounded difference*, it can also serve as a witness for concentration in termination.

Difference-bounded PRSM. A PRSM $\eta : \Sigma \to \mathbb{R}$ is *difference-bounded* if there exists real constants $a, b \in \mathbb{R}$ such that for every pair (σ, σ') of states in which σ' is a successor of σ, we have $a \leq \eta(\sigma') - \eta(\sigma) \leq b$.

To prove our main theorem (Theorem 8.3) regarding the use of difference-bounded PRSMs for concentration bounds on termination time of programs, we should first present the well-known Hoeffding's Inequality.

Theorem 8.2 (Hoeffding's Inequality [Hoeffding, 1994]). *Let $\{X_n\}_{n=1}^{\infty}$ be a supermartingale with respect to a filtration $\{\mathcal{F}_n\}_{n=1}^{\infty}$ and $\{[a_n, b_n]\}_{n=1}^{\infty}$ be a sequence of intervals of positive length in $\mathbb{R}$. If X_1 is a constant random variable and $X_n - X_{n-1} \in [a_n, b_n]$ almost-surely for all $n \in \mathbb{N}$, then*

$$\mathbb{P}(X_n - X_1 \geq \lambda) \leq e^{-\frac{2\cdot\lambda^2}{\sum_{k=2}^{n}(b_k - a_k)^2}}$$

for all $n \in \mathbb{N}$ and $\lambda > 0$.

Consider a difference-bounded PRSM η with parameters a, b. Recall that $X_n := \eta(\ell_n, \mathbf{v}_n)$. Define the stochastic process $\{Y_n\}_{n=1}^{\infty}$ by:

$$Y_n = X_n + \epsilon \cdot (\min\{T_S, n\} - 1).$$

The following lemma shows that $\{Y_n\}_{n=1}^{\infty}$ is a supermartingale and satisfies the requirements of Hoeffding's Inequaltiy.

Lemma 8.3. *$\{Y_n\}_{n\in\mathbb{N}}$ is a supermartingale and $Y_{n+1} - Y_n \in [a + \epsilon, b + \epsilon]$ almost surely for all $n \in \mathbb{N}$.*

Proof. Consider the following random variable:

$$U_n := \min\{T_S, n+1\} - \min\{T_S, n\},$$

and observe that this is equal to 1 if $T_S > n$ and 0 otherwise. Given that (i) the event $T_S > n$ is measurable in $\mathcal{F}_n$; and (ii) $X_n \geq 0$ iff $T_S > n$ (See conditions C2 and C3), we have

$$\begin{aligned}
\mathbb{E}(Y_{n+1} \mid \mathcal{F}_n) - Y_n &= \mathbb{E}(X_{n+1} \mid \mathcal{F}_n) - X_n + \epsilon \cdot \mathbb{E}(U_n \mid \mathcal{F}_n) \\
&= \mathbb{E}(X_{n+1} \mid \mathcal{F}_n) - X_n + \epsilon \cdot \mathbb{E}(\mathbf{1}_{T_S > n} \mid \mathcal{F}_n) \\
&= \mathbb{E}(X_{n+1} \mid \mathcal{F}_n) - X_n + \epsilon \cdot \mathbf{1}_{T_S > n} \\
&\le -\epsilon \cdot \mathbf{1}_{X_n \ge 0} + \epsilon \cdot \mathbf{1}_{T_S > n} \\
&= 0 \ .
\end{aligned}$$

Note that the inequality above is due to the fact that X_n is a ranking supermartingale. Moreover, since $T_S \le n$ implies $\ell_n = \perp$ and $X_{n+1} = X_n$ we have that $(X_{n+1} - X_n) = \mathbf{1}_{T_S > n} \cdot (X_{n+1} - X_n)$. Hence we have

$$\begin{aligned}
Y_{n+1} - Y_n &= X_{n+1} - X_n + \epsilon \cdot U_n \\
&= (X_{n+1} - X_n) + \epsilon \cdot \mathbf{1}_{T_S > n} \\
&= \mathbf{1}_{T_S > n} \cdot (X_{n+1} - X_n + \epsilon) \ .
\end{aligned}$$

Hence $Y_{n+1} - Y_n \in [a + \epsilon, b + \epsilon]$. □

We are now ready for our main theorem regarding concentration bounds:

Theorem 8.3. *Let S be a probabilistic transition system and η a difference-bounded PRSM of degree d with parameters a, b, ϵ, K as above. Then, for every scheduler s and initial valuation ν_0 and for all $n \in \mathbb{N}$, if $\epsilon \cdot (n-1) > \eta(\ell_0, \nu_0)$, we have*

$$\mathbb{P}^{\mathsf{s}}_{\nu_0}(T_S > n) \le e^{-\frac{2 \cdot (\epsilon \cdot (n-1) - \eta(\ell_0, \nu_0))^2}{(n-1) \cdot (b-a)^2}} .$$

Based on this theorem, a difference-bounded PRSM η implies a concentration bound of $\frac{\eta(\ell_0, \nu_0)}{\epsilon} + 2$. We now prove the theorem:

Proof. Let $W_0 := Y_1 = \eta(\ell_0, \nu_0)$. Fix any scheduler s. By Hoeffing's Inequality, for all $\lambda > 0$, we have $\mathbb{P}^{\mathsf{s}}_{\nu_0}(Y_n - W_0 \ge \lambda) \le e^{-\frac{2 \cdot \lambda^2}{(n-1) \cdot (b-a)^2}}$. Note that $T_S > n$ iff $X_n \ge 0$ by conditions C2 and C3 of PRSM. Let $\alpha = \epsilon \cdot (n-1) - W_0$ and $\hat{\alpha} = \epsilon \cdot (\min\{n, T_S\} - 1) - W_0$. Note that when $T_S > n$, α and $\hat{\alpha}$ coincide. Thus, for $\mathbb{P}(T_S > n) = \mathbb{P}(X_n \geq 0 \wedge T_S > n)$ we have

$$\begin{aligned}
\mathbb{P}(X_n \geq 0 \wedge T_S > n) &= \mathbb{P}((X_n + \alpha \geq \alpha) \wedge (T_S > n)) \\
&= \mathbb{P}((X_n + \hat{\alpha} \geq \alpha) \wedge (T_S > n)) \\
&\leq \mathbb{P}((X_n + \hat{\alpha} \geq \alpha)) \\
&= \mathbb{P}(Y_n - Y_1 \geq \epsilon \cdot (n-1) - W_0) \\
&\leq e^{-\frac{2 \cdot (\epsilon \cdot (n-1) - W_0)^2}{(n-1) \cdot (b-a)^2}}
\end{aligned}$$

for all $n > \frac{W_0}{\epsilon} + 1$. The first equality is obtained by simply adding α on both sides, and the second equality holds because when $T_S > n$ we have $\min\{n, T_S\} = n$ which ensures $\alpha = \hat{\alpha}$. The first inequality is obtained by simply dropping the conjunct $T_S > n$. The following equality is by definition, and the final inequality is an application of Theorem 8.2. □

Example 8.3. *Consider again our running example of Figure 8.1 with the invariant* Inv *given in Example 8.2. Let η be the function of Table 8.1. We can verify that the interval $[a, b] = [-10.2, 8.6]$ satisfies the conditions of difference-bounded PRSM:*

- *for all $x \in [1, 10]$, $\eta(2, x) - \eta(1, x) = -0.2$;*
- *for all $x \in [0, 1) \cup (10, 11]$, $-10.2 \leq \eta(\perp, x) - \eta(1, x) \leq -0.2$;*
- *for all $x \in [1, 10]$ and $i \in \{3, 4\}$, $\eta(i, x) - \eta(2, x) = -0.2$;*
- *for all $x \in [1, 10]$ and $i \in \{5, 6\}$, $-9.4 \leq \eta(i, x) - \eta(4, x) \leq 8.6$;*
- *for all $x \in [1, 10]$, $\eta(1, x-1) - \eta(5, x) = -0.2$;*
- *for all $x \in [1, 10]$, $\eta(1, x+1) - \eta(6, x) = -0.2$;*
- *for all $x \in [1, 10]$ and $r \in \{-1, 1\}$, $-9.6 \leq \eta(1, x+r) - \eta(3, x) \leq 8.4$.*

Therefore, by Theorem 8.3, assuming that the program is run with the initial value $x_0 = 5$, we deduce:

$$\mathbb{P}(T_S > 50000) \leq e^{-\frac{2 \cdot (0.2 \cdot 49999 - 30)^2}{49999 \cdot 18.8^2}} \approx 1.3016 \cdot 10^{-5}.$$

We end this section with a decidability result for the synthesis of PRSMs and difference-bounded PRSMs that can simply be obtained by applying quantifier elimination. However, note that quantifier elimination is extremely inefficient. As such, the next sections of this chapter focus on more efficient algorithms for synthesizing PRSMs.

Theorem 8.4. *Given a polynomial probabilistic transition system $S = (\mathbf{V}, \mathbf{L}, \mathbf{L}_p, \ell_0, I, \mathbf{\Theta})$ and a polynomial invariant Inv for S, for any fixed degree $d \in \mathbb{N}$, the problem of deciding whether there exists a (difference-bounded) PRSM of degree d for S is encodable in the existential theory of the reals and hence decidable.*

8.4 Synthesizing Polynomial Ranking Supermartingales

Our synthesis algorithm is quite similar to those of Chapters 6 and 7. As such, we forgo most details in this section. Our method is based on the positivstellensätze introduced in Section 2.6. Below, we fix an input polynomial probabilistic transition system $S = (\mathbf{V}, \mathbf{L}, \mathbf{L}_p, \ell_0, I, \mathbf{\Theta})$, an input polynomial invariant Inv and an input initial configuration (ℓ_0, ν_0) in which $\nu_0 \models I \ \wedge \ \mathsf{Inv}(\ell_0)$. We also assume that the degree d and technical parameter Υ are given as part of the input.

Our Algorithm. We present a succinct description of the key steps.

1. *Template η for a PRSM.* The algorithm constructs the set M_d of all monomials over $\mathbf{V}$ of degree no greater than d, and sets up a template PRSM η such that $\eta(\ell, \cdot)$ is the polynomial $\sum_{h \in M_d} a_{h,\ell} \cdot h$ where each $a_{h,\ell}$ is a new unknown variable. Our goal is to synthesize values for $a_{h,\ell}$'s so that η becomes a valid PRSM.

2. *Fixed Values for ϵ and K.* Based on Remark 8.2, the algorithm fixes ϵ to be 1 and K to be -1.

3. *Computing pre_η.* The algorithm symbolically computes pre_η. See Example 8.1. Note that all coefficients in pre_η are linear combinations of $a_{h,\ell}$'s.

4. *Generating Constraint Pairs.* The algorithm generates constraint pairs modeling (C1)–(C4). If the goal is to synthesize a difference-bounded PRSM, it also creates new

unknown variables for the parameters a and b and creates constraint pairs modeling difference-boundedness. For a more detailed treatment of constraint pairs see Section 7.4.1.

5. *Translating Constraint Pairs to Constraints over Unknown Variables.* The algorithm uses Putinar's Positivstellensatz (Theorem 2.2), Schmüdgen's Positivstellensatz (Theorem 2.3) or Handelman's Theorem (Theorem 2.1) to translate the constraint pairs into quadratic constraints. Moreover, it uses Υ as a bound on the degree of the sum-of-square polynomials or the number of multiplicands in monoid elements. This is the exact same process as in Sections 7.4.2 and 7.4.3, except that there is no need to synthesize positivity witnesses here (Remark 8.2).

6. *Solution via Semi-definite or Linear Programming.* The algorithm calls a Semi-definite Programming (SDP) solver for Schmüdgen's and Putinar's Positivstellensätze or Linear Programming (LP) solver for Handelman's Theorem in order to check the feasibility of the constraints generated in the previous step and optimize the runtime upper-bound $\frac{\eta(\ell_0,\mathrm{v}_0)-K}{\epsilon}$ (see Theorem 8.1). Note that feasibility implies the existence of a (difference-bounded) PRSM of degree d, which in turn implies finite termination. Similarly, the existence of a difference-bounded PRSM implies a concentration bound through Theorem 8.3.

Remark 8.3. *Unlike our algorithms in Chapters 6 and 7, we do not need to use general quadratic programming here and can simply use linear or semi-definite programming instead. This is because (i) the polynomials on the left-hand sides of constraint pairs generated in Step 4 above have fixed coefficients, i.e. there are no unknown variables in their coefficients, and (ii) the QP instance modeling sum-of-square polynomials is solvable using semi-definite programming (Remark 2.2).*

The arguments for soundness, semi-completeness, and complexity of our approach are similar to Chapters 6 and 7. We have the following theorems:

Theorem 8.5 (Soundness). *Given a polynomial probabilistic transition system $S = (\mathbf{V}, \mathbf{L}, \mathbf{L}_p, \ell_0, I, \mathbf{\Theta})$ and a polynomial invariant Inv, every solution obtained in Step 6 of our algorithm leads to a function η that is a valid (difference-bounded) PRSM of degree d for S with respect to Inv.*

Theorem 8.6 (Semi-completeness). *In the bounded reals model of computation, given a polynomial probabilistic transition system* $S = (\mathbf{V}, \mathbf{L}, \mathbf{L}_p, \ell_0, I, \mathbf{\Theta})$ *and a polynomial invariant* Inv*, let* η *be a valid PRSM of degree* d *for* S *with respect to* Inv*. Then, there exists a constant* Υ_η *such that for all* $\Upsilon > \Upsilon_\eta$ *the linear/semi-definite programming instance in Step 6 of the above algorithm has a solution that corresponds to* η*.*

Theorem 8.7 (Efficiency). *For a polynomial probabilistic transition system* $S = (\mathbf{V}, \mathbf{L}, \mathbf{L}_p, \ell_0, I, \mathbf{\Theta})$ *and a polynomial invariant* Inv*, both of constant degree, our algorithm runs in polynomial time, assuming that* d, Υ *are fixed constants.*

Proof. Similar to Chapters 6 and 7, our algorithm provides a polynomial-time reduction to quadratic programming. However, in this case, our QP instances are indeed LP or SDP instances. It is well-known that both LP and SDP are solvable in polynomial time [Grötschel et al., 2012]. □

8.5 Experimental Results

We now present experimental results for our algorithm over several classical programs.

Solvers. We implemented our approach in C++ and used the semi-definite programming tool SOSTOOLS [Papachristodoulou et al., 2013] (which in turn depends on SeDuMi [Sturm, 1999]) and the linear programming tool CPLEX [IBM, 2019] for solving the constraint systems.

Experimental Examples and Setup. We consider six classical examples of probabilistic programs exhibiting various types of non-linear behavior. Our examples are as follows:

- *Logistic Map* (Figure 8.2) is adapted from [Cousot, 2005]. It was previously handled by Lagrangian relaxation, whereas our approach solves it using linear programming;
- *Decay* (Figure 8.3) models a sequence of points converging stochastically to the origin;
- *Random Walk* (Figure 8.4) models a random walk within a bounded region defined through non-linear curves;

Example	**Method**	**SOSTOOLS**	**error**	$\eta(\ell_0, \cdot)$
Decay	Putinar	0.1248s	$\leq 10^{-9}$	$5282.3435 \cdot x^2 + 5282.3435 \cdot y^2 + 1$
Random Walk	Schmüdgen	0.7176s	$\leq 10^{-7}$	$-300 \cdot x^2 - 300 \cdot y^2 + 601$
Example	**Method**	**CPLEX**	-	$\eta(\ell_0, \cdot)$
Gambler's Ruin	Handelman	$\leq 10^{-2}$s	-	$33 \cdot x - 3 \cdot x^2$
Gambler's Ruin V.	Handelman	$\leq 10^{-2}$s	-	$-21 + 100 \cdot x - 70 \cdot y - 100 \cdot x^2 + 100 \cdot x \cdot y$
Logistic Map	Handelman	$\leq 10^{-2}$s	-	$1000500.7496 \cdot x$
Nested Loop	Handelman	$\leq 2 \cdot 10^{-2}$s	-	$48 + 160 \cdot n + (m - x) \cdot (800 \cdot n + 240)$

Table 8.2: Our Experimental Results over the Example Programs. The top portion shows results obtained using SDP and the bottom portion shows LP-based results.

- *Gambler's Ruin* (Figure 8.1) is our running example;
- *Gambler's Ruin Variant* (Figure 8.5) is a variant of Figure 8.1; and
- *Nested Loop* (Figure 8.6) is a nested loop with stochastic increments.

In all examples the invariants (shown in brackets) were straightforward and manually generated. Alternatively, one can use our approach in Chapter 6 to automatically generating the invariants. See our technical report [Chatterjee et al., 2016b] for more details.

Experimental Results. In Table 8.2, we present the experimental results, where "Method" shows whether we used Handelman's Theorem, Putinar's Positivstellensatz or Schmüdgen's Positivstellensatz to synthesize the PRSM, "SOSTOOLS/CPLEX" is the running time for these tools in seconds, "error" is the maximal numerical error of equality constraints added into SOSTOOLS, and $\eta(\ell_0, \cdot)$ is the polynomial for the initial label in the synthesized PRSM. All numbers except errors are rounded to 10^{-4}. The experimental results were obtained on Intel Core i7-2600 3.4 GHz machine with 16GB RAM.

Comparison with Previous Approaches. Except for Logistic Map, no previous approach can prove almost-sure termination over our example programs. For the Logistic Map example, our reduction is to linear programming whereas existing approaches [Cousot, 2005, Shen et al., 2013] reduce to semi-definite programming.

```
[$0 \leq a \leq 1 \wedge 0 \leq x \leq 1$]
while $0 \leq a \leq 0.999 \wedge 0.001 \leq x \leq 1$ do
    [$0 \leq a \leq 0.999 \wedge 0.001 \leq x \leq 1$]
    $x := a \cdot x \cdot (1 - x)$
od
```

Figure 8.2: Logistic Map

```
[$x^2 + y^2 \leq 2$]
while $0.1 \leq x^2 + y^2 \leq 1$ do
    [$0.1 \leq x^2 + y^2 \leq 1$]
    $\begin{pmatrix} x \\ y \end{pmatrix} := \begin{pmatrix} \mathrm{UNIF}(0.98, 1) \cdot x + 0.01 \cdot y \\ \mathrm{UNIF}(0.98, 1) \cdot y - 0.01 \cdot x \end{pmatrix}$
od
```

Figure 8.3: Decay

```
[$x^2 + y^2 \leq 2$]
while $x^2 + y \leq 1 \wedge x^2 - y \leq 1$ do
    [$x^2 + y \leq 1 \wedge x^2 - y \leq 1$]
    $\begin{pmatrix} x \\ y \end{pmatrix} := \begin{pmatrix} x + \mathrm{UNIF}(-0.1, 0.1) \\ y + \mathrm{UNIF}(-0.1, 0.1) \end{pmatrix}$
od
```

Figure 8.4: Random Walk

```
[$0.7 \leq x \leq y + 0.3$]
while $1 \leq x \leq y$ do
    [$1 \leq x \leq y$]
    if $\star$ do
        [$1 \leq x \leq y$]
        $x := x + \mathrm{UNIF}(-0.3, 0.3)$
    else
        [$1 \leq x \leq y$]
        if prob(0.5) do
            [$1 \leq x \leq y$]
            $x := x + 0.1$
        else
            [$1 \leq x \leq y$]
            $x := x - 0.1$
        fi
    fi
od
```

Figure 8.5: Gambler's Ruin Variant

```
[$x \leq m + 0.2 \wedge n \geq 0$]
while $x \leq m$ do
    [$x \leq m \wedge n \geq 0$]
    y := 0;
    [$x \leq m \wedge y \leq n + 0.2 \wedge n \geq 0$]
    while $y \leq n$ do
        [$x \leq m \wedge y \leq n \wedge n \geq 0$]
        $y := y + \mathrm{UNIF}(-0.1, 0.2)$
    od;
    [$x \leq m \wedge y \geq n \wedge n \geq 0$]
    $x := x + \mathrm{UNIF}(-0.1, 0.2)$
od
```

Figure 8.6: Nested Loop

References

[Adjé et al., 2015] Adjé, A., Garoche, P.-L., and Magron, V. (2015). Property-based polynomial invariant generation using sums-of-squares optimization. In *Static Analysis Symposium (SAS)*, pages 235–251.

[Adjé et al., 2010] Adjé, A., Gaubert, S., and Goubault, E. (2010). Coupling policy iteration with semi-definite relaxation to compute accurate numerical invariants in static analysis. In *European Symposium on Programming (ESOP)*, pages 23–42.

[Afzal et al., 2020] Afzal, M., Chakraborty, S., Chauhan, A., Chimdyalwar, B., Darke, P., Gupta, A., Kumar, S., M, C. B., Unadkat, D., and Venkatesh, R. (2020). VeriAbs : Verification by abstraction and test generation (competition contribution). In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 383–387.

[Albarghouthi et al., 2012a] Albarghouthi, A., Gurfinkel, A., and Chechik, M. (2012a). From under-approximations to over-approximations and back. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 157–172.

[Albarghouthi et al., 2012b] Albarghouthi, A., Li, Y., Gurfinkel, A., and Chechik, M. (2012b). Ufo: A framework for abstraction-and interpolation-based software verification. In *Conference on Computer-Aided Verification (CAV)*, pages 672–678.

[Alias et al., 2010] Alias, C., Darte, A., Feautrier, P., and Gonnord, L. (2010). Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *Static Analysis Symposium (SAS)*, pages 117–133.

[Alur et al., 2006] Alur, R., Dang, T., and Ivančić, F. (2006). Predicate abstraction for reachability analysis of hybrid systems. *ACM transactions on embedded computing systems (TECS)*, 5(1):152–199.

[Alur and Dill, 1990] Alur, R. and Dill, D. L. (1990). Automata for modeling real-time systems. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 322–335.

[Alur et al., 1995] Alur, R., Itai, A., Kurshan, R. P., and Yannakakis, M. (1995). Timing verification by successive approximation. *Information and Computation*, 118(1):142–157.

[Amato et al., 2007] Amato, C., Bernstein, D. S., and Zilberstein, S. (2007). Solving POMDPs using quadratically constrained linear programs. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2418–2424.

[Andersen and Andersen, 2018] Andersen, E. D. and Andersen, K. D. (2018). MOSEK optimization suite.

[Appel and Palsberg, 2003] Appel, A. W. and Palsberg, J. (2003). *Modern Compiler Implementation in Java.* Cambridge University Press, 2nd edition.

[Arnborg et al., 1987] Arnborg, S., Corneil, D. G., and Proskurowski, A. (1987). Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284.

[Arzt et al., 2014] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. (2014). FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 259–269.

[Asadi et al., 2020a] Asadi, A., Chatterjee, K., Fu, H., Goharshady, A. K., and Mahdavi, M. (2020a). Inductive reachability witnesses. *arXiv preprint arXiv:2007.14259.*

[Asadi et al., 2020b] Asadi, A., Chatterjee, K., Goharshady, A. K., Mohammadi, K., and Pavlogiannis, A. (2020b). Faster algorithms for quantitative analysis of Markov chains and Markov decision processes with small treewidth. *arXiv preprint arXiv:2004.08828.*

[Asadi et al., 2020c] Asadi, A., Chatterjee, K., Goharshady, A. K., Mohammadi, K., and Pavlogiannis, A. (2020c). Faster algorithms for quantitative analysis of MCs and MDPs with small treewidth. In *International Symposium on Automated Technology for Verification and Analysis (ATVA).*

[Ashok et al., 2017] Ashok, P., Chatterjee, K., Daca, P., Křetínský, J., and Meggendorfer, T. (2017). Value iteration for long-run average reward in Markov decision processes. In *Conference on Computer-Aided Verification (CAV)*, pages 201–221.

[Atig and Ganty, 2011] Atig, M. F. and Ganty, P. (2011). Approximating Petri net reachability along context-free traces. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 152–163.

[Atiyah and Macdonald, 1969] Atiyah, M. F. and Macdonald, I. G. (1969). *Introduction to Commutative Algebra.* Taylor and Francis.

[Babić et al., 2013] Babić, D., Cook, B., Hu, A. J., and Rakamarić, Z. (2013). Proving termination of nonlinear command sequences. *Formal Aspects of Computing*, 25(3):389–403.

[Back and Wright, 2012] Back, R.-J. and Wright, J. (2012). *Refinement calculus: a systematic introduction.* Springer.

[Bagnara et al., 2005] Bagnara, R., Rodríguez-Carbonell, E., and Zaffanella, E. (2005). Generation of basic semi-algebraic invariants using convex polyhedra. In *Static Analysis Symposium (SAS)*, pages 19–34.

[Baier and Katoen, 2008] Baier, C. and Katoen, J. (2008). *Principles of model checking.* MIT Press.

[Balarin and Sangiovanni-Vincentelli, 1993] Balarin, F. and Sangiovanni-Vincentelli, A. L. (1993). An iterative approach to language containment. In *Conference on Computer-Aided Verification (CAV)*, pages 29–40.

[Ball et al., 2011] Ball, T., Levin, V., and Rajamani, S. K. (2011). A decade of software model checking with SLAM. *Communications of the ACM*, 54(7):68–76.

[Ball and Rajamani, 2002] Ball, T. and Rajamani, S. K. (2002). The SLAM project: debugging system software via static analysis. In *Symposium on Principles of Programming Languages (POPL)*.

[Basu et al., 2007] Basu, S., Pollack, R., and Coste-Roy, M.-F. (2007). *Algorithms in real algebraic geometry*. Springer.

[Bellman, 1957] Bellman, R. (1957). A Markovian decision process. *Journal of Mathematics and Mechanics*, pages 679–684.

[Ben Sassi et al., 2015] Ben Sassi, M. A., Sankaranarayanan, S., Chen, X., and Ábrahám, E. (2015). Linear relaxations of polynomial positivity for polynomial Lyapunov function synthesis. *IMA Journal of Mathematical Control and Information*, 33(3):723–756.

[Benedikt et al., 2013] Benedikt, M., Lenhardt, R., and Worrell, J. (2013). LTL model checking of interval Markov chains. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 32–46.

[Berend and Tassa, 2010] Berend, D. and Tassa, T. (2010). Improved bounds on bell numbers and on moments of sums of random variables. *Probability and Mathematical Statistics*, 30(2):185–205.

[Berkelaar et al., 2003] Berkelaar, M., Eikland, K., and Notebaert, P. (2003). *lpsolve Linear Programming system*.

[Beyer, 2020] Beyer, D. (2020). Advances in automatic software verification: SV-COMP 2020. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 347–367.

[Beyer et al., 2007] Beyer, D., Henzinger, T. A., Jhala, R., and Majumdar, R. (2007). The software model checker Blast. *International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525.

[Beyer and Keremoglu, 2011] Beyer, D. and Keremoglu, M. E. (2011). CPAchecker: A tool for configurable software verification. In *Conference on Computer-Aided Verification (CAV)*, pages 184–190.

[Blackburn et al., 2006] Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. (2006). The DaCapo benchmarks: Java benchmarking development and analysis. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 169–190.

[Bodden, 2012] Bodden, E. (2012). Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *State of the Art in Java Program Analysis (SOAP)*, pages 3–8.

[Bodik et al., 2010] Bodik, R., Chandra, S., Galenson, J., Kimelman, D., Tung, N., Barman, S., and Rodarmor, C. (2010). Programming with angelic nondeterminism. In *Symposium on Principles of Programming Languages (POPL)*, pages 339–352.

[Bodlaender, 1994] Bodlaender, H. L. (1994). A tourist guide through treewidth. *Acta cybernetica*, 11.

[Bodlaender, 1996] Bodlaender, H. L. (1996). A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on computing*, 25(6):1305–1317.

[Bodlaender, 1998] Bodlaender, H. L. (1998). A partial k-arboretum of graphs with bounded treewidth. *Theoretical computer science*, 209:1–45.

[Bodlaender and Hagerup, 1998] Bodlaender, H. L. and Hagerup, T. (1998). Parallel algorithms with optimal speedup for bounded treewidth. *SIAM Journal on Computing*, 27(6):1725–1746.

[Borodin et al., 1995] Borodin, A., Irani, S., Raghavan, P., and Schieber, B. (1995). Competitive paging with locality of reference. *Journal of Computer and System Sciences*, 50(2):244–258.

[Bournez and Garnier, 2005] Bournez, O. and Garnier, F. (2005). Proving positive almost-sure termination. In *Conference on Rewriting Techniques and Applications (RTA)*, pages 323–337.

[Bradley et al., 2005a] Bradley, A. R., Manna, Z., and Sipma, H. B. (2005a). Linear ranking with reachability. In *Conference on Computer-Aided Verification (CAV)*, pages 491–504.

[Bradley et al., 2005b] Bradley, A. R., Manna, Z., and Sipma, H. B. (2005b). Termination of polynomial programs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 113–129.

[Brown, 2010] Brown, C. W. (2010). QEPCAD - quantifier elimination by partial cylindrical algebraic decomposition.

[Burgstaller et al., 2004] Burgstaller, B., Blieberger, J., and Scholz, B. (2004). On the tree width of Ada programs. In *Ada-Europe*, pages 78–90.

[Cadar and Sen, 2013] Cadar, C. and Sen, K. (2013). Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90.

[Calder et al., 1998] Calder, B., Krintz, C., John, S., and Austin, T. (1998). Cache-conscious data placement. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 139–149.

[Chakarov and Sankaranarayanan, 2013] Chakarov, A. and Sankaranarayanan, S. (2013). Probabilistic program analysis with martingales. In *Conference on Computer-Aided Verification (CAV)*, pages 511–526.

[Chakarov and Sankaranarayanan, 2014] Chakarov, A. and Sankaranarayanan, S. (2014). Expectation invariants for probabilistic program loops as fixed points. In *Static Analysis Symposium (SAS)*, pages 85–100.

[Chatterjee et al., 2016a] Chatterjee, K., Fu, H., and Goharshady, A. K. (2016a). Termination analysis of probabilistic programs through positivstellensatz's. In *International Conference on Computer Aided Verification (CAV)*.

[Chatterjee et al., 2016b] Chatterjee, K., Fu, H., and Goharshady, A. K. (2016b). Termination analysis of probabilistic programs through positivstellensatz's. *arXiv preprint arXiv:1604.07169*.

[Chatterjee et al., 2017a] Chatterjee, K., Fu, H., and Goharshady, A. K. (2017a). Non-polynomial worst-case analysis of recursive programs. In *International Conference on Computer Aided Verification (CAV)*.

[Chatterjee et al., 2019a] Chatterjee, K., Fu, H., and Goharshady, A. K. (2019a). Non-polynomial worst-case analysis of recursive programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*.

[Chatterjee et al., 2020a] Chatterjee, K., Fu, H., Goharshady, A. K., and Goharshady, E. K. (2020a). Polynomial invariant generation for non-deterministic recursive programs. *HAL preprint HAL:02015843*.

[Chatterjee et al., 2020b] Chatterjee, K., Fu, H., Goharshady, A. K., and Goharshady, E. K. (2020b). Polynomial invariant generation for non-deterministic recursive programs. In *ACM Conference on Programming Language Design and Implementation (PLDI)*.

[Chatterjee et al., 2018a] Chatterjee, K., Fu, H., Goharshady, A. K., and Okati, N. (2018a). Computational approaches for stochastic shortest path on succinct mdps. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

[Chatterjee et al., 2016c] Chatterjee, K., Fu, H., Novotný, P., and Hasheminezhad, R. (2016c). Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. In *Symposium on Principles of Programming Languages (POPL)*, pages 327–342.

[Chatterjee et al., 2019b] Chatterjee, K., Goharshady, A., and Goharshady, E. (2019b). The treewidth of smart contracts. In *Symposium on Applied Computing (SAC)*.

[Chatterjee et al., 2019c] Chatterjee, K., Goharshady, A. K., Goyal, P., Ibsen-Jensen, R., and Pavlogiannis, A. (2019c). Faster algorithms for dynamic algebraic queries in basic rsms with constant treewidth. *ACM Transactions on Programming Languages and Systems (TOPLAS)*.

[Chatterjee et al., 2016d] Chatterjee, K., Goharshady, A. K., Ibsen-Jensen, R., and Pavlogiannis, A. (2016d). Algorithms for algebraic path properties in concurrent systems of constant treewidth components. In *ACM Symposium on Principles of Programming Languages (POPL).*

[Chatterjee et al., 2018b] Chatterjee, K., Goharshady, A. K., Ibsen-Jensen, R., and Pavlogiannis, A. (2018b). Algorithms for algebraic path properties in concurrent systems of constant treewidth components. *ACM Transactions on Programming Languages and Systems (TOPLAS).*

[Chatterjee et al., 2020c] Chatterjee, K., Goharshady, A. K., Ibsen-Jensen, R., and Pavlogiannis, A. (2020c). Optimal and perfectly parallel algorithms for on-demand data-flow analysis. *arXiv preprint arXiv:2001.11070.*

[Chatterjee et al., 2020d] Chatterjee, K., Goharshady, A. K., Ibsen-Jensen, R., and Pavlogiannis, A. (2020d). Optimal and perfectly parallel algorithms for on-demand data-flow analysis. In *European Symposium on Programming (ESOP).*

[Chatterjee et al., 2018c] Chatterjee, K., Goharshady, A. K., Ibsen-Jensen, R., and Velner, Y. (2018c). Ergodic mean-payoff games for the analysis of attacks in cryptocurrencies. In *International Conference on Concurrency Theory (CONCUR).*

[Chatterjee et al., 2019d] Chatterjee, K., Goharshady, A. K., Okati, N., and Pavlogiannis, A. (2019d). Efficient parameterized algorithms for data packing. *HAL preprint HAL:01897615.*

[Chatterjee et al., 2019e] Chatterjee, K., Goharshady, A. K., Okati, N., and Pavlogiannis, A. (2019e). Efficient parameterized algorithms for data packing. In *ACM Symposium on Principles of Programming Languages (POPL).*

[Chatterjee et al., 2017b] Chatterjee, K., Goharshady, A. K., and Pavlogiannis, A. (2017b). JTDec: A tool for tree decompositions in soot. In *Symposium on Automated Technology for Verification and Analysis (ATVA).*

[Chatterjee et al., 2019f] Chatterjee, K., Goharshady, A. K., and Pourdamghani, A. (2019f). Hybrid mining: Exploiting blockchain's computational power for distributed problem solving. In *ACM Symposium on Applied Computing (SAC).*

[Chatterjee et al., 2019g] Chatterjee, K., Goharshady, A. K., and Pourdamghani, A. (2019g). Probabilistic smart contracts: Secure randomness on the blockchain. In *IEEE International Conference on Blockchain and Cryptocurrency (ICBC).*

[Chatterjee et al., 2018d] Chatterjee, K., Goharshady, A. K., and Velner, Y. (2018d). Quantitative analysis of smart contracts. In *European Symposium on Programming (ESOP).*

[Chatterjee et al., 2018e] Chatterjee, K., Henzinger, M., Loitzenbauer, V., Oraee, S., and Toman, V. (2018e). Symbolic algorithms for graphs and Markov decision processes with fairness objectives. In *Conference on Computer-Aided Verification (CAV).*

[Chatterjee and Henzinger, 2008] Chatterjee, K. and Henzinger, T. A. (2008). Value iteration. In *Model Checking*, pages 107–138.

[Chatterjee et al., 2010] Chatterjee, K., Henzinger, T. A., Jobstmann, B., and Singh, R. (2010). Measuring and synthesizing systems in probabilistic environments. In *Conference on Computer-Aided Verification (CAV)*, pages 380–395.

[Chatterjee and Łącki, 2013] Chatterjee, K. and Łącki, J. (2013). Faster algorithms for Markov decision processes with low treewidth. In *Conference on Computer-Aided Verification (CAV)*, pages 543–558.

[Chatterjee et al., 2017c] Chatterjee, K., Novotný, P., and Zikelic, D. (2017c). Stochastic invariants for probabilistic termination. In *Symposium on Principles of Programming Languages (POPL)*, pages 145–160.

[Chaudhuri, 2008] Chaudhuri, S. (2008). Subcubic algorithms for recursive state machines. In *Symposium on Principles of Programming Languages (POPL).*

[Chaudhuri and Zaroliagis, 2000] Chaudhuri, S. and Zaroliagis, C. D. (2000). Shortest paths in digraphs of small treewidth. part i: Sequential algorithms. *Algorithmica*, 27(3-4):212–226.

[Chen et al., 2017] Chen, C., Atamtürk, A., and Oren, S. S. (2017). A spatial branch-and-cut method for nonconvex QCQP with bounded complex variables. *Mathematical Programming*, 165(2):549–577.

[Chen et al., 2007] Chen, Y., Xia, B., Yang, L., Zhan, N., and Zhou, C. (2007). Discovering non-linear ranking functions by solving semi-algebraic systems. In *International Colloquium on Theoretical Aspects of Computing (ICTAC)*, pages 34–49.

[Chen et al., 2015] Chen, Y.-F., Hong, C.-D., Wang, B.-Y., and Zhang, L. (2015). Counterexample-guided polynomial loop invariant generation by lagrange interpolation. In *Conference on Computer Aided Verification (CAV)*, pages 658–674.

[Chonev, 2019] Chonev, V. (2019). Reachability in augmented interval Markov chains. In *Conference on Reachability Problems (RP)*, pages 79–92.

[Clarke et al., 2000] Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., and Veith, H. (2000). Counterexample-guided abstraction refinement. In *Conference on Computer-Aided Verification (CAV)*, pages 154–169.

[Clarke et al., 2018] Clarke, E. M., Henzinger, T. A., Veith, H., and Bloem, R. (2018). *Handbook of model checking*. Springer.

[Colón et al., 2003] Colón, M., Sankaranarayanan, S., and Sipma, H. (2003). Linear invariant generation using non-linear constraint solving. In *Conference on Computer-Aided Verification (CAV)*, pages 420–432.

[Colón and Sipma, 2001] Colón, M. A. and Sipma, H. B. (2001). Synthesis of linear ranking functions. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 67–81.

[Cousot, 2005] Cousot, P. (2005). Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 1–24.

[Cousot and Cousot, 1977] Cousot, P. and Cousot, R. (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages (POPL)*, pages 238–252.

[Cousot et al., 2005] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. (2005). The Astreé analyzer. In *European Symposium on Programming (ESOP)*, pages 21–30.

[Cousot and Halbwachs, 1978] Cousot, P. and Halbwachs, N. (1978). Automatic discovery of linear restraints among variables of a program. In *Symposium on Principles of Programming Languages (POPL)*, pages 84–96.

[Csallner et al., 2008] Csallner, C., Tillmann, N., and Smaragdakis, Y. (2008). DySy: dynamic symbolic execution for invariant inference. In *International Conference on Software Engineering (ICSE)*, pages 281–290.

[Cygan et al., 2015] Cygan, M., Fomin, F. V., Kowalik, Ł., Lokshtanov, D., Marx, D., Pilipczuk, M., Pilipczuk, M., and Saurabh, S. (2015). *Parameterized algorithms*. Springer.

[Czerwinski et al., 2019] Czerwinski, W., Lasota, S., Lazic, R., Leroux, J., and Mazowiecki, F. (2019). The reachability problem for Petri nets is not elementary. In *Symposium on Theory of Computing (STOC)*, pages 24–33.

[Darondeau et al., 2012] Darondeau, P., Demri, S., Meyer, R., and Morvan, C. (2012). Petri net reachability graphs: Decidability status of first order properties. *Logical Methods in Computer Science*, 8(4).

[Daws, 2004] Daws, C. (2004). Symbolic and parametric model checking of discrete-time Markov chains. In *International Colloquium on Theoretical Aspects of Computing (ICTAC)*, pages 280–294.

[De Alfaro et al., 2003] De Alfaro, L., Henzinger, T. A., and Majumdar, R. (2003). Discounting the future in systems theory. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 1022–1037.

[De Moura and Bjørner, 2008] De Moura, L. and Bjørner, N. (2008). Z3: An efficient smt solver. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340.

[de Oliveira et al., 2016] de Oliveira, S., Bensalem, S., and Prevosto, V. (2016). Polynomial invariants by linear algebra. In *Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 479–494.

[de Vries and Koutavas, 2011] de Vries, E. and Koutavas, V. (2011). Reverse Hoare logic. In *Software Engineering and Formal Methods (SEFM)*, pages 155–171.

[Dehnert et al., 2017] Dehnert, C., Junges, S., Katoen, J.-P., and Volk, M. (2017). A storm is coming: A modern probabilistic model checker. In *Conference on Computer-Aided Verification (CAV)*, pages 592–600.

[Dillig et al., 2013] Dillig, I., Dillig, T., Li, B., and McMillan, K. L. (2013). Inductive invariant generation via abductive inference. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 443–456.

[Ding and Kennedy, 1999] Ding, C. and Kennedy, K. (1999). Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 229–241.

[Ding and Kandemir, 2014] Ding, W. and Kandemir, M. (2014). CApRI: Cache-conscious data reordering for irregular codes. *ACM SIGMETRICS Performance Evaluation Review*, 42(1):477–489.

[Distefano et al., 2019] Distefano, D., Fähndrich, M., Logozzo, F., and O'Hearn, P. W. (2019). Scaling static analyses at facebook. *Communications of the ACM*, 62(8):62–70.

[Downey and Fellows, 2012] Downey, R. G. and Fellows, M. R. (2012). *Parameterized complexity*. Springer.

[Dubhashi and Panconesi, 2009] Dubhashi, D. P. and Panconesi, A. (2009). *Concentration of measure for the analysis of randomized algorithms*. Cambridge University Press.

[Durrett, 2019] Durrett, R. (2019). *Probability: theory and examples*. Cambridge university press.

[Esparza et al., 2012] Esparza, J., Gaiser, A., and Kiefer, S. (2012). Proving termination of probabilistic programs using patterns. In *Conference on Computer-Aided Verification (CAV)*, pages 123–138.

[Farkas, 1902] Farkas, J. (1902). Theorie der einfachen ungleichungen. *Journal für die reine und angewandte Mathematik*, (124):1–27.

[Farzan and Kincaid, 2015] Farzan, A. and Kincaid, Z. (2015). Compositional recurrence analysis. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 57–64.

[Fearnley, 2010] Fearnley, J. (2010). Exponential lower bounds for policy iteration. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 551–562.

[Feinberg, 2012] Feinberg, E. A. (2012). *Handbook of Markov decision processes.* Springer.

[Feng et al., 2017] Feng, Y., Zhang, L., Jansen, D. N., Zhan, N., and Xia, B. (2017). Finding polynomial loop invariants for probabilistic programs. In *Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 400–416.

[Ferrara et al., 2005] Ferrara, A., Pan, G., and Vardi, M. Y. (2005). Treewidth in verification: Local vs. global. In *Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, pages 489–503.

[Filar and Vrieze, 1996] Filar, J. and Vrieze, K. (1996). *Competitive Markov Decision Processes.* Springer.

[Fioriti and Hermanns, 2015] Fioriti, L. M. F. and Hermanns, H. (2015). Probabilistic termination: Soundness, completeness, and compositionality. In *Symposium on Principles of Programming Languages (POPL)*, pages 489–501.

[Floyd, 1993] Floyd, R. W. (1993). Assigning meanings to programs. In *Program Verification*, pages 65–81.

[Fomin et al., 2018] Fomin, F. V., Lokshtanov, D., Saurabh, S., Pilipczuk, M., and Wrochna, M. (2018). Fully polynomial-time parameterized computations for graphs and matrices of low treewidth. *ACM Transactions on Algorithms*, 14(3):34.

[Foster et al., 1953] Foster, F. G. et al. (1953). On the stochastic matrices associated with certain queuing processes. *The Annals of Mathematical Statistics*, 24(3):355–360.

[Gagniuc, 2017] Gagniuc, P. A. (2017). *Markov chains: from theory to implementation and experimentation*. Wiley.

[Garg et al., 2016] Garg, P., Neider, D., Madhusudan, P., and Roth, D. (2016). Learning invariants using decision trees and implication counterexamples. In *Symposium on Principles of Programming Languages (POPL)*, pages 499–512.

[Giacobazzi and Ranzato, 1997] Giacobazzi, R. and Ranzato, F. (1997). Completeness in abstract interpretation: A domain perspective. In *Algebraic Methodology and Software Technology (AMAST)*, pages 231–245.

[Giacobazzi et al., 2000] Giacobazzi, R., Ranzato, F., and Scozzari, F. (2000). Making abstract interpretations complete. *Journal of the ACM*, 47(2):361–416.

[Godefroid, 2007] Godefroid, P. (2007). Compositional dynamic test generation. In *Symposium on Principles of Programming Languages (POPL)*, pages 47–54.

[Goharshady et al., 2018] Goharshady, A. K., Behrouz, A., and Chatterjee, K. (2018). Secure credit reporting on the blockchain. In *IEEE International Symposium on Blockchain and its Applications*.

[Goharshady and Mohammadi, 2020] Goharshady, A. K. and Mohammadi, F. (2020). An efficient algorithm for computing network reliability in small treewidth. *Reliability Engineering and System Safety*.

[Golub and Van Loan, 1996] Golub, G. H. and Van Loan, C. F. (1996). *Matrix computations*. Johns Hopkins Universtiy Press.

[Gould et al., 2004] Gould, C., Su, Z., and Devanbu, P. (2004). JDBC checker: A static analysis tool for SQL/JDBC applications. In *International Conference on Software Engineering (ICSE)*, pages 697–698.

[Grigor'ev and Vorobjov, 1988] Grigor'ev, D. and Vorobjov, N. (1988). Solving systems of polynomial inequalities in subexponential time. *Journal of Symbolic Computation*, 5(1/2):37–64.

[Grötschel et al., 2012] Grötschel, M., Lovász, L., and Schrijver, A. (2012). *Geometric algorithms and combinatorial optimization*, volume 2.

[Grove and Torczon, 1993] Grove, D. and Torczon, L. (1993). Interprocedural constant propagation: A study of jump function implementation. In *Conference on Programming Language Design and Implementation (PLDI)*.

[Gulwani et al., 2009] Gulwani, S., Srivastava, S., and Venkatesan, R. (2009). Constraint-based invariant inference over predicate abstraction. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 120–135.

[Gurfinkel et al., 2006] Gurfinkel, A., Wei, O., and Chechik, M. (2006). YASM: A software model-checker for verification and refutation. In *Conference on Computer-Aided Verification (CAV)*, pages 170–174.

[Gustedt et al., 2002] Gustedt, J., Mæhle, O. A., and Telle, J. A. (2002). The treewidth of java programs. In *Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 86–97.

[Hahn et al., 2010] Hahn, E. M., Hermanns, H., Wachter, B., and Zhang, L. (2010). Param: A model checker for parametric Markov models. In *Conference on Computer-Aided Verification (CAV)*, pages 660–664.

[Hahn et al., 2009] Hahn, E. M., Hermanns, H., and Zhang, L. (2009). Probabilistic reachability for parametric Markov models. In *Model Checking Software*, pages 88–106.

[Halbwachs et al., 1997] Halbwachs, N., Proy, Y.-E., and Roumanoff, P. (1997). Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185.

[Han and Tseng, 2006] Han, H. and Tseng, C.-W. (2006). Exploiting locality for irregular scientific codes. *IEEE Transactions on Parallel and Distributed Systems*, 17(7):606–618.

[Handelman, 1988] Handelman, D. (1988). Representing polynomials by positive linear functions on compact convex polyhedra. *Pacific Journal of Mathematics*, 132(1):35–62.

[Harel and Tarjan, 1984] Harel, D. and Tarjan, R. E. (1984). Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355.

[Henzinger and Ho, 1994] Henzinger, T. and Ho, P.-H. (1994). Model checking strategies for linear hybrid systems.

[Henzinger et al., 2002] Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G. (2002). Lazy abstraction. In *Symposium on Principles of Programming Languages (POPL)*, pages 58–70.

[Higham, 2009] Higham, N. J. (2009). Cholesky factorization. *Wiley Interdisciplinary Reviews: Computational Statistics.*

[Hoare, 1969] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580.

[Hoeffding, 1994] Hoeffding, W. (1994). Probability inequalities for sums of bounded random variables. In *The Collected Works of Wassily Hoeffding*, pages 409–426.

[Holzmann, 1997] Holzmann, G. J. (1997). The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295.

[Hong, 1991] Hong, H. (1991). Comparison of several decision algorithms for the existential theory of the reals.

[Horn and Johnson, 1990] Horn, R. A. and Johnson, C. R. (1990). *Matrix analysis.* Cambridge university press.

[Horwitz et al., 1995] Horwitz, S., Reps, T., and Sagiv, M. (1995). Demand interprocedural dataflow analysis. *ACM SIGSOFT Software Engineering Notes.*

[Howard, 1960] Howard, R. A. (1960). Dynamic programming and Markov processes.

[Hrushovski et al., 2018] Hrushovski, E., Ouaknine, J., Pouly, A., and Worrell, J. (2018). Polynomial invariants for affine programs. In *Symposium on Logic in Computer Science (LICS)*, pages 530–539.

[Huang et al., 2019] Huang, M., Fu, H., Chatterjee, K., and Goharshady, A. K. (2019). Modular verification for almost-sure termination of probabilistic programs. In *ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[Humenberger et al., 2017] Humenberger, A., Jaroschek, M., and Kovács, L. (2017). Automated generation of non-linear loop invariants utilizing hypergeometric sequences. In *International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 221–228.

[IBM, 2019] IBM (2019). CPLEX optimizer: High-performance mathematical programming solver for linear programming, mixed-integer programming and quadratic programming.

[Jonsson and Larsen, 1991] Jonsson, B. and Larsen, K. G. (1991). Specification and refinement of probabilistic processes. In *Symposium on Logic in Computer Science (LICS)*, pages 266–277.

[Kaelbling et al., 1998] Kaelbling, L. P., Littman, M. L., and Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2):99–134.

[Kaelbling et al., 1996] Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285.

[Kapur, 2006] Kapur, D. (2006). Automatically generating loop invariants using quantifier elimination. In *Dagstuhl Seminar Proceedings*.

[Katoen et al., 2010] Katoen, J., McIver, A., Meinicke, L., and Morgan, C. C. (2010). Linear-invariant generation for probabilistic programs: - automated support for proof-based methods. In *Static Analysis Symposium (SAS)*, pages 390–406.

[Kemeny et al., 2012] Kemeny, J. G., Snell, J. L., and Knapp, A. W. (2012). *Denumerable Markov chains: with a chapter of Markov random fields by David Griffeath*. Springer.

[Kincaid et al., 2017] Kincaid, Z., Breck, J., Boroujeni, A. F., and Reps, T. W. (2017). Compositional recurrence analysis revisited. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 248–262.

[Kincaid et al., 2018] Kincaid, Z., Cyphert, J., Breck, J., and Reps, T. W. (2018). Non-linear reasoning for invariant synthesis. In *Symposium on Principles of Programming Languages (POPL)*, pages 54:1–54:33.

[Klaus Krause et al., 2019] Klaus Krause, P., Larisch, L., and Salfelder, F. (2019). The tree-width of C. *Discrete Applied Mathematics*.

[Kress-Gazit et al., 2009] Kress-Gazit, H., Fainekos, G. E., and Pappas, G. J. (2009). Temporal-logic-based reactive mission and motion planning. *IEEE transactions on robotics*, 25(6):1370–1381.

[Křetínský and Meggendorfer, 2017] Křetínský, J. and Meggendorfer, T. (2017). Efficient strategy iteration for mean payoff in Markov decision processes. In *Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 380–399.

[Kwiatkowska et al., 2011] Kwiatkowska, M., Norman, G., and Parker, D. (2011). PRISM 4.0: Verification of probabilistic real-time systems. In *Conference on Computer-Aided Verification (CAV)*, pages 585–591.

[Lanotte et al., 2007] Lanotte, R., Maggiolo-Schettini, A., and Troina, A. (2007). Parametric probabilistic transition systems for system design and analysis. *Formal Aspects of Computing*, 19(1):93–109.

[Lavaee, 2016] Lavaee, R. (2016). The hardness of data packing. In *Symposium on Principles of Programming Languages (POPL)*, pages 232–242.

[Le Gall, 2014] Le Gall, F. (2014). Powers of tensors and fast matrix multiplication. In *International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 296–303.

[Lee and Ryder, 1992] Lee, Y.-F. and Ryder, B. G. (1992). A comprehensive approach to parallel data flow analysis. In *International Conference on Supercomputing (ICS)*, pages 236–247.

[Lin et al., 2014] Lin, W., Wu, M., Yang, Z., and Zeng, Z. (2014). Proving total correctness and generating preconditions for loop programs via symbolic-numeric computation methods. *Frontiers of Computer Science*, 8(2):192–202.

[Linderoth, 2005] Linderoth, J. (2005). A simplicial branch-and-bound algorithm for solving quadratically constrained quadratic programs. *Mathematical Programming*, 103(2):251–282.

[Mahmoud, 2008] Mahmoud, H. (2008). *Pólya urn models.* Chapman and Hall/CRC.

[Majumdar and Sen, 2007] Majumdar, R. and Sen, K. (2007). Hybrid concolic testing. In *International Conference on Software Engineering (ICSE)*, pages 416–426.

[Manna and Pnueli, 1995] Manna, Z. and Pnueli, A. (1995). *Temporal verification of reactive systems: Safety.* Springer.

[Manna and Pnueli, 2012] Manna, Z. and Pnueli, A. (2012). *Temporal verification of reactive systems: safety.* Springer.

[Matousek and Gärtner, 2007] Matousek, J. and Gärtner, B. (2007). *Understanding and using linear programming.* Springer.

[Mayr, 1981] Mayr, E. W. (1981). An algorithm for the general Petri net reachability problem. In *Symposium on Theory of Computing (STOC)*, pages 238–246.

[McIver and Morgan, 2004] McIver, A. and Morgan, C. (2004). Developing and reasoning about probabilistic programs in pGCL. In *Pernambuco Summer School on Software Engineering*, pages 123–155.

[McIver et al., 2005] McIver, A., Morgan, C., and Morgan, C. C. (2005). *Abstraction, refinement and proof for probabilistic systems.* Springer.

[McMillan, 2008] McMillan, K. L. (2008). Quantified invariant generation using an interpolating saturation prover. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 413–427.

[Meurer et al., 2017] Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J. K., Singh, S., et al. (2017). SymPy: symbolic computing in Python. *PeerJ Computer Science*, 3:e103.

[Monniaux, 2001] Monniaux, D. (2001). An abstract analysis of the probabilistic termination of programs. In *Static Analysis Symposium (SAS)*, pages 111–126.

[Motwani and Raghavan, 1995] Motwani, R. and Raghavan, P. (1995). *Randomized algorithms*. Cambridge university press.

[Müller-Olm and Seidl, 2004] Müller-Olm, M. and Seidl, H. (2004). Computing polynomial program invariants. *Information Processing Letters*, 91(5).

[Naeem et al., 2010] Naeem, N. A., Lhoták, O., and Rodriguez, J. (2010). Practical extensions to the IFDS algorithm. Conference on Compiler Construction (CC).

[Nanda and Sinha, 2009] Nanda, M. G. and Sinha, S. (2009). Accurate interprocedural null-dereference analysis for Java. In *International Conference on Software Engineering (ICSE)*, pages 133–143.

[Ngo et al., 2018] Ngo, V. C., Carbonneaux, Q., and Hoffmann, J. (2018). Bounded expectations: resource analysis for probabilistic programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 496–512.

[Nguyen et al., 2012] Nguyen, T., Kapur, D., Weimer, W., and Forrest, S. (2012). Using dynamic analysis to discover polynomial and array invariants. In *International Conference on Software Engineering (ICSE)*, pages 683–693.

[Norris, 1998] Norris, J. R. (1998). *Markov chains*. Cambridge University Press.

[Obdržálek, 2003] Obdržálek, J. (2003). Fast mu-calculus model checking when tree-width is bounded. In *Conference on Computer-Aided Verification (CAV)*, pages 80–92.

[O'Hearn, 2020] O'Hearn, P. W. (2020). Incorrectness logic. In *Symposium on Principles of Programming Languages (POPL)*, pages 10:1–10:32.

[Oustry et al., 2019] Oustry, A., Tacchi, M., and Henrion, D. (2019). Inner approximations of the maximal positively invariant set for polynomial dynamical systems. *IEEE Control Systems Letters*, 3(3):733–738.

[Padon et al., 2016] Padon, O., McMillan, K. L., Panda, A., Sagiv, M., and Shoham, S. (2016). Ivy: safety verification by interactive generalization. *Conference on Programming Language Design and Implementation (PLDI)*, pages 614–630.

[Panagiotou and Souza, 2006] Panagiotou, K. and Souza, A. (2006). On adequate performance measures for paging. In *Symposium on Theory of Computing (STOC)*, pages 487–496.

[Papachristodoulou et al., 2013] Papachristodoulou, A., Anderson, J., Valmorbida, G., Prajna, S., Seiler, P., and Parrilo, P. A. (2013). *SOSTOOLS: Sum of squares optimization toolbox for MATLAB.*

[Petrank and Rawitz, 2002] Petrank, E. and Rawitz, D. (2002). The hardness of cache conscious data placement. In *Symposium on Principles of Programming Languages (POPL)*, pages 101–112.

[Pnueli, 1977] Pnueli, A. (1977). The temporal logic of programs. In *Symposium on Foundations of Computer Science (FOCS)*, pages 46–57.

[Podelski and Rybalchenko, 2004] Podelski, A. and Rybalchenko, A. (2004). A complete method for the synthesis of linear ranking functions. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 239–251.

[Prajna and Jadbabaie, 2004] Prajna, S. and Jadbabaie, A. (2004). Safety verification of hybrid systems using barrier certificates. In *Conference on Hybrid systems: Computation and Control (HSCC)*, pages 477–492.

[Puterman, 2014] Puterman, M. L. (2014). *Markov Decision Processes.* Wiley.

[Putinar, 1993] Putinar, M. (1993). Positive polynomials on compact semi-algebraic sets. *Indiana University Mathematics Journal*, 42(3):969–984.

[Quatmann and Katoen, 2018] Quatmann, T. and Katoen, J.-P. (2018). Sound value iteration. In *Conference on Computer-Aided Verification (CAV)*, pages 643–661.

[Ranzato, 2013] Ranzato, F. (2013). Complete abstractions everywhere. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 15–26.

[Rapoport et al., 2015] Rapoport, M., Lhoták, O., and Tip, F. (2015). Precise data flow analysis in the presence of correlated method calls. In *Static Analysis Symposium (SAS)*, pages 54–71.

[Reps, 1995] Reps, T. (1995). Demand interprocedural program analysis using logic databases. In *Applications of Logic Databases*, volume 296.

[Reps, 1998] Reps, T. (1998). Program analysis via graph reachability. *Information and software technology*, 40(11-12):701–726.

[Reps, 2000] Reps, T. (2000). Undecidability of context-sensitive data-dependence analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):162–186.

[Reps et al., 1995] Reps, T., Horwitz, S., and Sagiv, M. (1995). Precise interprocedural dataflow analysis via graph reachability. In *Symposium on Principles of Programming Languages (POPL)*, pages 49–61.

[Rival, 2005] Rival, X. (2005). Understanding the origin of alarms in Astrée. In *Static Analysis Symposium (SAS)*, pages 303–319.

[Robertson and Seymour, 1984] Robertson, N. and Seymour, P. D. (1984). Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36:49–64.

[Rodriguez and Lhoták, 2011] Rodriguez, J. and Lhoták, O. (2011). Actor-based parallel dataflow analysis. In *Conference on Compiler Construction (CC)*, pages 179–197.

[Rodríguez-Carbonell, 2018] Rodríguez-Carbonell, E. (2018). Some programs that need polynomial invariants in order to be verified.

[Rodríguez-Carbonell and Kapur, 2004] Rodríguez-Carbonell, E. and Kapur, D. (2004). Automatic generation of polynomial loop invariants: Algebraic foundations. In *International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 266–273.

[Rodríguez-Carbonell and Kapur, 2007] Rodríguez-Carbonell, E. and Kapur, D. (2007). Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Science of Computer Programming*, 64(1):54–75.

[Rothberg et al., 2018] Rothberg, E. et al. (2018). Gurobi optimizer reference manual. Technical report, Gurobi Optimization, LLC.

[Rountev et al., 2006] Rountev, A., Kagan, S., and Marlowe, T. (2006). Interprocedural dataflow analysis in the presence of large libraries. In *Conference on Compiler Construction (CC)*, pages 2–16.

[Sagiv et al., 1996] Sagiv, M., Reps, T., and Horwitz, S. (1996). Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*.

[Sankaranarayanan, 2011] Sankaranarayanan, S. (2011). Automatic abstraction of non-linear systems using change of bases transformations. In *Conference on Hybrid systems: Computation and Control (HSCC)*, pages 143–152.

[Sankaranarayanan et al., 2013] Sankaranarayanan, S., Chakarov, A., and Gulwani, S. (2013). Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 447–458.

[Sankaranarayanan et al., 2004] Sankaranarayanan, S., Sipma, H., and Manna, Z. (2004). Non-linear loop invariant generation using Gröbner bases. In *Symposium on Principles of Programming Languages (POPL)*, pages 318–329.

[Schmüdgen, 1991] Schmüdgen, K. (1991). The k-moment problem for compact semi-algebraic sets. *Mathematische Annalen*, 289(1):203–206.

[Schubert et al., 2019] Schubert, P. D., Hermann, B., and Bodden, E. (2019). PhASAR: An inter-procedural static analysis framework for C/C++. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 393–410.

[Shang et al., 2012] Shang, L., Xie, X., and Xue, J. (2012). On-demand dynamic summary-based points-to analysis. In *Symposium on Code Generation and Optimization (CGO)*, pages 264–274.

[Sharir and Pnueli, 1981] Sharir, M. and Pnueli, A. (1981). Two approaches to interprocedural data flow analysis. In *Program flow analysis: Theory and applications*. Prentice-Hall.

[Sharma and Aiken, 2016] Sharma, R. and Aiken, A. (2016). From invariant checking to invariant inference using randomized search. *Formal Methods in System Design*, 48(3):235–256.

[Shen et al., 2013] Shen, L., Wu, M., Yang, Z., and Zeng, Z. (2013). Generating exact non-linear ranking functions by symbolic-numeric hybrid method. *Journal of Systems Science and Complexity*, 26(2):291–301.

[Singh et al., 2015] Singh, G., Püschel, M., and Vechev, M. (2015). Making numerical program analysis fast. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 303–313.

[Singh et al., 2017] Singh, G., Püschel, M., and Vechev, M. (2017). Fast polyhedra abstract domain. In *Symposium on Principles of Programming Languages (POPL)*, pages 46–59.

[Sleator and Tarjan, 1985] Sleator, D. D. and Tarjan, R. E. (1985). Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208.

[Smith, 1998] Smith, J. E. (1998). A study of branch prediction strategies. In *International Symposium on Computer Architecture (ISCA)*, pages 202–215.

[Späth et al., 2019] Späth, J., Ali, K., and Bodden, E. (2019). Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. In *Symposium on Principles of Programming Languages (POPL)*, pages 48:1–48:29.

[Sturm, 1999] Sturm, J. F. (1999). Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. *Optimization methods and software*, 11(1-4):625–653.

[Sturmfels, 2002] Sturmfels, B. (2002). *Solving systems of polynomial equations.* American Mathematical Society.

[Tappler et al., 2019] Tappler, M., Aichernig, B. K., Bacci, G., Eichlseder, M., and Larsen, K. G. (2019). L^*-based learning of Markov decision processes. In *Symposium on Formal Methods (FM)*, pages 651–669.

[Thabit, 1982] Thabit, K. O. (1982). *Cache management by the compiler.* PhD thesis, Rice University.

[Thorup, 1998] Thorup, M. (1998). All structured programs have small tree width and good register allocation. *Information and Computation*, 142:159–181.

[Turing, 1936] Turing, A. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society.*

[Vallée-Rai et al., 2010] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. (2010). Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224.

[van Dijk et al., 2006] van Dijk, T., van den Heuvel, J.-P., and Slob, W. (2006). Computing treewidth with LibTW. Technical report, University of Utrecht.

[Vanderbei, 2006] Vanderbei, R. J. (2006). LOQO user's manual - version 4.05. Technical report, Princeton University.

[Walukiewicz, 2001] Walukiewicz, I. (2001). Pushdown processes: Games and model-checking. *Information and Computation*, 164(2):234–263.

[Wang et al., 2019] Wang, P., Fu, H., Goharshady, A. K., Chatterjee, K., Qin, X., and Shi, W. (2019). Cost analysis of nondeterministic probabilistic programs. In *ACM Conference on Programming Language Design and Implementation (PLDI).*

[Williams, 1991] Williams, D. (1991). *Probability with martingales.* Cambridge university press.

[Wolfram Research, 2020] Wolfram Research (2020). Mathematica, Version 12.0.

[Wulf and McKee, 1995] Wulf, W. A. and McKee, S. A. (1995). Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24.

[Yang et al., 2010] Yang, L., Zhou, C., Zhan, N., and Xia, B. (2010). Recent advances in program verification through computer algebra. *Frontiers of Computer Science in China*, 4(1):1–16.

[Zhang et al., 2006] Zhang, C., Ding, C., Ogihara, M., Zhong, Y., and Wu, Y. (2006). A hierarchical model of data locality. In *Symposium on Principles of Programming Languages (POPL)*, pages 16–29.

[Zhong et al., 2004] Zhong, Y., Orlovich, M., Shen, X., and Ding, C. (2004). Array regrouping and structure splitting using whole-program reference affinity. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 255–266.

[Zhu et al., 2019] Zhu, H., Xiong, Z., Magill, S., and Jagannathan, S. (2019). An inductive synthesis framework for verifiable reinforcement learning. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 686–701.