Faster Algorithms for Dynamic Algebraic Queries in Basic RSMs with Constant Treewidth

KRISHNENDU CHATTERJEE, IST Austria

AMIR KAFSHDAR GOHARSHADY, IST Austria

PRATEESH GOYAL, MIT

RASMUS IBSEN-JENSEN, University of Liverpool

ANDREAS PAVLOGIANNIS, Aarhus University

Interprocedural analysis is at the heart of numerous applications in programming languages, such as alias analysis, constant propagation, etc. Recursive state machines (RSMs) are standard models for interprocedural analysis. We consider a general framework with RSMs where the transitions are labeled from a semiring, and path properties are algebraic with semiring operations. RSMs with algebraic path properties can model interprocedural dataflow analysis problems, the shortest path problem, the most probable path problem, etc. The traditional algorithms for interprocedural analysis focus on path properties where the starting point is *fixed* as the entry point of a specific method. In this work, we consider possible multiple queries as required in many applications such as in alias analysis. The study of multiple queries allows us to bring in an important algorithmic distinction between the resource usage of the *one-time* preprocessing vs for *each individual* query. The second aspect that we consider is that the control flow graphs for most programs have constant treewidth.

Our main contributions are simple and implementable algorithms that support multiple queries for algebraic path properties for RSMs that have constant treewidth. Our theoretical results show that our algorithms have small additional one-time preprocessing, but can answer subsequent queries significantly faster as compared to the current algorithmic solutions for interprocedural dataflow analysis. We have also implemented our algorithms and evaluated their performance for performing on-demand interprocedural dataflow analysis on various domains, such as for live variable analysis and reaching definitions, on a standard benchmark set. Our experimental results align with our theoretical statements, and show that after a lightweight preprocessing, on-demand queries are answered much faster than the standard existing algorithmic approaches.

CCS Concepts: •**Theory of computation → Program analysis; Graph algorithms analysis;**

Additional Key Words and Phrases: Interprocedural analysis, Constant treewidth graphs, Dataflow analysis, Reachability and shortest path, Reachability and distance oracles.

ACM Reference Format:
Krishnendu Chatterjee, Amir Kafshdar Goharshady, Prateesh Goyal, Rasmus Ibsen-Jensen and Andreas Pavlogiannis, 2016. Faster Algorithms for Algebraic Path Properties in Recursive State Machines with Constant Treewidth. *ACM Trans. Program. Lang. Syst.* V, N, Article A (January YYYY), 44 pages.
DOI: 0000001.0000001

The research was partly supported by Austrian Science Fund (FWF) Grant No P23499- N23, FWF NFN Grant No S11407-N23 (RiSE/SHiNE), ERC Start grant (279307: Graph Games), the Austrian Science Fund (FWF) Schrodinger grant J-4220, the Facebook PhD Fellowship Program, and DOC Fellowship No 24956 of the Austrian Academy of Sciences (ÖAW).
Author's addresses: K. Chatterjee and A.K. Goharshady IST Austria (Institute of Science and Technology Austria), Klosterneuburg, Austria; P. Goyal, Massachusetts Institute of Technology, Cambridge, MA, USA; R. Ibsen-Jensen, University of Liverpool, Liverpool, United Kingdom; A. Pavlogiannis, Aarhus University, Denmark.

DOI: 0000001.0000001

1. INTRODUCTION

Interprocedural analysis and RSMs. Interprocedural analysis is one of the classic algorithmic problem in programming languages which is at the heart of numerous applications, ranging from alias analysis, to data dependencies (modification and reference side effect), to constant propagation, to live and use analysis [Reps et al. 1995; Sagiv et al. 1996; Callahan et al. 1986; Grove and Torczon 1993; Landi and Ryder 1991; Knoop et al. 1996; Cousot and Cousot 1977; Giegerich et al. 1981; Knoop and Steffen 1992; Naeem and Lhoták 2008; Zhang et al. 2014; Chatterjee et al. 2015b]. In seminal works [Reps et al. 1995; Sagiv et al. 1996] it was shown that a large class of interprocedural dataflow analysis problems can be solved in polynomial time. A standard model for interprocedural analysis is *recursive state machines (RSMs)* [Alur et al. 2005] (aka *supergraph* in [Reps et al. 1995]). A RSM is a formal model for control flow graphs of programs with recursion. We consider RSMs that consist of component state machines (CSMs), one for each method that has a unique entry and unique exit, and each CSM contains boxes which are labeled as CSMs that allows calls to other methods. This class of, so called, single-entry single-exit RSMs is computationally less expressive than multi-entry, multi-exit RSMs, and as expressive as the class of Basic Pushdown Systems, which are pushdown systems with a single state [Clarke et al. 2018].

Algebraic path properties. To specify properties of traces of a RSM we consider a very general framework, where edges of the RSM are labeled from a complete semiring (which subsumes bounded and finite distributive semirings), and we refer to the labels of the edges as weights. For a given path, the weight of the path is the semiring product of the weights on the edges of the path, and to choose among different paths we use the semiring plus operator. For example, (i) with Boolean semiring (with semiring product as AND, and semiring plus as OR) we can express the reachability property; (ii) with tropical semiring (with real-edge weights, semiring product as standard sum, and semiring plus as minimum) we can express the shortest path property; and (iii) with Viterbi semiring (with probability value on edges, semiring product as standard multiplication and semiring plus as maximum) we can express the most probable path property. The algebraic path properties expressed in our framework subsumes the IFDS/IDE frameworks [Reps et al. 1995; Sagiv et al. 1996] which consider finite semirings and meet over all paths as the semiring plus operator. Since IFDS/IDE are subsumed in our framework, the large and important class of dataflow analysis problems that can be expressed in IFDS/IDE frameworks can also be expressed in our framework.

On-demand analysis. Exhaustive data-flow analysis is computationally expensive and often unnecessary. A topic of great interest in the software engineering community is that of *on-demand* dataflow analysis [Babich and Jazayeri 1978; Zadeck 1984; Duesterwald et al. 1995; Horwitz et al. 1995; Reps 1995; Reps 1997; Yuan et al. 1997; Naeem et al. 2010]. On-demand analyses have several applications, such as (quoting from [Horwitz et al. 1995; Reps 1997]) (i) narrowing down the focus to specific points of interest, (ii) narrowing down the focus to specific data-flow facts of interest, (iii) reducing work in preliminary phases, (iv) side-stepping incremental updating problems, and (v) offering demand analysis as a user-level operation. On-demand analysis is also very useful for speculative optimizations in just-in-time compilers [Chen et al. 2004; Lin et al. 2004; Bebenita et al. 2010; Flückiger et al. 2017], where dynamic information can dramatically increase the precision of the analysis. In this setting, it is crucial that the the on-demand analysis runs fast, to incur as little overhead as possible.

A motivating example. As a toy motivating example, consider the partial program shown in Figure 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 0 in line 7. In turn, this depends on the value of the boolean variable b. Performing a value analysis from the entry of f reveals that x can have the values $\{0, 1\}$ in line 7. Hence, if the decision to compile h relies only on an offline static analysis, h is always compiled, even when not needed.

```
1 int x, y;
2 void f(boolean b){
3   y=0;
4   if(b)
5     y=1;
6   g();
7   if(x==0)
8     h(x);
9 }
```

```
 9 void g(){
10   x=y;
11 }

13 void h(int x){
14   //An expensive
15   //function
16 }
```

Fig. 1: A partial program. Whether the function h is called in line 8 depends on the value of the argument b of function f.

Now consider the case where the execution of the program is in line 5, and at this point the compiler decides on whether to compile h. It is clear that given this information, the set of possible values for x in line 7 is the singleton $\{1\}$ 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 will have the value 1. 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 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.

In this work, we address the above challenge algorithmically. We exploit the low-treewidth property of control-flow graphs to devise fast algorithms for on-demand analysis that have minimal preprocessing overhead.

Preprocess vs query. On-demand analyses can be naturally phrased in a preprocess vs query setting. In the preprocessing phase, the program is analyzed without knowledge of the precise analysis queries. Afterwards, in the query phase, analysis queries arrive in an online fashion (i.e., the analyzer is oblivious to future queries). In graph theoretic parlance, graph algorithms can consider two types of queries: (i) a *pair query* that given nodes u and v (called (u, v)-pair query) asks for the algebraic path property from u to v; and (ii) a *single-source* query that given a node u asks for the answer of (u, v)-pair queries for all nodes v. In this vocabulary, the traditional algorithms for offline interprocedural analysis has focused on the answer for *one* single-source query. This consideration opens up a wide spectrum with regards to the resources spent in each phase. On the one end we have no preprocessing, where each arising query is treated anew, as an offline analysis problem. On the other end we have complete preprocessing, where we precompute the answer to every possible query and store the answer in a lookup table. Hence, in the query phase, every query is answered by a simple table lookup. The key technical challenge faced by the static analyzer is to achieve the best possible tradeoff in this spectrum: spend as few resources as possible in the preprocessing phase (in terms of running time and space usage) so that, afterwards, on-demand queries are answered fast.

Treewidth property of control-flow graphs. A very well-known concept in graph theory is the notion of *treewidth* of a graph, which is a measure of how similar a graph is to a tree (a graph has treewidth 1 precisely if it is a tree) [Robertson and Seymour 1984]. The treewidth of a graph is defined based on a *tree decomposition* of the graph [Halin 1976] (see Section 2 for a formal definition). Beyond the mathematical elegance of the treewidth property for graphs, there are many classes of graphs which arise in practice and have constant treewidth. The most important example is that the flow graph for `goto`-free programs in many classic programming languages have constant treewidth [Thorup 1998]. The low treewidth of flow graphs 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. 2019].

	Preprocessing time	Space	Query		Reference
			Single-source	Pair	
Our Results	$O(n \cdot \log n + h \cdot b \cdot \log n)$	$O(n \cdot \log n)$	$O(n)$	$O(1)$	Theorem 4.2
	$O(n + h \cdot b \cdot \log n)$	$O(n)$	$O(n)$	$O(\log n)$	Theorem 4.2

Table I: Interprocedural same-context algebraic paths on RSMs with b boxes and constant treewidth, for stack height h.

	Preprocessing time	Space	Query		Reference
			Single-source	Pair	
IFDS/IDE (complete preprocessing)	$O(n^2 \cdot \lvert D\rvert^3)$	$O(n^2 \cdot \lvert D\rvert)$	$O(n \cdot \lvert D\rvert)$	$O(\lvert D\rvert)$	[Reps et al. 1995]
IFDS/IDE (no preprocessing)	-	$O(n \cdot \lvert D\rvert)$	$O(n \cdot \lvert D\rvert^3)$	$O(n \cdot \lvert D\rvert^3)$	[Reps et al. 1995]
Our Results	$O(n \cdot \log n \cdot \lvert D\rvert^3)$	$O(n \cdot \log n \cdot \lvert D\rvert^2)$	$O(n \cdot \lvert D\rvert^2)$	$O(\lvert D\rvert^2)$	Corollary 4.5
	$O((n + b \cdot \log n) \cdot \lvert D\rvert^3)$	$O(n \cdot \lvert D\rvert^2)$	$O(n \cdot \lvert D\rvert^2)$	$O(\log n \cdot \lvert D\rvert^2)$	Corollary 4.5
$\lvert D\rvert = \Omega(\log n)$	$O(n \cdot \lvert D\rvert^3)$	$O(n \cdot \lvert D\rvert^2)$	$O(n \cdot \lvert D\rvert^2 / \log n)$	$O(\lvert D\rvert^2 / \log n)$	Corollary 4.6

Table II: Interprocedural same-context algebraic paths on RSMs with b boxes and constant treewidth, where the semiring is over the subset of $|D|$ elements and the plus operator is the meet operator of the IFDS framework. Existing results are taken from [Reps et al. 1995]. Our results are obtained from Corollary 4.5 and Corollary 4.6.

	Preprocessing time	Space	Query		Reference
			Single-source	Pair	
Complete preprocessing[1]	$O(n^2 \cdot \log n)$	$O(n^2)$	$O(n)$	$O(1)$	[Schwoon 2002]
No preprocessing[2]	-	$O(n)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$	[Schwoon 2002]
Our Result	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n)$	$O(1)$	Corollary 4.7
	$O(n + b \cdot \log n)$	$O(n)$	$O(n)$	$O(\log n)$	Corollary 4.7

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

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

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

Algorithmic considerations. The current work focuses on the algorithmic problem of constructing algebraic path oracles for handling general algebraic path queries on RSMs, as well as on the special cases of IFDS/IDE and shortest-path semirings. In particular, we consider RSMs where every CSM has constant treewidth, and the algorithmic question of answering multiple single-source and multiple pair queries, where each query is a *same-context* query (a same-context query starts and ends with an empty stack, see [Chaudhuri 2008] for the significance of same-context queries). Although these problems are of great practical importance, this work makes an algorithmic treatment aimed at improved worst-case complexities. We address the practical side of our algorithms in an experimental evaluation which, although not extensive, gives a promising indication that our algorithms can be relevant in practice (e.g., wrt the hidden constant in the asymptotic complexity analysis).

Our contributions. Our main contributions are as follows:

(1) *(General result).* Since we consider arbitrary semirings (i.e., not restricted to finite semirings) we consider the stack-height-bounded problem, where the input contains a stack-height bound h, and we are interested in semiring distances as witnessed by interprocedural paths of stack height at most h. While in general for arbitrary semirings there does not exist a bound on the stack height, if the semiring does not have infinite descending chains, then the stack height bounded problem gives an exact answer to the general problem (i.e., without a bound on the stack height), for large enough h. This is especially the case for the main semirings of interest, as in the case of reachability and the case of IFDS. Our main result is an algorithm where the one-time preprocessing phase requires $O(n \cdot \log n + h \cdot b \cdot \log n)$ semiring operations, and then each subsequent bounded stack height pair query can be answered in constant number of semiring operations, where n is the number of nodes of the RSM and b the number of boxes (see Table I and Theorem 4.2).

(2) *IFDS/IDE* If we specialize our result to the IFDS/IDE setting with finite semirings from a finite universe of distributive functions $2^D \to 2^D$, and meet over all paths as the semiring plus operator, then we obtain the results shown in Table II (Corollary 4.5). For example, our approach with a factor of $O(\log n)$ overhead for one-time preprocessing, as compared to no preprocessing, can answer subsequent pair queries by a factor of $\Omega(n \cdot |D|)$ faster. Additionally, when $|D| = \Omega(\log n)$, our algorithm requires only $O(n \cdot |D|^3)$ preprocessing after which pair queries are answered in $O(|D|^2/\log n)$ time. Note that the complexity of the standard IFDS/IDE algorithm is $O(n \cdot |D|^3)$ for answering one single-source query, whereas in the same preprocessing time, our algorithm handles every pair query efficiently.

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

An important feature of our algorithms is that they are simple and implementable. Besides the theoretical improvements, we demonstrate the effectiveness of our approach for performing on-demand analysis on several standard benchmarks. We have used the tool JTDec [Chatterjee et al. 2017] for computing tree decompositions, and all benchmarks of our experimental results have small treewidth, and hence our treewidth considerations are justified. We have evaluated the performance of our algorithms for 6 different interprocedural dataflow analyses, expressed in the IFDS framework: control-flow reachability, unused variables, reaching definitions, live variables, simple uninitialized variables and possibly uninitialized variables. Our experiments show that our new treewidth-based algorithms succeed in answering both single-source and pair on-demand queries efficiently, only after a lightweight preprocessing.

Intuition and main technical contribution. Conceptually, the process of solving the algebraic-path problem on an RSM can be viewed as an iterative process: first, every control-flow graph of the RSM (corresponding to every method of the program) is analyzed independently, to solve the intraprocedural algebraic-path problem, from the entry to the exit of the module. This information is then summarized to the calling site of every invocation to the respective RSM, and the process repeats. The intuition behind our improvements is that, assuming the constant-treewidth property of the control-flow graphs, the tasks of (i) *updating* each control-flow graph with summary information and (ii *querying* each control-flow graph for the semiring distance between the entry and the exit nodes can be done efficiently. Our main technical contribution is a dynamic algorithm (also referred to as incremental algorithm in the literature on graph algorithms) that given a graph with constant treewidth, after a preprocessing phase of $O(n \cdot \log n)$ semiring operations supports (1) changing the label of an edge with $O(\log n)$ semiring operations; and (2) answering pair queries with $O(\log n)$ semiring operations; and (3) answering single-source queries with $O(n)$ semiring operations. These results are presented in Theorem 3.8.

Preliminary versions of this work have appeared in [Chatterjee et al. 2015a; Chatterjee et al. 2016].

1.1. Related Work

In this section we compare our work with several related work from interprocedural analysis as well as for constant treewidth property.

Interprocedural analysis. Interprocedural analysis is a classic algorithmic problem in static analysis and several diverse applications have been studied in the literature [Reps et al. 1995; Sagiv et al. 1996; Callahan et al. 1986; Grove and Torczon 1993; Landi and Ryder 1991; Knoop et al. 1996; Cousot and Cousot 1977; Giegerich et al. 1981; Knoop and Steffen 1992; Chatterjee et al. 2015a]. Our work is most closely related to the IFDS/IDE frameworks introduced in seminal works [Reps et al. 1995; Sagiv et al. 1996]. In both IFDS/IDE framework the semiring is finite, and they study the algorithmic question of solving one single-source query. While in our framework the semiring is not necessarily finite, we consider the stack height bounded problem. We also consider the multiple pair and single-source, same-context queries, and the additional restriction that RSMs have constant treewidth. Our general result specialized to finite semirings (where the stack height bounded problem coincides with the general problem) improves the existing best known algorithms for the IFDS/IDE framework where the RSMs have constant treewidth. Additionally, the shortest path problem cannot be expressed in the IFDS/IDE framework [Reps et al. 2005], but can be expressed in the GPR framework [Reps et al. 2005; Reps et al. 2007]. The GPR framework considers the more general problem on weighted pushdown systems. Although the RSMs that we consider in this work are a special case of pushdown systems, the GPR framework is efficient only for semirings of small height. For example, although the shortest path problem can be phrased in the GPR framework, the solution might take exponential time in the worst case. A solution to the shortest path problem for RSMs is presented in [Schwoon 2002], by replacing the work queue data structure of the GPR framework with a priority queue. The related problem of determining the minimum-mean cycle in RSMs was addressed in [Chatterjee et al. 2015b], without the constant-treewidth consideration. Finally, several works such as [Horwitz et al. 1995] ask for on-demand interprocedural analysis and algorithms to support dynamic updates, and our main technical contributions are algorithms to support dynamic updates in interprocedural analysis.

Recursive state machines (RSMs). Recursive state machines, which in general are equivalent to pushdown graphs, have been studied as a formal model for interprocedural analysis [Alur et al. 2005]. However, in comparison to pushdown graphs, RSMs are a more convenient formalism for interprocedural analysis. Games on recursive state machines with modular strategies have been considered in [Alur et al. 2006; Chatterjee and Velner 2012], and subcubic algorithm for general RSMs with reachability has been shown in [Chaudhuri 2008]. We focus on RSMs with unique entries and exits and with the restriction that the components have constant tree width. RSMs with unique entries and exits are less expressive than pushdown graphs, but remain a very natural model for efficient interprocedural analysis [Reps et al. 1995; Sagiv et al. 1996].

Treewidth of graphs. The notion of treewidth for graphs as an elegant mathematical tool to analyze graphs was introduced in [Robertson and Seymour 1984]. The significance of constant treewidth in graph theory is huge mainly because several problems on graphs become complexity-wise easier. Given a tree decomposition of a graph with low treewidth t, many NP-complete problems for arbitrary graphs can be solved in time polynomial in the size of the graph, but exponential in t [Arnborg and Proskurowski 1989; Bern et al. 1987; Bodlaender 1988; Bodlaender 1993; Bodlaender 2005].

Even for problems that can be solved in polynomial time, faster algorithms can be obtained for low treewidth graphs, for example, for the distance problem [Chaudhuri and Zaroliagis 1995]. The constant-treewidth property of graphs has also been used in the context of logic: Monadic Second Order (MSO) logic is a very expressive logic, and a celebrated result of [Courcelle 1990] showed that for constant-treewidth graphs the decision questions for MSO can be solved in polynomial time; and the result of [Elberfeld et al. 2010] shows that this can even be achieved in deterministic log-space. Dynamic algorithms for the special case of 2-treewidth graphs has been considered in [Bodlaender 1994] and extended to various tradeoffs by [Hagerup 2000]; and [Lacki 2013] shows how to maintain the strongly connected component decomposition under edge deletions for constant treewidth graphs. However, none of these works consider RSMs or interprocedural analysis. Various other models (such as probabilistic models of Markov decision processes and games played on graphs for synthesis) with the constant-treewidth restriction have also been considered [Chatterjee and Lacki 2013; Obdrzálek 2003]. The problem of computing a balanced tree decomposition for a constant treewidth graph was considered in [Reed 1992], and we use this algorithm in our preprocessing phase.

2. PRELIMINARIES

We will in this section give definitions related to semirings, graphs, and recursive state machines.

2.1. Semirings

Definition 2.1 (*Semirings*). We consider complete *semirings* $S = (\Sigma, \oplus, \otimes, \overline{\mathbf{0}}, \overline{\mathbf{1}})$ where Σ is a countable set, $\oplus$ and $\otimes$ are binary operators on Σ, and $\overline{\mathbf{0}}, \overline{\mathbf{1}} \in \Sigma$, and the following properties hold:

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

Additionally, we consider that

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

Conventionally, we let $\oplus(\emptyset) = \overline{\mathbf{0}}$ and $\otimes(\emptyset) = \overline{\mathbf{1}}$.

2.2. Graphs and tree decompositions

Definition 2.2 (*Graphs and weighted paths*). Let $G = (V, E)$ be a finite directed graph where V is a set of n nodes and $E \subseteq V \times V$ is an edge relation of m edges, along with a weight function $\mathsf{wt} : E \to \Sigma$ that assigns to each edge of G an element from Σ. A path $P : u \rightsquigarrow v$ is a sequence of nodes $(u_1, \dots, u_k)$ such that for each $1 \leq i < k$ we have $(u_i, u_{i+1}) \in E$. The length of P is $k - 1$. A path P is *simple* if no node repeats in the path (i.e., it does not contain a cycle). A single node is by itself a 0-length path. Given a path $P = (u_1, \dots, u_k)$, we use the set notation $u \in P$ to denote that u appears in P, and $A \cap P$ to refer to the set of nodes that appear in both P and A. The weight of P is $\otimes(P) = \bigotimes(\mathsf{wt}(u_1, u_2), \dots, \mathsf{wt}(u_{k-1}, u_k))$ if $|P| \geq 1$ else $\otimes(P) = \overline{\mathbf{1}}$. Given nodes $u, v \in V$, the distance $d(u, v)$ is defined as $d(u, v) = \bigoplus_{P : u \rightsquigarrow v} \otimes(P)$, and $d(u, v) = \overline{\mathbf{0}}$ if no such P exists.

Definition 2.3. A (rooted) tree $T = (V_T, E_T)$ is an undirected graph with a distinguished node r which is the root such that there is a unique simple path $P_u^v : u \rightsquigarrow v$ for each pair of nodes u, v. The *size* of T is $|V_T|$. Given a tree T with root r, the *level* $\mathsf{Lv}(u)$ of a node u is the length of the simple path P_u^r from u to the root r, and every node in P_u^r is an *ancestor* of u. If v is an ancestor of u, then u is a *descendant* of v. Note that a node u is both an ancestor and descendant of itself. For a pair of nodes $u, v \in V_T$, the *lowest common ancestor (LCA)* of u and v is the common ancestor of u

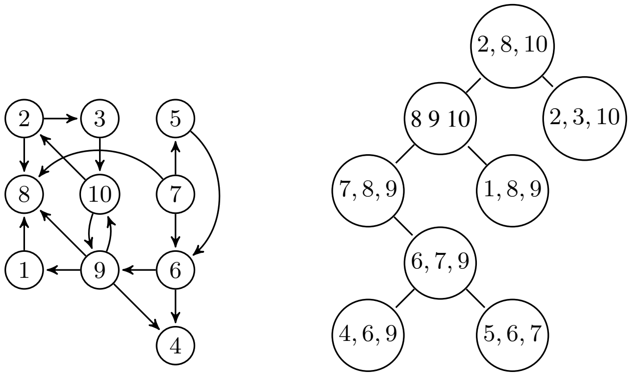


Fig. 2: A graph G of treewidth 2 (left) and a corresponding tree-decomposition T (right).

and v with the largest level. Given a node u with $\mathsf{Lv}(i) > 0$, the *parent* u of v is the unique ancestor of v in level $\mathsf{Lv}(v) - 1$, and v is a *child* of u. A *leaf* of T is a node with no children. For a node $u \in V_T$, we denote by $T(u)$ the subtree of T rooted in u (i.e., the tree consisting of all descendants of u). The tree T is *binary* if every node has at most two children. The *height* of T is $\max_u \mathsf{Lv}(u)$ (i.e., it is the length of the longest path P_u^r), and T is *balanced* if its height is bounded by $c \cdot \log |V_T|$, where c is a constant that does not depend on T. Given a tree T, a *connected component* $C \subseteq V_T$ of T is a set of nodes of T such that for every pair of nodes $u, v \in C$, the unique simple path P_u^v in T visits only nodes in C.

Definition 2.4 (*Tree decomposition and treewidth [Robertson and Seymour 1984]*). Given a graph G, a tree-decomposition $T = (V_T, E_T)$ is a tree with the following properties.

T1: $V_T = \{\mathsf{B}_1, \dots, \mathsf{B}_b :$ for all $1 \le i \le b.\ \mathsf{B}_i \subseteq V\}$ and $\bigcup_{\mathsf{B}_i \in V_T} \mathsf{B}_i = V$. That is, each node of T is a subset of nodes of G, and each node of G appears in some node of T.

T2: For all $(u, v) \in E$ there exists $\mathsf{B}_i \in V_T$ such that $u, v \in \mathsf{B}_i$. That is, the endpoints of each edge of G appear together in some node of T.

T3: For all B_i, B_j and any bag B_k that appears in the simple path $\mathsf{B}_i \rightsquigarrow \mathsf{B}_j$ in T, we have $\mathsf{B}_i \cap \mathsf{B}_j \subseteq \mathsf{B}_k$. That is, every node of G is contained in a contiguous subtree of T.

To distinguish between the nodes of G and the nodes of T, the sets B_i are called *bags*. The *width* of a tree-decomposition T is the size of the largest bag minus 1 and the *treewidth* of G is the width of a minimum-width tree decomposition of G. We refer to the treewidth t of G as *constant* if $t = O(1)$, i.e., the treewidth does not depend on G. It follows from the definition that if G has constant treewidth, then $m = O(n)$.

Example 2.5 (*Graph and tree decomposition*). The treewidth of a graph G is an intuitive measure which represents the proximity of G to a tree, though G itself is not a tree. The treewidth of G is 1 precisely if G is itself a tree [Robertson and Seymour 1984]. Consider an example graph and its tree decomposition shown in Figure 2. It is straightforward to verify that all the three conditions of tree decomposition are met. Each node in the tree is a bag, and labeled by the set of nodes it contains. Since each bag contains at most three nodes, the tree decomposition has width 2.

Notation on tree decompositions. Let G be a graph, $T = (V_T, E_T)$ a tree decomposition of G, and B_0 be the root of T. Denote with $\mathsf{Lv}(\mathsf{B}_i)$ the depth of B_i in T, with $\mathsf{Lv}(\mathsf{B}_0) = 0$. For $u \in V$, we say that a bag B is the *root bag* of u if B is the bag with the smallest level among all bags that contain u, i.e., $\mathsf{B}_u = \arg\min_{\mathsf{B} \in V_T :\, u \in \mathsf{B}} \mathsf{Lv}(\mathsf{B})$. By definition, there is exactly one root bag for each node u. We often write B_u for the root bag of node u, and denote with $\mathsf{Lv}(u) = \mathsf{Lv}(\mathsf{B}_u)$. We assume wlog that all tree decompositions mentioned in this work have the property that every leaf bag is the root bag of some node, as otherwise, we can remove that bag and obtain a valid tree decomposition. Finally, we denote with $\mathsf{B}_{(u,v)}$ the bag of the largest level that is the root bag of one of u, v.

Example 2.6 (*Root bags*). In the example of Figure 2, the bag $\{2, 8, 10\}$ is the root of T, the level of node 9 is $\mathsf{Lv}(9) = \mathsf{Lv}(\{8, 9, 10\}) = 1$, and the bag of the edge $(9, 1)$ is $\mathsf{B}_{(9,1)} = \{1, 8, 9\}$.

Separator property. A key property of a tree-decomposition $T = (V_T, E_T)$ of a graph G is that the nodes of each bag B form a **separator** of G. Removing B splits T into a number of connected components. The separator property states that every path between nodes that appear in bags of different components has to go through some node in B. This is formally stated in the following lemma.

LEMMA 2.7 ([BODLAENDER 1998, LEMMA 3]). *Consider a graph $G = (V, E)$, a tree-decomposition $T = (V_T, E_T)$ of G, and a bag B of T. Let $(C_i)_i$ be the components of T created by removing B from T, and let V_i be the set of nodes that appear in bags of component C_i. For every $i \neq j$, nodes $u \in V_i$, $v \in V_j$ and path $P : u \rightsquigarrow v$, we have that $P \cap \mathsf{B} \neq \emptyset$ (i.e., all paths between u and v go through some node in B).*

Using Lemma 2.7, we prove the following stronger version of the separator property, which will be useful throughout the paper.

LEMMA 2.8. *Consider a graph $G = (V, E)$ and a tree-decomposition $T = (V_T, E_T)$ of G. Let $u, v \in V$, and consider two distinct bags B_1 and B_j such that $u \in \mathsf{B}_1$ and $v \in \mathsf{B}_j$. Let $P' : \mathsf{B}_1, \mathsf{B}_2, \dots, \mathsf{B}_j$ be the unique simple path in T from B_1 to B_j. For each $i \in \{2, \dots, j\}$ and for each path $P : u \rightsquigarrow v$, there exists a node $x_i \in (\mathsf{B}_{i-1} \cap \mathsf{B}_i \cap P)$.*

PROOF. Fix a number $i \in \{2, \dots, j\}$. We argue that for each path $P : u \rightsquigarrow v$, there exists a node $x_i \in (\mathsf{B}_{i-1} \cap \mathsf{B}_i \cap P)$. We construct a tree T', which is similar to T except that instead of having an edge between bag B_{i-1} and bag B_i, there is a new bag B, that contains the nodes in $\mathsf{B}_{i-1} \cap \mathsf{B}_i$, and there is an edge between B_{i-1} and B and one between B and B_i. It is easy to see that T' satisfies the properties T1-T3 of a tree-decomposition of G. By Lemma 2.7, each bag B' in the unique path $P'' : \mathsf{B}_1, \dots, \mathsf{B}_{i-1}, \mathsf{B}, \mathsf{B}_i, \dots, \mathsf{B}_j$ in T' separates u from v in G. Hence, each path $u \rightsquigarrow v$ must go through some node in B, and the result follows. □

The following lemma states that for nodes u and v that appear in bags B, B', respectively, of the tree-decomposition $T = (V_T, E_T)$ of G, their distance can be written as a sum of distances $d(x_i, x_{i+1})$ between pairs of nodes (x_i, x_{i+1}) that appear in bags B_i that constitute the unique $\mathsf{B} \rightsquigarrow \mathsf{B}'$ path in T. This holds because every path $P : u \rightsquigarrow v$ can be decomposed to j subpaths of the form $x_i \rightsquigarrow x_{i+1}$. This yields a constructive way to compute $d(u, v)$ assuming that all distances $d(x_i, x_{i+1})$ have been already been computed.

LEMMA 2.9. *Consider a weighted graph $G = (V, E, \mathsf{wt})$ and a tree-decomposition $T = (V_T, E_T)$ of G. Let $u, v \in V$, and $P' : \mathsf{B}_1, \mathsf{B}_2, \dots, \mathsf{B}_j$ be a simple path in T such that $u \in \mathsf{B}_1$ and $v \in \mathsf{B}_j$. Let $A = \{u\} \times \left(\prod_{1<i\leq j} (\mathsf{B}_{i-1} \cap \mathsf{B}_i)\right) \times \{v\}$. Then $d(u, v) = \bigoplus_{(x_1,\dots,x_{j+1})\in A} \bigotimes_{i=1}^{j} d(x_i, x_{i+1})$.*

PROOF. Consider the set of all paths $\{P_\ell : u \rightsquigarrow v\}_\ell$, and we have $d(u, v) = \bigoplus_\ell \mathsf{wt}(P_\ell)$. By Lemma 2.8, for each path P_ℓ and $i \in \{1, \dots, j\}$, there exists some node $x_i^\ell \in (\mathsf{B}_{i-1} \cap \mathsf{B}_i \cap P_\ell)$. We let x_i range over the i-th node of each path of A, and we have

$$\begin{aligned} d(u,v) &= \bigotimes_{i=1}^{j} \bigoplus_{x_i, x_{i+1}} \bigoplus_{P: x_i \rightsquigarrow x_{i+1}} \mathsf{wt}(P) \\ &= \bigotimes_{i=1}^{j} \bigoplus_{x_i, x_{i+1}} d(x_i, x_{i+1}) \\ &= \bigoplus_{(x_1,\dots,x_{j+1})\in A} \bigotimes_{i=1}^{j} d(x_i, x_{i+1}) \end{aligned}$$

The desired result follows. □

The following lemma states that balanced tree decompositions of constant-treewidth graphs can be constructed efficiently.

LEMMA 2.10 ([BODLAENDER AND HAGERUP 1995, LEMMA 2]). *Given a graph $G = (V, E)$ of treewidth $t = O(1)$, a balanced tree decomposition of G with width $O(t)$ can be constructed in $O(n)$ time.*

The following crucial lemma states that given a tree decomposition of constant width, the *local distance* between every pair nodes that appear together in some bag can be computed in time linear in the size of the tree-decomposition.

LEMMA 2.11 ([CHAUDHURI AND ZAROLIAGIS 1995]). *Given a weighted graph $G = (V, E, \mathsf{wt})$ of treewidth t and a tree-decomposition $T = (V_T, E_T)$ of G of width $O(t)$, we can compute for all bags $\mathsf{B} \in V_T$ a* local distance map $\mathsf{LD}_\mathsf{B} : \mathsf{B} \times \mathsf{B} \to \Sigma$ *with* $\mathsf{LD}_\mathsf{B}(u, v) = d(u, v)$ *in total time $O(|V_T| \cdot t^3)$ and space $O(|V_T| \cdot t^2)$.*

Nicely rooted tree decompositions. A tree-decomposition $T = (V_T, E_T)$ of a graph G is *nicely rooted* if every bag is the root bag of at most one node of G.

LEMMA 2.12. *Given a tree decomposition $T = (V_T, E_T)$ of G of width $O(t)$ and $O(n)$ bags, a nicely rooted, binary tree decomposition $T' = (V_{T'}, E_{T'})$ of G of width $O(t)$ can be constructed in $O(n \cdot t)$ time. If $t = O(1)$ and T is balanced, then so is T'.*

PROOF. First, T can be turned into a binary tree decomposition T_1 by a standard tree-binarization process [Chaudhuri and Zaroliagis 1995, Fact 3], which increases the size by at most a factor 2. Hence T_1 remains balanced. Then, we can make T_1 nicely rooted simply by replacing each bag B which is the root of $k > 1$ nodes $x_1, \dots x_k$ with a chain of bags $\mathsf{B}_1, \dots, \mathsf{B}_k = \mathsf{B}$, where each B_i is the parent of B_{i+1}, and $\mathsf{B}_{i+1} = \mathsf{B}_i \cup \{x_{i+1}\}$. Note that this keeps the tree binary and increases its height by at most a factor t, hence if $t = O(1)$ and T is balanced, then the resulting tree is also balanced. □

Hence, combined with Lemma 2.10, a nicely rooted, balanced tree decomposition of a constant-treewidth graph can be constructed in $O(n)$ time.

Small tree decompositions. A tree-decomposition $T = (V_T, E_T)$ of a graph G of n nodes and treewidth t is called *small* if $|V_T| = O(\frac{n}{t})$.

LEMMA 2.13. *Given a tree decomposition $T = (V_T, E_T)$ of G of width $O(t)$ and $O(n)$ bags, a small, binary tree decomposition $T = (V_{T'}, E_{T'})$ of width $O(t)$ can be constructed in $O(n \cdot t)$ time. Moreover, if T is balanced, then so is T'.*

PROOF. Let $k = O(t)$ be the width of T. The construction is achieved using the following steps.

(1) Following the steps of [Bodlaender 1996, Lemma 2.4], we turn T to a *smooth* tree-decomposition $T_1 = (V_1, E_1)$, which has the properties that (i) for every bag $\mathsf{B} \in V_1$ we have $|\mathsf{B}| = k + 1$, and (ii) for every pair of bags $(\mathsf{B}_1, \mathsf{B}_2) \in E_1$ we have $|\mathsf{B}_1 \cap \mathsf{B}_2| = k$. The process of [Bodlaender 1996, Lemma 2.4] can be performed $O(n \cdot t)$ time and increases the height by at most a factor 2, hence if T is balanced, T_1 is also balanced, and by [Bodlaender 1996, Lemma 2.5], we have $|V_1| = O(n)$.

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

(3) We construct a tree-decomposition $T_3 = (V_3, E_3)$ by partitioning T_2 to disjoint connected components of size between $\frac{k}{2}$ and k each (the last component might have size less than $\frac{k}{2}$) and contracting each such component to a single bag in T_3. Since T_2 is smooth, the number of nodes in the union of the bags of each component is at most $2 \cdot k$. Hence the width of T_3 is $O(k)$. The partitioning is done as follows. We traverse T_2 bottom-up and group bags into components in a greedy way. In particular, given that the traversal is on a current bag B, we keep track of the number of bags i_{B} below B (not including B) that have not been grouped to a component yet. The first time we find $i_{\mathsf{B}} \geq t$, let B' be the child of B with the largest number $i_{\mathsf{B}'}$ among the children of B. We group B' and its ungrouped descendants into a new component C, and continue with the traversal. Observe that the size of C is $\frac{k}{2} \leq |C| < k$.

(4) Finally, we construct T' by turning T_3 to a binary tree-decomposition as in Step 2.

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

Hence, combined with Lemma 2.10, a small, balanced tree decomposition of a constant-treewidth graph can be constructed in $O(n)$ time.

The algebraic path problem on graphs of constant treewidth. Given a graph $G = (V, E)$, a balanced tree-decomposition $T = (V_T, E_T)$ of G with constant width $t = O(1)$, a complete semiring $(\Sigma, \oplus, \otimes, \overline{\mathbf{0}}, \overline{\mathbf{1}})$, and a weight function $\mathsf{wt} : E \to \Sigma$, the algebraic paths problem on input $u, v \in V$, asks for the distance $d(u, v)$ from node u to node v. In addition, we allow the weight function to change between successive queries. We measure the time complexity of our algorithms in number of operations, with each operation being either a basic machine operation, or an application of one of the operators of the semiring.

2.3. Recursive state machines

In this section we define recursive state machines (RSMs), which are a standard model for interprocedural analysis. Intuitively, an RSM represents a program as a collection of component state machines (CSMs), where each CSM represents a method of the program as a graph. Additionally, each CSM has distinguished nodes that represent method calls and returns to other CSMs. These notions are made clear in the following definitions.

Definition 2.14 (*RSMs and CSMs*). A *single-entry single-exit recursive state machine* (RSM from now on) over an alphabet Σ, as defined in [Alur et al. 2005], consists of a set $\{A_1, A_2, \dots, A_k\}$, such that for each $1 \leq i \leq k$, the *component state machine* (CSM) $A_i = (B_i, Y_i, V_i, E_i, \mathsf{wt}_i)$, where $V_i = N_i \cup \{en_i\} \cup \{ex_i\} \cup c_i \cup r_i$, consists of:

— A set B_i of *boxes*.

- A map Y_i, mapping each box in B_i to an index in $\{1, 2, \ldots, k\}$. We say that a box $b \in B_i$ *corresponds* to the CSM with index $Y_i(b)$.
- A set V_i of *nodes*, consisting of the union of the sets N_i, $\{en_i\}$, $\{ex_i\}$, c_i and r_i. The number n_i is the size of V_i. Each of these sets, besides V_i, are w.l.o.g. assumed to be pairwise disjoint.
  - The set N_i is the set of *internal nodes*.
  - The node en_i is the *entry node*.
  - The node ex_i is the *exit node*.
  - The set c_i is the set of *call nodes*. Each call node is a pair (x, b), where b is a box in B_i and x is the entry node $en_{Y_i(b)}$ of the corresponding CSM with index $Y_i(b)$.
  - The set r_i is the set of *return nodes*. Each return node is a pair (y, b), where b is a box in B_i and y is the exit node $ex_{Y_i(b)}$ of the corresponding CSM with index $Y_i(b)$.
- A set E_i of *internal edges*. Each edge is a pair in $(N_i \cup \{en_i\} \cup r_i) \times (N_i \cup \{ex_i\} \cup c_i)$.
- A map wt_i, mapping each edge in E_i to a label in Σ.

Definition 2.15 (*Control flow graph of CSMs and treewidth of RSMs*). Given a RSM $A = \{A_1, A_2, \ldots, A_k\}$, the *control flow graph* $G_i = (V_i, E'_i)$ for CSM A_i consists of V_i as the set of vertices and E'_i as the set of edges, where E'_i consists of the edges E_i of A_i and for each box b, each call node (v, b) of that box (i.e. for $v = en_{Y_i(b)}$) has an edge to each return node (v', b) of that box (i.e. for $v' = ex_{Y_i(b)}$). We say that the RSM has *treewidth* t, if t is the smallest integer such that for each index $1 \leq i \leq k$, the graph $G_i = (V_i, E'_i)$ has treewidth at most t. Programs are naturally represented as RSMs, where the control flow graph of each method of a program is represented as a CSM.

Example 2.16 (*RSM and tree decomposition*). Figure 3 shows an example of a program for matrix multiplication consisting of two methods (one for vector multiplication invoked by the one for matrix multiplication). The corresponding control flow graphs, and their tree decompositions that achieve treewidth 2 are also shown in the figure.

Box sequences. For a sequence L of boxes and a box b, we denote with $L \circ b$ the concatenation of L and b. Also, $\emptyset$ is the empty sequence of boxes.

Configurations and global edges. A *configuration* of a RSM is a pair (v, L), where v is a node in $(N_i \cup \{en_i\} \cup r_i)$ and L is a sequence of boxes. The *stack height* of a configuration (v, L) is the number of boxes in the sequence L. The set of *global edges* E are edges between configurations. The map wt maps each edge in E to a label in Σ. We have an edge between configuration $\mathcal{C}_1 = (v_1, L_1)$, where $v_1 \in V_i$, and configuration $\mathcal{C}_2 = (v_2, L_2)$ with label $\sigma = \mathsf{wt}(\mathcal{C}_1, \mathcal{C}_2)$ if and only if one of the following holds:

- **Internal edge:** v_2 is an *internal* node in N_i and each of the following (i) $L_1 = L_2$; and (ii) $(v_1, v_2) \in E_i$; and (iii) $\sigma = \mathsf{wt}_i((v_1, v_2))$.
- **Entry edge:** v_2 is the *entry* node $en_{Y_i(b)}$, for some box b, and each of the following (i) $L_1 \circ b = L_2$; and (ii) $(v_1, (v_2, b)) \in E_i$; and (iii) $\sigma = \mathsf{wt}_i((v_1, (v_2, b)))$.
- **Return edge:** $v_2 = (v, b)$ is a *return* node, for some exit node $v = ex_i$ and some box b and each of the following (i) $L_1 = L_2 \circ b$; and (ii) $(v_1, v) \in E_i$; and (iii) $\sigma = \mathsf{wt}_i((v_1, v))$.

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

Execution paths. An *execution path* is a sequence of configurations $P = \langle \mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_\ell \rangle$, such that for each integer i where $1 \leq i \leq \ell - 1$, we have that $(\mathcal{C}_i, \mathcal{C}_{i+1})$ is a global edge. The *length* of P is $|P| = \ell - 1$, and a single configuration is by itself is 0-length execution path. Also, we say that the stack height of an execution path is the maximum stack height of a configuration in the execution path. For a pair of configurations $\mathcal{C}, \mathcal{C}'$, the set $\mathcal{C} \rightsquigarrow \mathcal{C}'$, is the set of execution paths $\langle \mathcal{C}_1, \mathcal{C}_2, \dots, \mathcal{C}_\ell \rangle$, for any ℓ, where $\mathcal{C} = \mathcal{C}_1$ and $\mathcal{C}' = \mathcal{C}_\ell$. For a set X of execution paths, the set $\mathcal{B}(X, h) \subseteq X$ is the subset of execution paths, with stack height at most h. Given a complete semiring $(\Sigma, \oplus, \otimes, \overline{\mathbf{0}}, \overline{\mathbf{1}})$, the *weight* of a execution path $P = \langle \mathcal{C}_1, \mathcal{C}_2, \dots, \mathcal{C}_\ell \rangle$ is $\otimes(P) = \bigotimes(\mathsf{wt}(\mathcal{C}_1, \mathcal{C}_2), \dots, \mathsf{wt}(\mathcal{C}_{\ell-1}, \mathcal{C}_\ell))$ if $|P| \geq 1$ and $\mathsf{wt}(P) = \overline{\mathbf{1}}$ otherwise. Given configurations $\mathcal{C}, \mathcal{C}'$, the *configuration distance* $d(\mathcal{C}, \mathcal{C}')$ is defined as $d(\mathcal{C}, \mathcal{C}') = \bigoplus_{P : \mathcal{C} \rightsquigarrow \mathcal{C}'} \otimes(P)$ (the empty sum is $\overline{\mathbf{0}}$). Also, given configurations $\mathcal{C}, \mathcal{C}'$ and a stack height h, where c' is h-reachable from c, the *bounded height configuration distance* $d(\mathcal{C}, \mathcal{C}', h)$ is defined as $d(\mathcal{C}, \mathcal{C}', h) = \bigoplus_{P : \mathcal{B}(\mathcal{C} \rightsquigarrow \mathcal{C}', h)} \otimes(P)$. Note that the above definition of execution paths only allows for so called *valid* paths [Reps et al. 1995; Sagiv et al. 1996], i.e., paths that fully respect the calling contexts of an execution.

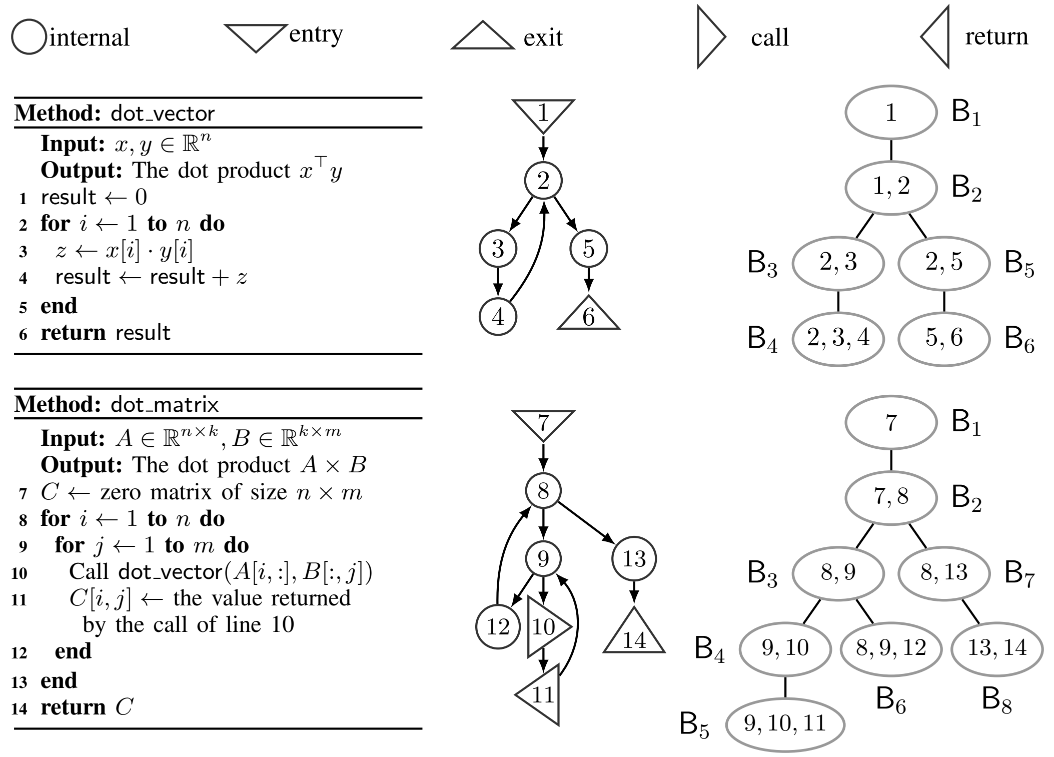
The algebraic path problem on RSMs of constant treewidth. Given (i) a RSM $A = \{A_1, A_2, \dots, A_k\}$; and (ii) for each $1 \leq i \leq k$ a balanced tree-decomposition $T_i = (V_{T_i}, E_{T_i})$ of the graph (V_i, E_i') with constant treewidth at most $t = O(1)$; and (iii) a complete semiring $(\Sigma, \oplus, \otimes, \overline{\mathbf{0}}, \overline{\mathbf{1}})$, the *algebraic path problem* on input nodes u, v, asks for the distance $d((u, \emptyset), (v, \emptyset))$, i.e. the distance between the configurations with the empty stack. Similarly, also given a height h, the *bounded height algebraic path problem* on input configurations c, c', asks for the distance $d((u, \emptyset), (v, \emptyset), h)$. When it is clear from the context, we will write $d(u, v)$ to refer to the algebraic path problem of nodes u and v on RSMs.

Remark 2.17. Note that the empty stack restriction implies that u and v are nodes of the same CSM. However, the paths from u to v are, in general, interprocedural, and thus involve invocations and returns from other CSMs. This formulation has been used before in terms of *same-context* [Chaudhuri 2008] and *same-level* [Reps et al. 1995] realizable paths and has several applications in program analysis, e.g. by capturing balanced parenthesis-like properties used in alias analysis [Sridharan et al. 2005].

2.4. Problems

A wide range of interprocedural problems can be formulated as bounded height algebraic path problems. Some examples follow.

(1) *Reachability* i.e., given nodes u, v in the same CSM, is there a path from u to v? The problem can be formulated on the boolean semiring $(\{\mathsf{True}, \mathsf{False}\}, \vee, \wedge, \mathsf{False}, \mathsf{True})$.
(2) *Shortest path* i.e., given a weight function $\mathsf{wt} : E \to \mathbb{R}_{\geq 0}$ and nodes u, v in the same CSM, what is the weight of the minimum-weight path from u to v? The problem can be formulated on the tropical semiring $(\mathbb{R}_{\geq 0} \cup \{\infty\}, \min, +, \infty, 0)$.
(3) *Most probable path* i.e., given a probability function $P : E \to [0, 1]$ and nodes u, v in the same CSM, what is the probability of the highest-probable path from u to v? The problem can be formulated on the Viterbi semiring $([0, 1], \max, \cdot, 0, 1)$.
(4) The class of *interprocedural, finite, distributive, subset (IFDS)* problems defined in [Reps et al. 1995]. Given a finite domain D, a universe of flow functions F containing distributive functions $f : 2^D \to 2^D$, a weight function $\mathsf{wt} : E \to F$ associates each edge with a flow function. The weight of an interprocedural path is then defined as the composition $\circ$ of the flow functions along its edges, and the IFDS problem given nodes u, v asks for the meet $\sqcap$ (union or intersection) of the weights of all $u \rightsquigarrow v$ paths. The problem can be formulated on the meet-composition semiring $(F, \sqcap, \circ, \emptyset, I)$, where I is the identity function.
(5) The class of *interprocedural distributive environment (IDE)* problems defined in [Sagiv et al. 1996]. This class of dataflow problems is an extension to IFDS, with the difference that the flow functions (called environment transformers) map elements from the finite domain D to values in an infinite set (e.g., of the form $f : D \to \mathbb{N}$). An environment transformer is denoted as $f[d \to \ell]$, meaning that the element $d \in D$ is mapped to value ℓ, while the mapping of all other elements remains unchanged. The problem can be formulated on the meet-environment-



```
Method: dot_vector
   Input: $x, y \in \mathbb{R}^n$
   Output: The dot product $x^\top y$
1  result $\leftarrow 0$
2  for $i \leftarrow 1$ to $n$ do
3    $z \leftarrow x[i] \cdot y[i]$
4    result $\leftarrow$ result $+ z$
5  end
6  return result
```

```
Method: dot_matrix
   Input: $A \in \mathbb{R}^{n\times k}, B \in \mathbb{R}^{k\times m}$
   Output: The dot product $A \times B$
7  $C \leftarrow$ zero matrix of size $n \times m$
8  for $i \leftarrow 1$ to $n$ do
9    for $j \leftarrow 1$ to $m$ do
10     Call dot_vector($A[i,:], B[:,j]$)
11     $C[i,j] \leftarrow$ the value returned
         by the call of line 10
12   end
13 end
14 return $C$
```

Fig. 3: Example of a program consisting of two methods, their control flow graphs $G_i = (V_i, E_i')$ where nodes correspond to line numbers, and the corresponding tree decompositions, each one achieving treewidth 2.

transformer semiring $(F, \sqcap, \circ, \emptyset, I)$, where I is the identity environment transformer, leaving every map unchanged.

Note that if we assume that the set of weights of all interprocedural paths in the system is finite, then the size of this set bounds the stack height h. Additionally, several problems can be formulated as algebraic path problems in which bounding the stack height can be viewed as an approximation to them (e.g., the probability of reaching a node v from a node u).

Model and word tricks. We consider the standard RAM model with word size $W = \Theta(\log n)$, where poly(n) is the size of the input. Our reachability algorithm (in Section 5) uses so called "word tricks" heavily. We use constant-time LCA queries which also use word tricks [Harel and Tarjan 1984; Bender and Farach-Colton 2000].

3. DYNAMIC ALGORITHMS FOR PREPROCESS, UPDATE AND QUERY

In the current section we present data structures that take as input a constant treewidth graph G of n nodes and a nicely rooted, balanced, binary tree decomposition $T = (V_T, E_T)$ of G, and achieve the following tasks:

(1) Preprocessing the tree-decomposition T to answer algebraic path queries fast.
(2) Updating the preprocessed T upon change of the weight wt(u, v) of an edge (u, v).
(3) Querying the preprocessed T to retrieve the distance $d(u, v)$ of any pair of nodes u, v.

Note that by Lemma 2.10 and Lemma 2.12, a nicely rooted, balanced tree decomposition can be constructed in time linear in the size of the input. In the following section we use the results of this section in order to preprocess RSMs fast, in order to answer interprocedural same-context algebraic path queries fast. Refer to Example 4.1 of Section 4 for an illustration on how these algorithms are executed on an RSM.

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

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

Informally, given a bag B, a U-shaped path in B is a path that traverses intermediate nodes whose root bag is either B or some descendant bag of B in T.

Example 3.1 (U-*shaped paths*). Here we present some examples of U-shaped paths on the tree decomposition of the control-flow graph of method $\mathsf{dot\_vector}$ of Figure 3.

(1) The path $3 \rightarrow 4 \rightarrow 2$ is U-shaped in bag B_3 since the root bag of the intermediate node 4 is B_4, which is a child of B_3. The same path is also U-shaped in bag B_4 since a node of the tree is considered ancestor of itself.
(2) The path $1 \rightarrow 2$ is U-shaped in bag B_2 since B_2 is the root bag of node 2.

In the following we present three algorithms for (i) preprocessing a tree decomposition, (ii) updating the data structures of the preprocessing upon a weight change $\mathsf{wt}(u, v)$ of an edge (u, v), and (iii) querying for the distance $d(u, v)$ for any pair of nodes u, v. The intuition behind the overall approach is that for every path $P : u \rightsquigarrow v$ and $z = \mathsf{argmin}_{x \in P} \mathsf{Lv}(x)$, the path P can be decomposed to paths $P_1 : u \rightsquigarrow z$ and $P_2 : z \rightsquigarrow v$. By Lemma 2.8, if we consider the path $P' : \mathsf{B}_u \rightsquigarrow \mathsf{B}_z$ and any bag $\mathsf{B}_i \in P'$, we can find nodes $x, y \in \mathsf{B}_i \cap P_1$ (not necessarily distinct). Then P_1 is decomposed to a sequence of U-shaped paths P_1^i, one for each such B_i, and the weight of P_1 can be written as the $\otimes$-product of the weights of P_1^i, i.e., $\otimes(P_1) = \bigotimes(\otimes(P_1^i))$. A similar observation holds for P_2. Hence, the task of preprocessing and updating is to summarize in each B_i the weights of all such U-shaped paths between all pairs of nodes appearing in B_i. To answer the query, the algorithm traverses upwards the tree T from B_u and B_v, and combines the summarized paths to obtain the weights of all such paths P_1 and P_2, and eventually P, such that $\otimes(P) = d(u, v)$.

Informal description of preprocessing. Algorithm $\mathsf{Preprocess}$ associates with each bag B a *local* U-*shaped distance* map $\mathsf{LUD}_{\mathsf{B}} : \mathsf{B} \times \mathsf{B} \rightarrow \Sigma$. Upon a weight change, algorithm Update updates the local U-shaped distance map of some bags. It will hold that after the preprocessing and each subsequent update, $\mathsf{LUD}_{\mathsf{B}}(u, v) = \bigoplus_{P : u \rightsquigarrow v} \{\otimes(P)\}$, where all P are U-shaped paths in B. Given this guarantee, we later present an algorithm for answering (u, v) queries with $d(u, v)$, the distance from u to v. Algorithm $\mathsf{Preprocess}$ is a dynamic programming algorithm. It traverses T bottom-up, and for a currently examined bag B that is the root bag of a node x, it calls the method Merge to compute the local U-shaped distance map $\mathsf{LUD}_{\mathsf{B}}$. In turn, Merge computes $\mathsf{LUD}_{\mathsf{B}}$ depending only on the local U-shaped distance maps $\mathsf{LUD}_{\mathsf{B}_i}$ of the children $\{\mathsf{B}_i\}$ of B, and uses the closure operator $*$ to capture possibly unbounded traversals of cycles whose smallest-level node is x. See Method 1 and Algorithm 2 for a formal description.

LEMMA 3.2. *At the end of* $\mathsf{Preprocess}$*, for every bag* B *and nodes* $u, v \in \mathsf{B}$*, we have* $\mathsf{LUD}_{\mathsf{B}}(u, v) = \bigoplus_{P : u \rightsquigarrow v} \{\otimes(P)\}$*, where all* P *are* U*-shaped paths in* B.

PROOF. The proof is by induction on the parents. Initially, B is a leaf, and hence the root bag of some node x (recall that we assume wlog that every leaf bag is the root bag of some node). Thus

Method 1: Merge

```
   Input: A bag $\mathsf{B}_x$ with children $\{\mathsf{B}_i\}_i$
   Output: A local U-shaped distance map $\mathsf{LUD}_{\mathsf{B}_x}$
1  Assign $\mathsf{wt}'(x,x) \leftarrow \left(\bigotimes\{\mathsf{LUD}_{\mathsf{B}_1}(x,x)^*,\dots,\mathsf{LUD}_{\mathsf{B}_j}(x,x)^*\}\right)^*$
2  foreach $u \in \mathsf{B}_x$ with $u \neq x$ do
3      Assign $\mathsf{wt}'(x,u) \leftarrow \bigoplus\{\mathsf{wt}(x,u),\mathsf{LUD}_{\mathsf{B}_1}(x,u),\dots,\mathsf{LUD}_{\mathsf{B}_j}(x,u)\}$
4      Assign $\mathsf{wt}'(u,x) \leftarrow \bigoplus\{\mathsf{wt}(u,x),\mathsf{LUD}_{\mathsf{B}_1}(u,x),\dots,\mathsf{LUD}_{\mathsf{B}_j}(u,x)\}$
5  end
6  foreach $u,v \in \mathsf{B}_x$ do
7      Assign $\delta \leftarrow \bigotimes(\mathsf{wt}'(u,x),\mathsf{wt}'(x,x),\mathsf{wt}'(x,v))$
8      Assign $\mathsf{LUD}_{\mathsf{B}_x}(u,v) \leftarrow \bigoplus\{\delta,\mathsf{LUD}_{\mathsf{B}_1}(u,v),\dots,\mathsf{LUD}_{\mathsf{B}_j}(u,v)\}$
9  end
```

ALGORITHM 2: Preprocess

```
   Input: A tree-decomposition $T=(V_T,E_T)$
   Output: A local U-shaped distance map $\mathsf{LUD}_{\mathsf{B}}$ for each bag $\mathsf{B} \in V_T$
1  Traverse $T$ bottom up and examine each bag $\mathsf{B}$ with children $\{\mathsf{B}_i\}_i$
2  if $\mathsf{B}$ is the root bag of some node $x$ then
3      Assign $\mathsf{LUD}_{\mathsf{B}} \leftarrow$ Merge on $\mathsf{B}$
4  else
5      foreach $u,v \in \mathsf{B}$ do
6          Assign $\mathsf{LUD}_{\mathsf{B}}(u,v) \leftarrow \bigoplus\{\mathsf{LUD}_{\mathsf{B}_1}(u,v),\dots,\mathsf{LUD}_{\mathsf{B}_j}(u,v)\}$
7      end
8  end
```

each such path P can only go through x, and hence will be captured by Preprocess. Now consider the case that the algorithm examines a bag B, and by the induction hypothesis the statement is true for all $\{\mathsf{B}_i\}$ children of B_x. The correctness follows easily if B is not the root bag of any node, since every such P is a U-shaped path in some child B_i of B, and Preprocess propagates the contents of all $\mathsf{LUD}_{\mathsf{B}_i}$ to $\mathsf{LUD}_{\mathsf{B}}$ in the else block of line 4. Now consider that B is the root bag some node x (recall that since the tree decomposition is nicely-rooted, B can be the root bag of at most one node), and any U-shaped path $P' : u \rightsquigarrow v$ that additionally visits x, and decompose it to paths $P_1 : u \rightsquigarrow x$, $P_2 : x \rightsquigarrow x$ and $P_3 : x \rightsquigarrow v$, such that x is not an intermediate node in either P_1 or P_3 (see Figure 4 for an illustration). By distributivity, we have:

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

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

of B, and we have by distributivity and the induction hypothesis

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

and the last expression equals $\mathsf{wt}'(x,x)$ from line 1 of Merge. It follows that in line 6 of Merge we have $\delta = \bigoplus_{P'} \otimes(P')$.

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

LEMMA 3.3. $\mathsf{Preprocess}$ *requires* $O(n)$ *semiring operations.*

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

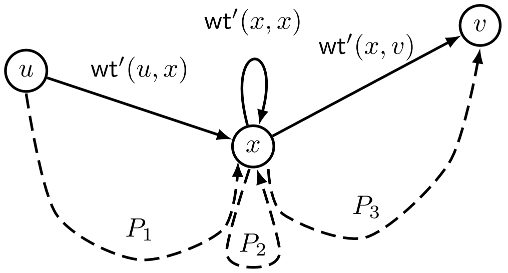


Fig. 4: Illustration of the inductive argument of $\mathsf{Preprocess}$ when the algorithm processes a bag B_x that is the root bag of node x. Every $u \rightsquigarrow v$ path is decomposed to three paths $P_1 : u \rightsquigarrow x$, $P_2 : x \rightsquigarrow x$ and $P_3 : x \rightsquigarrow v$ such that x is not an intermediate node of either P_1 or P_3. All paths P_1 and P_3 are U-shaped in some child bag of B_x, thus their weights have been captured in the corresponding LUD maps by the induction hypothesis, and hence as weights $\mathsf{wt}'(u,x)$ and $\mathsf{wt}'(x,v)$. The path P_2 is not necessarily U-shaped in B_x, as it might contain x as an intermediate node. However, P_2 is decomposed to paths $P_2^i : x \rightsquigarrow x$ such that x does not appear as an intermediate node of any P_2^i. Hence, every such path is also U-shaped, and the argument proceeds as in the case of P_1 and P_3. Note that this is where the algorithm makes use of the semiring closure operator $*$ to capture the effect of (unbounded) cycle traversals, since P_2 is a cycle around x.

Informal description of updating. Algorithm Update is called whenever the weight $\mathsf{wt}(x,y)$ of an edge of G has changed. Given the guarantee of Lemma 3.2, after Update has run on an edge update $\mathsf{wt}(x,y)$, it restores the property that for each bag B we have $\mathsf{LUD}_{\mathsf{B}}(u,v) = \bigoplus_{P:u\rightsquigarrow v}\{\otimes(P)\}$, where all P are U-shaped paths in B. See Algorithm 3 for a formal description.

LEMMA 3.4. *At the end of each run of* Update*, for every bag* B *and nodes* $u, v \in \mathsf{B}$*, we have* $\mathsf{LUD}_{\mathsf{B}}(u,v) = \bigoplus_{P:u\rightsquigarrow v}\{\otimes(P)\}$*, where all* P *are* U*-shaped paths in* B.

ALGORITHM 3: Update

```
Input: An edge $(x, y)$ with new weight $\mathsf{wt}(x, y)$
Output: A local U-shaped distance map $\mathsf{LUD_B}$ for each bag $\mathsf{B} \in V_T$
Assign $\mathsf{B} \leftarrow \mathsf{B}_{(x,y)}$, the highest bag containing the edge $(x, y)$
repeat
    Call Merge on B
    Assign $\mathsf{B} \leftarrow \mathsf{B}'$ where $\mathsf{B}'$ is the parent of B
until $\mathsf{Lv}(\mathsf{B}) = 0$
```

PROOF. First, by the definition of a U-shaped path P in B it follows that the statement holds for all bags not processed by Update, since for any such bag B and U-shaped path P in B, the path P cannot traverse (u, v). For the remaining bags, the proof follows an induction on the parents updated by Update, similar to that of Lemma 3.2. □

LEMMA 3.5. Update *requires* $O(\log n)$ *operations per update.*

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

Informal description of querying. Algorithm Query answers a (u, v) query with the distance $d(u, v)$ from u to v. Because of Lemma 2.8, every path $P : u \rightsquigarrow v$ is guaranteed to go through the least common ancestor (LCA) B_L of B_u and B_v, and possibly some of the ancestors B of B_L. Given this fact, algorithm Query uses the procedure Climb to climb up the tree from B_u and B_v until it reaches B_L and then the root of T. For each encountered bag B along the way, it computes maps $\delta_u(w) = \bigoplus_{P_1}\{\otimes(P_1)\}$, and $\delta_v(w) = \bigoplus_{P_2}\{\otimes(P_2)\}$ where all $P_1 : u \rightsquigarrow w$ and $P_2 : w \rightsquigarrow v$ are such that the root bag of each intermediate node y is a descendant of B. This guarantees that for path P such that $d(u, v) = \otimes(P)$, when Query examines the bag B_z that is the root bag of $z = \mathsf{argmin}_{x \in P}\mathsf{Lv}(x)$, it will be $d(u, v) = \bigotimes(\delta_u(z), \delta_v(z))$. Hence, for Query it suffices to maintain a current best solution δ, and update it with $\delta \leftarrow \bigoplus\{\delta, \bigotimes(\delta_u(x), \delta_v(x))\}$ every time it examines a bag B that is the root bag of some node x. Figure 5 presents a pictorial illustration of Query and its correctness. Method 4 presents the Climb procedure which, given a current distance map of a node δ, a current bag B and a flag Up, updates δ with the distance to (if Up = True), or from (if Up = False) each node in B. See Method 4 and Algorithm 5 for a formal description.

Method 4: Climb

```
Input: A bag B, a map $\delta$, a flag Up
Output: A new map $\delta$
Remove from $\delta$ all $w \notin \mathsf{B}$
Assign $\delta(w) \leftarrow \overline{\mathbf{0}}$ for all $w \in \mathsf{B}$ and not in $\delta$
if B is the root bag of some node $x$ then
    if Up then                                                      /* Climbing up */
        Update $\delta$ with $\delta(w) \leftarrow \bigoplus\{\delta(w), \bigotimes(\delta(x), \mathsf{LUD_B}(x, w))\}$
    else                                                          /* Climbing down */
        Update $\delta$ with $\delta(w) \leftarrow \bigoplus\{\delta(w), \bigotimes(\delta(x), \mathsf{LUD_B}(w, x))\}$
    end
return $\delta$
```

LEMMA 3.6. Query *returns* $\delta = d(u, v)$.

PROOF. Let $P : u \rightsquigarrow v$ be any path from u to v, and $z = \mathsf{argmin}_{x \in P}\mathsf{Lv}(x)$ the lowest level node in P. Decompose P to $P_1 : u \rightsquigarrow z$, $P_2 : z \rightsquigarrow v$, and it follows that $\otimes(P) = \bigotimes(\otimes(P_1), \otimes(P_2))$.

ALGORITHM 5: Query

```
   Input: A pair $(u, v)$
   Output: The distance $d(u, v)$ from $u$ to $v$
1  Initialize map $\delta_u$ with $\delta_u(w) \leftarrow \mathsf{LUD}_{\mathsf{B}_u}(u, w)$
2  Initialize map $\delta_v$ with $\delta_v(w) \leftarrow \mathsf{LUD}_{\mathsf{B}_v}(w, v)$
3  Assign $\mathsf{B}_L \leftarrow$ the LCA of $\mathsf{B}_u$, $\mathsf{B}_v$ in $T$
4  Assign $\mathsf{B} \leftarrow \mathsf{B}_u$
5  repeat
6      Assign $\mathsf{B} \leftarrow \mathsf{B}'$ where $\mathsf{B}'$ is the parent of $\mathsf{B}$
7      Call Climb on $\mathsf{B}$ and $\delta_u$ with flag Up set to True
8  until $\mathsf{B} = \mathsf{B}_L$
9  Assign $\mathsf{B} \leftarrow \mathsf{B}_v$
10 repeat
11     Assign $\mathsf{B} \leftarrow \mathsf{B}'$ where $\mathsf{B}'$ is the parent of $\mathsf{B}$
12     Call Climb on $\mathsf{B}$ and $\delta_v$ with flag Up set to False
13 until $\mathsf{B} = \mathsf{B}_L$
14 Assign $\mathsf{B} \leftarrow \mathsf{B}_L$
15 Assign $\delta \leftarrow \bigoplus_{x \in \mathsf{B}_L} \otimes(\delta_u(x), \delta_v(x))$
16 repeat
17     Assign $\mathsf{B} \leftarrow \mathsf{B}'$ where $\mathsf{B}'$ is the parent of $\mathsf{B}$
18     Call Climb on $\mathsf{B}$ and $\delta_u$ with flag Up set to True
19     Call Climb on $\mathsf{B}$ and $\delta_v$ with flag Up set to False
20     if $\mathsf{B}$ is the root bag of some node $x$ then
21         Assign $\delta \leftarrow \bigoplus\{\delta, \bigotimes(\delta_u(x), \delta_v(x))\}$
22 until $\mathsf{Lv}(\mathsf{B}) = 0$
23 return $\delta$
```

We argue that when Query examines B_z, it will be $\delta_u(z) = \bigoplus_{P_1} \otimes(P_1)$ and $\bigoplus_{P_2} \delta_v(z) = \otimes(P_2)$. We only focus on the $\delta_u(z)$ case here, as the $\delta_v(z)$ is similar. We argue inductively that when algorithm Query examines a bag B_x, for all $w \in \mathsf{B}_x$ we have $\delta_u(w) = \bigoplus_{P'}\{\otimes(P')\}$, where all P' are such that for each intermediate node y we have $\mathsf{Lv}(y) \geq \mathsf{Lv}(x)$. Initially (line 1), it is $x = u$, $\mathsf{B}_x = \mathsf{B}_u$, and every such P' is U-shaped in B_u, hence $\mathsf{LUD}_{\mathsf{B}_x}(x, w) = \bigoplus_{P'}\{\otimes(P')\}$ and $\delta_u(w) = \bigoplus_{P'}\{\otimes(P')\}$. Now consider that Query examines a bag B_x (Lines 7 and 18) and the claim holds for $\mathsf{B}_{x'}$ a descendant of B_x previously examined by Query. If x does not occur in P', it is a consequence of Lemma 2.8 that $w \in \mathsf{B}_{x'}$, hence by the induction hypothesis, P' has been considered by Query. Otherwise, x occurs in P' and decompose P' to P'_1, P'_2, such that P'_1 ends with the first occurrence of x in P', and it is $\otimes(P) = \bigotimes(\otimes(P'_1), \otimes(P'_2))$. Note that P'_2 is a U-shaped path in B_x, hence $\mathsf{LUD}_{\mathsf{B}_x}(x, w) = \bigoplus_{P'_2}\{\otimes(P'_2)\}$. Finally, as a consequence of Lemma 2.8, we have that $x \in \mathsf{B}_{x'}$, and by the induction hypothesis, $\delta_u(x) = \bigoplus_{P'_1}\{\otimes(P'_1)\}$. It follows that after Query processes B_x, it will be $\delta_u(w) = \bigoplus_{P'}\{\otimes(P')\}$. By the choice of z, when Query examines the bag B_z, it will be $\delta_u(z) = \bigoplus_{P_1}\{\otimes(P_1)\}$. A similar argument shows that at that point it will also be $\delta_v(z) = \bigoplus_{P_2}\{\otimes(P_2)\}$, hence at that point $\delta = \bigotimes(\otimes(P_1), \otimes(P_2)) = d(u, v)$. □

LEMMA 3.7. Query *requires* $O(\log n)$ *semiring operations.*

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

We conclude the results of this section with the following theorem.

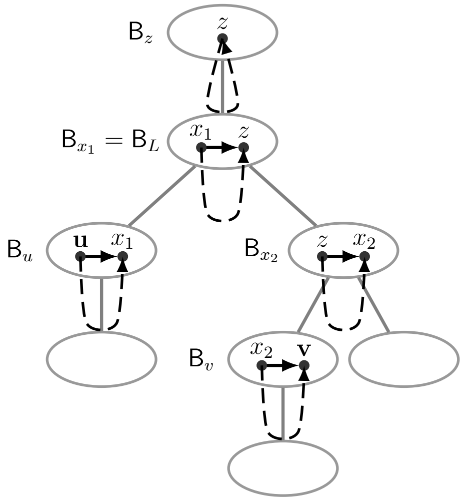


Fig. 5: Illustration of Query in computing the distance $d(u,v) = \otimes(P)$ as a sequence of U-shaped paths, whose weight has been captured in the local distance map of each bag. When B_z is examined, with $z = \mathsf{argmin}_{x \in P} \mathsf{Lv}(x)$, it will be $\delta_u(z) = d(u,z)$ and $\delta_v(z) = d(z,v)$, and hence by distributivity $d(u,v) = \bigotimes(\delta_u(z), \delta_v(z))$.

THEOREM 3.8. *Consider a graph $G = (V,E)$ of n nodes and treewidth $t = O(1)$, and a nicely rooted, balanced and binary tree-decomposition $T = (V_T, E_T)$ of G that has constant width. The following assertions hold:*

(*1*) Preprocess *requires $O(n)$ semiring operations;*
(*2*) Update *requires $O(\log n)$ semiring operations per edge weight update; and*
(*3*) Query *correctly answers distance queries in $O(\log n)$ semiring operations.*

Example 3.9 (*Illustration of* Preprocess, Update *and* Query). Here we illustrate the operations Preprocess, Update and Query on the tree decomposition of the control-flow graph of method dot_matrix in Figure 3 for the boolean semiring $(\{\mathsf{True}, \mathsf{False}\}, \vee, \wedge, \mathsf{False}, \mathsf{True})$ that expresses reachability.

Preprocess. We illustrate Preprocess along the path $\mathsf{B}_6 \rightsquigarrow \mathsf{B}_1$ of the tree decomposition. The algorithm traverses the tree decomposition bottom-up starting from B_6, which is passed as argument to Merge. The map $\mathsf{LUD}_{\mathsf{B}_6}$ contains the reachability information as obtained by the edge set of the algorithm, i.e., we have $\mathsf{LUD}_{\mathsf{B}_6}(8,9) = \mathsf{LUD}_{\mathsf{B}_6}(9,12) = \mathsf{LUD}_{\mathsf{B}_6}(12,8) = \mathsf{True}$, and all other entries of $\mathsf{LUD}_{\mathsf{B}_{12}}$ between different nodes equal False. These values of $\mathsf{LUD}_{\mathsf{B}_{12}}$ are stored in function wt' constructed in the first 5 lines of Merge. Note that B_6 is the root bag of node 12, and hence Merge will perform path-shortening on paths that go through node 12. In particular, after the loop in line 6 is executed, we also have that $\mathsf{LUD}_{\mathsf{B}_1 2}(9,8) = \mathsf{True}$ (i.e., the algorithm discovers new reachability information from 9 to 8), and Merge terminates. Next, Preprocess calls Merge on bag B_3 which is the root bag of node 9. Due to initialization, we have $\mathsf{LUD}_{\mathsf{B}_3}(8,9) = \mathsf{True}$, since $(8,9)$ is an edge of the graph, and thus $\mathsf{wt}'(8,9) = \mathsf{True}$ in line 4 of Merge. On the other hand, $\mathsf{LUD}_{\mathsf{B}_3}(9,8) = \mathsf{False}$, as $(9,8)$ is not an edge of the graph. However, due to the previous step, we have $\mathsf{LUD}_{\mathsf{B}_6}(9,8) = \mathsf{True}$, and line 3 of Merge sets $\mathsf{wt}'(9,8) = \mathsf{True}$. This shows how the reachability information discovered in the child bag B_6 is propagated to the parent bag B_3. After the loop of line 6 is executed, this reachability information is stored to the map $\mathsf{LUD}_{\mathsf{B}_3}$, i.e., we have $\mathsf{LUD}_{\mathsf{B}_3}(8,9) = \mathsf{LUD}_{\mathsf{B}_3}(9,8) = \mathsf{True}$. Finally, Preprocess processes bags B_2 and B_1 without discovering any new reachability information.

Update. We illustrate Update after inserting the reachability information $10 \rightsquigarrow 11$, i.e., setting $\mathsf{wt}(10, 11) = \mathsf{True}$. Before the update, we assume that we have $\mathsf{wt}(10, 11) = \overline{\mathbf{0}}$, i.e., there is no edge $(10, 11)$ in the graph. This is because 10 is a call node and 11 is the corresponding return node, hence it is not to be assumed that $10 \rightsquigarrow 11$ as the invocation might never return. By calling Update on the pair $(10, 11)$, the algorithm in line 1 identifies B_5 as the highest bag that contains both nodes, and calls Merge on that bag. In turn, Merge constructs the weight function wt' with $\mathsf{wt}'(9, 10) = \mathsf{wt}'(10, 11) = \mathsf{wt}'(11, 9) = \mathsf{True}$. Note that B_5 is the root bag of node 11, hence the loop in line 6 performs path shortening through node 11, and will set $\mathsf{LUD}_{\mathsf{B}_5}(10, 11) = \mathsf{LUD}_{\mathsf{B}_5}(10, 9) = \mathsf{True}$. After Merge has terminated, Update will move to the parent bag B_4 of B_5, which is passed to Merge. Due to the previous step, we have $\mathsf{LUD}_{\mathsf{B}_5}(10, 9) = \mathsf{True}$, and line 3 of Merge sets $\mathsf{wt}'(10, 9) = \mathsf{True}$. This bag is the root bag of node 10, and the path shortening in the loop of line 6 will set $\mathsf{LUD}_{\mathsf{B}_4}(10, 9) = \mathsf{True}$. After Merge terminates, Update will move to bags B_3, B_2 and B_1, and the corresponding invocations to Merge will not discover any new reachability information.

Query. We illustrate Query on the pair $(10, 13)$. The root bag of nodes 10 and 13 are B_4 and B_7, respectively, and their LCA bag is B_2. In line 1 and line 2, Query initializes the maps δ_{10} and δ_{13}, respectively, with $\delta_{10}(10) = \delta_{10}(9) = \mathsf{True}$ and $\delta_{13}(13) = \delta_{13}(8) = \mathsf{True}$. The first step is to execute the loop in line 5, which will call Climb on input the parent bag B_3 of B_4 and the map δ_{10}. Note that B_3 is the root bag of node 8, and since $\mathsf{LUD}_{\mathsf{B}_3}(8, 19) = \mathsf{True}$ due to the preprocessing, line 5 of Climb will insert $\delta_{10}(8) = \mathsf{True}$. Afterwards, Query will proceed with the parent bag B_2 of B_3, and the call to Climb will not modify the map δ_{10}. The second step is to execute the loop in line 10, which will call Climb on input the parent bag B_2 of B_7 and the map δ_{13}. Note that B_2 is the root bag of node 8, and since $\mathsf{LUD}_{\mathsf{B}_2}(7, 8) = \mathsf{True}$ due to the preprocessing, line 7 of Climb will insert $\delta_{13}(7) = \mathsf{True}$. Finally, line 15 of Query will set $\delta = \delta_{10}(8) \otimes \delta_{13}(8) = \mathsf{True}$, and the reachability relation between nodes 10 and 13 is discovered. Note that the loop of line 16 will further process the parent bag B_1 of B_2, without any effect in the returned value δ, hence Query will return True (i.e., indeed node 13 is reachable from node 10) as desired.

4. ALGORITHMS FOR CONSTANT TREEWIDTH RSMS

In this section we consider the bounded height algebraic path problem on RSMs of constant treewidth. That is, we consider (i) an RSM $A = \{A_1, A_2, \dots, A_k\}$, where A_i consists of n_i nodes and b_i boxes; (ii) a complete semiring $(\Sigma, \oplus, \otimes, \overline{\mathbf{0}}, \overline{\mathbf{1}})$; and (iii) a maximum stack height h. Our task is to create a data structure that after some preprocessing can answer queries of the form: Given a pair $((u, \emptyset), (v, \emptyset))$ of configurations compute $d((u, \emptyset), (v, \emptyset), h)$ (also recall Remark 2.17). For this purpose, we present the algorithm $\mathsf{RSMDistance}$, which performs such preprocessing using a data structure $\mathcal{D}$ consisting of the algorithms $\mathsf{Preprocess}$, Update and Query of Section 3. At the end of $\mathsf{RSMDistance}$ it will hold that algebraic path pair queries in a CSM A_i can be answered in $O(\log n_i)$ semiring operations. Although our algorithms apply to RSMs of arbitrary treewidth, they are efficient only when treewidth is small. We later present some additional preprocessing which suffers a factor of $O(\log n_i)$ in the preprocessing space, but reduces the pair query time to constant.

4.1. Algorithms For The Bounded Stack Height Problem

We start with our general solution to the bounded-stack height problem. We first describe the algorithm $\mathsf{RSMDistance}$ for preprocessing the RSM, and afterwards the algorithms for performing same-context queries.

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

(1) In a preprocessing phase, it computes a nicely rooted, balanced, binary tree decomposition $T_i = (V_{T_i}, E_{T_i})$ of each CSM G_i.
(2) It preprocesses the control flow graphs $G_i = (V_i, E'_i)$ of the CSMs A_i using Preprocess of Section 3, where the weight function wt_i for each G_i is extended such that $\mathsf{wt}_i((en, b), (ex, b)) = \overline{\mathbf{0}}$ for all pairs of call and return nodes to the same box b. This allows the computation of $d(u, v, 0)$ for all pairs of nodes (u, v), since no call can be made while still having zero stack height.
(3) Then, iteratively for each ℓ, where $1 \leq \ell \leq h$, given that we have a dynamic data structure $\mathcal{D}$ (concretely, an instance of the dynamic algorithms Update and Query from Section 3) for computing $d(u, v, \ell - 1)$, the algorithm does as follows: First, for each G_i whose entry to exit distance $d(en_i, ex_i, \ell - 1)$ has changed from the last iteration and for each G_j that contains a box pointing to G_i, it updates the call to return distance of the corresponding nodes, using Query.
(4) Then, it obtains the entry to exit distance $d(en_j, ex_j, \ell)$ to see if it was modified, and continues with the next iteration of $\ell + 1$.

See Algorithm 6 for a formal description.

ALGORITHM 6: RSMDistance

```
   Input: A set of control flow graphs $\mathcal{G} = \{G_i\}_{1 \leq i \leq k}$, stack height $h$
1  foreach $G_i \in \mathcal{G}$ do
2      Construct a nicely rooted, balanced, binary tree-decomposition $T_i = (V_{T_i}, E_{T_i})$
3      Call Preprocess on $T_i$
4  end
5  distances ← [Call Query on $(en_i, ex_i)$ of $G_i]_{1 \leq i \leq k}$
6  modified ← $\{1, \dots, k\}$
7  for $\ell \leftarrow 1$ to $h$ do
8      modified′ ← $\emptyset$
9      foreach $i \in$ modified do
10         foreach $G_j$ that contains boxes $b_{j_1}, \dots, b_{j_l}$ s.t. $Y_j(b_{j_x}) = i$ do
11             Call Update on $G_j$ for the weight change wt$((en_i, b_{j_l}), (ex_i, b_{j_x}))$ ← distances$[i]$
12             $\delta$ ← Query$(en_j, ex_j)$
13             if $\delta \neq$ distances$[j]$ then
14                 modified′ ← modified′ $\cup \{j\}$
15                 distances$[j]$ ← $\delta$
16         end
17     end
18     modified ← modified′
19 end
```

Correctness and logarithmic pair query time. The algorithm RSMDistance is described so that a proof by induction is straightforward for correctness. Initially, running the algorithm Preprocess from Section 3 on each of the graphs G_i allows queries for the distances $d(u, v, 0)$ for all pairs of nodes (u, v), since no method call can be made. Also, the induction follows directly since for every CSM A_i, updating the distance from call nodes (en, b) to the corresponding return nodes (ex, b) of every box b that corresponds to a CSM A_j whose distance $d(en_j, ex_j)$ was changed in the last iteration ℓ, ensures that the distance $d(u, v, \ell + 1)$ of every pair of nodes u, v in A_i is computed correctly. This is also true for the special pair of nodes en_i, ex_i, which feeds the next iteration of RSMDistance. Finally, RSMDistance requires $O(\sum_{i=1}^{k} n_i)$ time to construct a nicely rooted, balanced, binary tree decomposition (Lemma 2.10 and Lemma 2.12), $O(n)$ time to preprocess all G_i initially, and $O(\sum_{i=1}^{k}(b_i \cdot \log n_i))$ to update all G_i for one iteration of the loop of Line 10 (from Theorem 3.8). Hence, RSMDistance uses $O(\sum_{i=1}^{k}(n_i + h \cdot b_i \cdot \log n_i))$ preprocessing semiring operations. Finally, it is easy to verify that all preprocessing is done in $O(\sum_i n_i) = O(n)$ space.

	dot_vector						**dot_matrix**							
$\ell/\mathsf{LUD}_{\mathsf{B}_x}$	B_1	B_2	B_3	B_4	B_5	B_6	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
$\ell = 0$ (Preprocess)	–	$(1,2)$	$(2,3)$	$(2,3)$ $(3,4)$ $(4,2)$ $(\mathbf{3},\mathbf{2})$	$(2,5)$	$(5,6)$	–	$(7,8)$	$(8,9)$	$(9,10)$	$(9,10)$ $(11,9)$	$(8,9)$ $(9,12)$ $(12,8)$ $(\mathbf{9},\mathbf{8})$	$(8,13)$	$(13,14)$
$\ell = 1$ (Update)	–	$(1,2)$	$(2,3)$	$(2,3)$ $(3,4)$ $(4,2)$ $(3,2)$	$(2,5)$	$(5,6)$	–	$(7,8)$	$(8,9)$	$(9,10)$	$(9,10)$ $(11,9)$ $(\mathbf{10},\mathbf{11})$ $(\mathbf{10},\mathbf{9})$	$(8,12)$ $(9,12)$ $(12,8)$ $(9,8)$	$(8,13)$	$(13,14)$

(a)

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

(b)

Table IV: Illustration of RSMDistance on the tree decompositions of methods dot_vector and dot_matrix from Figure 3. Table (a) shows the local distance maps for each bag and stack height $\ell = 0, 1$. Table (b) shows how the distance query $d(1,6)$ in method dot_vector is handled.

After the last iteration of algorithm RSMDistance, we have a data structure $\mathcal{D}$ that occupies $O(n)$ space and answers distance queries $d(u, v, h)$ in $O(\log n_i)$ time, with $u, v \in V_i$, by calling Query from Theorem 3 for the distance $d(u, v)$ in G_i.

Example 4.1 (*Illustration of* RSMDistance). We now present a small example of how RSMDistance is executed on the RSM of Figure 3 for the case of reachability. In this case, for any pair of nodes (u, v), we have $d(u, v) = \mathsf{True}$ iff u reaches v. Table IV(a) illustrates how the local distance maps $\mathsf{LUD}_{\mathsf{B}_x}$ look for each bag B_x of each of the CSMs of the two methods dot_vector and dot_matrix. Each column represents the local distance map of the corresponding bag B_x, and an entry (u, v) means that $\mathsf{LUD}_{\mathsf{B}_x}(u, v) = \mathsf{True}$ (i.e., u reaches v). For brevity, in the table we hide self loops (i.e., entries of the form (u, u)) although they are stored by the algorithms. Initially, the stack height $\ell = 0$, and Preprocess is called for each graph (line 3). The new reachability relations discovered by Merge are shown in bold. Note that at this point we have $\mathsf{wt}(10, 11) = \mathsf{False}$ in method dot_matrix, as we do not know whether the call to method dot_vector actually returns. Afterwards, Query is called to discover the distance $d(1, 6)$ in method dot_vector (line 5). Table IV (b) shows the sequence in which Query examines the bags of the tree decomposition, and the distances δ_1, δ_6 and δ it maintains. When B_2 is examined, $\delta = \mathsf{True}$ and hence at the end Query returns $\delta = \mathsf{True}$. Finally, since Query returns $\delta = \mathsf{True}$, the weight $\mathsf{wt}(10, 11)$ between the call-return pair of nodes $(10, 11)$ in method dot_matrix is set to True. An execution of Update (line 11) with this update on the corresponding tree decomposition (Table IV(a) for $\ell = 1$) updates the entries $(10, 11)$ and $(10, 9)$ in $\mathsf{LUD}_{\mathsf{B}_5}$ of method dot_matrix (shown in bold). From this point, any same-context distance query can be answered in logarithmic time in the size of its CSM by further calls to Query.

Linear single-source query time. In order to handle single-source queries, some additional preprocessing is required. The basic idea is to use RSMDistance to process the graphs G_i, and then use additional preprocessing on each G_i by applying existing algorithms for graphs with constant treewidth. This is achieved using Lemma 2.11, which states that for each bag B of each tree-decomposition T_i, a *local distance map* $\mathsf{LD}_{\mathsf{B}} : \mathsf{B} \times \mathsf{B} \to \Sigma$ with $\mathsf{LD}_{\mathsf{B}}(u, v) = d(u, v)$ can be computed in time and space $O(n_i)$. After all such maps LD_{B} have been computed for each B, it is straightforward to answer single-source queries from some node u in linear time. The algorithm simply maintains a map $A : V_i \to \Sigma$, and initially $A(v) = d(u, v)$ for all $v \in \mathsf{B}_u$, and $A(v) = \overline{\mathbf{0}}$ otherwise. Then, it traverses T_i in a BFS manner starting at B_u, and for every encountered bag B and $v \in \mathsf{B}$, if $A(v) = \overline{\mathbf{0}}$, it sets $A(v) = \bigoplus_{z\in\mathsf{B}} \bigotimes(A(z), d(z, v))$. The correctness follows directly from Lemma 2.9. For constant treewidth, this results in a constant number of semiring operations per bag, and hence $O(n_i)$ time in total.

Constant pair query time. After RSMDistance has returned, it is possible to further preprocess the graphs G_i to reduce the pair query time to constant, while increasing the space by a factor of $\log n_i$. For constant treewidth, this can be obtained by adapting [Chaudhuri and Zaroliagis 1995, Theorem 10] to our setting, which in turn is based on a rather complicated algorithmic technique of [Alon and Schieber 1987]. We present a more intuitive, simpler and implementable approach that has a dynamic programming nature. In Section 6 we present some experimental results obtained by this approach.

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

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

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

An application of Lemma 2.8 inductively on the levels processed by the algorithm can be used to show that when a bag B is processed, for every node $u \in \mathsf{B}$ and $v \in V_i^{\mathsf{B}}$, we have $\mathsf{T}_u(v) = \bigoplus_{P:v\rightsquigarrow u} \otimes(P)$ and $\mathsf{F}_u(v) = \bigoplus_{P:u\rightsquigarrow v} \otimes(P)$. Finally, there are $O(n_i)$ semiring operations done at each level of T_i, and since there are $O(\log n_i)$ levels, $O(n_i \cdot \log n_i)$ operations are required in total. Hence, the space used is also $O(n_i \cdot \log n_i)$. We furthermore preprocess T_i in linear time and space to answer LCA queries in constant time (note that since T_i is balanced, this is standard). To answer a pair query u, v, it suffices to first obtain the LCA B of B_u and B_v, and it follows from Lemma 2.9 that $d(u, v) = \bigoplus_{z\in\mathsf{B}} \bigotimes\{T_z(u), F_z(v)\}$, which requires a constant number of semiring operations.

We conclude the results of this section with the following theorem. Afterwards, we obtain the results for the special cases of the IFDS/IDE framework, reachability and shortest path.

THEOREM 4.2. *Fix the following input: (i) an RSM $A = \{A_1, A_2, \dots, A_k\}$ with treewidth $t = O(1)$, where A_i consists of n_i nodes and b_i boxes; (ii) a complete semiring $(\Sigma, \oplus, \otimes, \overline{\mathbf{0}}, \overline{\mathbf{1}})$; and (iii) a maximum stack height h. Let $n = \sum_i n_i$.* RSMDistance *operates in the following complexity bounds.*

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

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

Remark 4.3. We note that the pair query time of Item 1 in Theorem 4.2 can be improved further to $O(\alpha(n_i))$ time, where $\alpha(n_i)$ is the inverse of the Ackermann function on input (n_i, n_i). This is achieved using [Chaudhuri and Zaroliagis 1995, Theorem 10, Item (ii)], instead of the process described above. This result is only of theoretical interest, as (i) the hidden constants are large, and (ii) the data structure for achieving such bounds is hard to be implemented in practice.

Remark 4.4. Our complexity analysis so far has neglected the precise dependency on treewidth, since we assume that the treewidth t of the RSM is bounded by a constant, i.e., $t = O(1)$. Here we clarify this dependency. Assuming that the tree decompositions in line 2 are given, all our algorithms have dependency of factor $O(t^2)$ in their complexity. This can be traced to the proofs Lemma 3.3, Lemma 3.5 and Lemma 3.7, which state explicitly that the time dependency of the algorithms Preprocess, Update and Query, respectively, is $O(t^2)$. Since, in general, computing the treewidth of a graph is NP-hard, the complexity of computing the tree decompositions in line 2 is at least 2^t, depending on the precise algorithm used.

4.2. IFDS Framework

In the special case where the algebraic path problem belongs to the IFDS/IDE framework, we have a meet-composition semiring $(F, \sqcap, \circ, \emptyset, I)$, where F is a set of distributive flow functions $2^D \to 2^D$, D is a set of data facts, $\sqcap$ is the meet operator (either union or intersection), $\circ$ is the flow function composition operator, and I is the identity flow function. For a fair comparison, the $\circ$ semiring operation does not induce a unit time cost, but instead a cost of $O(|D|)$ per data fact (as functions are represented as bipartite graphs [Reps et al. 1995]). Because the set D is finite and the meet operator is either union or intersection, it follows that the image of every data fact will be updated at most $|D|$ times. Hence, in the preprocessing phase where Preprocess(G_i) is called for each graph G_i, the total time spent for each G_i is $O(n_i \cdot |D|^3)$. Additionally, line 7 of RSMDistance needs to change so that instead of h iterations, the body of the loop is carried up to a fixpoint. The amortized cost for all edge updates per G_i is then $O(b_i \cdot \log n_i \cdot |D|^3)$ (as there are $|D|$ data facts).

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

COROLLARY 4.5 (IFDS/IDE). *Fix the following input (i) an RSM $A = \{A_1, A_2, \dots, A_k\}$ with treewidth $t = O(1)$, where A_i consists of n_i nodes and b_i boxes; and (ii) a meet-composition semiring $(F, \sqcap, \circ, \emptyset, I)$ where F is a set of distributive flow functions $D \to D$, $\circ$ is the flow function composition operator and $\sqcap$ is the meet operator. Let $n = \sum_i n_i$.* RSMDistance *operates in the following complexity bounds.*

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

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

A speedup for large data-fact domains. Here we outline a speedup for the algebraic path problem wrt the IFDS framework when the domain of data facts D is such that $|D| = \Omega(\log n)$. In this case, sets of data facts can be represented as bit sets, where the i-th bit of a bit set X is one iff it contains the i-th data fact. When $|D| = \Omega(\log n)$, such bit sets can be stored compactly in machine words. Since in the standard RAM model each machine word has size $\Theta(\log n)$, such a set X can be stored using $O(|D|/\log n)$ words. The meet $\sqcap$ (union/intersection) of two data-fact sets can be performed in $O(|D|/\log n)$ time, by computing the bit-wise OR/AND operation on the corresponding machine words. Similarly, a distributive flow function $f : 2^D \to 2^D$ can be represented using $O(|D|^2/\log n)$ words, by storing a bit set X_i for every data fact d_i for which $f(d_i) = X_i$. Using bit sets, every update of a flow function with a new data flow pair $d_i \to d_j$ incurs a $O(|D|/\log n)$ time cost, simply by performing the bit-wise OR/AND operation (depending on whether the meet operator is union/intersection) between the data-fact sets $f(d_i)$ and $f(d_j)$. Since there can be at most $|D|^2$ updates of data flow pairs $d_i \to d_j$ per graph edge, the total preprocessing cost for each graph G_i is $n_i \cdot |D|^3/\log n$. Similarly, in the update phase (line 7) the amortized cost per G_i is $b_i \cdot \log n_i \cdot |D|^3/\log n$. Finally, in the query phase, where we track data facts rather than data-flow functions, data-fact operations require $O(|D|/\log n)$ word operations per data fact. We thus arrive at the following corollary.

COROLLARY 4.6 (IFDS/IDE, LARGE DOMAIN). *Fix the following input (i) an RSM $A = \{A_1, A_2, \ldots, A_k\}$ with treewidth $t = O(1)$, where A_i consists of n_i nodes and b_i boxes; and (ii) a meet-composition semiring $(F, \sqcap, \circ, \emptyset, I)$ where F is a set of distributive flow functions $D \to D$ with $|D| = \Omega(\log n)$, $\circ$ is the flow function composition operator and $\sqcap$ is the meet operator. Let $n = \sum_i n_i$.* RSMDistance *operates in the following complexity bounds.*

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

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

Finally, we note that the special case of reachability is obtained by setting $|D| = 1$ in Corollary 4.5. In the next section we present some further improvements which allow to reduce the cost of preprocessing and querying.

4.3. Distances With Non-Negative Weights.

The distance (or shortest path) problem can be formulated on the tropical semiring $(\mathbb{R}_{\geq 0} \cup \{\infty\}, \min, +, \infty, 0)$. We consider that both semiring operators cost unit time (i.e., the weights occurring in the computation fit in a constant number of machine words). Since we consider non-negative weights, the distance between any pair of nodes is realized by an interprocedural path of stack height at most b, as no boxes need to appear more than once at any time in the stack of the path. Hence, we can solve the distance problem by setting $h = b$ in Theorem 4.2. However, our restriction to non-negative weights allows for a significant speedup, achieved by algorithm RSMDistanceTrop (see Algorithm 7). RSMDistanceTrop is obtained from RSMDistance by using a priority queue to store the distances from entries to exits. In each iteration of the while loop in Line 7 we extract the element of the queue with the smallest entry-to-exit distance, and update the entry-to-exit distances of all remaining elements in the queue that correspond to CSMs which invoke the CSM that corresponds to the extracted element. The algorithm has similar flavor to the classic Dijkstra's algorithm for distances on finite graphs with non-negative edge weights [Dijkstra 1959; Cormen et al. 2001]. The time complexity is $O(n)$ time for executing Preprocess, plus the time required for each execution of Update and Query. Note that Update is executed at least as many times as Query, and since both require time logarithmic in the size of the respective G_i, it suffices to count the total time spent on Update. Since Line 10 is executed at most once per box, the total time spent on Update is $O(\sum_{i=1}^{k} b_i \cdot \log n_i)$. We thus obtain the following corollary for distances with non-negative weights.

COROLLARY 4.7 (INTERPROCEDURAL SHORTEST PATHS). *Fix the following input (i) an RSM $A = \{A_1, A_2, \ldots, A_k\}$ with treewidth $t = O(1)$, where A_i consists of n_i nodes and b_i boxes;*

(ii) a tropical semiring $(\mathbb{R}_{\geq 0} \cup \{\infty\}, \min, +, \infty, 0)$. *Let* $n = \sum_i n_i$. RSMDistanceTrop *operates in the following complexity bounds.*

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

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

ALGORITHM 7: RSMDistanceTrop

```
Input: A set of control-flow graphs $\mathcal{G} = \{G_i\}_{1 \leq i \leq k}$
foreach $G_i \in \mathcal{G}$ do
    Construct a nicely rooted, balanced, binary tree-decomposition $T_i = (V_{T_i}, E_{T_i})$ of $G_i$
    Call Preprocess on $T_i$
end
distances $\leftarrow$ [Call Query on $(en_i, ex_i)$ of $G_i]_{1 \leq i \leq k}$
PriorityQueue$Q \leftarrow [(i,$ Call Query on $(en_i, ex_i)$ of $G_i)]_{1 \leq i \leq k}$
while $Q$ is not empty do
    $(i, \delta) \leftarrow Q.pop()$
    foreach $G_j$ that contains boxes $b_{j_1}, \dots, b_{j_l}$ s.t. $Y_j(b_{j_x}) = i$ do
        Call Update on $G_j$ for the weight change wt$((en_i, b_{j_l}), (ex_i, b_{j_x})) \leftarrow \delta$
        $\delta' \leftarrow$ Query$(en_j, ex_j)$
        if $\delta' <$ distances$[j]$ then
            distances[j] $\leftarrow \delta'$
            Decrease the key of $j$ in $Q$ to $\delta'$
    end
end
```

4.4. A Note On Non-Same-Context Queries

The focus of this work is on exploiting the constant-treewidth property of control-flow graphs for speeding up same-context interprocedural algebraic path queries. In full generality, interprocedural static analyses are also concerned with non-same-context queries, i.e., where the endpoints of a query are control nodes of different CSMs. We note that our techniques do not extend straightforwardly to general queries. Since non-same-context queries take place at the RSM level instead of the local CSM, fast algorithms for such queries will likely have to depend on the structural properties of the RSM rather than the CSM. However, although it is well known that the control-flow graphs of programs have small treewidth [Thorup 1998], this property is not guaranteed for the whole RSM (e.g., when viewed as a supergraph [Sagiv et al. 1996]). Although in practice a non-same-context query can be broken down to multiple same-context queries, this does not lead to complexity improvements (compared to doing the analysis offline) unless further restrictions are assumed about the structure of the RSM.

5. OPTIMAL REACHABILITY FOR LOW-TREEWIDTH GRAPHS

In this section we turn our attention to the problem of reachability on low-treewidth graphs. We present algorithms for building and querying a data-structure Reachability, which handles single-source and pair reachability queries over an input graph G of n nodes and treewidth t. In particular, we establish the following.

THEOREM 5.1. *Given a graph G of* n *nodes and treewidth t, let* $\mathcal{T}(G)$ *be the time and* $\mathcal{S}(G)$ *be the space required for constructing a balanced tree-decomposition* $T = (V_T, E_T)$ *of* $O(n)$ *bags* *and width $O(t)$ on the standard RAM with wordsize $W = \Theta(\log n)$. The data-structure* Reachability *correctly answers reachability queries and requires*

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

For constant-treewidth graphs we have that $\mathcal{T}(G) = O(n)$ and $\mathcal{S}(G) = O(n)$ (Lemma 2.10), and thus along with Theorem 5.1 we obtain the following corollary.

COROLLARY 5.2. *Given a graph G of n nodes and treewidth $t = O(1)$, the data-structure* Reachability *requires $O(n)$ preprocessing time and space, and correctly answers (i) pair reachability queries in $O(1)$ time, and (ii) single-source reachability queries in $O\left(\frac{n}{\log n}\right)$ time.*

Implications on the reachability of constant-treewidth RSMs. In conjunction with Theorem 4.2, we obtain the following corollary for RSMs. This is achieved by executing the algorithm RSMDistance as before, in order to infer in every CSM A_i the reachability information between every pair of call and return nodes (c, r). Afterwards, every corresponding graph G_i can be viewed independently, so that we can use Reachability to further preprocess it in order to handle same-context reachability queries. This approach yields the following corollary.

COROLLARY 5.3 (INTERPROCEDURAL REACHABILITY). *Fix the following input a (i) constant treewidth RSM $A = \{A_1, A_2, \dots, A_k\}$, where A_i consists of n_i nodes and b_i boxes. Let $n = \sum_i n_i$. Using $O(\sum_{i=1}^{k}(n_i + b_i \cdot \log n_i))$ preprocessing time and $O(n)$ space, one can correctly answer same-context algebraic pair queries in $O(1)$ time, and same-context algebraic single-source queries in $O(\frac{n_i}{\log n})$ time.*

In the remaining of this section we focus on a single input graph $G = (V, E)$ of treewidth t, without further mention of RSMs.

Intuition. Informally, the preprocessing consists of first obtaining a small, balanced and binary tree-decomposition T of G, and computing the local reachability information in each bag B (i.e., the pairs $(u, v) \in E^*$ with $u, v \in \mathsf{B}$) using Lemma 2.11. Then, the whole of preprocessing is done on T, by constructing two types of sets, which are represented as bit sequences and packed into words of length $W = \Theta(\log n)$. Initially, every node u receives an *index* i_u, such that for every bag B, the indices of nodes whose root bag is in $T(\mathsf{B})$ form a contiguous interval. Additionally, for every appearance of node u in a bag B, the node u receives a *local index* l_u^{B} in B. For brevity, a sequence $(A^0, A^1, \dots A^k)$ will be denoted by $(A^i)_{0 \leq i \leq k}$. When k is implied, we simply write $(A^i)_i$. The following two types of sets are constructed.

(1) Sets that store information about subtrees. Specifically, for every node u, the set F_u stores the relative indices of nodes v that can be reached from u, and whose root bag is in $T(\mathsf{B}_u)$. These sets are used to answer single-source queries.
(2) Sets that store information about ancestors. Specifically, for every node u, two sequences of sets are stored $(\mathsf{F}_u^i)_{0 \leq i \leq \mathsf{Lv}(u)}$, $(\mathsf{T}_u^i)_{0 \leq i \leq \mathsf{Lv}(u)}$, such that F_u^i (resp., T_u^i) contains the local indices of nodes v in the ancestor bag B_u^i of B_u at level i, such that $(u, v) \in E^*$ (resp., $(v, u) \in E^*$). These sets are used to answer pair queries.

The sets of the first type are constructed by a bottom-up pass, whereas the sets of the second type are constructed by a top-down pass. Both passes are based on the separator property of tree decompositions (recall Lemma2.7 and Lemma 2.8), which informally states that reachability properties between nodes in distant bags will be captured transitively, through nodes in intermediate bags.

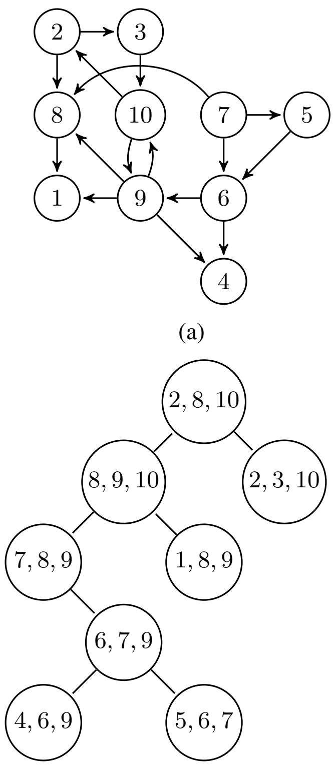
Reachability Preprocessing. We now give a formal description of the preprocessing of Reachability that takes as input a graph G of n nodes and treewidth t, and a balanced tree-decomposition $T = (V_T, E_T)$ of G that has $O(t)$. After the preprocessing, Reachability supports single-source and pair reachability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

(1) Turn T to a small, balanced binary tree-decomposition of G of width $O(t)$, using Lemma 2.13.
(2) Preprocess T to answer LCA queries in $O(1)$ time [Harel and Tarjan 1984].
(3) Compute the local distance map $\mathsf{LD}_\mathsf{B} : \mathsf{B} \times \mathsf{B} \to \mathbb{R}$ for every bag B w.r.t reachability, using Lemma 2.11. Hence, for any bag B and nodes $u, v \in \mathsf{B}$, we have $\mathsf{LD}_\mathsf{B}(u, v) = 1$ iff $(u, v) \in E^*$.
(4) Apply a pre-order traversal on T, and assign an incremental index i_u to each node u at the time the root bag B of u is visited. If there are multiple nodes u for which B is the root bag, assign the indices to those nodes in some arbitrary order. Additionally, store the number s_u of nodes whose root bag is in $T(\mathsf{B})$ and have index at least i_u. Finally, for each bag B and $u \in \mathsf{B}$, assign a unique local index l_u^B to u, and store in B the number of nodes (with multiplicities) a_B contained in all ancestors of B, and the number b_B of nodes in B.
(5) For every node u, initialize a bit set F_u of length s_u, pack it into words, and set the first bit to 1.
(6) Traverse T bottom-up, and for every bag B execute the following step. For every pair of nodes $u, v \in \mathsf{B}$ such that B is the root bag of v and $i_u < i_v$ and $\mathsf{LD}_\mathsf{B}(u, v) = 1$, insert F_v to the segment $[i_v - i_u, i_v - i_u + s_v]$ of F_u (the nodes reachable from v now become reachable from u, through v).
(7) For every node u initialize two sequences of bit sets $(\mathsf{T}_u^i)_{0 \leq i \leq \mathsf{Lv}(u)}$, $(\mathsf{F}_u^i)_{0 \leq i \leq \mathsf{Lv}(u)}$, and pack them into consecutive words. Each set T_u^i and F_u^i has size $b_{\mathsf{B}_u^i}$, where B_u^i is the ancestor of B_u at level i.
(8) Traverse T top-down, and for B the bag currently visited, for every node $x \in \mathsf{B}$, maintain two sequences of bit sets $(\overline{\mathsf{T}}_x^i)_{0 \leq i \leq \mathsf{Lv}(\mathsf{B})}$ and $(\overline{\mathsf{F}}_x^i)_{0 \leq i \leq \mathsf{Lv}(\mathsf{B})}$. Each set $\overline{\mathsf{T}}_x^i$ and $\overline{\mathsf{F}}_x^i$ has size b_{B^i}, where B^i is the ancestor of B at level i. Initially, B is the root of T (hence $\mathsf{Lv}(\mathsf{B}) = 0$), and set the position l_w^B of $\overline{\mathsf{F}}_x^0$ (resp., $\overline{\mathsf{T}}_x^0$) to 1 for every node w such that $\mathsf{LD}_\mathsf{B}(x, w) = 1$ (resp., $\mathsf{LD}_\mathsf{B}(w, x) = 1$). For each other bag B encountered in the traversal, do as follows. Let $S = \mathsf{B} \cap \mathsf{B}'$, where B' is the parent of B in T, and let x range over S.
    (a) For each node x, create a set $\overline{\mathsf{T}}_x$ (resp., $\overline{\mathsf{F}}_x$) of 0s of length b_B, and for every $w \in \mathsf{B}$ such that $\mathsf{LD}_\mathsf{B}(x, w) = 1$ (resp., $\mathsf{LD}_\mathsf{B}(w, x) = 1$), set the l_w^B-th bit of $\overline{\mathsf{F}}_x$ (resp., $\overline{\mathsf{T}}_x$) to 1. Append the set $\overline{\mathsf{T}}_x$ (resp., $\overline{\mathsf{F}}_x$) to $(\overline{\mathsf{T}}_x^i)_i$ (resp., $(\overline{\mathsf{F}}_x^i)_i$). Now each set sequence $(\overline{\mathsf{T}}_x^i)_i$ and $(\overline{\mathsf{F}}_x^i)_i$ has size $a_\mathsf{B} + b_\mathsf{B}$.
    (b) For each $u \in \mathsf{B}$ whose root bag is B, initialize set sequences $(\overline{\mathsf{F}}_u^i)_i$ and $(\overline{\mathsf{T}}_u^i)_i$ with 0s of length $a_\mathsf{B} + b_\mathsf{B}$ each, and set the bit at position l_u^B of $\overline{\mathsf{F}}_u^{\mathsf{Lv}(\mathsf{B})}$ and $\overline{\mathsf{T}}_u^{\mathsf{Lv}(\mathsf{B})}$ to 1. For every $w \in \mathsf{B}$ with $\mathsf{LD}_\mathsf{B}(u, w) = 1$ (resp., $\mathsf{LD}_\mathsf{B}(w, u) = 1$), insert $(\overline{\mathsf{F}}_w^i)_i$ to $(\overline{\mathsf{F}}_u^i)_i$ (resp., $(\overline{\mathsf{T}}_w^i)_i$ to $(\overline{\mathsf{T}}_u^i)_i$). Finally, set $(\mathsf{F}_u^i)_i$ equal to $(\overline{\mathsf{F}}_u^i)_i$ (resp., $(\mathsf{T}_u^i)_i$ equal to $(\overline{\mathsf{T}}_u^i)_i$).

Figure 6 illustrates the constructed sets on a small example.

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

LEMMA 5.4. *At the end of preprocessing, for every pair of nodes u and v with $i_u \leq i_v \leq i_u + s_u$, if there exists a path $P : u \rightsquigarrow v$ such that for every $w \in P$, we have $i_u \leq i_w \leq i_u + s_u$, then the $(i_v - i_u)$-th bit of* F_u *is* 1.



(a)

(c)

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

(b)

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

(d)

Fig. 6: (a), (c): A graph G and a tree-decomposition $T = (V_T, E_T)$ of G. (b): The sets F_u constructed from step 5 to answer single-source queries. The j-th bit of a set F_u is 1 iff $(u, v) \in E^*$, where v is such that $i_v - i_u = j$. (d): The set sequences $(\mathsf{F}_u^i)_i$ and $(\mathsf{T}_u^i)_i$ constructed from step 6 to answer pair queries, for $u = 6$. For every $i \in \{0, 1, 2, 3\}$ and ancestor B_6^i of B_6 at level i, every node $v \in \mathsf{B}_u^i$ is assigned a local index $l_v^{\mathsf{B}_6^i}$. The j-th bit of set F_6^i (resp. T_6^i) is 1 iff $(6, v) \in E^*$ (resp. $(v, 6) \in E^*$), where v is such that $l_v^{\mathsf{B}_6^i} = j$.

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

(1) If $|P_2| = 0$, the statement is true by taking $x = v$, since the 0-th bit of F_v is 1.
(2) If $|P_2| > 0$, examine the child B' of B in P_2. By Lemma 2.8, there exists $x \in \mathsf{B} \cap \mathsf{B}' \cap P$, and let $P_3 : x \rightsquigarrow v$. By the induction hypothesis there exists some $y \in \mathsf{B}' \cap P_3$ with $i_y \leq i_v \leq i_y + s_y$ and the $i_v - i_y$-th bit of F_y is 1. If $y \in \mathsf{B}$, we take $x = y$. Otherwise, B' is the root bag of y, and by the local distance computation of Lemma 2.11, it is $\mathsf{LD}_{\mathsf{B}'}(x, y) = 1$. By the choice of x, y we have that B_x is an ancestor of B_y. Thus, by construction we have $i_x < i_y$ and $s_x \geq s_y + i_y - i_x$, and hence $i_x \leq i_v \leq i_x + s_x$. Then in step 5, F_y is inserted in position $i_y - i_x$ of F_x, thus the bit at position $i_y - i_x + i_v - i_y = i_v - i_x$ of F_x will be 1, and we are done.

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

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

LEMMA 5.5. *At the end of preprocessing, for every node u, for every $v \in \mathsf{B}_u^i$ where B_u^i is the ancestor of B_u at level i, we have that if $(u,v) \in E^*$ (resp., $(v,u) \in E^*$), then the $l_v^{\mathsf{B}_u^i}$-th bit of F_u^i (resp., T_u^i) is* 1 .

LEMMA 5.6. *Given a graph G with n nodes and treewidth t, let $\mathcal{T}(G)$ be the time and $\mathcal{S}(G)$ be the space required for constructing a balanced tree-decomposition of G with $O(n)$ bags and width $O(t)$. The preprocessing phase of* Reachability *on G requires $O(\mathcal{T}(G) + n \cdot t^2)$ time and $O(\mathcal{S}(G) + n \cdot t)$ space.*

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

1. Using Lemma 2.13, this step requires $O(n \cdot t)$ time. From this point on, T consists of $b = O(\frac{n}{t})$ bags, has height $\eta = O(\log n)$, and width $t' = O(t)$.
2. By a standard construction for balanced trees, preprocessing T to answer LCA queries in $O(1)$ time requires $O(b) = O(\frac{n}{t})$ time.
3. By Lemma 2.11, this step requires $O(b \cdot t'^3) = O(\frac{n}{t} \cdot t^3) = O(n \cdot t^2)$ time and $O(b \cdot t'^2) = O(\frac{n}{t} \cdot t^2) = O(n \cdot t)$ space.
4. Every bag B is visited once, and each operation on B takes constant time. We make $O(t')$ such operations in B, hence this step requires $O(b \cdot t') = O(n)$ time in total.

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

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

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

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

$$\sum_{u\in V} 2\cdot\left\lceil\frac{a_{\mathsf{B}_u}+b_{\mathsf{B}_u}}{W}\right\rceil \leq 2\cdot n\cdot\left\lceil\frac{(t'+1)\cdot\eta}{W}\right\rceil = O(n\cdot t)$$

since $a_{\mathsf{B}_u} + b_{\mathsf{B}_u} \leq (t'+1)\cdot\eta$, $\eta = O(\log n)$ and $W = \Theta(\log n)$.

7. The space required is the space for storing the set sequences $(\overline{\mathsf{T}}_v^i)_i$ and $(\overline{\mathsf{F}}_v^i)_i$, which is $O(t^2)$ by a similar argument as in the previous item. The time required is $O(t)$ for initializing every new

set sequence $(\overline{\mathsf{T}}^i_u)_i$ and $(\overline{\mathsf{F}}^i_u)_i$ and this will happen once for each node u at its root bag B_u, hence the total time is $O(n \cdot t)$.

□

Reachability Querying. We now turn our attention to the querying phase.

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

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

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

LEMMA 5.7. *After the preprocessing phase of* Reachability, *pair and single-source reachability queries are answered correctly in* $O\left(\left\lceil \frac{t}{\log n} \right\rceil\right)$ *and* $O\left(\frac{n \cdot t}{\log n}\right)$ *time respectively.*

PROOF. Let $t' = O(t)$ be the width of the small tree-decomposition constructed in Step 1. The correctness of the pair query comes immediately from Lemma 5.5 and Lemma 2.7, which implies that every path $u \rightsquigarrow v$ must go through the LCA of B_u and B_v. The time complexity follows from the $O\left(\left\lceil \frac{t}{W} \right\rceil\right)$ word operations on the sets $\mathsf{F}_u^{\mathsf{Lv}(\mathsf{B})}$ and $\mathsf{T}_v^{\mathsf{Lv}(\mathsf{B})}$ of size $O(t)$ each.

Now consider the single-source query from a node u and let v be any node such that there is a path $P : u \rightsquigarrow v$. Let B be the LCA of $\mathsf{B}_u, \mathsf{B}_v$, and by Lemma 2.7, there is a node $y \in \mathsf{B} \cap P$. Let x be the last such node in P, and let $P' : x \rightsquigarrow v$ be the suffix of P from x. It follows that P' is a path such that for every $w \in P'$ we have $i_x \leq i_w \leq i_x + s_x$.

(1) If B_v is an ancestor of B_u, then necessarily $x = v$, and by Lemma 5.5, the l_v^{B}-th bit of $\mathsf{F}_u^{\mathsf{Lv}(\mathsf{B})}$ is 1. Then the algorithm sets the i_v-th bit of A to 1.
(2) Else, B_x is an ancestor of B_v (recall that a bag is an ancestor of itself), and by Lemma 5.4, the $(i_v - i_x)$-th bit of F_x is 1.
   (a) If B is B_u, the algorithm will insert F_x to the segment $[i_x, i_x + s_x]$ of A, thus the $i_x + i_v - i_x = i_v$-th bit of A is set to 1.
   (b) If B is not B_u, it can be seen that $j_{\min} \leq i_v \leq j_{\max}$, where $j_{\min}$ and $j_{\max}$ are the smallest and largest indices of nodes whose root bag is in $T(\mathsf{B}')$, with B' the child of B that is not ancestor of B_u. Since the $(i_v - i_x)$-th bit of F_x is 1, the $(i_v - j_{\min})$-th bit of the $[j_{\min}, j_{\max}]$ segment of F_x is 1, thus the $j_{\min} + i_v - j_{\min} = i_v$-th bit of A is set to 1.

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

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

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

6. EXPERIMENTAL RESULTS

In the previous sections, we have dealt with the algebraic path problem on RSMs in a purely algorithmic way. In this section we make an experimental evaluation of our algorithms on a standard benchmark set, for various types of specific data-flow analyses.

Experimental setup. We have implemented our preprocessing and query algorithms for the IFDS framework, which is the most widely-used static dataflow analysis framework. Our benchmark set consists of the DaCapo benchmark suit [Blackburn 2006] that contains several real-world Java applications. Every benchmark is represented as an RSM that consists of several CSMs, and each CSM corresponds to the control flow graph of a method of the benchmark. We have used the Soot framework [Vallée-Rai et al. 1999] for obtaining the control flow graphs, where every node of the graph corresponds to one Jimple statement of Soot. The tree decompositions were computed using the tool JTDec [Chatterjee et al. 2017], which computes tree decompositions of control flow graphs using a variant of Thorup's algorithm [Thorup 1998] for Java source code.

Compared algorithms. We have instantiated our preprocessing and query algorithms for the IFDS framework, for 6 different types of interprocedural data-flow analysis: *control-flow reachability*, *unused variables*, *reaching definitions*, *live variables*, *simple uninitialized variables* and *possibly uninitialized variables*. In each case, we compared the performance of our algorithms with three standard variants of the IFDS framework:

(1) The standard, offline IFDS algorithm of [Reps et al. 1995], where each query is treated as a new analysis.
(2) The complete preprocessing IFDS algorithm, where the transitive closure (wrt the respective analysis semiring) is computed in the preprocessing phase, and each query is answered by a simple table lookup.
(3) The On-Demand (OD) IFDS algorithm [Horwitz et al. 1995], which does no preprocessing, but uses memoization to speed-up subsequent queries in the query phase.

For our algorithm and the complete preprocessing (i.e., the ones that have an explicit preprocessing phase), we have imposed a timeout (TO) of 1 hour for performing the preprocessing. In the query phase, we have performed 500 random same-context single-source/pair queries in each benchmark, and computed the average time taken to answer each query. We have imposed a timeout of 1 hour in the query phase, and in cases where a timeout occurred, we have averaged the query time over the number of queries that were completed successfully by the timeout. Our experiments were run on a Debian machine with an Intel Xeon E5-1650 3.2GHz CPU, on a single thread.

Table V reports the statistics of our benchmark set. Recall that each benchmark is represented as an RSM. In each table entry, n denotes the number of nodes in the corresponding RSM, $|f|$ denotes the number of CSMs in the RSM[3], and t denotes the treewidth of the RSM. We see that in all cases, the treewidth of the corresponding RSM is small compared to its size. We now proceed with results on each individual analysis. In each case, $|D|$ denotes the size of the analysis domain (i.e., the set of data facts) in the IFDS formulation. Entries with $1\mu s$ denote that the corresponding running time is $1\mu s$ or less.

[3] We have excluded libraries from our analysis, to avoid having the large library determine the running times.

Benchmarks							
Name	n	$\lvert f\rvert$	t	Name	n	$\lvert f\rvert$	t
antlr	506	5	3	JFlex	17272	145	8
bloat	122	5	1	jython	264	5	2
chart	7688	26	3	luindex	716	7	4
eclipse	600	10	2	lusearch	1296	6	4
fop	138	3	2	pmd	546	5	3
hsqldb	4294	18	3	polyglot	41294	338	5
javac	404	13	3	xalan	950	12	3

Table V: Statistics of our benchmark set.

Benchmarks		Control-Flow Reachability									
		Preprocessing		Query							
				Single Source				Pair			
Name	$\lvert D\rvert$	Ours	Compl	Ours	Compl	No Pre	OD	Ours	Compl	No Pre	OD
antlr	1	$53ms$	$77ms$	$850\mu s$	$1\mu s$	$7ms$	$14ms$	$5\mu s$	$1\mu s$	$788\mu s$	$332\mu s$
bloat	1	$8ms$	$1ms$	$173\mu s$	$1\mu s$	$653\mu s$	$350\mu s$	$6\mu s$	$1\mu s$	$199\mu s$	$7\mu s$
chart	1	$421ms$	$2.89s$	$1ms$	$1\mu s$	$65ms$	$57ms$	$5\mu s$	$1\mu s$	$15ms$	$2ms$
eclipse	1	$20ms$	$35ms$	$304\mu s$	$1\mu s$	$2ms$	$9ms$	$5\mu s$	$1\mu s$	$871\mu s$	$245\mu s$
fop	1	$4ms$	$2ms$	$81\mu s$	$1\mu s$	$517\mu s$	$1ms$	$2\mu s$	$1\mu s$	$228\mu s$	$19\mu s$
hsqldb	1	$158ms$	$3.38s$	$3ms$	$1\mu s$	$134ms$	$26ms$	$6\mu s$	$1\mu s$	$8ms$	$1ms$
javac	1	$17ms$	$7ms$	$163\mu s$	$1\mu s$	$1ms$	$2ms$	$5\mu s$	$1\mu s$	$576\mu s$	$49\mu s$
JFlex	1	$661ms$	$2.80s$	$1ms$	$1\mu s$	$40ms$	$14ms$	$5\mu s$	$1\mu s$	$33ms$	$2ms$
jython	1	$9ms$	$5ms$	$138\mu s$	$1\mu s$	$1ms$	$3ms$	$5\mu s$	$1\mu s$	$399\mu s$	$62\mu s$
luindex	1	$36ms$	$36ms$	$427\mu s$	$1\mu s$	$5ms$	$6ms$	$5\mu s$	$1\mu s$	$1ms$	$190\mu s$
lusearch	1	$74ms$	$365ms$	$1ms$	$1\mu s$	$28ms$	$21ms$	$6\mu s$	$1\mu s$	$2ms$	$793\mu s$
pmd	1	$37ms$	$12ms$	$397\mu s$	$1\mu s$	$2ms$	$6ms$	$5\mu s$	$1\mu s$	$770\mu s$	$206\mu s$
polyglot	1	$1.65s$	$3.89s$	$900\mu s$	$1\mu s$	$91ms$	$25ms$	$5\mu s$	$1\mu s$	$84ms$	$2ms$
xalan	1	$82ms$	$69ms$	$734\mu s$	$1\mu s$	$6ms$	$10ms$	$5\mu s$	$1\mu s$	$1ms$	$451\mu s$

Table VI: Comparison table for control-flow reachability analysis.

6.1. Control-Flow reachability.

Our first analysis is simple control-flow reachability, and the goal is to decide for pairs (u, v) of control-flow nodes, whether v *may* be reachable from u in *some* same-context execution. Given that this is a *may* analysis, the meet operator is union, i.e. a node is reachable if it is reachable by at least one path. The domain of the analysis is the singleton set $D = \{\mathbf{1}\}$, i.e. we have one data fact per node, encoding whether it is reachable or not. For each edge (v_1, v_2) of the interprocedural control flow graph, its weight is defined as $\{\mathbf{1}\} \mapsto \{\mathbf{1}\}$ and $\emptyset \mapsto \emptyset$ (the identity function). In other words, if v_1 is reachable from some source node, then so is v_2. On the other hand, if we know that v_1 is not reachable from a specific source node, this fact cannot be used to deduce that v_2 is reachable. The results of control-flow reachability analysis are shown in Table VI.

Preprocessing. We see that our algorithm spends less than 1 second in the preprocessing of all benchmarks, except polyglot. On the other hand, the complete preprocessing spends more than 2 seconds in several benchmarks. We note that, in some cases, the complete preprocessing is faster than our algorithm. However, this happens on benchmarks where the overall preprocessing time is small for both algorithms, i.e., in orders of hundredths of a second. This is expected, as our algorithm is more involved than the complete preprocessing, which leads to larger hidden constants. On the other hand, when the size of the benchmark is large, our preprocessing is always faster. This finding is consistent in all analyses.

Querying. In the query phase we see that the complete preprocessing always requires the smallest time, which is expected as every query is a simple table lookup. Compared with the offline and on-demand algorithms, our queries are always faster, often by orders of magnitude. We note that control-flow reachability is a simple problem, and the exact running times are small enough to not have a practical significance. However, we included this analysis as it is a common basis to all other, more involved analyses. Indeed, every dataflow analysis needs to at least compute control-flow reachability. In the more involved examples that follow, the difference in running times is much more amplified.

6.2. Unused Variables.

Our second analysis is unused variables detection, which is a common analysis in IDEs such as JetBrains [ReSharper 2019] and Visual Studio [Warren et al. 2016]. Its goal is to identify the variables that *must* be *unused* until a point of the program (usually the endpoint of a method) is reached. Such variables can then be removed from the program to enhance its performance. This analysis has different domains in each method. In a method m, with local variables $v_1, \ldots, v_n$, the analysis domain is $\{\overline{v_1}, \ldots, \overline{v_n}\}$, where $\overline{v_i}$ denotes the fact that the variable v_i is unused. This is a *must* analysis and hence its meet operator is intersection, i.e. a variable is flagged as unused if it is unused in every execution. The results of unused variables analysis are shown in Table VII.

Example 6.1. Consider the program in Figure 7 (left). Its control-flow graph is given in Figure 7 (right). The edges are labeled with their weights, which are distributive data-flow functions. Note that we divided the node corresponding to line 4 in two. This is a standard practice (following [Reps et al. 1995]) to model the state before and after the method call.

We now review how the weights were assigned. In line 1, three new variables a, b and c are introduced, all of which are yet unused. Hence, the edge $(1, 2)$ adds three new data-flow facts $\bar{a}, \bar{b}$ and $\bar{c}$. In line 2, the variable a is assigned, but its value is not used. Hence, line 2 does not use any variables and therefore the edge $(2, 3)$ does not change the current set of data-flow facts. In contrast, the value of b is used in line 3, hence the edge $(3, 4)$ removes the data-flow fact $\bar{b}$. Now consider the edge $(8, 4')$ which returns control from g to f. Note that the variable c of f is used in the call to g iff the variable n of g is used in this call. This is reflected by adding the data-flow fact $\bar{c}$ at point $4'$ iff the fact $\bar{n}$ holds at 8.

Consider an arbitrary same-context path in the control-flow graph that starts at 1 and ends at 5 and assume that no initial data-flow facts hold at 1. By the time we reach 5, the data-flow fact $\bar{a}$ always holds, no matter which path was taken. Hence, a is an unused variable. However, after traversing the path $\langle 1, 2, 3, 4, 6, 7, 8, 4', 5\rangle$, the fact $\bar{c}$ does not hold. Given that the meet operator is intersection over all paths, this means that $\bar{c}$ does not hold at 5. Hence, c is not an unused variable.

Preprocessing. We see that the domain of the analysis is much larger than in the previous case of control-flow reachability, and varies per benchmark. The unused variables analysis is more involved than reachability, and this has an immediate effect on preprocessing times. Both our algorithm and the complete preprocessing use significantly more time for larger domains. However, our algorithm always terminates within one hour, and typically much earlier, whereas the complete preprocessing times out in 4 cases. In all cases, our algorithm preprocesses the corresponding RSM faster than complete preprocessing, and hence the advantages of our preprocessing technique become apparent in this analysis.

```
void f(int a, int b, int c) {
  a = 1;
  c = b;
  print(g(c));
}
int g(int n) {
  return n*2;
}
```

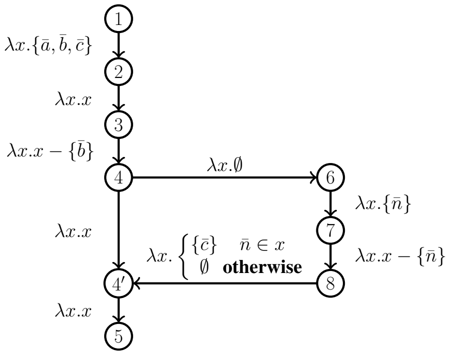

1
$\lambda x.\{\bar{a}, \bar{b}, \bar{c}\}$
2
$\lambda x.x$
3
$\lambda x.x - \{\bar{b}\}$
4
$\lambda x.\emptyset$
6
$\lambda x.\{\bar{n}\}$
7
$\lambda x.x - \{\bar{n}\}$
8
$\lambda x.x$
$\lambda x. \begin{cases} \{\bar{c}\} & \bar{n} \in x \\ \emptyset & \textbf{otherwise} \end{cases}$
4′
$\lambda x.x$
5

Fig. 7: An example program (left) and its unused variables analysis in IFDS (right).

Benchmarks		**Unused Variables**									
		Preprocessing		Query							
				Single Source				Pair			
Name	$\|D\|$	Ours	Compl	Ours	Compl	No Pre	OD	Ours	Compl	No Pre	OD
antlr	45	$2.37s$	$41.7s$	$10ms$	$1\mu s$	$1.99s$	$2.46s$	$3\mu s$	$1\mu s$	$800ms$	$26ms$
bloat	11	$17ms$	$20ms$	$300\mu s$	$1\mu s$	$5ms$	$1ms$	$2\mu s$	$1\mu s$	$2ms$	$37\mu s$
chart	146	$12m26s$	TO	$123ms$	TO	$6m23s$	$52.6s$	$10\mu s$	TO	$5m45s$	$122ms$
eclipse	30	$737ms$	$12.9s$	$4ms$	$1\mu s$	$515ms$	$566ms$	$5\mu s$	$1\mu s$	$185ms$	$22ms$
fop	11	$67ms$	$56ms$	$659\mu s$	$1\mu s$	$13ms$	$19ms$	$4\mu s$	$1\mu s$	$6ms$	$626\mu s$
hsqldb	271	$39m16s$	TO	$589ms$	TO	$23m4s$	$1m22s$	$12\mu s$	TO	$24m6s$	$211ms$
javac	21	$189ms$	$311ms$	$1ms$	$1\mu s$	$69ms$	$10ms$	$3\mu s$	$1\mu s$	$33ms$	$312\mu s$
JFlex	149	$11m55s$	TO	$76ms$	TO	$3m30s$	$5.09s$	$6\mu s$	TO	$3m7s$	$174ms$
jython	17	$76ms$	$276ms$	$669\mu s$	$1\mu s$	$38ms$	$55ms$	$2\mu s$	$1\mu s$	$17ms$	$2ms$
luindex	39	$2.63s$	$13.54s$	$9ms$	$1\mu s$	$1.30s$	$537ms$	$5\mu s$	$1\mu s$	$585ms$	$23ms$
lusearch	91	$1m2s$	$52m42s$	$62ms$	$4\mu s$	$37.8s$	$10.8s$	$9\mu s$	$1\mu s$	$20.5s$	$94ms$
pmd	30	$1.04s$	$2.56s$	$4ms$	$1\mu s$	$308ms$	$355ms$	$4\mu s$	$1\mu s$	$223ms$	$12ms$
polyglot	186	$31m6s$	TO	$60ms$	TO	$6m30s$	$20.7s$	$10\mu s$	TO	$7m21s$	$196ms$
xalan	32	$2.35s$	$35.5s$	$9ms$	$1\mu s$	$1.33s$	$640ms$	$4\mu s$	$1\mu s$	$714ms$	$24ms$

Table VII: Comparison table for unused variables analysis.

Querying. As expected, the complete preprocessing is the fastest in answering both single-source and pair queries, in the cases where the preprocessing phase was completed in time. However, we have seen that this comes at a cost of significantly larger preprocessing times. At the same time, our algorithm answers queries very fast, only at the cost of lightweight preprocessing. On the other hand, the offline and on-demand algorithms handle queries much more slowly. It is also worth noting that these two approaches experience high variance in their running times. This is expected, as e.g., the memoization heuristics of the on-demand algorithm have no theoretical guarantees.

6.3. Reaching definitions.

Our third analysis is reaching definitions. Reaching definitions is one of the most classic data-flow analyses and is often used as the textbook example for this family (see e.g. [Cooper and Torczon 2011; Appel and Palsberg 2002; Nielson et al. 2015]). In this analysis, the data-flow facts in D

```
1 void f() {
2   int a=1;
3   if(...)
4     a = a - 1;
5   else
6     a = 3 * a;
7 }
```

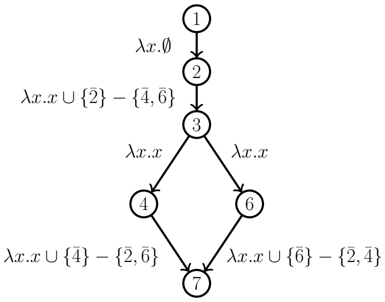

1
$\lambda x.\emptyset$
2
$\lambda x.x \cup \{\bar{2}\} - \{\bar{4}, \bar{6}\}$
3
$\lambda x.x$
$\lambda x.x$
4
6
$\lambda x.x \cup \{\bar{4}\} - \{\bar{2}, \bar{6}\}$
$\lambda x.x \cup \{\bar{6}\} - \{\bar{2}, \bar{4}\}$
7

Fig. 8: An example program (left) and its reaching definitions analysis in IFDS (right).

correspond to definition sites, i.e. points in the program where a value is assigned to a variable. Consider a definition site s that assigns a value to variable x. The data-flow fact corresponding to s holds at a node u of the control-flow graph if the value assigned at s *may* remain unchanged until the program reaches u, i.e. if there is a path from s to u in which no new value is assigned to x. Given that reaching definitions is a *may* analysis, its meet operator is union. The results for reaching definitions analysis are shown in Table VIII.

Example 6.2. Consider the program in Figure 8 (left). Its control-flow graph, and its weights for reaching definitions analysis, are given in Figure 8 (right). Here, we have $D = \{\bar{2}, \bar{4}, \bar{6}\}$, where $\bar{i}$ denotes that the definition in line i reaches the current node. Note that every time there is an assignment to the variable a, the new definition becomes active and deactivates every other definition of the same variable. Consider a reaching definitions analysis starting at node 1. Since reaching definitions is a *may* analysis, the data-flow fact $\bar{4}$ holds at node 7, because there is at least one path starting at 1 and ending at 7 which contains $\bar{4}$. The fact $\bar{6}$ has a similar situation. However, $\bar{2}$ does not hold at 7, because every path excludes it.

Preprocessing. This analysis uses the largest domain in all our experiments. Again, the complete preprocessing is consistently slower than our algorithm, and times out in 4 cases. On the other hand, our algorithm always completes in time, and the most challenging benchmark requires around 16 minutes.

Querying. The comparison in the query phase is qualitatively similar to that of the unused variables analysis. Although our algorithm is not as fast as the the complete preprocessing, its running times are small enough to make the difference negligible in practice. In addition, they are always smaller than each of offline and on-demand algorithms.

6.4. Live variables.

Our fourth analysis is live variables [Horwitz et al. 1995; Appel and Palsberg 2002], and its goal is to determine which variables *may* be live in specific program locations. A variable is considered live in a program location if its current value *may* be used later in the program execution, i.e. if its value might be read in the future, before first being overwritten by a new value. The domain of the analysis is similar to that of unused variables. In a method m, with local variables $v_1, \dots, v_n$, the analysis domain is $\{\overline{v_1}, \dots, \overline{v_n}\}$, where $\overline{v_i}$ denotes the fact that the variable v_i is live. However, unlike our previous examples, live variables is a *backwards* analysis. Concretely, whether a variable is live at a node v should be deduced from its liveness in *successors* of v. A standard trick to address this is by reversing the edges of the control-flow graph to perform this analysis [Bodden 2012].

Benchmarks		Reaching Definitions									
		Preprocessing		Query							
				Single Source				Pair			
Name	$\lvert D\rvert$	Ours	Compl	Ours	Compl	No Pre	OD	Ours	Compl	No Pre	OD
antlr	54	$2.16s$	$28.4s$	$17ms$	$3\mu s$	$2.37s$	$3.24s$	$3\mu s$	$1\mu s$	$337ms$	$20ms$
bloat	11	$18ms$	$7ms$	$384\mu s$	$1\mu s$	$2ms$	$1ms$	$5\mu s$	$1\mu s$	$1ms$	$39\mu s$
chart	170	$8m16s$	TO	$209ms$	TO	$3m29s$	$1m7s$	$11\mu s$	TO	$2m9s$	$85ms$
eclipse	31	$404ms$	$11.1s$	$3ms$	$1\mu s$	$203ms$	$530ms$	$3\mu s$	$1\mu s$	$71ms$	$17ms$
fop	12	$51ms$	$40ms$	$528\mu s$	$1\mu s$	$9ms$	$15ms$	$4\mu s$	$1\mu s$	$2ms$	$495\mu s$
hsqldb	289	$12m17s$	TO	$809ms$	TO	$5m22s$	$1m23s$	$8\mu s$	TO	$4m8s$	$114ms$
javac	21	$144ms$	$119ms$	$1ms$	$1\mu s$	$31ms$	$9ms$	$3\mu s$	$1\mu s$	$13ms$	$192\mu s$
JFlex	149	$9m26s$	TO	$102ms$	TO	$1m22s$	$5.09s$	$9\mu s$	TO	$55.5s$	$101ms$
jython	18	$69ms$	$102ms$	$755\mu s$	$1\mu s$	$19ms$	$45ms$	$4\mu s$	$1\mu s$	$7ms$	$1ms$
luindex	40	$1.51s$	$7.06s$	$6ms$	$1\mu s$	$595ms$	$619ms$	$3\mu s$	$1\mu s$	$192ms$	$19ms$
lusearch	95	$1m5s$	$33m31s$	$95ms$	$5\mu s$	$32.5s$	$11.5s$	$9\mu s$	$1\mu s$	$7.24s$	$64ms$
pmd	31	$637ms$	$929ms$	$3ms$	$1\mu s$	$157ms$	$441ms$	$3\mu s$	$1\mu s$	$81ms$	$12ms$
polyglot	212	$16m6s$	TO	$69ms$	TO	$2m8s$	$17.9s$	$10\mu s$	TO	$2m26s$	$165ms$
xalan	33	$1.54s$	$16.4s$	$6ms$	$1\mu s$	$1.49s$	$663ms$	$4\mu s$	$1\mu s$	$340ms$	$14ms$

Table VIII: Comparison table for reaching definitions analysis.

The results for live variables analysis are shown in Table IX. Although the running times differ from the previous analyses, the qualitative conclusions are the same.

Example 6.3. Figure 9 shows an example program (left) and its reversed control-flow graph labeled by weights for live variables analysis (right). Suppose that the analysis starts at node 6, i.e. endpoint of the program. At this point no variable is live, given that the program has just ended. The same is true at node 5, i.e. after execution of the print command, no variable is used and hence no variable is live. However, at point 4, the variable b is live, because its value needs to be printed at 5. However, b is not live at $3'$, because it will be overwritten (at 4) before ever being used again. Finally, note that b is live at 3, because its value will later be used in line 8. In contrast, a is not live at 3, because its value is not used in g[4].

6.5. Uninitialized variables.

Finally, we report on two variations of uninitialized variables analysis, namely, simple uninitialized variables and possibly-uninitialized variables. For these analyses, we follow the description in [Horwitz et al. 1995]. In both cases, the task is to determine the variables that *may* be uninitialized in specific program locations, i.e. the meet operator is union. The difference between these two analyses is very subtle. In the simple uninitialized variables analysis, a variable is considered to be initialized as soon as it appears in the left hand side of any assignment, no matter what appears on the right hand side. In contrast, in possibly-uninitialized variables analysis, a variable that appears on the left hand side of an expression whose right hand side contains another possibly-uninitialized variable is not considered to be initialized. In both cases, the data-flow facts domain is similar to previous analyses. In a method m, with local variables $v_1, \ldots, v_n$, the domain is $\{\overline{v_1}, \ldots, \overline{v_n}\}$, where $\overline{v_i}$ denotes the fact that the variable v_i is uninitialized.

[4] Alternatively, we could assume that the function call statement $g(a, b)$ uses the value of a, but this is a detrimental assumption and leads to an intraprocedural, rather than interprocedural, data-flow analysis.

```
void f(int a, int b) {
   print(a);
   g(a,b);
   b=2;
   print(b);
}
void g(int p, int q) {
   print(q);
}
```

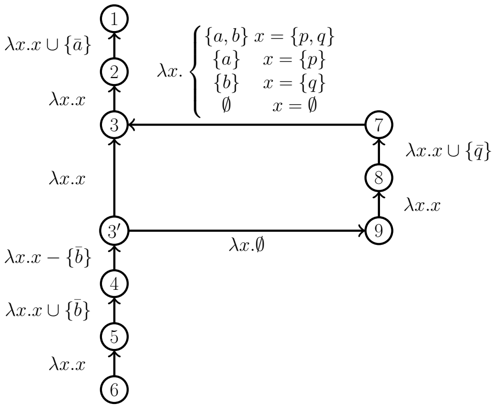

1
$\lambda x.x \cup \{\bar{a}\}$
2
$\lambda x.x$
$\lambda x. \begin{cases} \{a,b\} & x=\{p,q\} \\ \{a\} & x=\{p\} \\ \{b\} & x=\{q\} \\ \emptyset & x=\emptyset \end{cases}$
3
7
$\lambda x.x \cup \{\bar{q}\}$
$\lambda x.x$
8
$\lambda x.x$
3′
$\lambda x.\emptyset$
9
$\lambda x.x - \{\bar{b}\}$
4
$\lambda x.x \cup \{\bar{b}\}$
5
$\lambda x.x$
6

Fig. 9: An example program (left) and its live variables analysis in IFDS (right).

Benchmarks		Live Variables									
		Preprocessing		Query							
				Single Source				Pair			
Name	$\|D\|$	Ours	Compl	Ours	Compl	No Pre	OD	Ours	Compl	No Pre	OD
antlr	45	$2.42s$	$29.4s$	$16ms$	$2\mu s$	$2.04s$	$1.86s$	$5\mu s$	$1\mu s$	$220ms$	$19ms$
bloat	11	$18ms$	$14ms$	$369\mu s$	$1\mu s$	$4ms$	$958\mu s$	$4\mu s$	$1\mu s$	$1ms$	$31\mu s$
chart	146	$6m38s$	TO	$173ms$	TO	$3m56s$	$54.2s$	$7\mu s$	TO	$1m33s$	$96ms$
eclipse	30	$325ms$	$13.7s$	$3ms$	$1\mu s$	$465ms$	$405ms$	$3\mu s$	$1\mu s$	$67ms$	$24ms$
fop	11	$49ms$	$63ms$	$435\mu s$	$1\mu s$	$14ms$	$9ms$	$2\mu s$	$1\mu s$	$3ms$	$507\mu s$
hsqldb	271	$11m30s$	TO	$792ms$	TO	$4m53s$	$1m22s$	$12\mu s$	TO	$3m48s$	$145ms$
javac	21	$160ms$	$168ms$	$1ms$	$1\mu s$	$39ms$	$7ms$	$3\mu s$	$1\mu s$	$14ms$	$282\mu s$
JFlex	149	$9m56s$	TO	$106ms$	TO	$1m34s$	$3.36s$	$6\mu s$	TO	$54.6s$	$170ms$
jython	17	$88ms$	$224ms$	$1ms$	$1\mu s$	$27ms$	$50ms$	$4\mu s$	$1\mu s$	$8ms$	$1ms$
luindex	39	$2.04s$	$11.4s$	$9ms$	$2\mu s$	$811ms$	$454ms$	$6\mu s$	$1\mu s$	$189ms$	$21ms$
lusearch	91	$52.7s$	$32m5s$	$92ms$	$4\mu s$	$35.4s$	$9.20s$	$9\mu s$	$1\mu s$	$5.58s$	$76ms$
pmd	30	$732ms$	$1.39s$	$4ms$	$1\mu s$	$170ms$	$268ms$	$5\mu s$	$1\mu s$	$79ms$	$12ms$
polyglot	186	$43m31s$	TO	$59ms$	TO	$1m41s$	$18.1s$	$7\mu s$	TO	$1m58s$	$130ms$
xalan	32	$2.42s$	$40.1s$	$10ms$	$1\mu s$	$1.39s$	$416ms$	$6\mu s$	$1\mu s$	$235ms$	$15ms$

Table IX: Comparison table for live variables analysis.

The results for the analysis wrt simple uninitialized variables and possibly uninitialized variables are shown in Table X and Table XI, respectively. Although the running times differ from the previous analyses, the qualitative conclusions are the same.

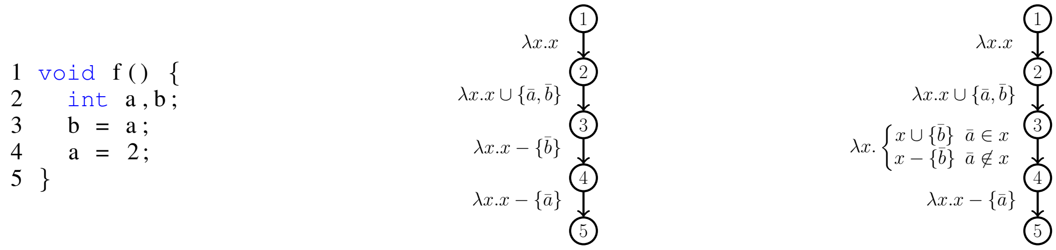
Example 6.4. Figure 10 shows an example program (left) and the modeling of simple uninitialized variables analysis (center) and possibly-uninitialized variables analysis (right) of this program in IFDS. Note that the only difference is in the weight of the edge $(3, 4)$. In line 3, the variable a is uninitialized when it is being assigned to b. In simple uninitialized variables analysis, b would be considered as initialized after line 3, whereas possibly-uninitialized variables analysis considers b as uninitialized.

Benchmarks		**Simple Uninitialized Variables**									
		Preprocessing		Query							
				Single Source				Pair			
Name	$\lvert D\rvert$	Ours	Compl	Ours	Compl	No Pre	OD	Ours	Compl	No Pre	OD
antlr	45	$1.45s$	$9.22s$	$10ms$	$2\mu s$	$1.17s$	$2.34s$	$3\mu s$	$1\mu s$	$207ms$	$13ms$
bloat	11	$14ms$	$7ms$	$321\mu s$	$1\mu s$	$3ms$	$1ms$	$4\mu s$	$1\mu s$	$1ms$	$34\mu s$
chart	146	$6m13s$	TO	$178ms$	TO	$2m1s$	$49.8s$	$11\mu s$	TO	$1m37s$	$40ms$
eclipse	30	$310ms$	$1.49s$	$3ms$	$1\mu s$	$192ms$	$565ms$	$3\mu s$	$1\mu s$	$66ms$	$12ms$
fop	11	$38ms$	$18ms$	$488\mu s$	$1\mu s$	$7ms$	$14ms$	$4\mu s$	$1\mu s$	$2ms$	$462\mu s$
hsqldb	271	$11m34s$	TO	$770ms$	TO	$4m39s$	$1m15s$	$8\mu s$	TO	$3m23s$	$100ms$
javac	21	$154ms$	$79ms$	$1ms$	$1\mu s$	$25ms$	$9ms$	$3\mu s$	$1\mu s$	$12ms$	$192\mu s$
JFlex	149	$7m41s$	TO	$110ms$	TO	$1m15s$	$4.36s$	$10\mu s$	TO	$52.2s$	$98ms$
jython	17	$62ms$	$50ms$	$723\mu s$	$1\mu s$	$16ms$	$47ms$	$4\mu s$	$1\mu s$	$6ms$	$1ms$
luindex	39	$1.23s$	$3.36s$	$7ms$	$1\mu s$	$518ms$	$672ms$	$3\mu s$	$1\mu s$	$171ms$	$12ms$
lusearch	91	$41.8s$	$8m49s$	$89ms$	$4\mu s$	$22.0s$	$10.3s$	$9\mu s$	$1\mu s$	$5.19s$	$52ms$
pmd	30	$458ms$	$454ms$	$3ms$	$1\mu s$	$134ms$	$421ms$	$3\mu s$	$1\mu s$	$72ms$	$6ms$
polyglot	186	$13m52s$	TO	$90ms$	TO	$1m31s$	$15.9s$	$10\mu s$	TO	$1m42s$	$128ms$
xalan	32	$1.23s$	$6.63s$	$10ms$	$1\mu s$	$575ms$	$667ms$	$4\mu s$	$1\mu s$	$215ms$	$10ms$

Table X: Comparison table for simple uninitialized variables analysis.

Benchmarks		**Possibly Uninitialized Variables**									
		Preprocessing		Query							
				Single Source				Pair			
Name	$\lvert D\rvert$	Ours	Compl	Ours	Compl	No Pre	OD	Ours	Compl	No Pre	OD
antlr	45	$2.26s$	$25.9s$	$16ms$	$2\mu s$	$1.61s$	$2.74s$	$3\mu s$	$1\mu s$	$218ms$	$18ms$
bloat	11	$17ms$	$8ms$	$370\mu s$	$1\mu s$	$3ms$	$1ms$	$5\mu s$	$1\mu s$	$1ms$	$38\mu s$
chart	146	$6m43s$	TO	$165ms$	TO	$1m59s$	$1m2s$	$11\mu s$	TO	$1m32s$	$53ms$
eclipse	30	$419ms$	$7.63s$	$3ms$	$1\mu s$	$349ms$	$528ms$	$5\mu s$	$1\mu s$	$63ms$	$16ms$
fop	11	$52ms$	$135ms$	$471\mu s$	$1\mu s$	$13ms$	$15ms$	$3\mu s$	$1\mu s$	$2ms$	$493\mu s$
hsqldb	271	$10m27s$	TO	$708ms$	TO	$4m34s$	$1m26s$	$12\mu s$	TO	$3m49s$	$103ms$
javac	21	$155ms$	$110ms$	$1ms$	$1\mu s$	$32ms$	$9ms$	$3\mu s$	$1\mu s$	$13ms$	$208\mu s$
JFlex	149	$10m42s$	TO	$76ms$	TO	$1m15s$	$4.84s$	$9\mu s$	TO	$52.7s$	$93ms$
jython	17	$63ms$	$98ms$	$718\mu s$	$1\mu s$	$21ms$	$61ms$	$3\mu s$	$1\mu s$	$7ms$	$1ms$
luindex	39	$1.42s$	$4.73s$	$9ms$	$1\mu s$	$564ms$	$833ms$	$3\mu s$	$1\mu s$	$175ms$	$16ms$
lusearch	91	$55.9s$	$44m6s$	$92ms$	$4\mu s$	$42.8s$	$8.87s$	$9\mu s$	$1\mu s$	$5.24s$	$60ms$
pmd	30	$506ms$	$879ms$	$3ms$	$1\mu s$	$157ms$	$395ms$	$3\mu s$	$1\mu s$	$72ms$	$8ms$
polyglot	186	$14m36s$	TO	$77ms$	TO	$1m33s$	$17.9s$	$10\mu s$	TO	$1m54s$	$132ms$
xalan	32	$1.86s$	$13.2s$	$9ms$	$1\mu s$	$704ms$	$591ms$	$6\mu s$	$1\mu s$	$331ms$	$18ms$

Table XI: Comparison table for possibly uninitialized variables analysis.



```
void f() {
  int a,b;
  b = a;
  a = 2;
}
```

1
$\lambda x.x$
2
$\lambda x.x \cup \{\bar{a}, \bar{b}\}$
3
$\lambda x.x - \{\bar{b}\}$
4
$\lambda x.x - \{\bar{a}\}$
5
1
$\lambda x.x$
2
$\lambda x.x \cup \{\bar{a}, \bar{b}\}$
3
$\lambda x. \begin{cases} x \cup \{\bar{b}\} & \bar{a} \in x \\ x - \{\bar{b}\} & \bar{a} \notin x \end{cases}$
4
$\lambda x.x - \{\bar{a}\}$
5

Fig. 10: An example program (left) and its uninitialized variables analyses in IFDS, including its simple uninitialized variables analysis (center) and its possibly-uninitialized variables analysis (right)

6.6. Experimental Conclusions

Our experiments with 6 IFDS-based dataflow analyses show that our new treewidth-based algorithms succeed in answering both single-source and pair queries efficiently, only after a lightweight preprocessing. In particular, in all cases that our preprocessing required more than a couple of minutes (ranging from about 6 minutes to 43 minutes), the complete preprocessing times out after 1 hour. In some cases, the complete preprocessing is at least 50 times slower than our preprocessing (e.g., in the unused variables analysis, the benchmark lusearch requires only 1 minute to be preprocessed by our algorithm, whereas the complete preprocessing requires 52 minutes). Since preprocessing times are typically large, this difference is noticeable. In the query phase, our algorithm requires more time than the complete preprocessing, as the latter only performs table lookups. However, the query times of our algorithm remain very small, and often the difference is negligible. In particular, our pair-queries are in the orders of microseconds, and thus appear to be within the time budget of a real-time analysis scenario (e.g., for just-in-time compilation).

The advantage of the offline and on-demand algorithms is that they perform no preprocessing. However, this comes at a significant cost in the query phase. The offline algorithm often requires several minutes per query, and although the on-demand algorithm is faster than the offline algorithm, it remains noticeably slower than our algorithm. We also note that the times we report are averages over 500 queries. Since the on-demand algorithm benefits only at the presence of multiple queries, we expect that its running time is significantly larger in the first queries. In fact, the average time for the first query of the on-demand algorithm coincides with the average time over all queries of the offline algorithm, as reported in our tables above, and is thus very slow.

As a final remark, we note that for benchmarks with more methods, we expect to have a larger preprocessing time, but not larger query time. This is because after the preprocessing phase, our algorithm treats the control-flow graph of each method independently. Hence, we expect that the running times reported here are robust with regards to variations in the number of methods. On the other hand, that this is not the case for the offline and on-demand algorithms, and hence benchmarks with more methods will lead to larger query times.

7. CONCLUSIONS

On-demand interprocedural static analysis has several advantages over offline analyses in various application domains, such as when performed as a user-level operation (e.g., during debugging) and when run by a just-in-time compiler performing speculative executions. The tasks of an on-demand static analyzer are naturally split between (i) a preprocessing phase, where the program is analyzed without knowledge of the precise analysis queries, and (ii) a query phase, where the analysis queries arrive in an online fashion (i.e., the analyzer is oblivious to future queries). Computationally, this yields a wide spectrum of the resources spent in each phase. The key technical challenge faced by the static analyzer is to achieve the best possible tradeoff in this spectrum: spend as few resources as possible in the preprocessing phase (in terms of running time and space usage) so that, afterwards, on-demand queries are answered fast.

In this work, we have taken an algorithmic approach to the challenge of on-demand interprocedural static analysis. A central part of our approach is the exploitation of a key structural property of control-flow graphs of typical programs, namely, the fact that they form the most representative family of constant-treewidth graphs. Given this property, we have developed algorithms for preprocessing and performing same-context queries, that offer strong complexity guarantees, and combine the best of the two endpoints in the preprocessing/querying spectrum: the preprocessing uses as much resources (time, space) as performing an offline analysis, and the querying uses as many resources as if we have had the complete preprocessing at our disposal. Besides the theoretical improvements, we have implemented our new, treewidth-based algorithms on a static analyzer and have evaluated their performance experimentally on a standard benchmark set, for various types of static analyses. Our results show that after a quick preprocessing, analysis queries are answered extremely fast, and hence the theoretical improvements are realized in practice.

Our formulation of the static analysis as an algebraic path problem implies that our results are not restricted to any particular static analysis, but are applicable to all analyses that admit an algebraic formulation (namely, distributive analyses). Our work leaves open one key challenge, namely, move from same-context queries to arbitrary queries where the endpoints belong to different functions of the program. For this problem, the techniques developed here might prove useful, e.g., by breaking such a distant query to multiple same-context queries. However, this treatment has no benefit with regards to the worst-case complexity, and the problem merits closer attention. It is also possible that treewidth alone is not sufficient to lead to algorithmic improvements, and further natural restrictions must be considered, for example, regarding the structure of the call graph.

References

Noga Alon and Baruch Schieber. 1987. *Optimal preprocessing for answering on-line product queries*. Technical Report. Tel Aviv University.

R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. W. Reps, and M. Yannakakis. 2005. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.* (2005).

R. Alur, S. La Torre, and P. Madhusudan. 2006. Modular strategies for recursive game graphs. *Theor. Comput. Sci.* (2006).

Andrew W Appel and Jens Palsberg. 2002. *Modern compiler implementation in Java*. Cambridge university press.

Stefan Arnborg and Andrzej Proskurowski. 1989. Linear time algorithms for NP-hard problems restricted to partial k-trees . *Discrete Appl Math* (1989).

Wayne A. Babich and Mehdi Jazayeri. 1978. The method of attributes for data flow analysis. *Acta Informatica* 10, 3 (1978).

Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. 2010. SPUR: A Trace-based JIT Compiler for CIL. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 708–725. DOI:https://doi.org/10.1145/1869459.1869517

Michael A. Bender and Martín Farach-Colton. 2000. The LCA Problem Revisited. In *LATIN 2000: Theoretical Informatics*. Springer Berlin Heidelberg.

M.W Bern, E.L Lawler, and A.L Wong. 1987. Linear-time computation of optimal subgraphs of decomposable graphs. *J Algorithm* (1987).

Stephen M. et al. Blackburn. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*.

Eric Bodden. 2012. Inter-procedural Data-flow Analysis with IFDS/IDE and Soot. In *SOAP*. ACM, New York, NY, USA.

HansL. Bodlaender. 1994. Dynamic algorithms for graphs with treewidth 2. In *Graph-Theoretic Concepts in Computer Science*. Springer.

HansL. Bodlaender. 2005. Discovering Treewidth. In *SOFSEM 2005: Theory and Practice of Computer Science*. LNCS, Vol. 3381. Springer.

HansL. Bodlaender and Torben Hagerup. 1995. Parallel algorithms with optimal speedup for bounded treewidth. Vol. 27. 1725–1746.

Hans L. Bodlaender. 1988. Dynamic programming on graphs with bounded treewidth. In *ICALP*. Springer.

Hans L. Bodlaender. 1993. A Tourist Guide through Treewidth. *Acta Cybern.* (1993).

Hans L. Bodlaender. 1996. A Linear-Time Algorithm for Finding Tree-Decompositions of Small Treewidth. *SIAM J. Comput.* (1996).

Hans L. Bodlaender. 1998. A partial k-arboretum of graphs with bounded treewidth. *TCS* (1998).

Bernd Burgstaller, Johann Blieberger, and Bernhard Scholz. 2004. On the Tree Width of Ada Programs. In *Reliable Software Technologies - Ada-Europe 2004*, Albert Llamosí and Alfred Strohmeier (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 78–90.

David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. 1986. Interprocedural Constant Propagation. In *CC*. ACM.

Krishnendu Chatterjee, Amir Goharshady, and Ehsan Goharshady. 2019. The Treewidth of Smart Contracts. In *SAC*.

Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Andreas Pavlogiannis. 2017. JTDec: A Tool for Tree Decompositions in Soot. In *Automated Technology for Verification and Analysis*, Deepak D'Souza and K. Narayan Kumar (Eds.). Springer International Publishing, Cham, 59–66.

Krishnendu Chatterjee, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. 2016. Optimal Reachability and a Space-Time Tradeoff for Distance Queries in Constant-Treewidth Graphs. In *24th Annual European Symposium on Algorithms, ESA 2016, August 22-24, 2016, Aarhus, Denmark*. 28:1–28:17.

Krishnendu Chatterjee, Rasmus Ibsen-Jensen, Andreas Pavlogiannis, and Prateesh Goyal. 2015a. Faster Algorithms for Algebraic Path Properties in Recursive State Machines with Constant Treewidth. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 97–109.

K. Chatterjee and J. Lacki. 2013. Faster Algorithms for Markov Decision Processes with Low Treewidth. In *CAV*.

Krishnendu Chatterjee, Andreas Pavlogiannis, and Yaron Velner. 2015b. Quantitative Interprocedural Analysis. In *POPL*.

Krishnendu Chatterjee and Yaron Velner. 2012. Mean-Payoff Pushdown Games. In *LICS*.

Swarat Chaudhuri. 2008. Subcubic Algorithms for Recursive State Machines. In *POPL*. ACM, New York, NY, USA.

Shiva Chaudhuri and Christos D. Zaroliagis. 1995. Shortest Paths in Digraphs of Small Treewidth. Part I: Sequential Algorithms. *Algorithmica* (1995).

Tong Chen, Jin Lin, Xiaoru Dai, Wei-Chung Hsu, and Pen-Chung Yew. 2004. Data Dependence Profiling for Speculative Optimizations. In *Compiler Construction*, Evelyn Duesterwald (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 57–72.

Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. 2018. *Handbook of Model Checking* (1st ed.). Springer Publishing Company, Incorporated.

Keith Cooper and Linda Torczon. 2011. *Engineering a compiler*. Elsevier.

T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. 2001. *Introduction To Algorithms*. MIT Press.

Brouno Courcelle. 1990. Graph Rewriting: An Algebraic and Logic Approach. In *Handbook of Theoretical Computer Science (Vol. B)*. MIT Press, Cambridge, MA, USA.

P. Cousot and R Cousot. 1977. Static determination of dynamic properties of recursive procedures. In *IFIP Conf. on Formal Description of Programming Concepts*, E.J. Neuhold (Ed.).

Edsger. W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numer. Math.* (1959).

Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. 1995. Demand-driven Computation of Interprocedural Data Flow *(POPL)*.

M. Elberfeld, A. Jakoby, and T. Tantau. 2010. Logspace Versions of the Theorems of Bodlaender and Courcelle. In *FOCS*.

Olivier Flückiger, Gabriel Scherer, Ming-Ho Yee, Aviral Goel, Amal Ahmed, and Jan Vitek. 2017. Correctness of Speculative Optimizations with Dynamic Deoptimization. *Proc. ACM Program. Lang.* 2, POPL, Article 49 (Dec. 2017), 28 pages. DOI:https://doi.org/10.1145/3158137

Robert Giegerich, Ulrich Möncke, and Reinhard Wilhelm. 1981. Invariance of Approximate Semantics with Respect to Program Transformations. In *3rd Conference of the European Co-operation in Informatics (ECI)*.

Dan Grove and Linda Torczon. 1993. Interprocedural Constant Propagation: A Study of Jump Function Implementation. In *PLDI*. ACM.

Jens Gustedt, OleA. Mæhle, and JanArne Telle. 2002. The Treewidth of Java Programs. In *Algorithm Engineering and Experiments*. Springer.

Torben Hagerup. 2000. Dynamic algorithms for graphs of bounded treewidth. *Algorithmica* (2000).

Rudolf Halin. 1976. S-functions for graphs. *Journal of Geometry* (1976).

D. Harel and R. Tarjan. 1984. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM J. Comput.* (1984).

Susan Horwitz, Thomas Reps, and Mooly Sagiv. 1995. Demand Interprocedural Dataflow Analysis. *SIGSOFT Softw. Eng. Notes* (1995).

Philipp Klaus Krause, Lukas Larisch, and Felix Salfelder. 2019. The tree-width of C. *Discrete Applied Mathematics* (03 2019). DOI:https://doi.org/10.1016/j.dam.2019.01.027

Jens Knoop and Bernhard Steffen. 1992. The Interprocedural Coincidence Theorem. In *CC*.

Jens Knoop, Bernhard Steffen, and Jürgen Vollmer. 1996. Parallelism for Free: Efficient and Optimal Bitvector Analyses for Parallel Programs. *ACM Trans. Program. Lang. Syst.* (1996).

J. Lacki. 2013. Improved Deterministic Algorithms for Decremental Reachability and Strongly Connected Components. *ACM Transactions on Algorithms* (2013).

William Landi and Barbara G. Ryder. 1991. Pointer-induced Aliasing: A Problem Classification. In *POPL*. ACM.

Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew, Roy Dz-Ching Ju, Tin-Fook Ngai, and Sun Chan. 2004. A Compiler Framework for Speculative Optimizations. *ACM Trans. Archit. Code Optim.* 1, 3 (Sept. 2004), 247–271. DOI:https://doi.org/10.1145/1022969.1022970

Nomair A. Naeem and Ondrej Lhoták. 2008. Typestate-like analysis of multiple interacting objects. In *OOPSLA*.

Nomair A. Naeem, Ondřej Lhoták, and Jonathan Rodriguez. 2010. Practical Extensions to the IFDS Algorithm *(CC)*.

Flemming Nielson, Hanne R Nielson, and Chris Hankin. 2015. *Principles of program analysis*. Springer.

Jan Obdrzálek. 2003. Fast Mu-Calculus Model Checking when Tree-Width Is Bounded. In *CAV*.

Bruce A. Reed. 1992. Finding Approximate Separators and Computing Tree Width Quickly. In *STOC*.

ThomasW. Reps. 1995. Demand Interprocedural Program Analysis Using Logic Databases. In *Applications of Logic Databases*. Vol. 296.

Thomas Reps. 1997. Program Analysis via Graph Reachability *(ILPS)*.

Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*.

Thomas Reps, Akash Lal, and Nick Kidd. 2007. Program Analysis Using Weighted Pushdown Systems. In *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*.

Thomas Reps, Stefan Schwoon, Somesh Jha, and David Melski. 2005. Weighted Pushdown Systems and Their Application to Interprocedural Dataflow Analysis. *Sci. Comput. Program.* (2005), 58.

JetBrains ReSharper. 2019. Code Inspection: Unused local variable. ReSharper 2019.1 Help. (2019). https://www.jetbrains.com/help/resharper/UnusedVariable.html

Neil Robertson and P.D Seymour. 1984. Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B* (1984).

Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theor. Comput. Sci.* (1996).

Stefan Schwoon. 2002. *Model-Checking Pushdown Systems*. Ph.D. Dissertation. Technischen Universität München.

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven Points-to Analysis for Java. In *OOPSLA*.

Mikkel Thorup. 1998. All Structured Programs Have Small Tree Width and Good Register Allocation. *Information and Computation* (1998).

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *CASCON '99*. IBM Press.

Genevieve Warren, Gordon Hogenson, Mike Jones, and Saisang Cai. 2016. CA1804: Remove unused locals. (2016). https://docs.microsoft.com/en-us/visualstudio/code-quality/ca1804-remove-unused-locals?view=vs-2019

Xin Yuan, Rajiv Gupta, and Rami Melhem. 1997. Demand-Driven Data Flow Analysis for Communication Optimization. *Parallel Processing Letters* 07, 04 (1997), 359–370.

Frank Kenneth Zadeck. 1984. Incremental Data Flow Analysis in a Structured Program Editor *(SIGPLAN)*.

Xin Zhang, Ravi Mangal, Mayur Naik, and Hongseok Yang. 2014. Hybrid top-down and bottom-up interprocedural analysis. In *PLDI*.