Faster Algorithms for Algebraic Path Properties in Recursive State Machines with Constant Treewidth *

Krishnendu Chatterjee
Rasmus Ibsen-Jensen
Andreas Pavlogiannis
IST Austria (Institute of Science and Technology Austria) Klosterneuburg, Austria
{krish.chat, ribsen, pavlogiannis}@ist.ac.at

Prateesh Goyal
IIT Bombay (Indian Institute of Technology Bombay) Mumbai, India
prateesh@iitb.ac.in

Abstract

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 a very 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 best-known solutions for several important problems, such as interprocedural reachability and shortest path. We provide a prototype implementation for interprocedural reachability and intraprocedural shortest path that gives a significant speed-up on several benchmarks.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program Analysis

* This work has been supported by the Austrian Science Foundation (FWF) under the NFN RiSE (S11405), FWF Grant P23499-N23, ERC Start grant (279307: Graph Games), and Microsoft faculty fellows award.

POPL '15, January 15–17, 2015, Mumbai, India.

ACM 978-1-4503-3300-9/15/01…$15.00.
http://dx.doi.org/10.1145/2676726.2676979

Keywords Interprocedural analysis, Constant treewidth graphs, Dataflow analysis, Reachability and shortest path.

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 [11, 14, 19, 22, 23, 29, 30, 32, 33, 36, 40, 45]. In seminal works [36, 40] 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)* [2] (aka *supergraph* in [36]). 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.

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 partially 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 [36, 40] 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.

Two important aspects. In the traditional algorithms for interprocedural analysis, the starting point is typically *fixed* as the entry point of a specific method. In graph theoretic parlance, graph algorithms can consider two types of queries: (i) a *pair query* that given nodes u and v (called (u, v)-pair query) asks for the 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. Thus the

traditional algorithms for interprocedural analysis has focused on the answer for *one* single-source query. Moreover, the existing algorithms also consider that the input control flow graph is arbitrary, and do not exploit the fact that most control flow graphs satisfy several elegant structural properties. In this work, we consider two new aspects, namely, (i) *multiple* pair and single-source queries, and (ii) exploit the fact that typically the control flow graphs of programs satisfy an important structural property called the *constant treewidth property*. We describe in details the two aspects.

- *Multiple queries.* We first describe the relevance of pair and multiple pair queries, and then the significance of even multiple single-source queries. In alias analysis, the question is whether two pointers may point to the same object, which is by definition modeled as a question between a pair of nodes. Similarly, e.g., in constant propagation, given a function call, a relevant question is whether some variable remains constant within the entry and exit of the function (in general it can be between a pair of nodes of the program). This shows that the pair query problem, and the multiple pair queries are relevant in many applications. Finally, consider a run-time optimization scenario, where the goal is to decide whether a variable remains constant from *now on*, and this corresponds to a single-source query, where the starting point is the current execution point of the program. Thus multiple pair queries and multiple single-source queries are relevant for several important static analysis problems.
- *Constant treewidth.* A very well-known concept in graph theory is the notion of *treewidth* of a graph, which is a measure of how similar a graph is to a tree (a graph has treewidth 1 precisely if it is a tree) [39]. The treewidth of a graph is defined based on a *tree decomposition* of the graph [26], 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 control flow graph for goto-free programs for many programming languages are of constant treewidth [42], and it was also shown in [24] that typically all Java programs have constant treewidth. An important property of constant-treewidth graphs is that the number of edges is at most a constant factor larger than the number of nodes. This has been considered in the comparison Tables 2 and 3.

Our contributions. In this work 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 [16] for the significance of same-context queries). In the analysis of multiple queries, there is a very important algorithmic distinction between *one-time* preprocessing (denoted as the preprocessing time), and the work done for each individual query (denoted as the query time). There are two endpoints in the spectrum of tradeoff between the preprocessing and query resources that can be obtained by using the classical algorithms for one single-source query, namely, (i) the *complete preprocessing*, and (ii) the *no preprocessing*. In complete preprocessing, the single-source answer is precomputed with every node as the starting point (for example, in graph reachability this corresponds to computing the all-pairs reachability problem with the classical BFS/DFS algorithm [17], or with fast matrix multiplication [21]). In no preprocessing, there is no preprocessing done, and the algorithm for one single-source query is used on demand for each individual query. We consider various other possible tradeoffs in preprocessing vs query time. 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 height of the stack is bounded by a parameter h. While in general for arbitrary semirings there does not exist a bound on the stack height, if the semiring contains subsets of a finite universe D, and the semiring plus operator is intersection or union, then solving the problem with sufficiently large bound on the stack height is equivalent to solving the problem without any restriction on stack height. Our main result is an algorithm where the one-time preprocessing phase requires $O(n \cdot \log n + h \cdot b \cdot \log n)$ semiring operations, and then each subsequent bounded stack height pair query can be answered in constant number of semiring operations, where n is the number of nodes of the RSM and b the number of boxes (see Table 1 and Theorem 3). 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 2 (Corollary 1). For example, our approach with a factor of $O(\log n)$ overhead for one-time preprocessing, as compared no preprocessing, can answer subsequent pair queries by a factor of $O(n \cdot |D|)$ faster. An important feature of our algorithms is that they are simple and implementable.
2. *(Reachability and shortest path).* We now discuss the significance of our result for the very important special cases of reachability and shortest path.
   - *(Reachability).* The result for reachability with full preprocessing, no preprocessing, and the various tradeoff that can be obtained by our approach is obtained from Table 2 by $|D| = 1$. For example for pair queries, full preprocessing requires quadratic time and space (for all-pairs reachability computation) and answers individual queries in constant time; no preprocessing requires linear time and space for individual queries; whereas with our approach (i) with almost-linear ($O(n \cdot \log n)$) preprocessing time and space we can answer individual queries in constant time, which is a significant (from quadratic to almost-linear) improvement over full preprocessing; or (ii) with linear space and almost-linear preprocessing time we can answer queries in logarithmic time, which is a huge (from linear to logarithmic) improvement over no preprocessing. For example, if we consider $O(n)$ pair queries, then both full preprocessing and no preprocessing in total require quadratic time, whereas our approach in total requires $O(n \cdot \log n + n \cdot \log n) = O(n \cdot \log n)$ time.
   - *(Shortest path).* We now consider the problem of shortest path, where the current best-known algorithm is for pushdown graphs [37, 38] and we are not aware of any better bounds for RSMs (that have unique entries and exits). The algorithm of [37] is a polynomial-time algorithm of degree four, and the full preprocessing requires $O(n^5)$ time and quadratic space, and can answer single-source (resp. pair) queries in linear (resp. constant time); whereas the no preprocessing requires $O(n^4)$ time and linear space for both single-source and pair queries. In contrast, we show that (i) with almost-quadratic ($O(n^2 \cdot \log n)$) preprocessing time and almost-linear space, we can answer single-source (resp. pair) queries in linear (resp. constant) time; or (i) with almost-quadratic preprocessing and linear space, we can answer single-source (resp. pair) queries in linear (resp. logarithmic) time. Thus our approach provides a significant theoretical improvement over the existing approaches.

   There are two facts that are responsible for our improvement, the first is that we consider that each CSM of the RSM has constant treewidth, and the second is the tradeoff of one-time preprocessing and individual queries. Also note that our results apply only to same-context queries.
3. *(Experimental results).* Besides the theoretical improvements, we demonstrate the effectiveness of our approach on several well-known benchmarks from programming languages. We use the tool for computing tree decompositions from [44], and all benchmarks of our experimental results have small treewidth. We have implemented our algorithms for reachabil-

	Preprocessing time	Space	Single source query	Pair query	Reference
Our	$O(\log n \cdot (n + h \cdot b))$	$O(n \cdot \log n)$	$O(n)$	$O(1)$	Theorem 3
Results	$O(\log n \cdot (n + h \cdot b))$	$O(n)$	$O(\log n)$	$O(n)$	Theorem 3

Table 1: Interprocedural same-context algebraic path problem on RSMs with b boxes and constant treewidth, for stack height h.

	Preprocessing time	Space	Single source query	Pair query	Reference
IDE/IFDS (complete preprocessing)	$O(n^2 \cdot \lvert D\rvert^3)$	$O(n^2 \cdot \lvert D\rvert)$	$O(\lvert D\rvert)$	$O(n \cdot \lvert D\rvert)$	[36, 40]
IDE/IFDS (no preprocessing)	-	$O(n \cdot \lvert D\rvert)$	$O(n \cdot \lvert D\rvert^3)$	$O(n \cdot \lvert D\rvert^3)$	[36, 40]
Our	$O(\lvert D\rvert^2 \cdot \log n \cdot (n + b \cdot \lvert D\rvert))$	$O(n \cdot \log n \cdot \lvert D\rvert^2)$	$O(n \cdot \lvert D\rvert^2)$	$O(\lvert D\rvert^2)$	Corollary 1
Results	$O(n \cdot \lvert D\rvert^2 + \log n \cdot (b \cdot \lvert D\rvert^3 + n))$	$O(n \cdot \lvert D\rvert^2)$	$O(n \cdot \lvert D\rvert^2)$	$O(\log n \cdot \lvert D\rvert^2)$	Corollary 1

Table 2: Interprocedural same-context algebraic path problem 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. The special case of reachability is obtained when $|D| = 1$.

	Preprocessing time	Space	Single-source query	Pair query	Reference
GPR (complete preprocessing)	$O(n^5)$	$O(n^2)$	$O(n)$	$O(1)$	[37, 38]
GPR (no preprocessing)	-	$O(n)$	$O(n^4)$	$O(n^4)$	[37, 38]
Our	$O(n^2 \cdot \log n)$	$O(n \cdot \log n)$	$O(n)$	$O(1)$	Corollary 2
Results	$O(n^2 \cdot \log n)$	$O(n)$	$O(n)$	$O(\log n)$	Corollary 2

Table 3: Interprocedural same-context shortest path for RSMs with constant treewidth.

ity (both intraprocedural and interprocedural) and shortest paths (only intraprocedural), and compare their performance against complete and no preprocessing approaches for same-context queries. Our experimental results show that our approach obtains a significant improvement over the existing approaches (of complete and no preprocessing).

Technical contribution. Our main technical contribution is a dynamic algorithm (also referred to as incremental algorithm in graph algorithm literature) 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 2.

Nice byproduct. Several previous works such as [27] have stated the importance and asked for the development of data structures and analysis techniques to support dynamic updates. Though our main results are for the problem where the RSM is given and fixed, our main technical contribution is a dynamic algorithm that can also be used in other applications to support dynamic updates, and is thus also of independent interest.

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 [11, 19, 22, 23, 29, 30, 32, 36, 40]. Our work is most closely related to the IFDS/IDE frameworks introduced in seminal works [36, 40]. 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. For example, the shortest path problem cannot be expressed in the IFDS/IDE framework [37], but can be expressed in the GPR framework [37, 38]. The GPR framework considers the more general problem of weighted pushdown graphs, whereas we show that with the restriction to constant treewidth RSMs the bounds for the best-known algorithm can be significantly improved. Finally, several works such as [27] 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 [2]. 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 [3, 13], and subcubic algorithm for general RSMs with reachability has been shown in [15]. 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 [36, 40].

Treewidth of graphs. The notion of treewidth for graphs as an elegant mathematical tool to analyze graphs was introduced in [39]. 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 [4, 5, 8–10]. 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 [16]. 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 [18] showed that for constant-treewidth graphs the decision questions for MSO can be solved in polynomial time; and the result of [20] 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 [7] and extended to various tradeoffs by [25]; and [31] 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 [12, 34]. The problem of computing a balanced tree decomposition for a constant treewidth graph was considered in [35], and we use this algorithm in our preprocessing phase. More importantly, in the context of programming languages, it was shown by [42] that the control flow graph for goto-free programs for many programming languages have constant treewidth. This theoretical result was subsequently followed up in several practical approaches, and it was shown in [24] that though in theory Java programs might not have constant treewidth, in practice Java programs do have constant treewidth. We also use the existing tree-decomposition tool developed by [44] in our experimental results.

2. Definitions

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

2.1 Semirings

Definition 1 (Semirings)**.** We consider partially complete *semirings* $(\Sigma, \oplus, \otimes, \overline{\mathbf{0}}, \overline{\mathbf{1}})$ 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 associative, commutative, and $\overline{\mathbf{0}}$ is the neutral element,
2. $\otimes$ is associative, and $\overline{\mathbf{1}}$ is the neutral element,
3. $\otimes$ distributes over $\oplus$,
4. $\oplus$ is infinitely associative,
5. $\otimes$ infinitely distributes over $\oplus$,
6. $\overline{\mathbf{0}}$ absorbs in multiplication, i.e., $\forall a \in \Sigma : a \otimes \overline{\mathbf{0}} = \overline{\mathbf{0}}$.

Additionally, we consider that semirings are equipped with a *closure* operator *, such that $\forall s \in \Sigma : \; s^* = \overline{\mathbf{1}} \oplus (s \otimes s^*) = \overline{\mathbf{1}} \oplus (s^* \otimes s)$.

2.2 Graphs and tree decomposition

Definition 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 edges $(e_1, \ldots, e_k)$ and each $e_i = (x_i, y_i)$ is such that $x_1 = u$, $y_k = v$, and for all $1 \leq i \leq k-1$ we have $y_i = x_{i+1}$. 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 $P = (e_1, \ldots, e_k)$, the weight of P is $\otimes(P) = \bigotimes(\mathsf{wt}(e_1), \ldots, \mathsf{wt}(e_k))$ 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 3 (Tree decomposition and treewidth [10, 39])**.** Given a graph $G = (V, E)$, a *tree-decomposition* $\mathrm{Tree}(G) = (V_T, E_T)$ is a tree such that the following conditions hold:

1. $V_T = \{B_0, \ldots, B_{n'-1} : \text{for all } 0 \leq i \leq n'-1, B_i \subseteq V\}$ and $\bigcup_{B_i \in V_T} B_i = V$.
2. For all $(u, v) \in E$ there exists $B_i \in V_T$ such that $u, v \in B_i$.
3. For all i, j, k such that there exist paths $B_i \rightsquigarrow B_k$ and $B_k \rightsquigarrow B_j$ in $\mathrm{Tree}(G)$, we have $B_i \cap B_j \subseteq B_k$.

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

Example 1 (Graph and tree decomposition)**.** The treewidth of a graph G is an intuitive measure which represents the proximity of G to a tree, though G itself not a tree. The treewidth of G is 1 precisely if G is itself a tree [39]. Consider an example graph and its tree decomposition shown in Figure 1. It is straightforward to verify that all the three conditions of tree decomposition are met. Each node in the tree is a bag, and labeled by the set of nodes it contains. Since each bag contains at most three nodes, the tree decomposition by definition has treewidth 2.

Intuitive meaning of tree decomposition. In words, the tree-decomposition $\mathrm{Tree}(G)$ is a tree where every node (bag) is subset of nodes of G, such that: (1) every vertex in G belongs to some bag; (2) every edge in G also belongs to some bag; and (3) for every node v of G, for every subpath in $\mathrm{Tree}(G)$, if v appears in the endpoints of the path, then it must appear all along the path.

Separator property. Given a graph G and its tree decomposition $\mathrm{Tree}(G)$, note that for each bag B in $\mathrm{Tree}(G)$, if we remove the set of nodes in the bag, then the graph splits into possibly multiple components (i.e., each bag is a separator for the graph). In other words, every bag acts as a *separator* of the graph.

Notations for tree decomposition. Let G be a graph, $T = \mathrm{Tree}(G)$, and B_0 be the root of T. Denote with $\mathsf{Lv}(B_i)$ the depth of B_i in T, with $\mathsf{Lv}(B_0) = 0$. For $u \in V$, we say that a bag B *introduces* u if B is the bag with the smallest level among all bags that contain u, i.e., $B_u = \arg\min_{B \in V_T :\, u \in B} \mathsf{Lv}(B)$. By definition, there is exactly one bag introducing each node u. We often write B_u for the bag that introduces the node u, and denote with $\mathsf{Lv}(u) = \mathsf{Lv}(B_u)$. Finally, we denote with $B_{(u,v)}$ the bag of the highest level that introduces one of u, v. A tree-decomposition $\mathrm{Tree}(G)$ is *semi-nice* if $\mathrm{Tree}(G)$ is a binary tree, and every bag introduces at most one node.

Example 2. In the example of Figure 1, the bag $\{2, 8, 10\}$ is the root of $\mathrm{Tree}(G)$, the level of node 9 is $\mathsf{Lv}(9) = \mathsf{Lv}(\{8, 9, 10\}) = 1$, and the bag of the edge $(9, 1)$ is $B_{(9,1)} = \{1, 8, 9\}$.

Theorem 1. *(1) For every graph there exists a semi-nice tree decomposition that achieves the treewidth of G and uses $n' = O(n)$ bags [28]. (2) For constant treewidth graphs, a balanced tree decomposition can be obtained in $O(n \cdot \log n)$ time (i.e., every simple path $B_0 \rightsquigarrow B_i$ in $\mathrm{Tree}(G)$ has length $O(\log n)$) [35].*

The algebraic path problem on graphs of constant treewidth. Given $G = (V, E)$, a balanced, semi-nice tree-decomposition $\mathrm{Tree}(G)$ of G with constant treewidth $t = O(1)$, a partially complete semiring $(\Sigma, \oplus, \otimes, \overline{\mathbf{0}}, \overline{\mathbf{1}})$, a weight function $\mathsf{wt} : E \to \Sigma$, the algebraic path 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.

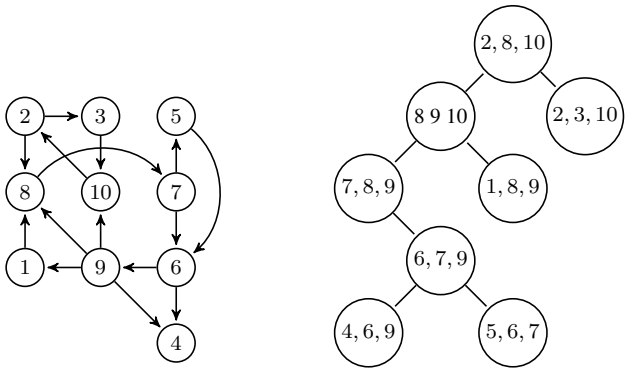

Figure 1: A graph G with treewidth 2 (left) and a corresponding tree-decomposition $\mathrm{Tree}(G)$ (right).

2.3 Recursive state machines

Definition 4 (RSMs and CSMs)**.** A *single-entry single-exit recursive state machine* (RSM from now on) over an alphabet Σ, as defined in [2], consists of a set $\{A_1, A_2, \ldots, A_k\}$, such that for each $1 \leq i \leq k$, the *component state machine* (CSM) $A_i = (B_i, Y_i, V_i, E_i, \mathtt{wt}_i)$, 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 $\mathtt{wt}_i$, mapping each edge in E_i to a label in Σ.

Definition 5 (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 3 (RSM and tree decomposition)**.** Figure 2 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 $\mathtt{wt}$ maps each edge in E to a label in Σ. We have that there is an edge between configuration $c_1 = (v_1, L_1)$, where $v_1 \in V_i$, and configuration $c_2 = (v_2, L_2)$ with label $\sigma = \mathtt{wt}(c_1, c_2)$ if and only if one of the following holds:

- **Internal edge:** We have that 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) $\sigma = \mathtt{wt}_i((v_1, v_2))$.
- **Entry edge:** We have that 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) $\sigma = \mathtt{wt}_i((v_1, (v_2, b)))$.
- **Return edge:** We have that $v_2 = (v, b)$ 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) $\sigma = \mathtt{wt}_i((v_1, v))$.

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 and labels $P = \langle c_1, \sigma_1, c_2, \sigma_2 \ldots, \sigma_{\ell-1}, c_\ell \rangle$, such that for each integer i where $1 \leq i \leq \ell - 1$, we have that $(c_i, c_{i+1}) \in E$ and $\sigma_i = \mathtt{wt}(c_i, c_{i+1})$. We call ℓ the *length* of P. Also, we say that the stack height of a execution path is the maximum stack height of a configuration in the execution path. For a pair of configurations c, c', the set $c \rightsquigarrow c'$, is the set of execution paths $\langle c_1, \sigma_1, c_2, \sigma_2 \ldots, \sigma_{\ell-1}, c_\ell \rangle$, for any ℓ, where $c = c_1$ and $c' = c_\ell$. For a set S of execution paths, the set $B(S, h) \subseteq S$ is the subset of execution paths, with stack height at most h. Given a partially complete semiring $(\Sigma, \oplus, \otimes, \overline{\mathbf{0}}, \overline{\mathbf{1}})$, the *distance* of a execution path $P = \langle c_1, \sigma_1, c_2, \sigma_2 \ldots, \sigma_{\ell-1}, c_\ell \rangle$ is $\otimes(P) = \bigotimes(\sigma_1, \ldots, \sigma_{\ell-1})$ (the empty product is $\overline{\mathbf{1}}$). Given configurations c, c', the *configuration distance* $d(c, c')$ is defined as $d(c, c') = \bigoplus_{P: c \rightsquigarrow c'} \otimes(P)$ (the empty sum is $\overline{\mathbf{0}}$). Also, given configurations c, c' and a stack height h, where c' is h-reachable from c, the *bounded height configuration distance* $d(c, c', h)$ is defined as $d(c, c', h) = \bigoplus_{P: B(c \rightsquigarrow c', h)} \otimes(P)$. Note that the above definition of execution paths only allows for so called *valid* paths [36, 40], i.e., paths that fully respect the calling contexts of an execution.

The algebraic path problem on RSMs of constant tree-width. Given (i) a RSM $A = \{A_1, A_2, \ldots, A_k\}$; and (ii) for each $1 \leq i \leq k$ a balanced, semi-nice tree-decomposition $\mathrm{Tree}(A_i) := \mathrm{Tree}((V_i, E_i'))$ with constant treewidth at most $t = O(1)$; and (iii) a partially 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 1. 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* [15] and *same-level* [36] realizable paths

and has several applications in program analysis, e.g. by capturing balanced parenthesis-like properties used in alias analysis [41].

2.4 Problems

We note that a wide range of interprocedural problems can be formulated as bounded height algebraic path problems.

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 [36]. 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 [40]. 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-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., shortest path with negative interprocedural cycles, or probability of reaching a node v from a node u).

3. Dynamic Algorithms for Preprocess, Update and Query

In the current section we present algorithms that take as input a constant treewidth graph G and a balanced, semi-nice tree-decomposition $\mathrm{Tree}(G)$ (recall Theorem 1), and achieve the following tasks:

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

In the following section we use the results of this section in order to preprocess RSMs fast, with the purpose of answering interprocedural same-context algebraic path queries fast. Refer to Example 4 of Section 4 for an illustration on how these algorithms are executed on an RSM.

First we establish the following lemma which captures the main intuition behind tree decompositions, namely, that bags B of the tree-decomposition $\mathrm{Tree}(G)$ are separators between nodes of G that belong to disconnected components of $\mathrm{Tree}(G)$ once B is removed.

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

Proof. Fix a number $i \in \{1, \ldots, j-1\}$. We argue that for each path $P : u \rightsquigarrow v$, there exists a node $x_i \in (B_i \cap B_{i+1} \cap P)$. We construct a tree $\mathrm{Tree}'(G)$, which is similar to $\mathrm{Tree}(G)$ except that instead of having an edge between bag B_i and bag B_{i+1}, there is a new bag B, that contains the nodes in $B_i \cap B_{i+1}$, and there is an edge between B_i and B and one between B and B_{i+1}. It is easy to see that $\mathrm{Tree}'(G)$ forms a tree decomposition of G. Let $\mathcal{C}_1$, $\mathcal{C}_2$ be the two components of $\mathrm{Tree}(G)$ separated be B, and w.l.o.g. $u \in \mathcal{C}_1$ and $v \in \mathcal{C}_2$. It follows by the definition of tree decomposition that B is a separator of $\bigcup_{B' \in \mathcal{C}_1} B'$ and $\bigcup_{B' \in \mathcal{C}_2} B'$. Hence, each path $u \rightsquigarrow v$ must go through some node x_i in B, and by construction $x_i \in B_i \cap B_{i+1}$. □

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

1. Either $|P| > 1$ and for all intermediate nodes $w \in P$, we have $\mathsf{Lv}(w) \geq \mathsf{Lv}(B)$,
2. or $|P| \leq 1$ and B is B_u or B_v.

Informally, given a bag B, a U-shaped path in B is a path that traverses intermediate nodes that are introduced in B and its descendants in $\mathrm{Tree}(G)$. In the following we present three algorithms for (i) preprocessing a tree decomposition, (ii) updating the data structures of the preprocessing upon a weight change $\mathsf{wt}(u, v)$ of an edge (u, v), and (iii) querying for the distance $d(u, v)$ for any pair of nodes u, v. The intuition behind the overall approach is that for every path $P : u \rightsquigarrow v$ and $z = \mathsf{argmin}_{x \in P} \mathsf{Lv}(x)$, the path P can be decomposed to paths $P_1 : u \rightsquigarrow z$ and $P_2 : z \rightsquigarrow v$. By Lemma 1, if we consider the path $P' : B_u \rightsquigarrow B_z$ and any bag $B_i \in P'$, we can find nodes $x, y \in B_i \cap P_1$ (not necessarily distinct). Then P_1 is decomposed to a sequence of U-shaped paths P_1^i, one for each such B_i, and the weight of P_1 can be written as the $\otimes$-product of the weights of P_1^i, i.e., $\otimes(P_1) = \bigotimes(\otimes(P_1^i))$. Similar observation holds for P_2. Hence, the task of preprocessing and updating is to summarize in each B_i the weights of all such U-shaped paths between all pairs of nodes appearing in B_i. To answer the query, the algorithm traverses upwards the tree $\mathrm{Tree}(G)$ from B_u and B_v, and combines the summarized paths to obtain the weights of all such paths P_1 and P_2, and eventually P, such that $\otimes(P) = d(u, v)$.

Informal description of preprocessing. Algorithm Preprocess associates with each bag B a *local distance* map $\mathsf{LD}_B : B \times B \to \Sigma$. Upon a weight change, algorithm Update updates the local distance map of some bags. It will hold that after the preprocessing and each subsequent update, $\mathsf{LD}_B(u, v) = \bigoplus_{P : u \rightsquigarrow v} \{\otimes(P)\}$, 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 Preprocess is a dynamic programming algorithm. It traverses $\mathrm{Tree}(G)$ bottom-up, and for a currently examined bag B introducing a node x, it calls the method

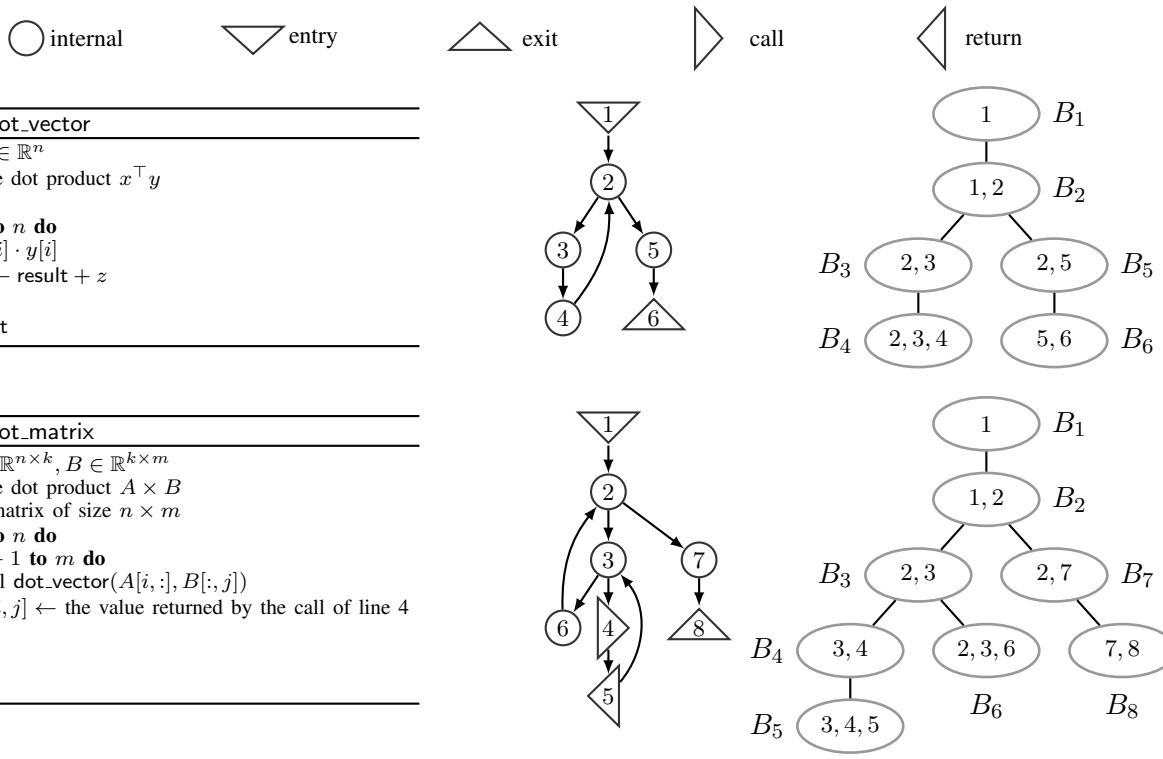

Figure 2: Example of a program consisting of two methods, their control flow graphs $G_i = (V_i, E_i')$ where nodes correspond to line numbers, and the corresponding tree decompositions, each one achieving treewidth 2.

Merge to compute the local distance map LD_B. In turn, Merge computes LD_B depending only on the local distance maps LD_{B_i} of the children $\{B_i\}$ of B, and uses the closure operator $*$ to capture possibly unbdounded traversals of cycles whose smallest-level node is x. See Method 1 and Algorithm 2 for a formal description.

Method 1: Merge

Input: A bag B_x with children $\{B_i\}$
Output: A local distance map LD_{B_x}

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

Lemma 2. *At the end of* Preprocess*, for every bag B and nodes $u, v \in B$, we have $\mathsf{LD}_B(u,v) = \bigoplus_{P: u \rightsquigarrow v}\{\otimes(P)\}$, where all P are* U*-shaped paths in B.*

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

Algorithm 2: Preprocess

Input: A tree-decomposition $\mathrm{Tree}(G) = (V_T, E_T)$
Output: A local distance map LD_B for each bag $B \in V_T$

1 Traverse $\mathrm{Tree}(G)$ bottom up and examine each bag B with children $\{B_i\}$
2 **if** B *introduces some node* x **then**
3 Assign $\mathsf{LD}_B \leftarrow$ Merge on B
4 **else**
5 **foreach** $u, v \in B$ **do**
6 Assign $\mathsf{LD}_B(u,v) \leftarrow \bigoplus\{\mathsf{LD}_{B_1}(u,v), \ldots, \mathsf{LD}_{B_j}(u,v)\}$
7 **end**
8 **end**

$P_1 : u \rightsquigarrow x$, $P_2 : x \rightsquigarrow x$ and $P_3 : x \rightsquigarrow v$, such that x is not an intermediate node in either P_1 or P_3, and we have by distributivity:

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

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 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

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

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

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

Lemma 3. Preprocess *requires $O(n)$ semiring operations.*

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

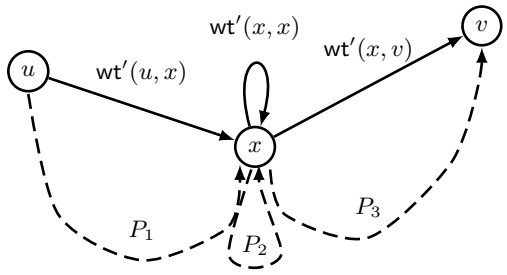

Figure 3: Illustration of the inductive argument of Preprocess.

Informal description of updating. Algorithm Update is called whenever the weight $\mathsf{wt}(x,y)$ of an edge of G has changed. Given the guarantee of Lemma 2, after Update has run on an edge update $\mathsf{wt}(x,y)$, it restores the property that for each bag B we have $\mathsf{LD}_B(u,v) = \bigoplus_{P: u \rightsquigarrow v} \{\otimes(P)\}$, where all P are U-shaped paths in B. See Algorithm 3 for a formal description.

Algorithm 3: Update

Input: An edge (x,y) with new weight $\mathsf{wt}(x,y)$
Output: A local distance map LD_B for each bag $B \in V_T$

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

Lemma 4. *At the end of each run of* Update*, for every bag B and nodes $u, v \in B$, we have $\mathsf{LD}_B(u,v) = \bigoplus_{P: u \rightsquigarrow v} \{\otimes(P)\}$, where all P are U-shaped paths in B.*

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 2. □

Lemma 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 $B_{(u,v)}$ to the root. Recall that the height of $\mathrm{Tree}(G)$ is $O(\log n)$ (Theorem 1), 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 1, every path $P : u \rightsquigarrow v$ is guaranteed to go through the least common ancestor (LCA) B_L of B_u and B_v, and possibly some of the ancestors B of B_L. Given this fact, algorithm Query uses the procedure Climb to climb up the tree from B_u and B_v until it reaches B_L and then the root of $\mathrm{Tree}(G)$. For each encountered bag B along the way, it computes maps $\delta_u(w) = \bigoplus_{P_1} \{\otimes(P_1)\}$, and $\delta_v(w) = \bigoplus_{P_2} \{\otimes(P_2)\}$ where all $P_1 : u \rightsquigarrow w$ and $P_2 : w \rightsquigarrow v$ are such that each intermediate node y in them has been introduced in B. This guarantees that for path P such that $d(u,v) = \otimes(P)$, when Query examines the bag B_z introducing $z = \mathsf{argmin}_{x \in P} \mathsf{Lv}(x)$, 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 introducing some node x. Figure 4 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 δ, a flag Up
Output: A new map δ

1 Remove from δ all $w \notin B$
2 Assign $\delta(w) \leftarrow \overline{\mathbf{0}}$ for all $w \in B$ and not in δ
3 **if** *B introduces node x* **then**
4 **if** Up **then** /* Climbing up */
5 Update δ with $\delta(w) \leftarrow \bigoplus\{\delta(w), \bigotimes(\delta(x), \mathsf{LD}_B(x,w))\}$
6 **else** /* Climbing down */
7 Update δ with $\delta(w) \leftarrow \bigoplus\{\delta(w), \bigotimes(\delta(x), \mathsf{LD}_B(w,x))\}$
8 **end**
9 **return** δ

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

Algorithm 5: Query

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

Initialize map δ_u with $\delta_u(w) \leftarrow \mathsf{LD}_{B_u}(u, w)$
Initialize map δ_v with $\delta_v(w) \leftarrow \mathsf{LD}_{B_v}(w, v)$
Assign $B_L \leftarrow$ the LCA of B_u, B_v in $\mathrm{Tree}(G)$
Assign $B \leftarrow B_u$
repeat
    Assign $B \leftarrow B'$ where B' is the parent of B
    Call Climb on B and δ_u with flag Up set to True
until $B = B_L$
Assign $B \leftarrow B_v$
repeat
    Assign $B \leftarrow B'$ where B' is the parent of B
    Call Climb on B and δ_v with flag Up set to False
until $B = B_L$
Assign $B \leftarrow B_L$
Assign $\delta \leftarrow \bigoplus_{x \in B_L} \otimes(\delta_u(x), \delta_v(x))$
repeat
    Assign $B \leftarrow B'$ where B' is the parent of B
    Call Climb on B and δ_u with flag Up set to True
    Call Climb on B and δ_v with flag Up set to False
    if B *introduces node* x **then**
        Assign $\delta \leftarrow \bigoplus\{\delta, \bigotimes(\delta_u(x), \delta_v(x))\}$
until $\mathsf{Lv}(B) = 0$
return δ

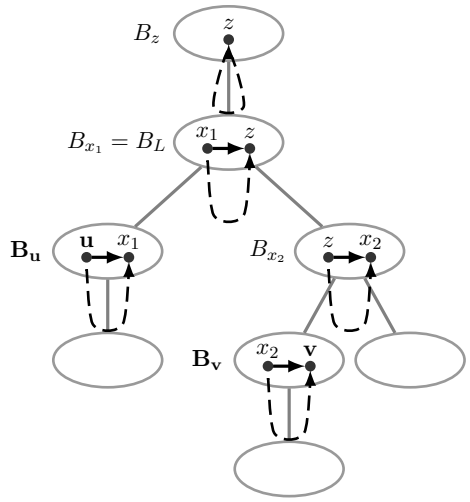

Figure 4: Illustration of Query in computing the distance $d(u, v) = \otimes(P)$ as a sequence of U-shaped paths, whose weight has been captured in the local distance map of each bag. When B_z is examined, with $z = \mathsf{argmin}_{x \in P}\mathsf{Lv}(x)$, it will be $\delta_u(z) = d(u, z)$ and $\delta_v(z) = d(z, v)$, and hence by distributivity $d(u, v) = \bigotimes(\delta_u(z), \delta_v(z))$.

ter 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 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 $\mathrm{Tree}(G)$ is $O(\log n)$ (Theorem 1), and the result follows. □

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

Theorem 2. *Consider a graph* $G = (V, E)$ *and a balanced, semi-nice tree-decomposition* $\mathrm{Tree}(G)$ *of constant treewidth. 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.*

Witness paths. Our algorithms so far have only been concerned with returning the distance $d(u, v)$ of the pair query u, v. When the semiring lacks the closure operator (i.e., for all $s \in \Sigma$ it is $s^* = \overline{\mathbf{1}}$), as in most problems e.g., reachability and shortest paths with positive weights, the distance from every u to v is realized by an acyclic path. Then, it is straightforward to also obtain a witness path, i.e., a path $P : u \rightsquigarrow v$ such that $\otimes(P) = d(u, v)$, with some minor additional preprocessing. Here we outline how.

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

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 $A = \{A_1, A_2, \ldots, A_k\}$, where A_i consists of n_i nodes and b_i boxes; (ii) a partially complete semiring $(\Sigma, \oplus, \otimes, \overline{\mathbf{0}}, \overline{\mathbf{1}})$; and (iii) a maximum stack height h. Our task is to create a datastructure 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 1). For this purpose, we present the algorithm RSMDistance, which performs such preprocessing using a datastructure $\mathcal{D}$ consisting of the algorithms Preprocess, Update and Query of Section 3. At the end of RSMDistance it will hold that algebraic path pair queries in a CSM A_i can be answered in $O(\log n_i)$ semiring operations. We later present some additional preprocessing which suffers a factor of $O(\log n_i)$ in the preprocessing space, but reduces the pair query time to constant.

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

1. First, 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.

2. Then, iteratively for each ℓ, where $0 \leq \ell \leq h-1$, given that we have a dynamic datastructure $\mathcal{D}$ (concretely, an instance of the dynamic algorithms Update and Query from Section 3) for computing $d(u, v, \ell)$, the algorithm does as follows: First, for each G_i whose entry to exit distance $d(En_i, Ex_i)$ 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.
3. Then, it obtains the entry to exit distance $d(En_j, Ex_j)$ 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 ∈ 𝒢 do
2      Construct the tree-decomposition Tree(G_i)
3      Call Preprocess on Tree(G_i)
4  end
5  distances ← [Call Query on (En_i, Ex_i) of G_i]_{1≤i≤k}
6  modified ← {1, ..., k}
7  for ℓ ← 0 to h − 1 do
8      modified′ ← ∅
9      foreach i ∈ modified do
10         foreach G_j that contains boxes b_{j_1}, ..., b_{j_l} s.t. Y_j(b_{j_x}) = i do
11             Call Update on G_j for the weight change
               wt((en, b_{j_l}), (ex, b_{j_x})) ← distances[i]
12             Call Query on (En_j, Ex_j)
13             if d(En_j, Ex_j) ≠ distances[j] then
14                 modified′ ← modified′ ∪ {j}
15                 distances[j] ← d(En_j, Ex_j)
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 $O(\sum_{i=1}^{k}(n_i \cdot \log n_i))$ time to construct a balanced tree decomposition (Theorem 1), $O(n)$ time to preprocess all G_i initially, and $O(\sum_{i=1}^{k}(b_i \cdot \log n_i))$ to update all G_i for one iteration of the loop of Line 4 (from Theorem 2). Hence, RSMDistance uses $O(\sum_{i=1}^{k}(n_i \cdot \log n_i + h \cdot b_i \cdot \log n_i))$ preprocessing semiring operations. Finally, it is easy to verify that all preprocessing is done in $O(n)$ space.

After the last iteration of algorithm RSMDistance, we have a datastructure $\mathcal{D}$ 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. We now present a small example of how RSMDistance is executed on the RSM of Figure 2 for the case of reachability. In this case, for any pair of nodes (u, v), we have $d(u, v) = \mathsf{True}$ iff u reaches v. Table 4(a) illustrates how the local distance maps LD_{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 $\mathsf{LD}_{B_x}(u, v) = \mathsf{True}$ (i.e., u reaches v). For brevity, in the table we hide self loops (i.e., entries of the form (u, u)) although they are stored by the algorithms. Initially, the stack height $\ell = 0$, and Preprocess is called for each graph (line 3). The new reachability relations discovered by Merge are shown in bold. Note that at this point we have $\mathsf{wt}(4, 5) = \mathsf{False}$ in method dot_matrix, as we do not know whether the call to method dot_vector actually returns. Afterwards, Query is called to discover the distance $d(1, 6)$ in method dot_vector (line 5). Table 4(b) shows the sequence in which Query examines the bags of the tree decomposition, and the distances δ_1, δ_6 and δ it maintains. When B_2 is examined, $\delta = \mathsf{True}$ and hence at the end Query returns $\delta = \mathsf{True}$. Finally, since Query returns $\delta = \mathsf{True}$, the weight $\mathsf{wt}(4, 5)$ between the call-return pair of nodes $(4, 5)$ in method dot_matrix is set to True. An execution of Update (line 11) with this update on the corresponding tree decomposition (Table 4(a) for $\ell = 1$) updates the entries $(4, 5)$ and $(4, 3)$ in LD_{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. For graphs with constant treewidth, an extension of Lemma 7 from [16] allows us to precompute the distance $d(u, v)$ for every pair of nodes $u, v \in V_i$ that appear in the same bag of $\mathrm{Tree}(G_i)$. The computation required is similar to Preprocess, with the difference that this time $\mathrm{Tree}(G_i)$ is traversed top-down instead of bottom-up. Additionally, for each examined bag B, a Floyd-Warshall algorithm is run in the graph G_i induced by B, and all pairs of distances are updated. It follows from Lemma 7 of [16] that for constant treewidth, this step requires $O(n_i)$ time and space.

After all distances $d(u, v)$ have been computed for each B, it is straightforward to answer single-source queries from some node u in linear time. The algorithm simply maintains a map $A : V_i \to \Sigma$, and initially $A(v) = d(u, v)$ for all $v \in B_u$, and $A(v) = \overline{\mathbf{0}}$ otherwise. Then, it traverses $\mathrm{Tree}(G_i)$ in a BFS manner starting at B_u, and for every encountered bag B and $v \in B$, if $A(v) = \overline{\mathbf{0}}$, it sets $A(v) = \bigoplus_{z \in B} \bigotimes(A(z), d(z, v))$. 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 Theorem 10 from [16] to our setting, which in turn is based on a rather complicated algorithmic technique of [1]. We present a more intuitive, simpler and implementable approach that has a dynamic programming nature. In Section 5 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 $d(u, v)$ for every pair of nodes u, v that appear in the same bag, at no overhead. To handle pair queries in constant time, we further traverse each $\mathrm{Tree}(G_i)$ one last time, bottom-up, and for each node u we store maps $F_u, T_u : V_i^{B_u} \to \Sigma$, where $V_i^{B_u}$ is the subset of V_i of nodes that appear in B_u and its descendants in $\mathrm{Tree}(G_i)$. The maps are such that $F_u(v) = d(u, v)$ and $T_u = d(v, u)$. Hence, F_u stores

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

(a)

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

(b)

Table 4: Illustration of RSMDistance on the tree decompositions of methods dot_vector and dot_matrix from Figure 2. 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.

the distances from u to nodes in $V_i^{B_u}$, and T_u stores the distances from nodes in $V_i^{B_u}$ to u. The maps are computed in a dynamic programming fashion, as follows:

1. Initially, the maps F_u and T_u are constructed for all u that appear in a bag B which is a leaf of $\mathrm{Tree}(G_i)$. The information required has already been computed as part of the preprocessing for answering single-source queries. Then, $\mathrm{Tree}(G_i)$ is traversed up, level by level.
2. When examining a bag B such that the computation has been performed for all its children, for every node $u \in B$ and $v \in V_i^B$, we set $F_u(v) = \bigoplus_{z \in B} \bigotimes \{d(u,z), F_z(v)\}$, and similarly for $T_u = \bigoplus_{z \in B} \bigotimes \{d(z,u), T_z(v)\}$.

An application of Lemma 1 inductively on the levels processed by the algorithm can be used to show that when a bag B is processed, for every node $u \in B$ and $v \in V_i^B$, we have $T_u(v) = \bigoplus_{P: v \rightsquigarrow u} \otimes(P)$ and $F_u(v) = \bigoplus_{P: u \rightsquigarrow v} \otimes(P)$. Finally, there are $O(n_i)$ semiring operations done at each level of $\mathrm{Tree}(G_i)$, and since there are $O(\log n_i)$ levels, $O(n_i \cdot \log n_i)$ operations are required in total. Hence, the space used is also $O(n_i \cdot \log n_i)$. We furthermore preprocess $\mathrm{Tree}(G_i)$ in linear time and space to answer LCA queries in constant time (note that since $\mathrm{Tree}(G_i)$ is balanced, this is standard). To answer a pair query u, v, it suffices to first obtain the LCA B of B_u and B_v, and it follows from Lemma 1 that $d(u,v) = \bigoplus_{z \in B} \bigotimes \{T_z(u), F_z(v)\}$, 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 3. *Fix the following input: (i) a constant treewidth RSM $A = \{A_1, A_2, \ldots, A_k\}$, where A_i consists of n_i nodes and b_i boxes; (ii) a partially complete semiring $(\Sigma, \oplus, \otimes, \overline{\mathbf{0}}, \overline{\mathbf{1}})$; and (iii) a maximum stack height h.* RSMDistance *uses $O(\sum_{i=1}^{k}(n_i \cdot \log n_i + h \cdot b_i \cdot \log n_i))$ preprocessing semiring operations and*

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

IFDS/IDE framework. In the special case where the algebraic path problem belongs to the IFDS/IDE framework, we have a meet-composition semiring $(F, \sqcap, \circ, \emptyset, I)$, 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 [36]). 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. Then, 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 per G_i is then $b_i \cdot \log n_i \cdot |D|^3$ (as there are $|D|$ data facts), and we have the following corollary (also see Table 2).

Corollary 1 (IFDS/IDE)**.** *Fix the following input a (i) constant treewidth RSM $A = \{A_1, A_2, \ldots, A_k\}$, where A_i consists of n_i nodes and b_i boxes; 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.*

1. *Algorithm* RSMDistance *uses $O(\sum_{i=1}^{k}(n_i \cdot |D|^2 + b_i \cdot \log n_i \cdot |D|^3 + n_i \cdot \log n_i))$ preprocessing time, $O(n \cdot |D|^2)$ space, and 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. *Algorithm* RSMDistance *uses $O(\sum_{i=1}^{k}(n_i \cdot \log n_i \cdot |D|^2 + b_i \cdot \log n_i \cdot |D|^3))$ preprocessing time, $O(|D|^2 \cdot \sum_{i=1}^{k}(n_i \cdot \log n_i))$ space, and 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.*

Reachability. The special case of reachability is obtained by setting $|D| = 1$ in Corollary 1.

Shortest paths. The 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). Because we consider non-negative weights, it follows that the distance between any pair of nodes is realized by a path that traverses every

entry node at most once. Hence, we set $h = k$ in Theorem 3, and obtain the following corollary for shortest paths (also see Table 3).

Corollary 2 (Shortest paths)**.** *Fix the following input a (i) constant treewidth RSM $A = \{A_1, A_2, \ldots, A_k\}$, where A_i consists of n_i nodes and b_i boxes; (ii) a tropical semiring $(\mathbb{R}_{\geq 0} \cup \{\infty\}, \min, +, \infty, 0)$.* RSMDistance *uses $O(\sum_{i=1}^{k}(n_i \cdot \log n_i + k \cdot b_i \cdot \log n_i))$ preprocessing time and:*

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

Interprocedural witness paths. As in the case of simple graphs from Section 3, we can retrieve a witness path for any distance $d(u, v, h)$ that is realized by acyclic interprocedural paths $P : (u, \emptyset) \rightsquigarrow (v, \emptyset)$, without affecting the stated complexities. The process is straightforward. Let A_i contain the pair of nodes u, v on which the query is asked. Initially, we obtain the witness intraprocedural path $P' : u \rightsquigarrow v$, as described in Section 3. Then, we proceed recursively to obtain a witness path P_j between the entry En_j and exit Ex_j nodes of every CSM A_j such that P' contains an edge between a call node (en, b) and a return node (ex, b) with $Y_i(B) = j$. That is, we reconstruct a witness path for every call to a CSM whose weight has been summarized locally in A_i. This process constructs an interprocedural witness path $P : u \rightsquigarrow v$ such that $\otimes(P) = d(u, v)$ in $O(|P|)$ time.

5. Experimental Results

Set up. We have implemented our algorithms for linear-time single-source and constant-time pair queries presented in Section 4 and have tested them on graphs obtained from the DaCapo benchmark suit [6] that contains several, real-world Java applications. Every benchmark is represented as a RSM that consists of several CSMs, and each CSM corresponds to the control flow graph of a method of the benchmark. We have used the Soot framework [43] for obtaining the control flow graphs, where every node of the graph corresponds to one Jimple statement of Soot, and the tool of [44] to obtain their tree decompositions. Our experiments were run on a standard desktop computer with a 3.4GHz CPU, on a single thread.

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

- *(Reachability).* For every G_i, the complete preprocessing in the case of reachability is done by executing n_i DFSs, one from each source node. The single-source query from u is answered by executing one DFS from u, and the pair query u, v is done similarly, but we stop as soon as v is reached. We note that this methodology correctly answers interprocedural same-context reachability queries.
- *(Shortest path).* For shortest path we perform intraprocedural analysis on each G_i. We assign both positive and negative weights to each edge of G_i uniformly at random from the range $[-10, 10]$. For general semiring path properties, the Bellman-Ford algorithm [17] is a very natural one, which in the case of shortest path can handle positive and negative weights, as long as there is no negative cycle. To have a meaningful comparison with Bellman-Ford (as a representative of a general semiring framework), we consider both positive and negative weights, but do not allow negative cycles. For complete preprocessing we run the classical Floyd-Warshall algorithm (which computes all-pairs shortest paths and is a generalization of Bellman-Ford). Under no preprocessing, for every single-source and pair query we run the Bellman-Ford algorithm.

Results. Our experimental results are shown in Table 5.

1. The average treewidth of control flow graphs is confirmed to be very small, and does not scale with the size of the graph. In fact, even the largest treewidth is four.
2. The preprocessing time of our algorithm is significantly less than the complete preprocessing, by factor of 1.5 to 4 times in case of reachability, and by orders of magnitude in case of shortest path.
3. In both reachability and shortest path, all queries are handled significantly faster after our preprocessing, than no preprocessing. We also note that for shortest path queries, Bellman-Ford answers single-source and pair queries in the same time, which is significantly slower than both our single-source and pair queries. Finally, we note that for single-source reachability queries, though we do not provide theoretical improvement over DFS (Table 2), the one-time preprocessing information allows for practical improvements.

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

Description of Table 5. In the table, the second (resp. third) column shows the average number of nodes (resp. treewidth) of CSMs of each benchmark. The running times of preprocessing are gathered by averaging over all CSMs in each benchmark. The running times of querying are gathered by averaging over all possible single-source and pair queries in each CSM, and then averaging over all CSMs in each benchmark.

6. Conclusions

In this work we considered constant treewidth RSMs since control flow graphs of most programs have constant treewidth. We presented algorithms to handle multiple same-context algebraic path queries, where the weights belong to a partially complete semiring. Our algorithms have small additional one-time preprocessing, but answer subsequent queries significantly faster than no preprocessing both in terms of theoretical bounds as well as in practice, even for basic problems such as reachability and shortest path. While in this work we focused on RSMs with unique entries and exits, an interesting theoretical question is to extend our results to RSMs with multiple entries and exists.

Acknowledgements. We thank anonymous reviewers for helpful comments to improve the presentation of the paper.

References

[1] N. Alon and B. Schieber. Optimal preprocessing for answering on-line product queries. Technical report, Tel Aviv University, 1987.

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

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

Benchmarks			Interprocedural Reachability						Intraprocedural Shortest path					
			Preprocessing		Query				Preprocessing		Query			
					Single		Pair				Single		Pair	
	n	t	Our	Complete	Our	No Prepr.	Our	No Prepr.	Our	Complete	Our	No Prepr.	Our	No Prepr.
antlr	698	1.0	76316	136145	15.3	166.3	0.15	14.34	221578	$1.13{\cdot}10^7$	251	24576	0.36	24576
bloat	696	2.3	27597	54335	3.9	72.5	0.10	14.34	87950	$1.15{\cdot}10^7$	257	25239	0.37	25239
chart	1159	1.5	22191	90709	2.3	80.9	0.13	22.32	125468	$1.24{\cdot}10^8$	398	88856	0.39	88856
eclipse	656	1.6	37010	138905	6.7	239.1	0.19	15.76	152293	$1.07{\cdot}10^7$	533	23639	0.46	23639
fop	1209	1.7	30189	91795	2.9	60.6	0.12	43.0	153728	$3.94{\cdot}10^8$	1926	113689	2.71	113689
hsqldb	698	1.0	55668	180333	13.0	219.0	0.14	13.89	215063	$1.23{\cdot}10^7$	236	24322	0.36	24322
jython	748	1.5	43609	68687	7.2	85.7	0.11	12.84	159085	$1.42{\cdot}10^7$	386	29958	0.32	29958
luindex	885	1.3	36015	142005	5.6	202.7	0.16	26.44	163108	$2.97{\cdot}10^7$	258	51192	0.37	51192
lusearch	885	1.3	51375	189251	12.8	211.4	0.13	26.01	219015	$2.90{\cdot}10^7$	254	50719	0.34	50719
pmd	644	1.4	31483	52527	2.5	83.9	0.13	12.5	140974	$9.14{\cdot}10^6$	327	22572	0.37	22572
xalan	698	1.0	57734	138420	8.0	235.0	0.19	14.28	186695	$1.10{\cdot}10^7$	380	24141	0.43	24141
Jflex	1091	1.6	51431	91742	3.1	50.8	0.11	20.46	154818	$1.24{\cdot}10^8$	231	83093	0.36	83093
muffin	1022	1.7	29905	66708	2.6	52.7	0.10	18.57	125938	$1.02{\cdot}10^8$	265	80878	0.38	80878
javac	711	1.8	32981	59793	4.8	75.2	0.11	11.86	117390	$1.31{\cdot}10^7$	370	26180	0.34	26180
polyglot	698	1.0	68643	150799	12.2	184.5	0.14	14.14	228758	$1.15{\cdot}10^7$	244	24400	0.35	24400

Table 5: Average statistics gathered from our experiments on the DaCapo benchmark suit. Times are in microseconds.

[4] S. Arnborg and A. Proskurowski. Linear time algorithms for NP-hard problems restricted to partial k-trees . *Discrete Appl Math*, 1989.

[5] M. Bern, E. Lawler, and A. Wong. Linear-time computation of optimal subgraphs of decomposable graphs. *J Algorithm*, 1987.

[6] S. M. e. a. Blackburn. The dacapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, 2006.

[7] H. Bodlaender. Dynamic algorithms for graphs with treewidth 2. In *Graph-Theoretic Concepts in Computer Science*, LNCS. Springer, 1994.

[8] H. Bodlaender. Discovering treewidth. In *SOFSEM 2005: Theory and Practice of Computer Science*, volume 3381 of *LNCS*. Springer, 2005.

[9] H. L. Bodlaender. Dynamic programming on graphs with bounded treewidth. In *ICALP*, LNCS. Springer, 1988.

[10] H. L. Bodlaender. A tourist guide through treewidth. *Acta Cybern.*, 1993.

[11] D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. In *CC*. ACM, 1986.

[12] K. Chatterjee and J. Lacki. Faster algorithms for Markov decision processes with low treewidth. In *CAV*, 2013.

[13] K. Chatterjee and Y. Velner. Mean-payoff pushdown games. In *LICS*, 2012.

[14] K. Chatterjee, A. Pavlogiannis, and Y. Velner. Quantitative interprocedural analysis. In *POPL*, 2015.

[15] S. Chaudhuri. Subcubic algorithms for recursive state machines. In *POPL*, New York, NY, USA, 2008. ACM.

[16] S. Chaudhuri and C. D. Zaroliagis. Shortest Paths in Digraphs of Small Treewidth. Part I: Sequential Algorithms. *Algorithmica*, 1995.

[17] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction To Algorithms*. MIT Press, 2001.

[18] B. Courcelle. Graph rewriting: An algebraic and logic approach. In *Handbook of Theoretical Computer Science (Vol. B)*. MIT Press, Cambridge, MA, USA, 1990.

[19] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E. Neuhold, editor, *IFIP Conf. on Formal Description of Programming Concepts*, 1977.

[20] M. Elberfeld, A. Jakoby, and T. Tantau. Logspace versions of the theorems of bodlaender and courcelle. In *FOCS*, 2010.

[21] M. J. Fischer and A. R. Meyer. Boolean Matrix Multiplication and Transitive Closure. In *SWAT (FOCS)*. IEEE Computer Society, 1971.

[22] R. Giegerich, U. Möncke, and R. Wilhelm. Invariance of approximate semantics with respect to program transformations. In *3rd Conference of the European Co-operation in Informatics (ECI)*, 1981.

[23] D. Grove and L. Torczon. Interprocedural constant propagation: A study of jump function implementation. In *PLDI*. ACM, 1993.

[24] J. Gustedt, O. Mæhle, and J. Telle. The treewidth of java programs. In *Algorithm Engineering and Experiments*, LNCS. Springer, 2002.

[25] T. Hagerup. Dynamic algorithms for graphs of bounded treewidth. *Algorithmica*, 2000.

[26] R. Halin. S-functions for graphs. *Journal of Geometry*, 1976.

[27] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. *SIGSOFT Softw. Eng. Notes*, 1995.

[28] T. Kloks. *Treewidth, Computations and Approximations*. LNCS. Springer, 1994.

[29] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *CC*, 1992.

[30] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Trans. Program. Lang. Syst.*, 1996.

[31] J. Lacki. Improved deterministic algorithms for decremental reachability and strongly connected components. *ACM Transactions on Algorithms*, 2013.

[32] W. Landi and B. G. Ryder. Pointer-induced aliasing: A problem classification. In *POPL*. ACM, 1991.

[33] N. A. Naeem and O. Lhoták. Typestate-like analysis of multiple interacting objects. In *OOPSLA*, 2008.

[34] J. Obdrzálek. Fast mu-calculus model checking when tree-width is bounded. In *CAV*, 2003.

[35] B. A. Reed. Finding approximate separators and computing tree width quickly. In *STOC*, 1992.

[36] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, New York, NY, USA, 1995. ACM.

[37] T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.*, 2005.

[38] T. Reps, A. Lal, and N. Kidd. Program analysis using weighted pushdown systems. In *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*, LNCS. 2007.

[39] N. Robertson and P. Seymour. Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B*, 1984.

[40] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.*, 1996.

[41] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for java. In *OOPSLA*, 2005.

[42] M. Thorup. All Structured Programs Have Small Tree Width and Good Register Allocation. *Information and Computation*, 1998.

[43] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *CASCON '99*. IBM Press, 1999.

[44] T. van Dijk, J.-P. van den Heuvel, and W. Slob. Computing treewidth with libtw. Technical report, University of Utrecht, 2006.

[45] X. Zhang, R. Mangal, M. Naik, and H. Yang. Hybrid top-down and bottom-up interprocedural analysis. In *PLDI*, 2014.