Exploiting the Sparseness of Control-flow and Call Graphs for Efficient and On-demand Algebraic Program Analysis *

Giovanna Kobus Conrado, Amir Kafshdar Goharshady, Kerim Kochekov, Yun Chen Tsai, Ahmed Khaled Zaher

Exploiting the Sparseness of Control-flow and Call Graphs for Efficient and On-demand Algebraic Program Analysis*

GIOVANNA KOBUS CONRADO, AMIR KAFSHDAR GOHARSHADY, KERIM KOCHEKOV, YUN CHEN TSAI, and AHMED KHALED ZAHER, Hong Kong University of Science and Technology, Hong Kong

Algebraic Program Analysis (APA) is a ubiquitous framework that has been employed as a unifying model for various problems in data-flow analysis, termination analysis, invariant generation, predicate abstraction and a wide variety of other standard static analysis tasks. APA models program summaries as elements of a regular algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$. Suppose that a summary in A is assigned to every transition of the program and that we aim to compute the effect of running the program starting at line s and ending at line t. APA first computes a regular expression ρ capturing all program paths of interest. In case of intraprocedural analysis, ρ models all paths from s to t, whereas in the interprocedural case it models all interprocedurally-valid paths, i.e. paths that go back to the right caller function when a callee returns. This regular expression ρ is then interpreted over the algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$ to obtain the desired result. Suppose the program has n lines of code and each evaluation of an operation in the regular algebra takes $O(k)$ time. It is well-known that a single APA query, or a set of queries with the same starting point s, can be answered in $O(n \cdot \alpha(n) \cdot k)$, where α is the inverse Ackermann function.

In this work, we consider an on-demand setting for APA: the program is given in the input and can be preprocessed. The analysis has to then answer a large number of on-line queries, each providing a pair (s, t) of program lines which are the start and end point of the query, respectively. The goal is to avoid the significant cost of running a fresh APA instance for each query. Our main contribution is a series of algorithms that, after a lightweight preprocessing of $O(n \cdot \lg n \cdot k)$, answer each query in $O(k)$ time. In other words, our preprocessing has almost the same asymptotic complexity as a single APA query, except for a sub-logarithmic factor, and then every future query is answered instantly, i.e. by a constant number of operations in the algebra. We achieve this remarkable speedup by relying on certain structural sparsity properties of control-flow and call graphs (CFGs and CGs). Specifically, we exploit the fact that control-flow graphs of real-world programs have a tree-like structure and bounded treewidth and nesting depth and that their call graphs have small treedepth in comparison to the size of the program. Finally, we provide experimental results demonstrating the effectiveness and efficiency of our approach and showing that it beats the runtime of classical APA by several orders of magnitude.

CCS Concepts: • **Theory of computation** → **Program analysis**; **Program verification**; **Program reasoning**; **Parameterized complexity and exact algorithms**.

Additional Key Words and Phrases: Algebaric Program Analysis, Parameterized Algorithms, Graph Sparsity, Treewidth, Treedepth, Data-flow Analysis

*The research was partially supported by the Hong Kong Research Grants Council ECS Project Number 26208122. G.K. Conrado, K. Kochekov and A.K. Zaher were supported by the Hong Kong PhD Fellowship Scheme (HKPFS). Authors are ordered alphabetically.

Authors' address: Giovanna Kobus Conrado, gkc@connect.ust.hk; Amir Kafshdar Goharshady, goharshady@cse.ust.hk; Kerim Kochekov, kkochekov@connect.ust.hk; Yun Chen Tsai, yctsai@connect.ust.hk; Ahmed Khaled Zaher, akazaher@connect.ust.hk, Department of Computer Science and Engineering, Department of Mathematics, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong.

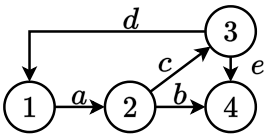
2475-1421/2023/10-ART292

https://doi.org/10.1145/3622868

ACM Reference Format:
Giovanna Kobus Conrado, Amir Kafshdar Goharshady, Kerim Kochekov, Yun Chen Tsai, and Ahmed Khaled Zaher. 2023. Exploiting the Sparseness of Control-flow and Call Graphs for Efficient and On-demand Algebraic Program Analysis. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 292 (October 2023), 32 pages. https://doi.org/10.1145/3622868

1 INTRODUCTION

Algebraic Program Analysis (APA). Algebraic Program Analysis traces its roots to the algebraic approach of solving path problems in graphs as exemplified by Backhouse and Carré [1975] and Tarjan [1981b]. The elegant idea in Tarjan [1981b] was to first compute a regular expression ρ capturing all paths of interest in a given graph and then reinterpret ρ in a different algebraic structure $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$, which is often a Kleene algebra. More formally, if A is a Kleene algebra, then the reinterpretation is a homomorphism from regular expressions to A. For example, suppose we are interested in finding the length of a shortest path from vertex 1 to 4 in the graph $G = (V, E)$ above in which every edge has a real weight assigned by a function $[\![.]\!] : E \to \mathbb{R}$. We can first compute a regular expression ρ that captures all paths from 1 to 4. For example, we can set $\rho := (a \cdot c \cdot d)^* \cdot (a \cdot b + a \cdot c \cdot e)$. We then consider the algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$ in which the parts are defined as follows:



$$A := \mathbb{R} \cup \{-\infty, +\infty\} \quad x \oplus y := \min\{x, y\} \quad x \otimes y := x + y \quad x^{\circledast} := \begin{cases} 0 & x \geq 0 \\ -\infty & x < 0 \end{cases} \quad \overline{0} := +\infty \quad \overline{1} := 0$$

Replacing every edge in ρ with its weight and reinterpreting the result in A will yield the shortest path from 1 to 4, since the $\oplus$ operator combines two paths by choosing the shorter one and the operator $\otimes$ concatenates paths by adding their lengths. More specifically, the distance is

$$[\![\rho]\!] := ([\![a]\!] \otimes [\![c]\!] \otimes [\![d]\!])^{\circledast} \otimes ([\![a]\!] \otimes [\![b]\!] \oplus [\![a]\!] \otimes [\![c]\!] \otimes [\![e]\!]) .$$

Thus, to obtain the shortest path, the first step was to compute ρ and the second step was to reinterpret it in A. While shortest path is interesting in its own right, algebraic program analysis is particularly focused on the case where the graph G is the control-flow graph (CFG) of a program P, or a combination of its control-flow and call graph (CG), so that the paths in G model runs of P, and the elements in A are program summaries. The idea is to assign a semantic meaning $[\![e]\!] \in A$ to every edge $e \in E$ and extend the semantics to paths and regular expressions, in a way that ensures $[\![\rho]\!]$ is the desired result of the static analysis. For example, a standard data-flow analysis such as Kildall [1973] can be modeled in this framework by an algebra in which every element $a \in A$ is a transformer function, mapping data facts that hold before the execution of a program fragment to those that may/must hold after its execution. Here, $\otimes$ is function composition and $\oplus$ is the join/meet operator. Thus, $[\![\rho]\!]$ yields the join/meet-over-all-paths answer. See Section 2 for a more detailed example.

Notation. Throughout this work, we write $[\![x]\!]$ to denote an element of the algebra A that corresponds to or summarizes x. When x is an edge of the graph, then $[\![x]\!]$ is simply its assigned semantics. When x is a regular expression, $[\![x]\!]$ is its reinterpretation in A. Finally, when x is a function or a subprogram, $[\![x]\!]$ summarizes all paths from the starting point of x to its endpoint. In other words, $[\![x]\!]$ is the reinterpretation of the regular expression that captures all paths from the start to the end of x.

The APA framework models a wide variety of static analysis tasks, including data-flow [Kildall 1973; Reps et al. 1995; Sagiv et al. 1996], recurrence analysis [Kincaid et al. 2017], predicate abstraction and loop summarization [Kroening et al. 2008], termination analysis [Zhu and Kincaid 2021] and invariant generation [Kincaid et al. 2018]. See Breck [2020] and Kincaid et al. [2021] for a more detailed treatment and a recent survey/tutorial on APA. APA provides a dualistic contrast to the classical approach of iterative abstract interpretation. Most static analyses can be thought of as ways of solving a system of possibly recursive semantic equations. As beautifully put in Kincaid et al. [2021], APA's approach is to first find a closed-form solution, i.e. the regular expression ρ, and then interpret it based on program semantics. This is in contrast to the classical approach in abstract interpretation which first interprets and then solves the equations by an iterative method.

Intraprocedural APA. If our underlying graph G is the control-flow graph of a single function in a program P, then the APA approach above is said to be intraprocedural. At the heart of intraprocedural APA is the algorithm of Tarjan [1981a] which can compute the regular path expression ρ in $O(n \cdot \alpha(n))$ where n is the number of vertices and edges* in the graph G and α is the inverse Ackermann function. This algorithm is applicable to reducible flow graphs, which contain almost all real-world control-flow graphs. Tarjan's algorithm produces a representation of the path expression as an expression tree with $O(n \cdot \alpha(n))$ vertices. This bound is tight, i.e. there are cases where the representation's size is $\Omega(n \cdot \alpha(n))$. Thus, the overall runtime of an intraprocedural APA is $O(n \cdot \alpha(n) \cdot k)$ assuming that the evaluation of each atomic operation in the algebra, i.e. $\oplus, \otimes$ and $\circledast$, takes $O(k)$ time. Crucially, this is the runtime when the start vertex is fixed. More specifically, given a fixed starting vertex s, which usually corresponds to the entry of the function, Tarjan's algorithm computes a regular expression ρ_t for every vertex t. This expression models all paths from the fixed vertex s to t in G. We then reinterpret ρ_t in $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$.

Interprocedural APA [Kincaid et al. 2021, Section 4]. The challenge in an interprocedural APA, analyzing the entirety of a program P which can include many functions, is that we can no longer rely directly on the algorithm of Tarjan [1981a]. Suppose our graph $G = (V, E)$ is an interprocedural control-flow graph, containing CFGs of each of the functions in P, together with interprocedural edges that model function calls and returns. Given $s, t \in V$, Tarjan's algorithm produces a regular expression ρ capturing all paths from s to t in G. However, not every path in G corresponds to a valid execution of P. For example, suppose that P has three functions f_1, f_2, g and both f_1 and f_2 call g in their code. An execution that starts with f_1 and goes into g should return back to f_1 when g reaches its endpoint. However, in the graph G, the endpoint of g also has an edge that goes back to f_2. Thus, G contains a path that starts in f_1, goes through g and returns to the wrong caller f_2. Such paths are called interprocedurally invalid and should not be included in the analysis. See Reps et al. [1995] for a more formal explanation of this point. To overcome this difficulty, interprocedural APA often consists of three steps [Cousot and Cousot 1977; Sharir and Pnueli 1978]:

(1) For every function g of the program P, a summary $[\![g]\!] \in A$ is computed. $[\![g]\!]$ models the behavior and effects of g, including those of any descendant functions that might be (transitively) called by g.
(2) Whenever a function f contains a call to g, an edge e is added from the vertex of the call site in f to its corresponding return site in f. Moreover, we set $[\![e]\!]$ based on $[\![g]\!]$ such that the entire execution of the call to g can be summarized by a single edge in the graph. We also remove the edge from the endpoint of g back to f.
(3) The intraprocedural APA algorithm is applied on the modified graph obtained in the previous step.

*A control-flow graph with n vertices has $O(n)$ edges.

The trick above reduces interprocedural APA to the intraprocedural case. Intuitively, any run of the program that reaches the call to g in f is either going to return from that call, which is modeled by e, or reach the analysis endpoint t without returning, which is why we have removed the edge from g back to f. See Section 2 for an example. Of course, the sticking point is how to compute the summaries in Step (1) above. This can be done either using the Newtonian program analysis technique of Esparza et al. [2010] or its more recent enhancement in Reps et al. [2017]. See Kincaid et al. [2021, Section 4] for more details and examples. Note that, although each APA query has a fixed start s and endpoint t, the summaries in Step (1) have to be computed only once and can then be reused for different values of s and t. Since computing function summaries is an orthogonal problem, in this work, we assume they are given as part of the input and account for recursion.

On-Demand Analysis. In this work, we consider the on-demand setting for APA. In our setting, the initial input consists of a program P, a regular algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$, a semantic function $[\![.]\!] : E \to A$ mapping every transition of the program to an element of the algebra, and a summary $[\![f]\!] \in A$ for every function f in P. The algorithm is allowed some time to preprocess this input and then receives a large number of on-line APA queries. Each query provides two program points s and t and asks for the result of the APA analysis starting from s and ending at t. The goal is to avoid running a new APA instance for each query and instead find a more scalable solution. In other words, our goal is to spend $o(n \cdot \alpha(n) \cdot k)$ time per query so that answering many queries is strictly faster than handling each query separately.

Function Summaries. We do not consider the summarization step, i.e. Step (1) above, and instead assume that the function summaries are given as part of the input. We also assume that the summaries handle recursion and its resulting fixed-points. This is due to the following reasons: (i) Computing function summaries is an orthogonal and well-studied task with well-known solutions [Esparza et al. 2010; Reps et al. 2017], as well as analysis-specific worklist algorithms [Reps et al. 1995], and (ii) Even a naïve algorithm that computes a new regular expression r for every query can still reuse the summaries. Hence, even without our contributions in this work, the summaries never had to be computed more than once. Thus, the challenge is in speeding up Steps (2) and (3).

Motivation for On-Demand Analysis. On-demand analyses are quite common in the static analysis literature. For example, Babich and Jazayeri [1978]; Chatterjee et al. [2020]; Duesterwald et al. [1995]; Goharshady and Zaher [2023]; Horwitz et al. [1995]; Sridharan et al. [2005]; Yan et al. [2011]; Zheng and Rugina [2008] and Reps [1993] are some of the works that consider an on-demand variant of an analysis that can be modeled in the APA framework. Thus, our work can be seen as their unification and extension. To quote from Chatterjee et al. [2020] and Reps [1993]: on-demand analyses are especially important for just-in-time compilers and their speculative optimizations. They also reduce the overall runtime of the analysis by (i) potentially reusing the information obtained in previous queries to speed up future queries, (ii) narrowing down the focus to specific points or facts of interest, (iii) reducing the work in preliminary phases by avoiding an exhaustive computation of all query results in preprocessing, (iv) side-stepping incremental updating problems, and (v) offering the analysis as a user-level operation that can be used by the programmers when debugging. Note that during debugging, the programmer is not interested in the result of an analysis whose startpoint s is the beginning of the program, but would rather have an on-demand analysis starting at the current point.

Structrual Sparsity Parameters. The main high-level insight that allows us to speed up on-demand APA is to recognize that control-flow and call graphs (CFGs and CGs) of programs are not arbitrarily complex. Instead, they have many natural structural sparsity properties that can be exploited to obtain faster algorithms. Indeed, the classical algorithm of Tarjan [1981a] which is the basis of modern APA is itself a faster version of Kleene's NFA-to-regex translation [Kleene 1956]

and the speedup was achieved by exploiting the fact that Gaussian elimination can be performed more efficiently on reducible flow graphs. Unfortunately, the assumptions of Tarjan [1981a] are not enough for our case. Instead, we consider several structural sparsity parameters for both CFGs and CGs. More specifically, we rely on nesting depth or treewidth for CFGs, and treedepth for CGs. These parameters are formally defined in Section 3, and the next paragraphs provide an intuitive summary of the process.

Sketch of Intraprocedural Results. We first start with intraprocedural on-demand APA and show how to make it more efficient assuming that the analyzed program P is structured and has a bounded nesting depth. While most real-world programs have a small nesting depth, one can of course write structured programs with an arbitrary depth. Thus, we then extend our approach to graphs with bounded treewidth. Treewidth [Robertson and Seymour 1986] is one of the most commonly-used parameters in static analysis and model checking [Aiswarya 2022]. It is a measure of tree-likeness of graphs. Intuitively, a graph with treewidth w can be decomposed into parts of size at most $w + 1$ that are connected to each other in a tree-like manner. This is called a tree decomposition. It is well-known that structured programs have CFGs with a treewidth of at most 7 [Thorup 1998].

Sketch of Interprocedural Results. We then turn our attention to the more challenging problem of interprocedural APA. To handle interprocedural queries and speed up on-demand APA, we have to consider not only the CFGs but also the CG. Moreover, we cannot simply apply the interprocedural-to-intraprocedural reduction mentioned above since this leads to a graph that does not share the structural properties of the original CFGs, i.e. the treewidth can get arbitrarily large. Thus, the reduction above is not applicable to the on-demand setting. To overcome this challenge, we exploit two other graph sparsity parameters over the CGs. The first parameter is the number of call sites in each function and the second parameter is the CG's treedepth. Treedepth is a cousin of treewidth that, intuitively, models the extent to which a graph looks like a shallow tree.

Small Treedepth Assumption. The speedups we obtain for interprocedural APA are reliant on the assumption that the treedepth of the call graph is relatively small in comparison to the size of the program. There is a recent work [Goharshady and Zaher 2023] that establishes the small-treedepth property experimentally. In our experimental results, we observed the same phenomenon, i.e. the treedepth was no more than 133 for standard benchmarks with more than a hundred thousand nodes. Intuitively, call graphs of real-world programs are expected to have small treedepth, in comparison to the size of the program, because most functions only call a small subset of previously defined functions, so the call graph hardly forms large cliques or grids, which are necessary for a large treedepth. Even a long chain of *distinct* non-recursive functions $f_1, f_2, \ldots, f_m$ such that each f_i calls f_{i+1} would only have a treedepth of $O(\log m)$. Note that we exploit the treedepth of the call graph without any limits on the size of the function-call stack, which can grow arbitrarily large by recursion. Functions that call themselves recursively have no impact on this treedepth. Finally, we note that it is theoretically possible to create adversarial programs whose call graphs have arbitrarily large treedepth. For example, consider a program with m functions in which every function calls every other function. The call graph of this program is a complete graph with treedepth $m - 1$. We believe such programs are not realistic.

Our Contributions. Based on the discussion above, our specific contributions are as follows:

- ***Intraprocedural APA***
  - *Exploiting Nesting Depth:* Assuming the CFG has bounded nesting depth, we provide an algorithm for on-demand intraprocedural APA that takes $O(n \cdot \log\log n \cdot k)$ time in preprocessing and answers each APA query in $O(k)$. Recall that n is the number of lines of code in the program and k is the time needed to evaluate an atomic operation in the algebra.

This algorithm has a very similar runtime to classical APA even on one query, differing only by a factor of $\log\log(n)/\alpha(n)$. The improvements get much more pronounced as the number of queries grows. Our algorithm spends only $O(k)$ time per each extra query, whereas classical APA takes $O(n \cdot \alpha(n) \cdot k)$.
  - *Exploiting Treewidth:* Assuming the CFG has bounded treewidth, we provide another algorithm for on-demand APA that takes $O(n \cdot \log n \cdot k)$ time in preprocessing and then answers each APA query in $O(k)$. Again, the improvements over classical APA are huge. Our preprocessing is slower than a single APA only by a sub-logarithmic factor of $\log n/\alpha(n)$. This is more than compensated for by our much faster query time. We spend only $O(k)$ to answer each query, whereas the classical baseline takes $O(n \cdot \alpha(n) \cdot k)$.
- ***Interprocedural APA***
  - *Exploiting Treedepth:* Assuming that the CG has small treedepth and that there is a constant bound on the number of function call sites in each function, we present an algorithm for interprocedural on-demand APA that matches our intraprocedural runtime bounds and provides the exact same improvements over classical APA. Specifically, our algorithm spends either $O(n \cdot \log n \cdot k)$ or $O(n \cdot \log\log n \cdot k + m \cdot \log m \cdot k)$ time in preprocessing, where m is the number of functions, and then answers each interprocedural APA query in $O(k)$. Our runtime dependence on the treedepth is cubic.
- ***Experimental Results.*** Finally, we provide experimental results over real-world programs from the literature, demonstrating the effectiveness of our approach and its significant gains in efficiency, beating the classical APA's runtime by several orders of magnitude.

Novelty. To the best of our knowledge, this is the first work to obtain faster runtimes for on-demand APA in comparison with repeated application of classical APA. While similar results exist for special cases of APA, as mentioned further below, none of the previous works were able to handle general APA in an on-demand setting. Thus, our approach can be seen as a unification and extension of all previous on-demand algorithms for special cases of APA, such as data-flow. In terms of algorithmic ideas, the techniques used in our nesting-depth-based algorithm are entirely novel. More importantly, our algorithm for treewidth-based APA uses an elegant combination of tree decompositions and centroid decompositions, which was not known before and helps avoid the treewidth blowup present in prior treewidth-based static analyses in the literature.

Related Works. We consider several families of related works:

- *Treewidth-based Model Checking.* Treewidth is arguably the most commonly used graph parameter in static analysis and model checking. It was initially used for the problem of register allocation in compiler optimization [Thorup 1998], but soon found more applications, notably in model checking when specifications are given in the monadic second order logic of graphs [Borie et al. 1992; Courcelle 1990; Kneis and Langer 2008], μ-calculus [Obdrzálek 2003], local restrictions of LTL [Ferrara et al. 2005] or Datalog and finite-variable logics [Dalmau et al. 2002]. Treewidth has also been used in the analysis of Markov chains and decision processes and to compute quantitative properties such as mean payoff [Asadi et al. 2020; Chatterjee et al. 2021]. Recently, the work Conrado et al. [2023] showed that CFGs have bounded pathwidth, too. See Aiswarya [2022] for a survey of treewidth-based results in verification.
- *On-demand Algorithms for Special Cases of APA.* Standard formulations of data-flow analysis are special cases of APA. This holds both for the intraprocedural analyses of Kildall [1973] and the interprocedural (IFDS) framework of Reps et al. [1995]. There has been extensive research focused on producing on-demand variants of these data-flow analyses [Babich and Jazayeri 1978; Duesterwald et al. 1995; Horwitz et al. 1995; Reps 1993]. While these works

provide significant practical gains in efficiency, their only theoretical guarantee is that of same worst-case runtime, i.e. they are guaranteed not to use more time than a repeated application of classical data-flow. Chatterjee et al. [2020] provided the first on-demand interprocedural data-flow analysis with a theoretically-improved runtime bound. However, it was severely limited and could only handle *same-context* interprocedural queries, i.e. queries limited to paths that leave the function-call stack unchanged. Notably, it used treewidth as a parameter. Thus, our work can be seen as a significant extension of Chatterjee et al. [2020] which (a) can handle general APA instead of just data-flow, and (b) is not limited to same-context queries. If applied to data-flow analysis, our algorithms achieve the same complexity as Chatterjee et al. [2020]. Another recent work in this direction is Goharshady and Zaher [2023], which also uses treedepth for IFDS analysis. In comparison, we handle the more general case of APA, instead of IFDS, and obtain the same runtime bounds. There are also on-demand approaches to other special cases of APA such as alias and points-to analyses [Sridharan et al. 2005; Yan et al. 2011; Zheng and Rugina 2008]. However, these directions are not as well-studied as on-demand data-flow. Finally, the APA-based recurrence analyses in Kincaid et al. [2017] have a compositional nature that makes them amenable to on-demand settings.

- *Semiring-based Program Analysis.* Perhaps the closest works to ours are Chatterjee et al. [2019b, 2016, 2017, 2018]. These works consider a different variant of APA in which summaries are taken from a semiring $(A, \oplus, \otimes, \overline{0}, \overline{1})$ instead of a Kleene/regular algebra. Note the absence of the Kleene star operator $\circledast$. They then use parameterization by treewidth to obtain efficient on-demand algorithms. Thus, it is fair to say we have been inspired by them as we are exploiting the same parameter. However, these approaches have important limitations that are absent in our work: (a) much like Chatterjee et al. [2020] they can only handle same-context queries, and (b) they fix a constant bound h on the stack height, whereas we consider all valid paths with no limit on the height of the function-call stack. On the other hand, Chatterjee et al. [2018] handles concurrent programs whereas our setting is single-threaded. These being said, the most important difference between our work and Chatterjee et al. [2019b, 2016, 2020, 2018] is that we use an entirely different and novel treewidth-based algorithm. The algorithms in these prior works require *balanced* tree decompositions that are in turn obtained from Elberfeld et al. [2010] and Bodlaender and Hagerup [1998] at the expense of a linear blowup in the treewidth. We sidestep balancing using centroid decompositions.

2 ALGEBRAIC PROGRAM ANALYSIS

In this section, we formally define our notation and the algebraic program analysis (APA) problem in both intraprocedural and interprocedural settings.

Programs, Functions and CFGs. We define our programs in a manner similar to Reps et al. [1995]. A *program* P is a collection $P = \{f_1, f_2, \ldots, f_m\}$ of *functions*. Each function f_i is defined by a *control-flow graph* (CFG) $G_i = (V_i, E_i, \top_i, \bot_i, C_i, R_i, \Phi_i)$ in which:

- V_i is a finite set of vertices, $\top_i \in V_i$ is a distinguished *start* vertex and $\bot_i \in V_i$ is a distinguished *end* vertex. Intuitively, each vertex corresponds to one line of the program, except that function calls are broken into two lines as described below. $\top_i$ corresponds to the first line of f_i and $\bot_i$ to its last line.
- $E_i \subseteq V_i \times V_i$ is a finite set of directed edges, modeling the flow of control in the program.
- $C_i \subseteq V_i$ is a (potentially empty) set of *call site* vertices and is in one-to-one correspondence with $R_i \subseteq V_i$ which is the set of *return site* vertices. In other words, every $c \in C_i$ has a corresponding $r_c \in R_i$. There is an edge from c to r_c, which is the only outgoing edge of c and the only incoming edge of r_c. The sets C_i and R_i are disjoint.

```
1  void f(int *p, int *&q){
2   unsigned int temp; cin>>temp;
3   while(temp--)
4    swap(p, q);
5  }
6  int main(){
7   int *p, *q;
8   p = new int(100);
9   f(p, q);
10  return *q;
11 }
```

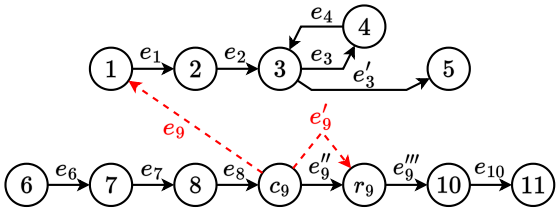


Fig. 1. A C++ Program (left) and its CFGs (right). The added edges in the augmented graph are dashed in red.

- $\Phi_i : C_i \to \{f_1, \ldots, f_m\}$ assigns a function to each call site. Intuitively, when the program reaches $c \in C_i$, it calls the function $\Phi_i(c)$, and when that function's execution ends, control returns back to the corresponding return site r_c.

We let $V := \bigcup_{i=1}^{m} V_i$ be the set of all vertices in the entire program and define G, E, C, R, and Φ analogously. A *path* in G is a sequence $e_1 = (v_1, v_2), e_2 = (v_2, v_3), \ldots, e_k = (v_k, v_{k+1})$ of edges in E where each edge starts at the endpoint of the previous edge.

Example. Figure 1 shows a program written in C++ and its CFGs (black edges).

Regular Expressions. The set of *regular expressions* ρ over the graph G is defined using the following grammar:

$$\rho ::= e \mid \varnothing \mid \epsilon \mid \rho + \rho \mid \rho \cdot \rho \mid \rho^* \qquad e \in E$$

We often use parentheses when writing regular expressions and otherwise assume that * takes precedence over $\cdot$ which in turn precedes $+$. We use ρ^k as syntactic sugar defined by $\rho^0 := \epsilon$ and $\rho^k := \rho^{k-1} \cdot \rho$ for $k \geq 1$. Every regular expression ρ defines a set $\langle \rho \rangle$ of strings over E :

$$\langle e \rangle := \{e\} \qquad \langle \varnothing \rangle := \varnothing \qquad \langle \epsilon \rangle := \{\epsilon\} \qquad \langle \rho_1 + \rho_2 \rangle := \langle \rho_1 \rangle \cup \langle \rho_2 \rangle$$

$$\langle \rho_1 \cdot \rho_2 \rangle := \{xy \mid x \in \langle \rho_1 \rangle \wedge y \in \langle \rho_2 \rangle\} \qquad \langle \rho^* \rangle := \bigcup_{k=0}^{\infty} \langle \rho^k \rangle$$

Here, ϵ is the empty string.

Path Expressions [Tarjan 1981b]. Not every string of edges appearing in $\langle \rho \rangle$ is necessarily a path in the graph G. We say that ρ is a *path expression* of type (s, t) if every string appearing in $\langle \rho \rangle$ is a path from s to t in G. Furthermore, we say ρ is an (s, t)-*summary* if $\langle \rho \rangle$ is exactly the set of all paths from s to t in G. The work Tarjan [1981a] provides an algorithm that computes an (s, t)-summary in time $O(n \cdot \alpha(n))$ where n is the number of vertices in the CFG.

Example. In Figure 1, we can compute a $(1, 5)$-summary $\rho = (e_1 \cdot e_2) \cdot (e_3 \cdot e_4)^* \cdot e_3'$.

Regular Algebra. A *regular algebra* $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$ consists of a non-empty *universe* set A, two distinguished elements $\overline{0}, \overline{1} \in A$, together with two binary operations $\oplus : A \times A \to A$ (known as *choice*, *branching* or *addition*) and $\otimes : A \times A \to A$ (known as *sequencing*, *concatenation* or *multiplication*), as well as a unary operation $\circledast : A \to A$ (known as *iteration* or *Kleene star*). With a slight misuse of notation, we do not distinguish between the algebra and its universe and use A to denote both. The precedence of operators is $\circledast > \otimes > \oplus$.

(Re)interpretation. Let $[\![.]\!] : E \to A$ be a *semantic function* that assigns an algebra element $[\![e]\!] \in A$ to each edge $e \in E$. Additionally, let ρ be a regular expression as above. The $[\![.]\!]$-*interpretation* or

reinterpretation of ρ in A is denoted by $[\![\rho]\!]$ and defined as follows:

$$[\![e]\!] := [\![e]\!] \quad [\![\varnothing]\!] := \overline{0} \quad [\![\epsilon]\!] := \overline{1} \quad [\![\rho_1 + \rho_2]\!] := [\![\rho_1]\!] \oplus [\![\rho_2]\!] \quad [\![\rho_1 \cdot \rho_2]\!] := [\![\rho_1]\!] \otimes [\![\rho_2]\!] \quad [\![\rho^*]\!] := [\![\rho]\!]^{\circledast}.$$

Example. Suppose our goal is to do a null-pointer analysis on the program of Figure 1. Let $D = \{\overline{p}, \overline{q}\}$ where $\overline{x}$ models the fact that x might be null. Let A be the set of data-flow transformer functions of the form $2^D \to 2^D$, mapping the set of facts before the execution of a program fragment to those that might hold afterwards. In this example, we have $[\![e_7]\!] := \lambda X.\ D$, $[\![e_8]\!] := \lambda X.\ X - \{\overline{p}\}$ and

$$[\![e_4]\!] := \lambda X.\ \ (\overline{q} \in X\ ?\ \{\overline{p}\} : \varnothing) \cup (\overline{p} \in X\ ?\ \{\overline{q}\} : \varnothing)\,.$$

For every other black edge e, its interpretation $[\![e]\!]$ is the identity function $\lambda X.\ X$. We define the addition operation as union and the multiplication as function composition. The iteration operator leads to the fixed-point that is the union of all compositions. More formally,

$$a \oplus b := \lambda X.\ a(X) \cup b(X) \qquad a \otimes b := \lambda X.\ b(a(X)) \qquad a^{\circledast} := \lambda X.\ \cup_{k=0}^{\infty} a^k(X)$$

Recall that we already know a $(1, 5)$-summary $\rho = (e_1 \cdot e_2) \cdot (e_3 \cdot e_4)^* \cdot e_3'$. Let's reinterpret ρ in our algebra A. We have $[\![\rho]\!] = ([\![e_1]\!] \otimes [\![e_2]\!]) \otimes ([\![e_3]\!] \otimes [\![e_4]\!])^{\circledast} \otimes [\![e_3']\!] = (\lambda X.\ X)^2 \otimes (\lambda X.\ X \otimes [\![e_4]\!])^{\circledast} \otimes \lambda X.\ X = [\![e_4]\!]^{\circledast} = \lambda X.\ (X = \varnothing\ ?\ X : D)$. In other words, $[\![\rho]\!]$ is a data-flow transformer function that tells us the following: if none of p and q are null at the beginning of line 1, then none of them will be null at the end of line 5. However, even if one of them might be null at 1, then either of them might be null at 5.

Intraprocedural APA. Our first APA algorithm considers the case of a single function. Our input has the following parts:

- A regular algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$;
- A program P consisting of a single function f with no function call vertices;
- A semantic function $[\![.]\!] : E \to A$ mapping each edge to an element of the algebra.

Each APA query provides two vertices $s, t \in V$. The analysis should respond to this query by outputting a single element $a \in A$ where $a = [\![\rho]\!]$ for an (s, t)-summary ρ.

Path Equivalence. We say two elements $a, b \in A$ are *path-equivalent* and write $a \equiv b$ if there exist two path expressions ρ_a, ρ_b such that $[\![\rho_a]\!] = a$, $[\![\rho_b]\!] = b$ and $\langle \rho_a \rangle = \langle \rho_b \rangle$. Note that a query might not have a unique answer as different (s, t)-summaries might lead to different interpretations in A but all answers are path-equivalent. The answer is unique if A is a Kleene algebra (defined below). Although having more than one possible answer is technically allowed and our algorithms can handle it, this usually happens only when APA is combined with abstract interpretation [Kincaid et al. 2021]. In these cases, we normally have a concrete algebra that is Kleene and an abstract one which is not. Moreover, any abstract solution in A is equally acceptable. See Kincaid et al. [2021] and Tarjan [1981b] for examples of such analyses and more discussion.

Kleene Algebra [Kozen 1990]. In an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$, we say $a \prec b$ if $a \oplus b = b$. The algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$ is a *Kleene algebra* if it satisfies the following requirements for all $a, b, c \in A$:

- *Associativity:* $a \oplus (b \oplus c) = (a \oplus b) \oplus c$ and $a \otimes (b \otimes c) = (a \otimes b) \otimes c$;
- *Commutativity of Addition:* $a \oplus b = b \oplus a$;
- *Distributivity:* $a \otimes (b \oplus c) = a \otimes b \oplus a \otimes c$ and $(b \oplus c) \otimes a = b \otimes a \oplus c \otimes a$;
- *Identity Elements:* $a \oplus \overline{0} = \overline{0} \oplus a = a$ and $a \otimes \overline{1} = \overline{1} \otimes a = a$;
- *Idempotence:* $a \oplus a = a$;
- *Annihilation:* $a \otimes \overline{0} = \overline{0} \otimes a = \overline{0}$;
- As a result of the previous requirements, $\prec$ is a partial order on A. We further require:
  - *Unfolding:* $\overline{1} \oplus a \otimes a^{\circledast} \prec a^{\circledast}$ and $\overline{1} \oplus a^{\circledast} \otimes a \prec a^{\circledast}$.
  - *Induction:* $a \otimes b \prec b \Rightarrow a^{\circledast} \otimes b \prec b$ and $b \otimes a \prec b \Rightarrow b \otimes a^{\circledast} \prec b$.

Note that the algebra of regular expressions with union, concatenation and Kleene iteration is a Kleene algebra. Many classical static analyses, such as data-flow [Kildall 1973; Reps et al. 1995], use program summaries that form a Kleene algebra. Kleene algebras also satisfy the following desirable property: If ρ_1 and ρ_2 are two (s,t)-summaries and we reinterpret them in a Kleene algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$ using the same semantic function, we get $[\![\rho_1]\!] = [\![\rho_2]\!]$. In other words, if we use a Kleene algebra in our APA, then we always get the same analysis result no matter what path expression is used to summarize all the paths from s to t, and thus the answer to every query is unique. Nevertheless, our algorithms do not assume that $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$ is a Kleene algebra and are applicable to any regular algebra. In some steps of the algorithms, it might seem like we are assuming A is a Kleene algebra, e.g. we do not put parentheses when multiplying a series of elements, but this is not the case. Instead, we mean to emphasize the fact that multiplication order does not matter since any path-equivalent answer is acceptable and regular expressions form a Kleene algebra. We now turn to the interprocedural case of APA.

Function Summaries. A *function summarizer* is a function $[\![.]\!] : \{f_1, \ldots, f_m\} \to A$ that assigns an element $[\![f_i]\!]$ of the algebra A to every function f_i in P. We call $[\![f_i]\!]$ the *summary* of f_i. Intuitively, $[\![f_i]\!]$ models the effect of executing f_i, including the actions taken by other functions that are recursively called within f_i. In practice, such summaries are computed either by the Newtonian program analysis approach of Esparza et al. [2010] or its recent extension with tensor products [Reps et al. 2017]. For some analyses, there are also faster dedicated summarization procedures such as the worklist algorithm of Reps et al. [1995] for data-flow.

Example. In Figure 1, since f does not call any other functions, any $(1,5)$-summary captures all possible execution paths of f. We already have a $(1,5)$-summary ρ. However, note that in the call to f at line 9, the variable p is passed by value but q is passed by reference. Thus, q is the same variable in both the main function and f, but each function has its own p. Hence, $[\![f]\!]$ cannot affect $\overline{p}$ in the main function. In other words, we have $[\![f]\!] := \lambda X.\ ([\![\rho]\!](X) - \{\overline{p}\}) \cup (X \cap \{\overline{p}\}) = \lambda X.\ (X = \varnothing\ ?\ \varnothing : \{\overline{q}\}) \cup (X \cap \{\overline{p}\})$. Simply put, we summarized the effects of f as follows: f does not affect the nullity of p. If either p or q is null when the call to f is made, then q might be null at the end of f's execution. Otherwise, q is guaranteed to be non-null after f's execution.

Augmented Graph [Reps et al. 1995]. Given a program P consisting of functions $f_1, \ldots, f_m$, we define the *augmented graph* $\widehat{G} = (V, \widehat{E}) := (V, E \cup E_C \cup E_R)$, where E_C is a set of *interprocedural call edges* and E_R is a set of *intraprocedural call-to-return* edges. For every call vertex $c \in C$, there is an edge $e_c \in E_C$ and another edge $e'_c \in E_R$. Suppose that $\Phi(c) = f_i$, i.e. c calls the function f_i, then e_c would be an edge from c to $\top_i$ and e'_c a new edge from c to its corresponding return site r_c.

Extending the Semantic Function. Given that we added new edges to our graph, we have to extend our semantics. The edge $e'_c = (c, r_c)$ is supposed to summarize the effect of the function call. Thus, we set $[\![e'_c]\!] := [\![(c, \top_i)]\!] \otimes [\![f_i]\!] \otimes [\![(\bot_i, r_c)]\!]$. The edge $e_c = (c, \top_i)$ should flow information from the caller function (point c) to the callee f_i. Thus $[\![e_c]\!]$ is defined based on the particular APA application. Similarly, the remaining edge between c and r_c should be used to transfer local information.

Intuition. The paths in the augmented graph $\widehat{G}$ correspond to interprocedurally valid execution paths in P, i.e. paths that return to the correct callee when a called function's execution ends. To see this, consider an execution path of the program P that reaches point c and makes a call to f_i. There are two cases: Either this call to f_i returns, in which case the entire execution of f_i is summarized by $[\![e'_c]\!] = [\![f_i]\!]$, or the analysis endpoint t is reached before f_i returns, in which case we can take the edge e_c. The subtle point here is that there is no edge from the endpoint of f_i back to either c or r_c, thus a path that takes e_c is committing not to return from f_i's execution.

Example. Figure 1 shows the additional edges in the augmented graph by dashed red lines. Since e_9 simply passes p and q to f, we have $[\![e_9]\!] := \lambda X.\ X$. However, e'_9 is supposed to short-circuit an execution of f and thus we set $[\![e'_9]\!] := [\![f]\!]$. Moreover, the edge e''_9 should pass the information about nullity of p from c_9 to r_9. Thus, we set $[\![e''_9]\!] := \lambda X.\ X \cap \{\overline{p}\}$. Suppose our goal is to see which variables might be null at line 10 assuming the program starts at line 6. We compute a $(6, 10)$-summary regular expression in $\widehat{G}$ which is simply $e_6 \cdot e_7 \cdot e_8 \cdot (e'_9 + e''_9) \cdot e'''_9$. We then interpret it in A to obtain $\lambda X.\ X \otimes \lambda X.\ D \otimes \lambda X.\ X - \{\overline{p}\} \otimes ([\![e'_9]\!] \oplus \lambda X.\ X \cap \{\overline{p}\}) \otimes \lambda X.\ X$. After a tedious calculation, the result is $\lambda X.\ \{\overline{q}\}$, meaning that no matter which combination of variables are null at line 6, the variable q might be null at line 10.

Interprocedural APA. Based on the intuition above, the interprocedural variant of on-demand APA has an input consisting of the following parts:

- A regular algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$;
- A program $P = \{f_1, f_2, \ldots, f_m\}$ together with a summary $[\![f_i]\!] \in A$ for each function $f_i \in P$;
- A semantic function $[\![.]\!] : \widehat{E} \to A$ that maps each edge of the augmented graph $\widehat{G}$ to an element of the algebra.

Each interprocedural APA query provides two vertices $s, t \in V$. The analysis should respond by outputting an element $a \in A$ where $a = [\![\rho]\!]$ for an (s, t)-summary ρ over the augmented graph $\widehat{G}$.

Call Graph (CG). The *call graph* of P is a directed graph $H = (\{f_1, \ldots, f_m\}, E_H)$ having one vertex for each function in P and an edge (f_i, f_j) if the function f_i has a direct call to f_j, i.e. $(f_i, f_j) \in E_H$ iff $\exists c \in V_i \quad \Phi(c) = f_j$. While control-flow graphs G_i have nice structural sparsity properties that we will exploit for faster on-demand APA, these properties are not preserved by the augmented graph $\widehat{G}$. Thus, our algorithms instead work with the call graph H and the G_i's. In our example, the call graph has a single edge from main to f.

3 GRAPH SPARSITY PARAMETERS AND DECOMPOSITIONS

We now define the graph structural sparsity parameters that will be used in our algorithms. Throughout this section, suppose that a graph $G = (V, E)$ is fixed. The parameters and decompositions defined below assume an undirected G. To apply them to a directed graph, we simply ignore the directions of the edges. Thus, every graph in the rest of this section is undirected.

Depth Decompositions [Nesetril and de Mendez 2006]. A *depth decomposition* of G is a tree / forest $T = (V, E_T)$ on the same vertex set as G such that every edge $e \in E$ of G connects a vertex to one of its ancestors or descendants in T. Let A_v be the set of ancestors of v in T. If there is a path from u to v in G, then it is straightforward to see that it has to visit $A_u \cap A_v$, since we should start from u and go to an ancestor/descendant each time, until we reach v. Thus, the highest internal vertex has to be an ancestor of both u and v. Intuitively, $A_u \cap A_v$ is a small cutset in G that separates u from v. Figure 2 (center) shows a depth decomposition of the left graph. Edges of the original graph are traced in dotted red lines to show that they go between a vertex and an ancestor/descendant in the tree.

Treedepth [Nesetril and de Mendez 2006]. The *treedepth* of a graph is defined as the smallest depth among all its depth decompositions. Intuitively, treedepth is a measure of how much a given graph resembles a star or a shallow tree. It is experimentally seen that call graphs of real-world programs have small treedepth [Goharshady and Zaher 2023]. Intuitively, this is because the functions of a program are typically implemented in a chronological order and each function calls a small number of those implemented before it and potentially also a few that are implemented after it (in case of non-simple recursion) [Goharshady and Zaher 2023]. For any fixed constant d, there is an algorithm that decides whether an input graph has treedepth at most d in linear time and also

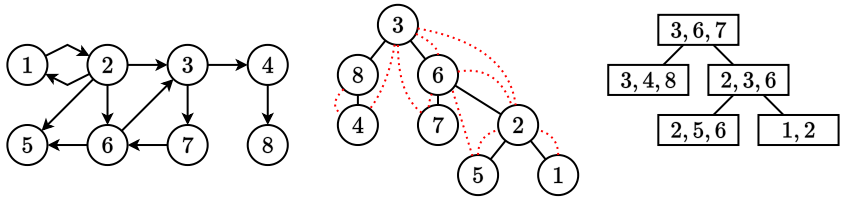


Fig. 2. A graph $G = (V, E)$ (left), a depth decomposition of G (center) and a tree decomposition of G (right).

produces the witnessing depth decomposition if the answer is positive. Moreover, the treedepth of a graph G is always greater than or equal to its treewidth [Nesetril and de Mendez 2006].

Tree Decompositions [Robertson and Seymour 1986]. A *tree decomposition* of G is a tree $T = (\mathcal{B}, E_T)$ such that:

- Each node $b \in \mathcal{B}$ of T is called a *bag* and contains a set $V_b \subseteq V$ of the vertices of G.
- The bags cover all vertices of G, i.e. $\bigcup_{b\in\mathcal{B}} V_b = V$.
- For every edge $(u, v) \in E$, there is a bag $b \in \mathcal{B}$ that contains both endpoints, i.e. $u, v \in V_b$.
- Every vertex $v \in V$ appears in a connected subtree of T. Formally, suppose b_3 is a bag on the unique path from b_1 to b_2 in T. Then, every vertex that appears in both b_1 and b_2 has to appear in b_3, as well, i.e. $V_{b_1} \cap V_{b_2} \subseteq V_{b_3}$.

The *width* of a tree decomposition is the size of its largest bag minus 1, i.e. $w(T) := \max_{b\in\mathcal{B}} |V_b| - 1$. Figure 2 (right) shows a tree decomposition of the graph on the left with width 2. Much like depth decompositions, tree decompositions also allow us to find small cuts. If there is a path π from u to v in G and if u appears in a bag b_u and v in b_v in T, then π has to intersect every bag that appears on the unique path from b_u to b_v in T. Thus, if b and b' are two neighboring bags in T, their intersection $V_b \cap V_{b'}$ is a cut in G [Cygan et al. 2015, Chapter 7]. This is called the *cut property* of tree decompositions.

Treewidth [Robertson and Seymour 1986]. The *treewidth* of G is the smallest width among all of its tree decompositions. Intuitively, a graph with treewidth w can be decomposed into smaller parts by repeatedly finding cuts of size w and removing them. Another intuition is that it can be decomposed into small parts (bags) of size at most $w + 1$ that are themselves connected in a tree-like manner. It is well-known that control-flow graphs of structured `goto`-free programs have a treewidth of at most 7 [Chatterjee et al. 2019a; Gustedt et al. 2002; Thorup 1998]. This result comes with an algorithm that produces the corresponding tree decompositions in linear time. Moreover, each `goto` statement can increase the treewidth by at most one.

Motivation for Decompositions. The main algorithmic importance of treedepth and treewidth is that they enable dynamic programming approaches over the respective decompositions [Bodlaender 1988]. Thus, many efficient algorithms designed for trees can be extended to graphs with bounded treedepth/treewidth, potentially by incurring an exponential runtime dependence on the parameter. Therefore, a large number of NP-hard problems become tractable when limited to the families of sparse graphs with bounded treewidth/treedepth [Ahmadi et al. 2022b; Chatterjee et al. 2019c; Cygan et al. 2015; Goharshady and Mohammadi 2020; Meybodi et al. 2022; Nesetril and de Mendez 2006; Niedermeier 2004] and even problems with PTIME solutions can achieve an improved runtime over these sparse graphs [Ahmadi et al. 2022a; Asadi et al. 2020; Fomin et al. 2018]. In our case, since it is known that CFGs and CGs have bounded treewidth and treedepth, respectively, these are the ideal parameters to exploit for efficient APA.

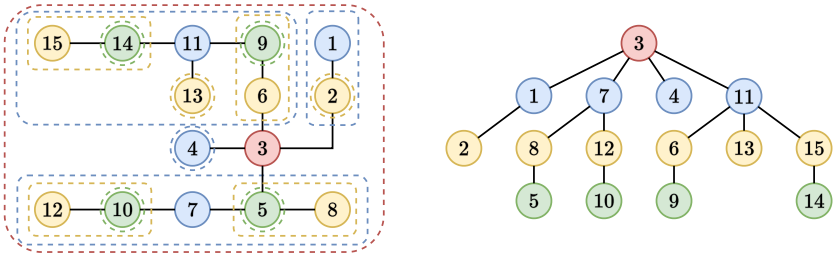


Fig. 3. A tree T (left) and a centroid decomposition T' of T (right) [Carpanese 2018].

Centroid [Jordan 1869]. In a tree $T = (V_T, E_T)$ with n vertices, a *centroid* is a vertex whose removal breaks the tree into connected components of size at most $n/2$ each. Every tree has at least one centroid.

Centroid Decompositions. Given a tree $T = (V_T, E_T)$, a *centroid decomposition* of T is a rooted tree T' on the same vertex set. We define T' recursively: First, find a centroid c of T and put it as the root of T'. Then, remove c from T to break it into connected components $T_1, T_2, \ldots, T_m$. Next, find a centroid decomposition T_i' of each T_i. Finally, connect all the T_i' decompositions together by making c the joint parent of all their roots. Figure 3 shows a tree and a centroid decomposition. The dotted regions show the subtrees T_i at each step. This figure is adapted from Carpanese [2018], which is an excellent tutorial on centroid decompositions. The centroid decomposition has a depth of $O(\lg n)$ since we are cutting the size of the tree in half or less each time we find a centroid. As in previous cases, we have a cut property in centroid decompositions, too: Suppose that u and v are two vertices in T and π is the unique path between them. Let l be the lowest common ancestor of u and v in T'. The path π is guaranteed to visit l. Finally, it is notable that a centroid decomposition can be computed in linear time [della Giustina et al. 2019].

4 INTRAPROCEDURAL ON-DEMAND APA

In this section, we provide two algorithms for intraprocedural on-demand APA, i.e. the case where the program is a single function with no recursive calls. Since we have only one function, we simplify our CFG notation to $G = (V, E, \top, \bot)$. Our first algorithm is more efficient and assumes we are given a structured program with loops and branches, but no arbitrary jumps (`goto` statements). It uses the nesting depth of the program as a parameter that is presumed to be small and obtains a preprocessing time of $O(n \cdot \log \log n \cdot k)$, after which it can answer each query in $O(k)$. This handles the vast majority of real-world programs as `goto` statements and large nesting depths are highly discouraged by programming style standards. However, there are also real-world programs that have large nesting depth or a few `goto` statements in each function. To handle these, we provide a second algorithm with a slightly higher preprocessing time of $O(n \cdot \log n \cdot k)$ which only assumes the treewidth of the CFG is bounded. It is well-known that a structured program with g `goto` statements has a treewidth of at most $7 + g$ [Thorup 1998] and that human-written programs never have a treewidth larger than 5 in practice [Gustedt et al. 2002].

4.1 Exploiting Nesting Depth

Overview. Our first algorithm exploits the fact that real-world programs usually have bounded nesting depth, which is the maximum number of nested statements in a block of code. At a high level, our algorithm consists of the following steps:

(1) *Linearization:* The algorithm recursively obtains summaries for each subprogram, e.g. each `while` loop. This allows us to intuitively flatten the program and focus on straight-line queries.
(2) *Construction of a Square-root Tree:* The algorithm constructs a data structure known as *square-root tree*, which helps us efficiently compute summaries of straight-line segments.
(3) *Answering an APA query:* We break each query into smaller parts, one for each nesting depth, and use the square-root tree to answer each part efficiently. Since the nesting depth is bounded, we have a constant number of parts.

We now formalize these ideas in more detail.

Structured Programs. A *simple program* is defined using the following grammar:

$$P := P; P \mid \mathtt{branch}_l\ P, P, \ldots, P\ \mathtt{end}_l \mid \mathtt{loop}_l\ P\ \mathtt{end}_l \mid \mathtt{break}_l \mid \mathtt{continue}_l \mid \sigma$$

Here, σ is an atomic operation that has no bearing on the control flow. `branch` captures multi-way branching caused by statements such as `if` and `switch` in common programming languages. We also have a generalized `loop` structure that can be instantiated to model `for` or `while` loops by assigning a suitable semantic meaning $[\![e]\!]$ to each edge e of the CFG. Finally, note that our `break` and `continue` statements are labeled and can apply to any enclosing loop. We call these programs simple since they have only one procedure/function.

CFGs. Given a simple program P we define its CFG $G_P = (V_P, E_P, \top_P, \bot_P)$ recursively as follows:

P	V_P	E_P	$\top_P$	$\bot_P$
σ	$\{\sigma\}$	$\varnothing$	σ	σ
$\mathtt{continue}_l$	$\{\mathtt{continue}_l\}$	$\varnothing$	$\mathtt{continue}_l$	$\mathtt{continue}_l$
$\mathtt{break}_l$	$\{\mathtt{break}_l\}$	$\varnothing$	$\mathtt{break}_l$	$\mathtt{break}_l$
$P_1; P_2$	$V_{P_1} \cup V_{P_2}$	$E_{P_1} \cup E_{P_2} \cup \{(\bot_{P_1}, \top_{P_2})\}$	$\top_{P_1}$	$\bot_{P_2}$
$\mathtt{branch}_l\ P_1, P_2, \ldots, P_m\ \mathtt{end}_l$	$\bigcup_{i=1}^{m} V_{P_i} \cup \{\mathtt{branch}_l, \mathtt{end}_l\}$	$\bigcup_{i=1}^{m} (E_{P_i} \cup \{(\mathtt{branch}_l, \top_{P_i}), (\bot_{P_i}, \mathtt{end}_l)\})$	$\mathtt{branch}_l$	$\mathtt{end}_l$
$\mathtt{loop}_l\ P' \mathtt{end}_l$	$V_{P'} \cup \{\mathtt{loop}_l, \mathtt{end}_l\}$	$E_{P'} \cup \{(\mathtt{loop}_l, \top_{P'}), (\bot_{P'}, \mathtt{loop}_l), (\mathtt{loop}_l, \mathtt{end}_l)\} \cup \{(v, \mathtt{loop}_l) \mid v \in V_{P'} \wedge v \equiv \mathtt{continue}_l\} \cup \{(v, \mathtt{end}_l) \mid v \in V_{P'} \wedge v \equiv \mathtt{break}_l\}$	$\mathtt{loop}_l$	$\mathtt{end}_l$

A pictorial representation is provided further below. Note that we distinguish between different instances of $\mathtt{break}_l$ or $\mathtt{continue}_l$, even when they apply to the same loop. Moreover, we assume that every loop label l has a constant number of corresponding break and continue statements.

Nesting Depth. We define the nesting depth $d(P)$ of a program P as the maximum number of nested loop and branch statements. More formally, $d(\sigma) := d(\mathtt{continue}_l) := d(\mathtt{break}_l) := 0$, $d(P_1; P_2) = \max\{d(P_1), d(P_2)\}$, $d(\mathtt{branch}_l\ P_1, P_2, \ldots, P_m\ \mathtt{end}_l) := 1 + \max_{i=1}^{m} d(P_i)$, and finally $d(\mathtt{loop}_l\ P' \mathtt{end}_l) := 1 + d(P')$. We assign a *level* $\lambda(v)$ to each vertex v of our CFG: a vertex appearing in an innermost block of code has a level of 0. A vertex that appears in a block which itself includes another block nested in it has a level of 1, and so on. Figure 4 shows an example Python program with the levels of each line.

In this section, we assume our CFG G_P is obtained using the definition above from a program P that has a constant depth $d(P) = \delta$. We will relax these restrictions in the next section. Recall that we assume every edge e of the CFG has a semantic meaning $[\![e]\!] \in A$. An APA query is of the form $q(G, s, t)$ and returns $[\![\rho]\!]$ for a path expression ρ that summarizes all paths from s to t in G. We are now ready to present our algorithm.

APA Preprocessing. Our preprocessing is structurally recursive. Before considering any program P, we first recursively preprocess all its maximal subprograms P' that have the smaller depth $d(P) - 1$. This enables us to make APA queries on each such P'. After all such subprograms are preprocessed, our algorithm performs the following steps on P :

```
x = 1                    2
while x < 100:           2
 x+=1                    1
 if x%2 == 1:            1
  x *= 3                 0
 x+=1                    1
y=10                     2
```

Fig. 4. An example program (left) and the level of each statement (right).

Preprocessing Step 1: Linearizing the CFG. In this step, our goal is to obtain a *linearized CFG* $L_P = (V_P^L, E_P^L)$ in which: (i) $V_P^L := \{v \in V_P \mid \lambda(v) = d(P)\}$ is the set of all vertices that appear in G_P but not in any $G_{P'}$, (ii) L_P is a union of disjoint directed paths, and (iii) for any two vertices $s, t \in V_P^L$, we have $q(L_P, s, t) \equiv q(G_P, s, t)$. Intuitively, L_P is a simplified version of G_P in which we have summarized all the loops and branches and thus our graph is simply a path. If we ever have to answer a query between two points in P that were not inside a nested loop or branch, we can perform this query in the linearized CFG L_P instead of the much larger original CFG G_P and we are guaranteed to obtain a path-equivalent result. To obtain L_P, we start with G_P and apply the following reductions (See Figure 5):

- *Branch Summarization:* If a branching operation $\mathsf{branch}_l\ P_1, P_2, \ldots, P_m\ \mathsf{end}_l$ appears directly in P, i.e. not nested within another loop or branch, then we perform the following operations for each P_i :
  - Compute $[\![\rho_i]\!] := q(G_{P_i}, \top_{P_i}, \bot_{P_i})$. Note that we can perform this APA query since P_i has already been preprocessed before P.
  - Add a new edge e_{P_i} from branch_l to end_l and set
$$[\![e_{P_i}]\!] := [\![(\mathsf{branch}_l, \top_{P_i})]\!] \otimes [\![\rho_i]\!] \otimes [\![(\bot_{P_i}, \mathsf{end}_l)]\!].$$
  - Delete all vertices in V_{P_i}.

  As an example, see Figure 5. The top part of the changes from the first graph to the second graph show the effects of branch summarization.
- *Loop Summarization:* Summarizing loops is a bit more complicated than branches due to the possible existence of break and continue statements. Let $\mathsf{loop}_l\ P'\ \mathsf{end}_l$ be a loop that appears directly in P, not nested within another loop or branch. Every iteration of the loop either (1) terminates normally and goes back to the header loop_l, or (2) reaches a $\mathsf{continue}_l$ statement and goes to loop_l, or (3) reaches a break_l statement and transitions to end_l. We have to add summaries for each of these cases separately. Thus, we do the following:
  - Compute $[\![\eta_1]\!] := q(G_{P'}, \top_{P'}, \bot_{P'})$ and add a self-loop $e^1_{P'}$ to loop_l with $[\![e^1_{P'}]\!] := [\![(\mathsf{loop}_l, \top_{P'})]\!] \otimes [\![\eta_1]\!] \otimes [\![(\bot_{P'}, \mathsf{loop}_l)]\!]$. This summarizes any iteration of type (1). Note that it does not add any information about iterations of type (2) and (3) above since any break_l or $\mathsf{continue}_l$ vertex is a dead-end in $G_{P'}$.
  - For every vertex $v \in V_{P'}$ that is a $\mathsf{continue}_l$, compute $[\![\eta^2_v]\!] := q(G_{P'}, \top_{P'}, v)$. Add a self-loop e^2_v from loop_l to itself and set $[\![e^2_v]\!] := [\![(\mathsf{loop}_l, \top_{P'})]\!] \otimes [\![\eta^2_v]\!] \otimes [\![(v, \mathsf{loop}_l)]\!]$. This summarizes an iteration of type (2).
  - Similarly, for each vertex $v \in V_{P'}$ that is a break_l, compute $[\![\eta^3_v]\!] := q(G_{P'}, \top_{P'}, v)$. Add an edge e^3_v from loop_l to end_l and set $[\![e^3_v]\!] := [\![(\mathsf{loop}_l, \top_{P'})]\!] \otimes [\![\eta^3_v]\!] \otimes [\![(v, \mathsf{end}_l)]\!]$.
  - Finally, delete all vertices in $V_{P'}$.

  The bottom part of the changes from the first graph to the second graph in Figure 5 is an example of loop summarization.

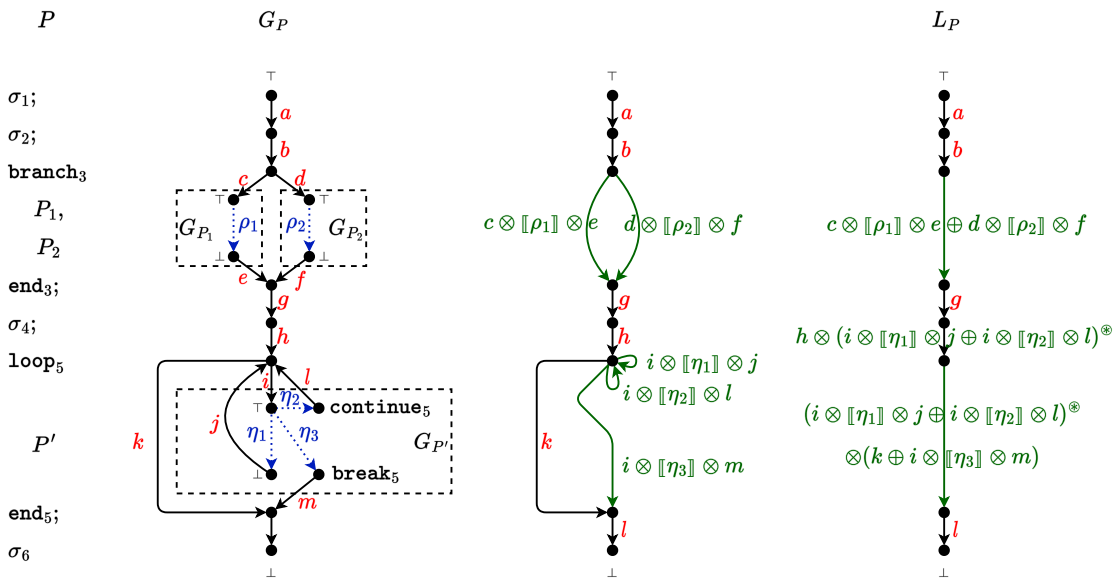


Fig. 5. The process of obtaining the linearized control-flow graph L_P. The letters in red are the semantics of the edges. Subgraphs are shown by dashed boxes. Blue letters are path expressions in the subgraphs. New edges are shown in green. Note that our algorithm is not symbolic and every green expression is actually evaluated to a single element in A which is saved as the semantics of the edge.

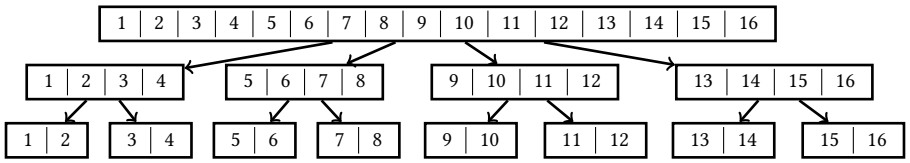


Fig. 6. A square-root tree for an array of size 16.

- *Eliminating Multi-edges and Self-loops:* Applying the summarizations above leads to a graph containing multiple edges between the same pair of vertices and self-loops, e.g. Figure 5 (middle graph). To achieve a linear CFG L_P we first eliminate multi-edges. If $e_1, e_2, \ldots, e_m$ are all the edges from a vertex u to another vertex v, we remove them and instead add a single new edge $e = (u, v)$ with $[\![e]\!] := [\![e_1]\!] \oplus [\![e_2]\!] \oplus \ldots \oplus [\![e_m]\!]$. Next, we have to eliminate self-loops. Let $e_1, e_2, \ldots, e_m$ be self-loops on vertex u. Take every incoming edge of the form $e' = (v, u)$ and replace $[\![e']\!]$ with $[\![e']\!] \otimes \chi_u$ where $\chi_u := ([\![e_1]\!] \oplus [\![e_2]\!] \oplus \ldots \oplus [\![e_m]\!])^{\circledast}$. Similarly, for every outgoing edge $e'' = (u, v)$ replace $[\![e'']\!]$ with $\chi_u \otimes [\![e'']\!]$. As an example, see the changes between the last two graphs in Figure 5. We also save χ_u for future use.

Since we always preserve path equivalence, the process above produces a linearized CFG L_P with the desired properties[†]

[†]For brevity, we only present the case where L_P is a connected path. Technically, it is possible that L_P is not a single path but instead a union of disjoint paths. This can happen if P contains a break or continue statement for an outer loop that encloses the entirety of P. In this case, our algorithm simply processes each connected component of L_P separately.

Preprocessing Step 2: Generating a Square-root Tree. Our linearized CFG L_P is a path. Let us denote it by $v_1, e_1, v_2, e_2, \ldots, e_m, v_{m+1}$. We know that if an APA query of the form $q(G_p, v_i, v_{j+1})$ is requested, we can answer it in L_p instead of G_p. More precisely, if $j+1 < i$, there is no path and the answer is $\overline{0}$. If $j+1 = i$, then the answer is simply χ_{v_i}, which is already computed, and if $j+1 > i$, the answer is

$$[\![e_i]\!] \otimes [\![e_{i+1}]\!] \otimes \ldots \otimes [\![e_j]\!]. \tag{1}$$

Thus, we would like to do some extra preprocessing so that we can later compute expressions of form (1) quickly. The classical solution is to create a segment tree [de Berg et al. 2000] on L_P. This problem can also be solved by the methods of Alon and Schieber [1987]. However, to gain more efficiency, we opt for a different data structure, called a square-root tree [Kernozhitsky et al. 2022]. This data structure is defined by a root node, which represents a segment of length n, and has $\sqrt{n}$ children, each representing consecutive segments of length $\sqrt{n}$ of the original array. The tree is then defined recursively for each of the children. This structure is exemplified in Figure 6 for a simple array. The steps taken to preprocess L_P are as follows:

- Ensure the number m of edges is a power of two by adding dummy edges with semantics $\overline{1}$ to the end of the path L_P. If $m \le 2$, stop.
- Compute the evaluation of all prefixes of the form $[\![e_1]\!] \otimes [\![e_2]\!] \otimes \ldots [\![e_j]\!]$ and all suffixes of the form $[\![e_i]\!] \otimes [\![e_{i+1}]\!] \otimes \ldots [\![e_m]\!]$.
- Let m' be the smallest power of two such that $m' \ge \sqrt{m}$. Break L_P into paths $L_P^1, L_P^2, \ldots, L_P^{m''}$ each of size m'.
- Compute the evaluation of all "big-step" expressions of the form $[\![L_P^i]\!] \otimes [\![L_P^{i+1}]\!] \otimes \ldots \otimes [\![L_P^j]\!]$. Note that there are only $O(m)$ such expressions.
- Create a tree in which each vertex models a path and the L_P^i's are children of L_P.
- Recursively process each subpath L_P^i and thus extend the tree.

This concludes our APA preprocessing.

Motivation for Square-root Tree. Intuitively, our goal is to effeciently compute expressions of the form (1), which correspond to a segment from index i to index j of an array. In a traditional segment tree, each node has two children and the subarrays assigned to the children are the left and right halves of the parent's subarray. This leads to a tree of depth $O(\log n)$. Thus, computing each expression of form (1) can take logarithmic time. In contrast, in a square-root tree, each node has $\Theta(\sqrt{m})$ children, where m is the size of the subarray assigned to this node. This ensures that the tree is shallower and has a depth of only $O(\log \log n)$. The lemma below formalizes this point:

Lemma 4.1. *Assuming an APA query on a preprocessed subprogram can be answered in $O(k)$, the entire APA preprocessing as described above takes $O(n \cdot \log \log n \cdot k)$ time.*

Proof. Since every statement in the CFG is summarized once, the linearization step makes $O(n)$ queries and performs an additional $O(n)$ operations in the algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$. Thus, this part takes $O(n \cdot k)$. Let $d(m)$ denote the depth of our tree when we have m elements. Our root has $O(\sqrt{m})$ children, each with $O(\sqrt{m})$ elements, but the number of children does not matter for the depth. Thus, we have $d(m) = 1 + d(\sqrt{m})$. If we define $l := \log m$ and $D(l) := d(2^l) = d(m)$, we get $d(m) = D(l) = 1 + D(l/2)$. The latter can be solved by Master's theorem leading to $D(l) \in O(\log l)$ which is equivalent to $d(m) \in O(\log \log m)$. So the square-root tree has a depth of $O(\log \log m)$. Let m_i denote the number of vertices represented by the root of the i-th square-root tree. Based on the discussion above, creating this square-root tree takes $O(m_i \cdot \log \log m_i \cdot k)$ time. However the sum of all m_i's is n. Thus, the total runtime is $O(n \cdot \log \log n \cdot k)$. □

We now set the stage for our APA query phase.

Lemma 4.2. *Using the square-root tree created in the preprocessing procedure above, an expression of form (1) can be evaluated in $O(k)$.*

Proof. We aim to evaluate $[\![e_i]\!] \otimes [\![e_{i+1}]\!] \otimes \ldots \otimes [\![e_j]\!]$. Let us look at the first level of our tree. If the segment $[e_i, e_j]$ transcends several subpaths $L^{i'}, L^{i'+1}, \ldots, L^{j'-1}, L^{j}$, we have

$$[\![e_i]\!] \otimes [\![e_{i+1}]\!] \otimes \ldots \otimes [\![e_j]\!] = ([\![e_i]\!] \otimes \ldots \otimes [\![e_{i' \cdot m'}]\!]) \otimes \Big([\![L^{i'+1}]\!] \otimes \ldots \otimes [\![L^{j'-1}]\!] \Big) \otimes \Big([\![e_{(j'-1)\cdot m'+1}]\!] \otimes \ldots \otimes [\![e_j]\!] \Big).$$

The first part is a suffix, the second a big-step expression and the third a prefix. They are all precomputed. Thus, the query only needs to perform two operations in the algebra which takes $O(k)$. The only remaining case is if $[e_i, e_j]$ is entirely inside one subpath $L^{i'}$. In this case, a similar analysis should be performed, but at a lower level of the tree. Given that all paths in the tree have lengths that are powers of two, we can find the suitable level by simple bitwise operations on i and j which take $O(1)$. □

Predecessors and Successors at Higher Levels. Let $v \in V_P$ be a vertex of our CFG and $\lambda \geq \lambda(v)$. A vertex u at level λ is an *λ-successor* of v if there exists a path π from v to u such that (i) all vertices in π are at level λ or lower, and (ii) u is the first/only vertex in π that has level λ. Similarly, u is an *λ-predecessor* of v if there exists a path from u to v such that (i) all vertices in π are at level λ or lower, and (ii) u is the last/only vertex in π that has level λ. Every vertex v has exactly one λ-predecessor and at most two λ-successors at any level $\lambda \geq \lambda(v)$. More specifically, if $\lambda = \lambda(v)$, then v is its own unique λ-predecessor and λ-successor. Otherwise, v is nested within another structure at level λ. If v is nested within $\mathsf{branch}_l\ P_1, P_2, \ldots, P_m\ \mathsf{end}_l$ with $\lambda(\mathsf{branch}_l) = \lambda(\mathsf{end}_l) = \lambda$, then the only λ-predecessor of v is branch_l and its only λ-successor is end_l. Similarly, if v is nested within $\mathsf{loop}_l\ P'\ \mathsf{end}_l$ with $\lambda(\mathsf{loop}_l) = \lambda(\mathsf{end}_l) = \lambda$, then its only λ-predecessor is loop_l but both loop_l and end_l can be λ-successors since an execution that starts at v might break/continue the loop. For every vertex v and its every λ-successor u, we can find a summary $[\![v \to u]\!]$ of all paths π defined as above in $O(k)$. More specifically, let P' be the subprogram nested inside u that contains v, we define and compute:

$$[\![v \to v]\!] := q(L_v, v, v) \qquad [\![v \to u]\!] := \bigoplus_{u' \in P' \,:\, (u',u) \in E_P} q(G_{P'}, v, u') \otimes [\![(u', u)]\!] \quad \text{for } u \neq v.$$

Here, L_v is the linearized CFG at level $\lambda(v)$ that contains v. Recall that a query on a linearized CFG can be answered in $O(k)$ by Lemma 4.2. The second part recursively calls a constant number of APA queries on P' which has a lower depth. Note that there are constantly many possible vertices u' since we assumed every loop is broken/continued in constantly many places. Similarly, if u is the λ-predecessor of v we have:

$$[\![v \to v]\!] := q(L_v, v, v) \qquad [\![u \to v]\!] := [\![(u, \top_{P'})]\!] \otimes q(G_{P'}, \top_{P'}, v) \quad \text{for } u \neq v.$$

Again, the former is computed using Lemma 4.2 and the latter is an APA query in the program P' that has lower depth than λ.

Remark. The values $[\![u \to v]\!]$ or $[\![v \to u]\!]$ above can be computed either in preprocessing or in each APA query when they are needed. This choice has no effect on the runtime complexity. Below, we assume they are computed in query.

Answering an APA Query. Based on the ideas above, we are finally ready to show how a general APA query can be answered. Suppose we are given an APA query $q(G_P, s, t)$ and P is a program with nesting depth δ. Let us consider all paths from s to t in G_P. We can divide these paths in two sets: (1) those that never visit a vertex at level δ, and (2) those that do visit a vertex at level δ. We find an answer for each part separately and then use the $\oplus$ operation in our algebra:

(1) Paths of type (1) can exist only if there is a subprogram P' of depth $\delta - 1$ that contains both s and t. In this case, we can recursively compute $a_1 := q(G_{P'}, s, t)$. If no such P' exists, we set $a_1 := \overline{0}$.
(2) By definition, any path that starts at s, gets to level δ, and then proceeds to reach t has to go through one of the δ-successors of s and the δ-predecessor of t. Thus, we compute $a_2 := \bigoplus_{s'} [\![s \to s']\!] \otimes q(L_P, s', t') \otimes [\![t' \to t]\!]$, where s' is a δ-successor of s and t' is the δ-predecessor of t. The first and last multiplicands are computed as explained above and the middle multiplicand is obtained by Lemma 4.2.

Finally, our algorithm returns $a_1 \oplus a_2$ as the desired value for $q(G_P, s, t)$.

Lemma 4.3. *The algorithm above answers an intraprocedural query $q(G_P, s, t)$ in $O(k)$.*

Proof. Recall that we assumed the depth δ is a constant. We prove the lemma by induction on δ. The case $\delta = 0$ is trivial. For $\delta > 0$, our algorithm makes constantly many recursive APA queries on subprograms P' with depth $\delta - 1$ and constantly many linearized queries using Lemma 4.2, each of which takes $O(k)$ by induction hypothesis and Lemma 4.2. □

Finally, we have our main theorem, which is a direct result of Lemmas 4.1 and 4.3:

Theorem 4.4. *Given a program P with bounded nesting depth, an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$, and a semantic function $[\![.]\!] : E \to A$ that assigns an element of A to every edge of the CFG G_P, our algorithm preprocesses P in $O(n \cdot \log\log n \cdot k)$ time and then answers each intraprocedural APA query $q(G_P, s, t)$ in $O(k)$, where n is the size of G_P and k is the time needed to perform a single atomic operation in A.*

Runtime Dependence on the Nesting Depth. The dependence on the nesting depth δ has been ommited from the complexity since δ is assumed to be a constant. The complexity of the preprocessing does not depend on the nesting depth, as the linearization and the building of the square-root tree depend only on the number of vertices in the CFG. The query phase makes a constant number of queries per nesting depth, according to Lemma 4.3, and therefore takes $O(k \cdot \delta)$.

4.2 Exploiting Treewidth

In this section, we provide an alternative algorithm for on-demand intraprocedural APA that does not depend on nesting depth. Instead, we assume the given control-flow graph G has bounded treewidth τ. This assumption is shown to hold for real-world structured programs both theoretically [Thorup 1998] and in practice [Gustedt et al. 2002]. Since we handle programs of any nesting depth, our setting is strictly more general than Section 4.1. The tradeoff is an increase in our preprocessing time from $O(n \cdot \log\log n \cdot k)$ to $O(n \cdot \log n \cdot k)$.

Overview. The central idea in our algorithm is to consider a centroid decomposition of a tree decomposition and then precompute summaries from every bag to each of its ancestors in the centroid decomposition. Note that every bag has at most $O(\log n)$ ancestors. We then show that every APA query can be broken down into a combination of a constant number of precomputed summaries, which allows us to answer queries with a constant number of operations in the algebra. Our algorithm executes the following steps:

(1) *Computing a Decomposition:* The algorithm computes a tree decomposition of the CFG using an external tool.
(2) *Same-bag Summaries:* The algorithm finds a summary between every two nodes of the CFG that appear somewhere in the same bag in the tree decomposition. This can be done using a variant of the Floyd-Warshall algorithm for bounded-treewidth graphs.
(3) *Centroid Decomposition*: The algorithm computes a centroid decomposition of the tree decomposition. This is a tree of logarithmic height that allows us to write all $O(n^2)$ summaries

between pairs of vertices in the original tree as a combination of $O(n \cdot \log n)$ precomputed summaries.

(4) *Precomputed Summaries:* The algorithm computes $O(n \cdot \log n)$ summaries using the structure of the centroid decomposition. Intuitively, we compute summaries between every node and its ancestors/descendants.

(5) *Answering an APA query:* Given an APA query, the algorithm breaks it down into a constant number of possibilities based on the precomputed summaries and thus computes the result in $O(k)$.

Motivation for Decompositions. As mentioned, our goal is to precompute a small number of path summaries, so that we can write any given query as a combination of a constant number of precomputed summaries. For this, we are considering a centroid decomposition of a tree decomposition of our CFG. This combination of decompositions might look strange in the first glance, but actually exploits a natural shared property of both types of decompositions, namely the cut property. Consider a graph G, a tree decomposition T of G and a centroid decomposition T' of T. Suppose the query provides two vertices u and v in G. Let b_u and b_v be two bags in T that contain u and v, respectively. By the cut property of tree decompositions, any path from u to v in G has to visit every bag that is on the path from b_u to b_v in T. Now, let b be the lowest common ancestor of b_u and b_v in T'. By the cut property of centroid decompositions, b is on the path from b_u to b_v in T. Thus, every path from u to v in G has to intersect V_b. Therefore, if we already have precomputed summaries between u and v on the one hand, and every vertex in V_b on the other hand, we can cover all possible paths from u to v by simply iterating over the intermediate vertex that is in V_b.

We are now ready to provide a more formal and detailed description of the algorithm.

APA Preprocessing. Our preprocessing uses a novel and clever combination of tree decompositions and centroid decompositions. We precompute the answers to certain types of APA queries. The structure of these queries is such that any other query's result can be obtained from them by a constant number of operations in the algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$.

Preprocessing Step 1: Computing a Tree Decomposition. We start by computing a tree decomposition $T = (\mathcal{B}, E_T)$ of our control-flow graph G. This can be done using standard algorithms [Bodlaender 1996; Thorup 1998].

Preprocessing Step 2: In-bag Summaries. In this step, for every bag $b \in \mathcal{B}$ and every two vertices $u, v \in V_b$, we compute and save $q(G, u, v)$. We also add a new summary edge $e = (u, v)$ with $[\![e]\!] := q(G, u, v)$. Moreover, despite adding edges to G, this step keeps T a valid tree decomposition for G since the added edges are always to a pair of vertices that appear in the same bag.

Of course, it would not be efficient to perform all these queries separately. Instead, this step is achieved by a recursive algorithm. Suppose $b_l \in \mathcal{B}$ is a leaf bag in the tree decomposition T. We say a vertex $v \in V$ is *special* if the only bag it appears in is b_l. Our first subgoal is to remove b_l from T and all special vertices from G while preserving the query values between any pair of non-special vertices (or changing them to path-equivalent values). We perform a local all-pairs summarization in b_l, i.e. for any two vertices $u, v \in V_{b_l}$, we add a new edge $e_{u,v}$ in G from u to v with $[\![e_{u,v}]\!] := q(G[V_l], u, v)$. In other words, we summarize all paths that are entirely in V_{b_l} by direct edges. Since the algebra of regular expressions is idempotent, this preserves path equivalence. Now, let T^- be the result of removing b_l from T and G^- be the graph obtained by removing all special vertices from G. We verify that T^- is a tree decomposition of G^-. By definition, every non-special vertex appears in a bag in T^-. Consider an edge between two non-special vertices. If this edge was covered by a bag other than b_l, then it remains covered. Otherwise, since both endpoints are non-special, they are both in the parent of b_l. Finally, removing a leaf cannot disconnect the previously-connected subtree of bags containing any particular vertex. We recursively compute

in-bag summaries $q(G^-, u, v)$ for all pairs (u, v) of non-special vertices that appear in the same bag in T^- and add the resulting corresponding summary edges in G^- and G.

For any such pair (u, v), we claim $q(G, u, v) \equiv q(G^-, u, v)$. Consider a path π from u to v in G. If there are no special vertices in π, then the same path exists in G^-. Otherwise, decompose $\pi = \pi_1 \cdot a' \cdot a \cdot a'' \cdot \pi_2$, where a is the first special vertex in π. Since a is special, it only appears in the bag b_l in T. Thus, all neighbors of a must also appear in b_l given that a tree decomposition has to cover all edges. Therefore, $a', a'' \in V_{b_l}$ and we have a direct summary edge $e_{a',a''}$. Thus, we can replace π with $\pi' := \pi_1 \cdot a' \cdot a'' \cdot \pi_2$ without loss of information. Continually applying this process, we obtain a path that does not have any special vertices and is included in G^-. So, $q(G, u, v) \equiv q(G^-, u, v)$.

Finally, we have to find in-bag summaries for our special vertices, too. To do this, we simply run another all-pairs summarization in b_l. Suppose $u, v \in V_{b_l}$. After this second all-pairs summarization in b_l, we claim there is now a direct (u, v)-edge that summarizes all paths from u to v. Consider one such path π and decompose it into $\pi = \pi_1 \cdot a \cdot \pi_2 \cdot b \cdot \pi_3$ in which all vertices in $\pi_1 \cdot a$ are in V_{b_l}, none of the vertices in π_2 are in V_{b_l} and finally $b \in V_{b_l}$. The vertices a and b are not special, because special vertices do not have edges to outside V_{b_l}. Thus, both a and b appear in the parent bag p of b_l. The subpath $a \cdot \pi_2 \cdot b$ is entirely in G^- and $a, b \in V_p$ are in the same bag. Thus, there is a direct edge between a and b that summarizes not only $a \cdot \pi_2 \cdot b$ but also every other path from a to b in G^-. We can shorten π to $\pi' := \pi_1 \cdot a \cdot b \cdot \pi_3$ without losing any information. Repeating the same process leads to a path that is entirely within V_{b_l}. This path is summarized by the all-pairs summarization in b_l.

Example. Figure 7 (top) shows part of a graph G and its tree decomposition T. Here, b_l is a leaf bag containing $\{1, 2, 3, 4\}$ of which 1 and 2 are special vertices that only appear in b_l and 3 and 4 are non-special vertices that appear in the parent bag p, too. Suppose that our goal is to perform a shortest-path analysis. The initial situation in the bag b_l is shown in (i). We first apply an all-pairs shortest path in $G[V_{b_l}]$, i.e. only inside this bag. This uncovers three new edges shown in (ii). After adding these edges, we can be sure that there is a shortest path from 4 to 3 (resp. 3 to 4) that does not have to go through the special vertices 1 and 2. Thus, we can remove 1, 2 and the bag b_l and recursively perform an in-bag shortest path summarization in the rest of the graph. Note that even after removing b_l, 3 and 4 appear in the same bag p. So, this recursive call creates direct edges between them showing their shortest path in G (iii). Finally, we perform another all-pairs shortest path computation in $G[V_{b_l}]$. This creates new edges (iv) that summarize distances for special vertices. We do not need to reconsider the rest of the graph since all pertinent shortest paths outside V_{b_l} are already summarized by the red edges in (iii).

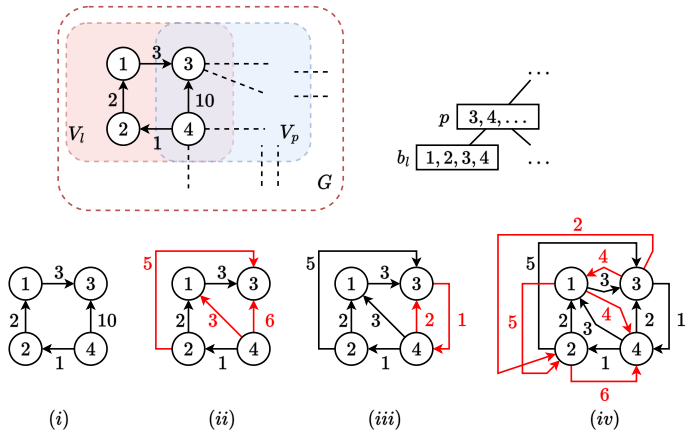


Fig. 7. Computing In-bag Summaries. Dashed lines denote edges that may exist and have an endpoint outside b_l. Every vertex also has a self-loop of weight 0 which is not shown.

Remark. As in Section 4.1, whenever our algorithm creates multi-edges between the same pair of vertices, we remove them and add a single edge labeled with the sum of their labels. If it creates a self-loop e on vertex u, we left-multiply the label of every outgoing edge of u by $[\![e]\!]$ and right-multiply every incoming edge by $[\![e]\!]$. Finally, if there are two self-loops e_1, e_2 on u we replace them with a single self-loop e and set $[\![e]\!] := ([\![e_1]\!] \oplus [\![e_2]\!])^{\circledast}$. These simplifications do not change our runtime complexity and preserve path equivalence. It is also noteworthy that Step 2 is the only part of our preprocessing that uses the $\circledast$ operator.

Preprocessing Step 3: Computing a Centroid Decomposition. We compute a centroid decomposition $T' = (\mathcal{B}, E_{T'})$ of our tree decomposition T. This is done using a standard algorithm such as della Giustina et al. [2019]. Note that T' is a centroid decomposition of a tree decomposition, thus every node in T' is a bag. We also preprocess T' for lowest common ancestor (LCA) queries using Gabow and Tarjan [1983] or Bender et al. [2005]. The reason we care about LCA is as follows: Let $b_u, b_v \in \mathcal{B}$ and $\beta \in \mathcal{B}$ be the LCA of b_u and b_v in the centroid decomposition T'. There is a unique path π_T from b_u to b_v in the original tree T. This path π has to visit β. In other words, the path in the original tree has to visit the LCA in the centroid decomposition[‡]. For example, in Figure 3, the LCA of 9 and 15 in the centroid decomposition T' is 11. The path between 9 and 15 in the original tree T is $9, 11, 14, 15$ which visits 11.

Preprocessing Step 4: Ancestor-Descendant Summaries. In this step, for every bag $b_1 \in \mathcal{B}$, every vertex $u \in V_{b_1}$, every descendant b_2 of b_1 in the centroid decomposition T', and every $v \in V_{b_2}$, we compute and save $q(G, u, v)$ and $q(G, v, u)$. In other words, if v appears in some bag in our centroid decomposition and u appears in an ancestor of that bag in the centroid decomposition, then we compute and remember both $q(G, u, v)$ and $q(G, v, u)$.

To find these values, take any bag $b_1 \in \mathcal{B}$ and let T_{b_1} be the subtree of T whose chosen centroid was b_1 when creating the centroid decomposition. In Figure 3, such subtrees are shown by dashed boxes. For example, T_{11} consists of $6, 9, 11, 13, 14$ and 15. Note that we are considering a subtree of T not T'. By construction of T', a bag b_2 is a descendant of b_1 in T' if and only if b_2 appears in T_{b_1}. In our example, the T'-descendants of 11 are $\{6, 9, 11, 13, 14, 15\}$. Thus, we need to compute $q(G, u, v)$ and $q(G, v, u)$ for every $u \in V_{b_1}$ and $v \in V_{b_2}$ where $b_2 \in T_{b_1}$. Take the subtree T_{b_1} and root it at b_1. Pick a vertex $u \in V_{b_1}$, a bag $b_2 \in T_{b_1}$ and another vertex $v \in V_{b_2}$. If $b_2 = b_1$ then $q(G, u, v)$ and $q(G, v, u)$ are already computed in the previous step (in-bag summaries). Otherwise, let p be the parent of b_2 in T_{b_1}. Since p is on the path from b_2 to b_1 in T, any path from $u \in V_{b_1}$ to $v \in V_{b_2}$ in G has to intersect $V_p \cap V_{b_2}$. This is a rephrasing of the cut property in tree decompositions. We have:

$$q(G, u, v) \equiv \bigoplus_{w \in V_p \cap V_{b_2}} q(G, u, w) \otimes q(G, w, v).$$

Similarly,

$$q(G, v, u) \equiv \bigoplus_{w \in V_p \cap V_{b_2}} q(G, v, w) \otimes q(G, w, u).$$

The equations above lead to a simple top-down dynamic programming algorithm over T_{b_1}. We pick the bags b_2 in a top-down order from T_{b_1} and apply these equations to compute $q(G, u, v)$ and $q(G, v, u)$ for each $u \in V_{b_1}$ and $v \in V_{b_2}$. Note that when we are at b_2, we have already processed its parent p. Thus, we have the values $q(G, u, w)$ and $q(G, w, u)$ for every $w \in V_p$. Moreover, since w and v appear in the same bag b_2, we already have $q(G, w, v)$ and $q(G, v, w)$ from our in-bag summarization step.

[‡]This is a direct consequence of the way we defined a centroid decomposition. See della Giustina et al. [2019]; Jordan [1869] and Carpanese [2018] for further discussion of this point.

Lemma 4.5. *Assuming the treewidth τ is a constant, the entire APA preprocessing as described above takes $O(n \cdot \log n \cdot k)$ time.*

Proof. Steps 1 and 3 use previously-known methods which take $O(n)$. Step 2 performs two all-pairs summarizations in every bag $b \in \mathcal{B}$. However, the bag size is at most $\tau + 1 \in O(1)$ and thus each all-pairs summarization in $G[V_b]$ takes $O(k)$. Given that we have $O(n)$ bags, Step 2's total runtime is $O(n \cdot k)$. In Step 4, we consider $O(n \cdot \log n)$ pairs of bags (b_1, b_2) since b_1 has to be an ancestor of b_2 in the centroid decomposition T' which has a depth of $O(\log n)$. Finally, we spend $O(k)$ time for each pair of considered bags, since the bag sizes are $O(1)$. □

Lemma 4.6. *Let $u, v \in V$ be two vertices of G and b_u, b_v two arbitrary bags in $\mathcal{B}$ such that $u \in V_{b_u}$ and $v \in V_{b_v}$. Let β be the lowest common ancestor of b_u and b_v in the centroid decomposition T'. Then, every path π from u to v in G has to visit β, i.e. $\pi \cap V_\beta \neq \emptyset$.*

Proof. First, forget that T is a tree decomposition and look at it simply as a tree. Since T' is a centroid decomposition of T and β is the T'-LCA of b_u and b_v, we know that the path from b_u to b_v in T has to visit β. Now, let us remember that T is a tree decomposition. Since β is a bag on the path from b_u to b_v in T, based on the cut property of tree decompositions, any path π from u to v in G has to intersect V_β. □

Answering an APA Query. Given an intraprocedural APA query $q(G, u, v)$ we answer it in the following steps:

- Pick two bags $b_u, b_v \in \mathcal{B}$ such that $u \in b_u$ and $v \in b_v$. If $b_u = b_v$, then $q(G, u, v)$ is already computed as an in-bag summary and is returned.
- Let β be the lowest common ancestor of b_u and b_v in the centroid decomposition T'.
- Compute and return

$$q(G, u, v) = \bigoplus_{w \in V_\beta} q(G, u, w) \otimes q(G, w, v). \tag{2}$$

The answer is guaranteed to be correct based on Lemma 4.6.

Lemma 4.7. *The algorithm above answers an intraprocedural query $q(G, u, v)$ in $O(k)$.*

Proof. We can find b_u and b_v in $O(1)$. For this, it suffices to keep track of just one of the bags containing each vertex when we compute the tree decomposition T. Similarly, the lowest common ancestor β is computed in $O(1)$ using the algorithm of Gabow and Tarjan [1983] or Bender et al. [2005]. All expressions on the right-hand side of (2) are precomputed and there are $O(1)$ of them since $|V_\beta| \leq \tau + 1 \in O(1)$. Thus, we perform constantly many operations in the algebra, taking $O(k)$. □

Our main theorem is a direct consequence of Lemmas 4.5 and 4.7:

Theorem 4.8. *Given a (control-flow) graph $G = (V, E)$, an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$, and a semantic function $[\![.]\!] : E \to A$ that assigns an element of A to every edge of G, and assuming that G has bounded treewidth τ, our algorithm preprocesses G in $O(n \cdot \log n \cdot k)$ time and then answers each intraprocedural APA query $q(G, s, t)$ in $O(k)$, where n is the size of G and k is the time needed to perform a single atomic operation in A.*

Runtime Dependence on the Treewidth. The dependence on the treewidth τ was not mentioned in the complexity analysis above since it is assumed to be a constant. This being said, our algorithm runs in polynomial time with respect to the treewidth. We can assume we are given a tree decomposition of width τ as part of the input, since it can be computed by standard methods [Bodlaender 1996; Thorup 1998]. Given this tree decomposition, the runtime of each step is as follows:

- Preprocessing Step 2 runs a variant of the Floyd-Warshall algorithm on each bag, taking $O(n \cdot k \cdot \tau^3)$ time.
- Preprocessing Step 3 runs a linear algorithm on the number of bags of the tree decompositon, which is $O(n \cdot \tau)$.
- Preprocessing Step 4 considers $O(n \cdot \log n \cdot \tau^3)$ pairs of bags, per the proof of Lemma 4.5, and each of these pairs is calculated in $O(k \cdot \tau)$ time, therefore the total runtime of this step is $O(n \cdot \log n \cdot k \cdot \tau^3)$
- The complexity of answering an APA query follows from Equation (2) and is $O(k \cdot \tau)$

So the overall runtime of our algorithm, considering the dependence on τ, is $O(n \cdot \log n \cdot k \cdot \tau^3)$ for preprocessing and $O(k \cdot \tau)$ for answering an APA query.

5 INTERPROCEDURAL ON-DEMAND APA

In this section, we consider interprocedural APA. Recall that our program consists of m functions $f_1, f_2, \ldots, f_m$, each modeled by a CFG $G_i = (V_i, E_i, \top_i, \bot_i, C_i, R_i, \Phi_i)$. We also have the augmented graph $\widehat{G} = (V, \widehat{E})$ as defined in Section 2 and a semantic function $[\![.]\!] : \widehat{E} \to A$. Our goal is to answer APA queries of the form $q(\widehat{G}, s, t)$. Unfortunately, we cannot directly apply the algorithms of Section 4 since $\widehat{G}$ might not be sparse and its treewidth might be arbitrarily large. Moreover, nesting depth is not even well-defined for $\widehat{G}$. Instead, recall that we assume the CG H has bounded treedepth and that every function f_i contains a constant number of function call nodes.

Overview. There are two main ideas behind our interprocedural algorithm: (i) any execution path from a function f_1 to a function f_2 can be broken down into a same-context path in f_1, followed by a path in the call graph, followed by a same-context path in f_2. The first and third part can be handled by our intraprocedural algorithms over the augmented CFGs; (ii) since the call graph has small treedepth, it also has small treewidth. Thus, we can apply our previous intraprocedural algorithm on the call graph, instead of the CFGs. This helps us handle the second part of the path. Hence, our interprocedural solution in this section essentially extends the algorithm in Section 4.2. It consists of the following steps:

(1) *Computing the Augmented CFG:* Our algorithm computes the augmented CFG for each funtion f_i and preprocesses it using one of the intraprocedural approaches of the previous sections.
(2) *Summaries:* The algorithm assigns summaries to each edge of the call graph, representing all paths starting at one endpoint of that edge and ending in the other.
(3) *Call-graph Preprocessing:* The algorithm preprocesses the call graph, instead of control-flow graphs, using the algorithm of Section 4.2. Note that the call graph is assumed to have small treedepth, which entails small treewidth. Thus, we can apply the algorithm designed for CFGs in Section 4.2 to the call graph.
(4) *Answering an interprocedural APA query:* The algorithm breaks the query into intraprocedural parts, covered by the CFGs, and an intraprocedural path in the CG. For the former, it uses the summaries computed in the first step above and for the latter, it relies on the results of the call-graph preprocessing.

Augmented CFGs. For every function f_i, we define the *augmented CFG* of f_i as $\widehat{G}_i := \widehat{G}[V_i]$. Intuitively, $\widehat{G}_i$ contains the call-to-return-site edges that summarize function calls in f_i, but it excludes any interprocedural edge between f_i and other functions. Note that $\widehat{G}_i$ has the same nesting depth and treewidth as the CFG G_i. Thus, our algorithms in Section 4 can be applied to answer APA queries of the form $q(\widehat{G}_i, s, t)$.

Preprocessing. The fundamental idea in our algorithm for interprocedural on-demand APA is to consider the call graph $H = (\{f_1, \ldots, f_m\}, E_H)$ of the program and assign a semantic meaning $[\![e]\!] \in A$ to every interprocedural edge $e \in E_H$. Then, we break down an APA query in $\widehat{G}$ to a series

of APA queries on the call graph H and augmented control-flow graphs $\widehat{G}_i$. These subqueries are then handled by our algorithms of Section 4.

Preprocessing Step 1: Intraprocedural Preprocessing. We compute every augmented CFG $\widehat{G}_i$ and preprocess it using one of the intraprocedural approaches of Sections 4.1 or 4.2.

Preprocessing Step 2: Assigning Semantics to CG Edges. Let $e = (f_i, f_j) \in E_H$ be an edge in our call graph. We define $[\![e]\!]$ such that e summarizes all paths in $\widehat{G}$ that start at the beginning of f_i and end as soon as they reach the beginning of f_j. By construction of $\widehat{G}$, these paths correspond to executions in our program that start at $\top_i$, potentially call some intermediary functions, wait for all of them to return, and then finally call f_j. Thus, the last call to f_j in these paths has to be made from a call site c in f_i. Therefore, we can compute $[\![e]\!]$ by a set of queries in $\widehat{G}_i$ instead of $\widehat{G}$. More precisely, we compute:

$$[\![e]\!] = \bigoplus_{c \in C_i \,:\, \Phi_i(c) = f_j} q(\widehat{G}_i, \top_i, c) \otimes [\![(c, \top_j)]\!].$$

Preprocessing Step 3: CG Preprocessing. Using the semantics assigned in the previous step, we preprocess our CG H for intraprocedural APA according to the algorithm of Section 4.2. Note that we cannot apply the algorithm of Section 4.1 since H does not have a well-defined or bounded nesting depth. However, since we know H has small treedepth, we are sure it has small treewidth, too, and thus Section 4.2 is applicable.

Lemma 5.1. *The runtime of the entire preprocessing phase above is $O(n \cdot \log n \cdot k)$ if our treewidth-based intraprocedural algorithm is used in Step 1 and $O(n \cdot \log\log n \cdot k + m \cdot \log m \cdot k)$ if the nesting depth-based variant is utilized.*

Proof. In Step 2, each call node c is used once in the computation of the $[\![e]\!]$'s. Thus, we perform $O(n)$ queries in augmented CFGs $\widehat{G}_i$ plus $O(n)$ computations in the algebra. □

Answering Interprocedural APA Queries. Suppose we are given a query $q(\widehat{G}, s, t)$. With a slight misuse of notation, let f_s be the function containing s and f_t defined similarly. Let π be an arbitrary path from s to t in $\widehat{G}$ and decompose it as $\pi = \pi_1 \cdot c_1 \cdot \pi_2 \cdot c_2 \cdot \pi_3$ where c_1 is the first call node in π which is *not* immediately followed by its corresponding return node r_1 and c_2 is the last such call node in π. Intuitively, c_1 corresponds to the first function call that is triggered in π but does not return before reaching t and c_2 to the last such call. Since we reach t at the end of π, we are sure that c_2 is calling f_t and π_3 is an intraprocedural path in $\widehat{G}_t$. Similarly, we know that c_1 is a call vertex in f_s. Suppose c_1 calls the function $f_i := \Phi_s(c_1)$. Then the subpath $\pi_2 \cdot c_2$ goes from $\top_i$ to $\top_t$. Therefore, it is captured by $q(H, f_i, f_t)$. Based on this discussion, we answer $q(\widehat{G}, s, t)$ by outputting the result of the following computation:

$$q(\widehat{G}, s, t) \equiv \bigoplus_{c_1 \in C_s \,\wedge\, \Phi_s(c_1) = f_i} q(\widehat{G}_s, s, c_1) \otimes [\![(c_1, \top_i)]\!] \otimes q(H, f_i, f_t) \otimes q(\widehat{G}_t, \top_t, t). \tag{3}$$

Lemma 5.2. *The algorithm above answers an interprocedural query $q(\widehat{G}, s, t)$ in $O(k)$.*

Proof. Each of the $q(\cdot, \cdot, \cdot)$ expressions on the right-hand side of (3) can be computed by intraprocedural APA queries of Section 4 in $O(k)$. Also, the function f_s is assumed to have a constant number of function call nodes in it. Thus, we try constantly many c_1's. □

Finally, we have the main theorem of this work, which is a direct corollary of Lemmas 5.1 and 5.2:

Theorem 5.3. *Given a program P with n lines of code, consisting of m functions $f_1, \ldots, f_m$, and assuming that the call graph and all control-flow graphs of P have bounded treewidth, our algorithm above preprocesses P in time $O(n \cdot \log n \cdot k)$ or $O(n \cdot \log\log n \cdot k + m \cdot \log m \cdot k)$ and is then able to answer each interprocedural APA query in $O(k)$.*

Runtime Dependence on the Treedepth. In the theorem above, we did not include the runtime dependence on parameters which were assumed to be small. If our CG has a treedepth of d, and our CFGs have a treewidth of τ and nesting depth δ, then the runtime of each step of our algorithm is as follows:

- Analogously to the analysis of Lemma 5.1, the runtime of the entire preprocessing phase is $O(n \cdot \log n \cdot k \cdot \tau^3 + m \cdot \log m \cdot k \cdot d^3)$ if our treewidth-based intraprocedural algorithm is used in Step 1 and $O(n \cdot \log \log n \cdot k + m \cdot \log m \cdot k \cdot d^3)$ if the nesting depth-based variant is utilized.
- Analogously to the analysis of Lemma 5.2, the complexity of answering APA queries is $O(k \cdot \tau + k \cdot d)$ if our treewidth-based intraprocedural algorithm is used in Step 1 and $O(k \cdot \delta + k \cdot d)$ if the nesting depth-based variant is utilized.

Thus, our interprocedural algorithm, in all its variants, has polynomial runtime dependence on each of the parameters, including the treedepth. We remark that a further optimization provided in Goharshady and Zaher [2023] decreases the preprocessing dependence on the treedepth from cubic to linear and can be directly applied in our setting, too.

Space Complexity. In all of our algorithms, as well as the classical method of Tarjan, the memory complexity is the same as the time complexity except that k, the time needed to perform one operation in the algebra, should be replaced by k', i.e. the space needed to store one element of the algebra. This is because our runtime is always the number of dynamic programming variables multiplied by k. Since we have to store these variables, each of them will take k' units of space. Similarly, Tarjan's algorithm creates a representation of the regular expressions that takes $\Theta(n \cdot \alpha(n))$ space. Reinterpreting this representation requires saving $\Theta(n \cdot \alpha(n))$ elements of the algebra.

6 EXPERIMENTAL RESULTS

Implementation and Machine. We implemented our algorithm of Section 4.2 and those of Tarjan [1981a,b] in C++ and used tools from Dell et al. [2017] and Kowalik et al. [2020] to compute decompositions. All results were obtained on an Intel i7-11800H machine (2.30 GHz, 8 cores, 16 threads) with 12 GB of RAM, running Microsoft Windows 11.

Analyses and Benchmarks. We implemented the algorithm of Section 4.2 since it is the only one that extends to the interprocedural case in Section 5. The algorithm of Section 4.1 can only be applied intraprocedurally given that the notion of nesting depth is not well-defined in call graphs. We then considered two different use-cases of APA:

- *Data-flow Analysis:* Our first experiment applied our approach to algebras modeling standard interprocedural data-flow analyses following the framework of Reps et al. [1995]. Specifically, we modeled dead-code elimination, possibly-uninitialized variables, and null-pointer analysis. In this experiment, each element of our algebra A was a data-flow transformer function mapping the data-flow facts that held before a program fragment to those that might hold after its execution. We used the exact same transformer functions as in Reps et al. [1995]:
  - For dead-code elimination, our transformer functions are always identity, i.e. they only model reachability without keeping track of any extra data-facts.
  - For possibly-uninitialized variables, our set of data-facts is the same as the set of program variables and each transformer maps variables that were uninitialized before executing the current transition to those that are uninitialized afterwards. Specifically, the variable on the LHS of an assignment might become uninitialized if one of the variables on the RHS was uninitialized before.
  - Null-pointer analysis is handled similarly to possibly-uninitialized variables, except that we only keep track of reference variables. We did not consider pointer aliasing.

In all three cases above, our $\otimes$ operation is function composition and our $\oplus$ operation is union. This is following Reps et al. [1995]. As benchmarks, we used 13 real-world Java programs from the well-known DaCapo benchmark suite [Blackburn et al. 2006]. We also used Soot [Bodden 2012; Vallée-Rai et al. 1999] to obtain CFGs and CGs, as well as the elements of our algebra.

- *Predicate Abstraction:* In our second experiment, we performed the predicate abstraction of Reps et al. [2017, Section 5] over boolean programs. In this analysis, every element of the algebra, and thus the semantics of every edge, is a Binary Decision Diagram (BDD) mapping valuations of boolean variables before the execution of a program fragment to those after its execution. We used the BDD library of Lind-Nielsen [1999] in our implementation. As benchmarks, we took 54 boolean programs generated from Windows device drivers provided by Ball et al. [2010] and Ball and Rajamani [2000]. We used the exact same predicate abstraction, BDDs and algebraic operators as in these works. For details of the transformer functions and BDDs, we refer to Ball and Rajamani [2000, Section 3].

The choice of these benchmarks was due to the presence of ample experiments on these two families of problems in the literature. Thus, we could directly compare with the previous (non-parameterized) algorithms for APA.

Baseline and Time Limits. We compared our approach with classical APA, i.e. our own implementation of the algorithms of Tarjan [1981a,b]. For each benchmark, we first obtained function summaries using a chaotic iteration method converging to a fixed-point. The time for computing summaries is reported in the S columns of Tables 1 and 2. We then used the programs, function summaries and edge semantics as the input. Thus, our approach and the baseline received the exact same inputs. Also, they both used the same libraries for computations in the algebra. In each experiment, we set a time limit of *1 hour per benchmark* and asked *1 million distinct queries* on each benchmark, generated and ordered uniformly at random. In cases where the benchmark was small and the number of possible queries was less than a million, we asked them all in a uniformly-chosen random order. For each approach on each benchmark, we calculated its average runtime (AR) as its total runtime divided by the number of queries it successfully answered. Note that our preprocessing was counted in our algorithm's average runtime.

Treewidth and Treedepth Values. The theoretical bounds of the treewidth on the CFGs hold experimentally. In our experiments, the maximum observed treewidth was 7 and the average was 4.46. The small treedepth assumption also holds in real-world programs with hundreds of thousands of lines of code. In our experiments on the DaCapo benchmarks and Windows device drivers, the maximum observed treedepths were 133 and 101, respectively. The average treedepths were 41.77 and 48.94, respectively. Thus, our small treedepth assumption is experimentally justified, though not theoretically proven. Also, the bound on CG treedepth are much larger than the bound on CFG treewidth: 133 vs 7. Finally, our algorithms' runtime dependence on the treedepth is polynomial, so even a treedepth of 133 leads to huge gains in efficiency.

Table 1. Our experimental results on data-flow analysis. tw denotes the maximum treewidth among CFGs, td is the treedepth of the CG, S is the time spent computing function summaries, Prec is the preprocessing time in seconds, OAR and BAR are our AR and the baseline's AR, respectively, and are both in microseconds. Finally, B/O is the ratio of baseline's AR to ours.

				Dead-code Elimination					Null-pointer Analysis					Possibly Uninitialized Variables				
	$\lvert\widehat{G}\rvert$	tw	td	Prec	OAR	S	BAR	B/O	Prec	OAR	S	BAR	B/O	Prec	OAR	S	BAR	B/O
hsqldb	1881	3	4	0.04	5	0.01	490	105.75	1.25	79	0.19	3116	39.65	2.44	147	0.24	4985	34.00
xalan	2204	3	4	0.05	4	0.01	431	101.68	1.17	67	0.18	2642	39.38	1.83	103	0.20	4021	39.22
avrora	4632	3	13	0.04	7	0.03	552	76.93	0.07	8	0.05	630	75.24	0.08	9	0.03	604	67.94
fop	9473	4	11	0.25	11	0.02	1207	112.82	4.82	76	0.17	4256	56.14	10.33	122	0.27	4725	38.68
luindex	22207	4	15	3.13	67	0.08	1496	22.42	11.99	70	0.21	4615	66.11	526.64	679	1.17	110513	162.74
lusearch	29380	4	14	3.48	65	0.09	2314	35.60	12.37	66	0.24	3962	59.94	579.75	705	1.42	101392	143.73
eclipse	38500	5	27	0.48	9	0.14	8486	975.44	2.94	35	0.39	10401	298.87	45.30	98	6.54	14349	146.87
antlr	43485	4	44	0.55	13	0.12	1128	86.10	1.67	17	0.30	2744	164.30	4.10	18	0.47	2902	164.90
pmd	76578	4	50	2.87	28	0.21	19288	698.85	6.40	30	0.34	23406	793.42	57.15	82	1.10	21609	264.49
sunflow	105510	4	100	1.42	33	0.33	1467	43.92	13.13	44	1.02	4212	95.72	68.74	115	2.79	4573	39.92
jython	110446	6	65	2.68	29	0.24	8948	308.54	14.78	44	1.01	13791	313.43	117.61	144	2.13	14958	103.63
chart	122517	7	63	2.22	34	0.35	1899	56.35	39.06	74	1.05	4163	55.95	92.73	118	1.75	3104	26.35
bloat	128501	7	133	2.12	45	0.42	11777	262.88	33.44	75	1.48	14552	195.33	45.85	77	1.95	14104	182.22
Average	53486	**4.46**	**41.77**	1.49	26.92	0.16	4575.62	**222.10**	11.01	52.69	0.43	7114.62	**173.34**	119.43	185.92	1.54	23218.38	**108.82**

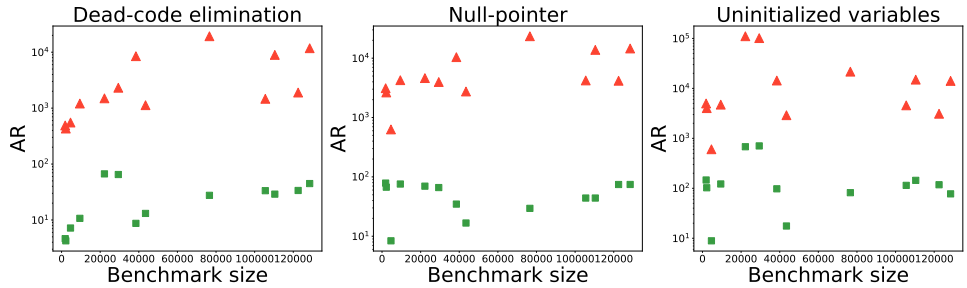


Fig. 8. Average runtimes of our approach (green squares) and the baseline (red triangles) over the data-flow benchmarks. The y axis is in logarithmic scale.

Experimental Results on Data-flow. Figure 8 and Table 1 show our experimental results on data-flow. Our approach routinely outperforms classical APA by two orders of magnitude. The columns labeled B/O report the ratio of baseline's average runtime to ours.

Experimental Results on Boolean Programs. Our predicate abstraction experiment produced even starker improvements in contrast to the baseline, since they are a much more complicated family of standard benchmarks that exemplify the wider expressibility of APA in comparison to data-flow (Table 2). Indeed, in these examples each element of the algebra is a full-fledged binary decision diagram (BDD). Figure 9 shows the AR obtained by our approach and classical APA over each of the benchmarks. Note that the y axis in this figure is in logarithmic scale.

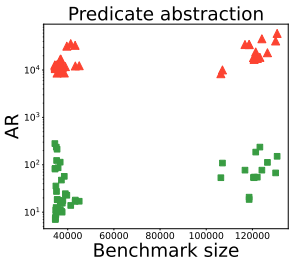


Fig. 9. AR comparison on predicate abstraction benchmarks.

Table 2. Our experimental results on boolean programs.

				Predicate Abstraction				
Benchmark	$\lvert\widehat{G}\rvert$	tw	td	Prec	OAR	S	BAR	B/O
...hw__IrqlReturn	34433	4	38	62.69	280	5.92	12825	45.73
...hw__IrqlExPassive	34506	4	38	12.42	81	1.84	12185	149.97
...Miniport_Driver_Function	34538	3	25	17.17	84	0.08	12181	145.13
...HW__WdfInterruptLockRelease	34612	4	30	0.55	7	0.07	11002	1546.14
...HW__SpinlockRelease	34629	4	30	0.64	7	0.13	11127	1581.61
...HW__WdfWaitlockRelease	34631	4	30	0.63	8	0.81	10699	1398.9
...hw__IrqlDispatch	34876	4	38	4.74	36	0.68	11294	316.46
...HW__InvalidReqAccess	34877	4	28	0.85	9	0.23	11043	1240.6
...hw__CheckAddDevice	34947	4	40	0.71	11	0.07	11483	1012.09
...hw__CheckDriverUnload	34947	4	40	0.7	11	0.09	11192	1004.86
...hw__CheckIrpMjPnp	34947	4	40	0.67	11	0.11	11206	1011.64
...hw__IrqlKeDispatchLte	34980	4	39	1.03	13	70.05	10972	875.2
...hw__MarkingQueuedIrps	35048	4	39	141.75	236	0.24	12703	53.88
...HW__WdfSpinlockRelease	35140	4	29	2.85	27	0.39	11126	406.52
...hw__CancelSpinlockRelease	35186	4	40	0.65	10	0.39	11147	1063.16
...hw__IrqlExAllocatePool	35194	4	38	20.52	123	2.71	12565	101.94
...hw__AddDeviceBus	35227	4	40	0.69	11	0.09	11162	1007.71
...sys__KmdfIrql	35395	3	12	21.66	213	8.87	8620	40.44
...ndis6__SpinLock	35462	3	26	4.66	18	0.06	11447	625.21
...ndis6__SpinLockDpr	35462	3	26	4.47	19	5.27	11212	604.9
..._hw__CancelSpinLock	35546	4	40	0.89	12	0.12	13209	1111.13
...NetBuffer_Function	35842	3	25	18.56	87	0.09	12089	139.76
...classpnp__WmiComplete	36662	4	31	0.84	11	0.16	11421	1072.69
...sys__KmdfIrql	36742	3	21	21.07	114	0.07	9484	83.44
...sfloppy__WmiComplete	36775	4	31	0.77	16	13.4	11384	691.11
...sfloppy__WmiForward	36810	4	31	0.78	11	0.13	16927	1488.24
..._WmiForward	37011	4	31	0.7	18	1.82	16961	969.07
...sys__KmdfIrql	37271	3	18	5.85	48	0.12	8823	185.43
..._hw__QueuedSpinLockRelease	37429	4	42	0.57	10	0.12	11343	1095.63
...hw__ExclusiveResourceAccessRelease	37573	4	42	0.7	10	0.17	11302	1129.87
...HW__WdfSpinlock	37656	4	32	2.15	16	0.1	11759	758.2
...hw__SpinlockRelease	37896	4	42	1.59	18	1.22	11675	662.62
...sys__KmdfIrql	38511	3	15	3.43	57	0.19	8654	151.68
...hw__SpinLock	38980	4	42	2.73	25	0.26	11800	480.69
...disk__MarkIrpPending2	39620	4	32	0.87	23	0.1	31858	1399.62
...cdrom__PagedCodeAtD0	41308	4	33	0.75	14	0.08	36009	2597.11
...sfloppy__DispatchRoutine	43203	4	32	1	17	0.1	33180	1963.1
...classpnp__StartIoRoutine	43298	4	32	0.95	18	0.12	12169	675.25
...classpnp__IrpCancelField	44979	4	33	1.19	17	0.94	12317	723.7
...cdrom__KmdfIrql	106339	6	61	15.08	53	5.36	8352	156.56
...cdrom__InvalidReqAccessLocal	107022	6	72	35.45	109	1.18	10035	92.25
...PeriodicTimer	116788	6	91	20.9	77	0.23	34463	446.86
...NdisStallExecution_Delay	118573	6	97	3.58	21	0.43	34732	1673.65
...NdisAllocateMemoryWithTagPriority	118595	6	93	3.04	19	1	34251	1844.83
...IrqlSetting_Function	120461	7	97	15.93	54	0.63	18231	337.28
...Irql_Timer_Function	120746	6	93	16.04	53	7.34	16714	313.17
...Irql_SendRcv_Function	120834	7	93	17.07	55	6.8	18137	329.06
...Irql_StatusIndication_Function	121440	6	95	57.09	185	1.17	24058	129.75
...Irql_Miniport_Driver_Function	122115	6	95	17.89	55	1.85	17515	319.73
...Irql_Synch_Function	123241	6	95	79.99	236	4.51	18505	78.27
...Irql_NetBuffer_Function	124106	6	97	23.84	77	1.29	45635	592.09
...Irql_Miscellaneous_Function	126513	6	95	35.61	112	1.83	23152	206.02
...atheros__SpinLockDpr	130062	6	97	26.03	67	1.22	40997	608.34
...atheros__SpinLock	130710	7	101	62.26	151	3.31	59090	391.06
Average	59994.33	**4.46**	**48.94**	14.73	57.06	2.88	17100.41	**724.62**

7 CONCLUSION

We provided novel scalable algorithms for on-demand interprocedural APA. Our algorithms exploit the fact that CFGs of real-world programs have small treewidth and their CGs have small treedepth. After a preprocessing that takes $O(n \cdot \log n \cdot k)$, where n is the size of the program and k is the time needed for an operation in the underlying algebra, our approach answers each APA query instantly, i.e. in $O(k)$. We also provided experimental results on real-world programs showing that our method outperforms classical APA by several orders of magnitude.

REFERENCES

Ali Ahmadi, Krishnendu Chatterjee, Amir Kafshdar Goharshady, Tobias Meggendorfer, Roodabeh Safavi, and Đorde Zikelic. 2022a. Algorithms and Hardness Results for Computing Cores of Markov Chains. In *FSTTCS*, Vol. 250. 29:1–29:20.

Ali Ahmadi, Majid Daliri, Amir Kafshdar Goharshady, and Andreas Pavlogiannis. 2022b. Efficient approximations for cache-conscious data placement. In *PLDI*. 857–871.

C Aiswarya. 2022. How treewidth helps in verification. *ACM SIGLOG News* 9, 1 (2022), 6–21.

Noga Alon and Baruch Schieber. 1987. Optimal preprocessing for answering on-line product queries. https://citeseerx.ist.psu.edu/document?repid=rep1&doi=cf740240d3a7440e23e92a09bf590cb70544cf4f

Ali Asadi, Krishnendu Chatterjee, Amir Kafshdar Goharshady, Kiarash Mohammadi, and Andreas Pavlogiannis. 2020. Faster Algorithms for Quantitative Analysis of MCs and MDPs with Small Treewidth. In *ATVA*. 253–270.

Wayne A. Babich and Mehdi Jazayeri. 1978. The Method of Attributes for Data Flow Analysis: Part II. Demand Analysis. *Acta Informatica* 10 (1978), 265–272.

Roland C Backhouse and Bernard A Carré. 1975. Regular algebra applied to path-finding problems. *IMA Journal of Applied Mathematics* 15, 2 (1975), 161–186.

Thomas Ball, Ella Bounimova, Vladimir Levin, Rahul Kumar, and Jakob Lichtenberg. 2010. The Static Driver Verifier Research Platform. In *CAV*, Vol. 6174. 119–122.

Thomas Ball and Sriram K. Rajamani. 2000. Bebop: A Symbolic Model Checker for Boolean Programs. In *SPIN*, Vol. 1885. 113–130.

Michael A Bender, Martin Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. 2005. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms* 57, 2 (2005), 75–94.

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*. ACM, 169–190.

Eric Bodden. 2012. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *SOAP@PLDI*. 3–8.

Hans L. Bodlaender. 1988. Dynamic Programming on Graphs with Bounded Treewidth. In *ICALP*, Vol. 317. 105–118.

Hans L. Bodlaender. 1996. A Linear-Time Algorithm for Finding Tree-Decompositions of Small Treewidth. *SIAM J. Comput.* 25, 6 (1996), 1305–1317.

Hans L. Bodlaender and Torben Hagerup. 1998. Parallel Algorithms with Optimal Speedup for Bounded Treewidth. *SIAM J. Comput.* 27, 6 (1998), 1725–1746.

Richard B Borie, R Gary Parker, and Craig A Tovey. 1992. Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. *Algorithmica* 7 (1992), 555–581.

Jason Breck. 2020. *Enhancing Algebraic Program Analysis.* University of Wisconsin.

Igor Carpanese. 2018. A Visual Introduction to Centroid Decomposition. https://medium.com/carpanese/an-illustrated-introduction-to-centroid-decomposition-8c1989d53308

Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Ehsan Kafshdar Goharshady. 2019a. The treewidth of smart contracts. In *SAC*. 400–408.

Krishnendu Chatterjee, Amir Kafshdar Goharshady, Prateesh Goyal, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. 2019b. Faster Algorithms for Dynamic Algebraic Queries in Basic RSMs with Constant Treewidth. *ACM Trans. Program. Lang. Syst.* 41, 4 (2019), 23:1–23:46.

Krishnendu Chatterjee, Amir Kafshdar Goharshady, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. 2016. Algorithms for algebraic path properties in concurrent systems of constant treewidth components. In *POPL*. 733–747.

Krishnendu Chatterjee, Amir Kafshdar Goharshady, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. 2020. Optimal and Perfectly Parallel Algorithms for On-demand Data-Flow Analysis. In *ESOP*. 112–140.

Krishnendu Chatterjee, Amir Kafshdar Goharshady, Nastaran Okati, and Andreas Pavlogiannis. 2019c. Efficient parameterized algorithms for data packing. In *POPL*. 53:1–53:28.

Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Andreas Pavlogiannis. 2017. JTDec: A Tool for Tree Decompositions in Soot. In *ATVA*, Vol. 10482. 59–66.

Krishnendu Chatterjee, Rasmus Ibsen-Jensen, Amir Kafshdar Goharshady, and Andreas Pavlogiannis. 2018. Algorithms for Algebraic Path Properties in Concurrent Systems of Constant Treewidth Components. *ACM Trans. Program. Lang. Syst.* 40, 3 (2018), 9:1–9:43.

Krishnendu Chatterjee, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. 2021. Faster algorithms for quantitative verification in bounded treewidth graphs. *Formal Methods Syst. Des.* 57, 3 (2021), 401–428.

Giovanna Kobus Conrado, Amir Kafshdar Goharshady, and Chun Kit Lam. 2023. The Bounded Pathwidth of Control-Flow Graphs. In *OOPSLA*. 232:1–232:26.

Bruno Courcelle. 1990. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Information and computation* 85, 1 (1990), 12–75.

Patrick Cousot and Radhia Cousot. 1977. Static Determination of Dynamic Properties of Recursive Procedures. In *Formal Description of Programming Concepts*. 237–278.

Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. 2015. *Parameterized algorithms*. Springer.

Víctor Dalmau, Phokion G. Kolaitis, and Moshe Y. Vardi. 2002. Constraint Satisfaction, Bounded Treewidth, and Finite-Variable Logics. In *CP*. Springer, 310–326.

Mark de Berg, Marc van Kreveld, Mark Overmars, Otfried Cheong Schwarzkopf, Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. 2000. More geometric data structures: Windowing. *Computational Geometry: algorithms and applications* (2000), 211–233.

Holger Dell, Christian Komusiewicz, Nimrod Talmon, and Mathias Weller. 2017. The PACE 2017 Parameterized Algorithms and Computational Experiments Challenge: The Second Iteration. In *IPEC*, Vol. 89. 30:1–30:12.

Davide della Giustina, Nicola Prezza, and Rossano Venturini. 2019. A New Linear-Time Algorithm for Centroid Decomposition. In *SPIRE*. 274–282.

Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. 1995. Demand-driven Computation of Interprocedural Data Flow. In *POPL*. ACM Press, 37–48.

Michael Elberfeld, Andreas Jakoby, and Till Tantau. 2010. Logspace Versions of the Theorems of Bodlaender and Courcelle. In *FOCS*. IEEE Computer Society, 143–152.

Javier Esparza, Stefan Kiefer, and Michael Luttenberger. 2010. Newtonian program analysis. *J. ACM* 57, 6 (2010), 33:1–33:47.

Andrea Ferrara, Guoqiang Pan, and Moshe Y. Vardi. 2005. Treewidth in Verification: Local vs. Global. In *LPAR*. 489–503.

Fedor V. Fomin, Daniel Lokshtanov, Saket Saurabh, Michal Pilipczuk, and Marcin Wrochna. 2018. Fully Polynomial-Time Parameterized Computations for Graphs and Matrices of Low Treewidth. *ACM Trans. Algorithms* 14, 3 (2018), 34:1–34:45.

Harold N. Gabow and Robert Endre Tarjan. 1983. A Linear-Time Algorithm for a Special Case of Disjoint Set Union. In *STOC*. 246–251.

Amir Kafshdar Goharshady and Fatemeh Mohammadi. 2020. An efficient algorithm for computing network reliability in small treewidth. *Reliab. Eng. Syst. Saf.* 193 (2020), 106665.

Amir Kafshdar Goharshady and Ahmed Khaled Zaher. 2023. Efficient Interprocedural Data-Flow Analysis Using Treedepth and Treewidth. In *VMCAI*. 177–202.

Jens Gustedt, Ole A. Mæhle, and Jan Arne Telle. 2002. The Treewidth of Java Programs. In *ALENEX*, Vol. 2409. 86–97.

Susan Horwitz, Thomas W. Reps, and Shmuel Sagiv. 1995. Demand Interprocedural Dataflow Analysis. In *FSE*. 104–115.

Camille Jordan. 1869. Sur les assemblages de lignes. *Journal für die reine und angewandte Mathematik* 70 (1869), 185–190.

Alexander Kernozhitsky, Anton Älgmyr, Oleksandr Kulkov, and Wiktor Kuchta. 2022. Sqrt Tree. https://cp-algorithms.com/data_structures/sqrt-tree.html

Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *POPL*. 194–206.

Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas W. Reps. 2017. Compositional recurrence analysis revisited. In *PLDI*. ACM, 248–262.

Zachary Kincaid, John Cyphert, Jason Breck, and Thomas W. Reps. 2018. Non-linear reasoning for invariant synthesis. In *POPL*. 54:1–54:33.

Zachary Kincaid, Thomas W. Reps, and John Cyphert. 2021. Algebraic Program Analysis. In *CAV*. 46–83.

Stephen Kleene. 1956. Representation of events in nerve nets and finite automata. *Automata studies* 34 (1956), 3–41.

Joachim Kneis and Alexander Langer. 2008. A Practical Approach to Courcelle's Theorem. In *MEMICS (Electronic Notes in Theoretical Computer Science)*, Vol. 251. 65–81.

Lukasz Kowalik, Marcin Mucha, Wojciech Nadara, Marcin Pilipczuk, Manuel Sorge, and Piotr Wygocki. 2020. The PACE 2020 Parameterized Algorithms and Computational Experiments Challenge: Treedepth. In *IPEC*, Vol. 180. 37:1–37:18.

Dexter Kozen. 1990. On Kleene Algebras and Closed Semirings. In *MFCS*. 26–47.

Daniel Kroening, Natasha Sharygina, Stefano Tonetta, Aliaksei Tsitovich, and Christoph M. Wintersteiger. 2008. Loop Summarization Using Abstract Transformers. In *ATVA*. 111–125.

Jørn Lind-Nielsen. 1999. BuDDy: A binary decision diagram package. (1999).
Mohsen Alambardar Meybodi, Amir Kafshdar Goharshady, Mohammad Reza Hooshmandasl, and Ali Shakiba. 2022. Optimal Mining: Maximizing Bitcoin Miners' Revenues from Transaction Fees. In *Blockchain*. 266–273.
Jaroslav Nesetril and Patrice Ossona de Mendez. 2006. Tree-depth, subgraph coloring and homomorphism bounds. *Eur. J. Comb.* 27, 6 (2006), 1022–1041.
Rolf Niedermeier. 2004. Ubiquitous Parameterization - Invitation to Fixed-Parameter Algorithms. In *MFCS*. 84–103.
Jan Obdrzálek. 2003. Fast Mu-Calculus Model Checking when Tree-Width Is Bounded. In *CAV*. 80–92.
Thomas W. Reps. 1993. Demand Interprocedural Program Analysis Using Logic Databases. In *ILPS*. 163–196.
Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*. 49–61.
Thomas W. Reps, Emma Turetsky, and Prathmesh Prabhu. 2017. Newtonian Program Analysis via Tensor Product. *ACM Trans. Program. Lang. Syst.* 39, 2 (2017), 9:1–9:72.
Neil Robertson and Paul D. Seymour. 1986. Graph Minors. II. Algorithmic Aspects of Tree-Width. *J. Algorithms* 7, 3 (1986), 309–322.
Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. 1996. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theor. Comput. Sci.* 167 (1996), 131–170.
Micha Sharir and Amir Pnueli. 1978. *Two approaches to interprocedural data flow analysis*. Courant Institute of Mathematical Sciences.
Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. In *OOPSLA*. ACM, 59–76.
Robert Endre Tarjan. 1981a. Fast Algorithms for Solving Path Problems. *J. ACM* 28, 3 (1981), 594–614.
Robert Endre Tarjan. 1981b. A Unified Approach to Path Problems. *J. ACM* 28, 3 (1981), 577–593.
Mikkel Thorup. 1998. All structured programs have small tree width and good register allocation. *Information and Computation* 142, 2 (1998), 159–181.
Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *CASCON*. 13.
Dacong Yan, Guoqing Xu, and Atanas Rountev. 2011. Demand-driven context-sensitive alias analysis for Java. In *ISSTA*. ACM, 155–165.
Xin Zheng and Radu Rugina. 2008. Demand-driven alias analysis for C. In *POPL*. ACM, 197–208.
Shaowei Zhu and Zachary Kincaid. 2021. Termination analysis without the tears. In *PLDI*. 1296–1311.

Received 2023-04-14; accepted 2023-08-27