Fast Graph Simplification for Interleaved-Dyck Reachability

YUANBO LI and QIRUN ZHANG, Georgia Institute of Technology
THOMAS REPS, University of Wisconsin-Madison

Many program-analysis problems can be formulated as graph-reachability problems. Interleaved Dyck language reachability (InterDyck-reachability) is a fundamental framework to express a wide variety of program-analysis problems over edge-labeled graphs. The InterDyck language represents an intersection of multiple matched-parenthesis languages (i.e., Dyck languages). In practice, program analyses typically leverage one Dyck language to achieve context-sensitivity, and other Dyck languages to model data dependencies, such as field-sensitivity and pointer references/dereferences. In the ideal case, an InterDyck-reachability framework should model multiple Dyck languages *simultaneously*.

Unfortunately, precise InterDyck-reachability is undecidable. Any practical solution must over-approximate the exact answer. In the literature, a lot of work has been proposed to over-approximate the InterDyck-reachability formulation. This article offers a new perspective on improving both the precision and the scalability of InterDyck-reachability: we aim at simplifying the underlying input graph G. Our key insight is based on the observation that if an edge is not contributing to any InterDyck-paths, we can safely eliminate it from G. Our technique is orthogonal to the InterDyck-reachability formulation and can serve as a pre-processing step with any over-approximating approach for InterDyck-reachability. We have applied our graph simplification algorithm to pre-processing the graphs from a recent InterDyck-reachability-based taint analysis for Android. Our evaluation of three popular InterDyck-reachability algorithms yields promising results. In particular, our graph-simplification method improves both the scalability and precision of all three InterDyck-reachability algorithms, sometimes dramatically.

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

Additional Key Words and Phrases: Static analysis, CFL-reachability

ACM Reference format:
Yuanbo Li, Qirun Zhang, and Thomas Reps. 2022. Fast Graph Simplification for Interleaved-Dyck Reachability. *ACM Trans. Program. Lang. Syst.* 44, 2, Article 11 (May 2022), 28 pages.
https://doi.org/10.1145/3492428

11

Portions of this work appeared in the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation [13].
This work was supported, in part, by a gift from Rajiv and Ritu Batra; by Facebook under a Probability and Programming Research Award; by Amazon under an Amazon Research Award in automated reasoning; by the United States **National Science Foundation** (**NSF**) under grants No. 1917924 and No. 2114627; by the **Defense Advanced Research Projects Agency** (**DARPA**) under grant N66001-21-C-4024; and by ONR under grants N00014-17-1-2889 and N00014-19-1-2318. The first author was partially supported by the Facebook Graduate Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the above sponsoring entities.
Authors' addresses: Y. Li and Q. Zhang, School of Computer Science, Georgia Institute of Technology. 266 Ferst Dr NW, Atlanta, GA 30332; emails: {yuanboli, qrzhang}@gatech.edu; T. Reps, Computer Sciences Department, University of Wisconsin, 1210 W. Dayton St., Madison, WI 53703; email: reps@cs.wisc.edu.

0164-0925/2022/05-ART11 $15.00
https://doi.org/10.1145/3492428

1 INTRODUCTION

The L **language-reachability** (**L-reachability**) framework is a popular model to formulate many program-analysis problems [16]. An L-reachability instance $Reach\langle L, G\rangle$ contains (1) a formal language L that formalizes the analysis problem, and (2) an edge-labeled graph G that represents the program under analysis. Two nodes are L-reachable in G iff there exists a path joining them, and the path string belongs to L. In the literature, the most popular L-reachability formulation is Dyck-reachability [11, 27]. A Dyck language essentially generates well-balanced parentheses, which can be used to capture well-paired program properties, such as function calls/returns [17, 18, 24], pointer references/dereferences [29, 30], locks/unlocks [10, 15], and field reads/writes [9, 26, 27].

A natural generalization of Dyck-reachability is **Interleaved Dyck-reachability** (**InterDyck-reachability**) [9, 14, 17, 28]. The Interleaved Dyck language denotes the intersection of multiple Dyck languages based on an interleaving operator $\odot$. For instance, let L_1 and L_2 be two Dyck languages that generate matched parentheses and matched brackets, respectively. The string "$\llparenthesis \llbracket \llparenthesis \rrbracket \rrparenthesis \rrparenthesis$" belongs to the language InterDyck $= L_1 \odot L_2$, because both parentheses and brackets are properly matched. InterDyck-reachability is much more expressive than Dyck-reachability, and in practice, brings tremendous precision improvements in client analyses. In particular, almost all recent work on context-sensitive, field-sensitive analysis has adopted the InterDyck-reachability formulation to achieve both the context- and field-sensitivities simultaneously [9, 21, 28].

Unfortunately, solving InterDyck-reachability is computationally hard because the InterDyck-reachability problem is, in general, undecidable [17]. Therefore, any practical analysis must approximate the exact answer. In practice, it is quite challenging to develop a suitable *over-approximative* InterDyck-reachability framework that offers a sweet spot in the trade-off between precision and scalability. InterDyck is a prototypical example of a non-context-free language [8]. Traditional approaches employ less expressive but polynomial-time decidable language-reachability frameworks, such as **context-free-language reachability** (**CFL-reachability**) to over-approximate InterDyck-reachability [9, 23, 26]. For example, the recent work by Späth et al. proposed synchronized pushdown systems to compute a sound solution for InterDyck-reachability [21]. The work by Zhang and Su proposed **linear-conjunctive-language reachability** (**LCL-reachability**) to precisely describe the InterDyck-reachability formulation [28]. However, the LCL-reachability algorithm is inherently an over-approximation. To the best of our knowledge, all previous efforts on the InterDyck-reachability problem attempt to improve either on the L-reachability formulation or the L-reachability algorithm.

In this article, we attack the InterDyck-reachability problem from a new angle. Consider an InterDyck-reachability instance $Reach\langle L, G\rangle$. Unlike existing approaches that improve either the L-reachability formulation or the algorithm, our approach focuses on simplifying the input graph G in $Reach\langle L, G\rangle$. Specifically, we give an efficient algorithm to simplify the input graphs by eliminating "useless" graph edges. The benefits of graph simplification are two-fold. First, working with smaller graphs improves the scalability of all existing approaches for InterDyck-reachability. Second, because all InterDyck-reachability algorithms are inherently over-approximative, they could achieve better precision by working with graphs that contain fewer edges. The technical challenge, however, is to design a graph-simplification algorithm that is both effective (i.e., it should remove as many "useless" edges as possible) and efficient (i.e., as a pre-processing step, it should run much faster than the InterDyck-reachability algorithm itself).

Consider an InterDyck language $L_1 \odot L_2 \ldots \odot L_N$, where for each $i \in [1, N]$, L_i is a Dyck language. Our enabling insight is to decompose the undecidable InterDyck-reachability problem in the input graph G into N subcubic-time Dyck-reachability problems in a new graph G'.

The new graph G' is a relaxation of the original graph G that is *bidirected*, i.e., for each edge $u \xrightarrow{(\!|_i} v$ labeled by an open-parenthesis $(\!|_i$, there exists its corresponding close-parenthesis edge $v \xrightarrow{|\!)_i} u$, and vice versa. The decomposition of the InterDyck-reachability problem into N Dyck-reachability problems transforms the undecidable problem into N subcubic-time problems [2]. The relaxation of graph G transforms each subcubic-time Dyck-reachability problem into a bidirected Dyck-reachability problem, which can be solved in almost linear time [1]. After the relaxation, if an edge contributes to an InterDyck-path in G, the corresponding edge must contribute to a Dyck-path in G'. G' contains more edges than G, and hence more paths than G. Therefore, we can safely *delete all non-contributing edges* that are not involved in any InterDyck-paths in G', as well as in G. The problem then becomes one of identifying non-contributing edges in G'.

The bidirected-graph relaxation from G to G' plays a significant role for graph simplification. Dyck-reachability in the bidirected graph G' has special properties that allow us to identify non-contributing edges much faster than if it were attempted in G. In particular, given an input graph G with n nodes and m edges, we give an efficient algorithm that simplifies G in $O(m+n \cdot \alpha(n))$ time, where α denotes the inverse Ackermann function. This graph-simplification algorithm is asymptotically faster than the fastest $O(mn)$-time InterDyck-reachability algorithm [28]. We also propose a specialized graph-simplification algorithm—which has the same complexity—for when the input graph G is already bidirected. The techniques are general and can be used as a preprocessing step for any existing InterDyck-reachability algorithms.

We have implemented the graph-simplification algorithm, and evaluated it on a recent InterDyck-reachability-based taint analysis for Android [9]. In particular, we tested graph simplification with three popular InterDyck-reachability algorithms, based on CFL-reachability [16], **synchronized pushdown system reachability** (**SPDS**) [21], and LCL-reachability [28]. Our experimental results are encouraging: the graph simplification technique significantly improves both the performance and the precision of the client analyses.

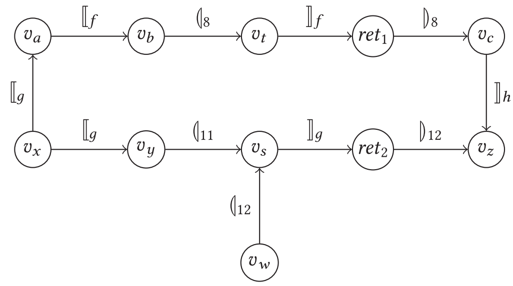
- We found that, on average, it is 2.63× faster to (i) run the simplification algorithm on digraph G—thereby creating simplified digraph G_f—and then (ii) run an InterDyck-reachability algorithm A on G_f, compared to running A directly on the original graph G.
- In the experiments with LCL-reachability, we found that the cost of running the simplification algorithm is recouped for all examples that require more than seven seconds to run in the original graphs.
- The number of reachable pairs returned by the analysis based on the simplified graph G_f is reduced to 64.92% compared to the number obtained by running the analysis on G. Moreover, the analysis run on G_f uses 57.37% memory for the analysis running on G.

Our work makes the following contributions:

- We propose a novel graph-simplification framework for InterDyck-reachability. Our technique reduces input-graph size, and is compatible with all existing sound InterDyck-reachability algorithms.
- Given a graph with n nodes and m edges, we give a fast simplification algorithm that runs in $O(m + n \cdot \alpha(n))$ time. In practice, our algorithm scales linearly with the graph size.
- We evaluate our technique based on a variety of InterDyck-reachability algorithms for taint analysis. Our empirical results show that graph simplification is beneficial: running an analysis on a simplified graph (plus graph simplification) is faster than running the analysis on the original graph. With simplified graphs, all evaluated algorithms yield more precise results and use less memory as well.

```
1  class T { T f; T g; T h; }
2  T getF(T v_t){ return v_t.f;
       }
3  T getG(T v_s){ return v_s.g;
       }
4
5  T v_a, v_b, v_c, v_y, v_z, v_w; ...
6  v_a.g = v_x;
7  v_b.f = v_a;
8  v_c = getF(v_b);
9  v_z = v_c.h
10 v_y.g = v_x;
11 getG(v_y)
12 v_z = getG(v_w)
```

(a) Example Java code.



(b) InterDyck-reachability graph for taint analysis. Bracket edges (*i.e.*, $[\![_f$ and $]\!]_f$) represent field reads and writes *w.r.t.* a field name f. Parenthesis edges (*i.e.*, $(\!|_l$ and $|\!)_l$) represent calls and returns *w.r.t.* the line number l of the call-sites.

Fig. 1. Motivating taint-analysis example.

This work focuses on graph simplification for the InterDyck-reachability problem. However, our graph simplification algorithm can also be applied to Dyck-reachability problems because our proposed $O(m + n \cdot \alpha(n))$ graph simplification algorithm is significantly faster than the subcubic Dyck-reachability algorithms.

The remainder of the article is organized as follows: Section 2 motivates graph simplification. Section 3 gives definitions and the problem formulation. Section 4 presents the idea of eliminating non-contributing edges. Section 5 gives the simplification algorithm. Section 6 describes our evaluation. Section 7 discusses related work. Section 8 concludes.

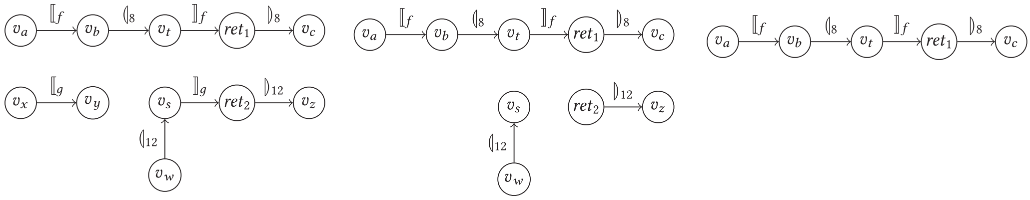
2 MOTIVATING EXAMPLE

We motivate our graph-simplification method using a formulation of taint analysis as an InterDyck-reachability problem [9]. Consider the simple Java-like program in Figure 1. For every pair of variables, the taint analysis checks whether a tainted value can potentially flow between them.

InterDyck-reachability for taint analysis. Figure 1(b) gives the graph G that encodes the taint-analysis problem for the program in Figure 1(a) as an InterDyck-reachability problem. In particular, nodes in G represent the variables in the program, and edges represent the assignments and calls/returns. Each edge is labeled with either a bracket or a parenthesis. Specifically, the brackets (i.e., $[\![$ and $]\!]$) represent field reads/writes, and parentheses (i.e., $(\!|, |\!)$) represent calls and returns. For a path in G to represent the flow of a tainted value, both brackets and parentheses must be properly matched. Let L_b and L_p be Dyck languages of brackets and parentheses respectively. Due to the work of Huang et al. [9], the taint analysis can be formulated as an InterDyck-reachability problem over G, where InterDyck $= L_b \odot L_p$.

InterDyck-reachability algorithms. The problem of InterDyck-reachability is undecidable [17]. We briefly describe three popular over-approximation algorithms for the InterDyck-reachability.

— *CFL-reachability algorithm* [16]. The intersection of a regular language and a context-free language is still context-free. Therefore, we can over-approximate one Dyck language in InterDyck using a regular language. For instance, let R_b be the regular language that over-approximates L_b. The reachability problem could be solved by a CFL-reachability algorithm based on the $CFL = R_b \cap L_p$.



(a) Graph G_1 after the first iteration. The field-write edge $v_x \xrightarrow{\llbracket_g} v_a$, the field-read edge $v_c \xrightarrow{\rrbracket_h} v_z$, the call edge $v_y \xrightarrow{\llparenthesis_{11}} v_s$ have been eliminated.

(b) Graph G_2 after the second iteration. The field-write edge $v_x \xrightarrow{\llbracket_g} v_y$, the field-read edge $v_s \xrightarrow{\rrbracket_g} ret_2$, and the corresponding nodes have been eliminated.

(c) Graph G_3 after the final iteration. The edge $v_w \xrightarrow{\llparenthesis_{12}} v_s$ and the edge $ret_2 \xrightarrow{\rrparenthesis} v_z$ have been eliminated from the graph.

Fig. 2. Overview of the graph-simplification procedure on the taint-analysis example.

— *SPDS-reachability algorithm* [21]. The SPDS also over-approximates the InterDyck-reachability. SPDS encodes calls/returns and field reads/writes as separate CFL-reachability problems, and intersects the results.
— *LCL-reachability algorithm* [28]. The InterDyck languages belong to the class of LCLs. Therefore, an LCL-reachability formulation can precisely encode an InterDyck-reachability problem. Zhang and Su [28] give the LCL rules based on trellis automata, which is an alternative form of the LCL grammar for the InterDyck language, and propose a worklist-based algorithm that computes an over-approximating solution for LCL-reachability.

Graph simplification. Recall that our key idea for graph simplification is to eliminate graph edges that are not contributing to any InterDyck-paths. Our simplification algorithm is iterative. Intuitively, the eliminated edges identified by a previous iteration can be used to identify additional non-contributing edges in later iterations. Figure 2 provides an overview of the results for the taint-analysis example after selected iterations of the edge-elimination algorithm. Figure 2(a) shows the simplification result after the first iteration. The simplification algorithm identifies that the edges $v_x \xrightarrow{\llbracket_g} v_a$, $v_c \xrightarrow{\rrbracket_h} v_z$, and $v_y \xrightarrow{\llparenthesis_{11}} v_s$ cannot be involved in any InterDyck-paths, so the algorithm removes these edges. Their removal now allows $v_x \xrightarrow{\llbracket_g} v_y$ and $v_s \xrightarrow{\rrbracket_g} ret_2$ to be identified as edges not contributing for InterDyck-reachability, and the second iteration removes them. Figure 2(b) gives the simplification result after the second iteration, and Figure 2(c) shows the final result after no additional removal steps are possible.

Benefits of graph simplification. The graph simplification is iterative. Figures 2(a) and (b) give two intermediate steps based on the first and second applications, respectively, of our graph simplification algorithm (Algorithm 2 in Section 5.2). Figure 2(c) shows the final graph G_f. Compared with the original graph in Figure 1(b), the number of edges in G_f has been reduced from 11 to 4, and the number of nodes has been reduced from 11 to 5. It is immediate that any InterDyck-reachability algorithm runs faster on G_f because G_f is only half the size of the original graph G. Table 1 gives the InterDyck-reachable node pairs and demonstrates another benefit of graph simplification—namely, when various (over-approximative) InterDyck-reachability algorithms run on the simplified graph, a more precise answer might be obtained. For the graph in Figure 1(b), as demonstrated in Table 1, both the CFL-reachability algorithm and SPDS-reachability algorithm benefit in terms of precision by running on the simplified graph. LCL-reachability algorithm does not obtain a more precise solution on this specific example; however, our experimental results in Section 6 show that the LCL-reachability algorithm can also benefit from graph simplification in terms of precision.

Table 1. Precision Improvement by Graph Simplification

Graph	InterDyck-Reachable Node Pairs		
	CFL	LCL	SPDS
Original	$\left\{ \begin{array}{c} (v_x, v_z), \\ (v_a, v_c) \end{array} \right\}$	$\{(v_a, v_c)\}$	$\left\{ \begin{array}{c} (v_x, v_z), \\ (v_a, v_c) \end{array} \right\}$
Simplified	$\{(v_a, v_c)\}$	$\{(v_a, v_c)\}$	$\{(v_a, v_c)\}$

We discuss the impact of graph simplification on different InterDyck-reachability algorithms.

- *CFL-reachability algorithm.* The CFL-reachability algorithm in Figure 1(b) computes a false-positive reachable pair (v_x, v_z). This pair is introduced by the path $p_1 = v_x \xrightarrow{[\![_g} v_a \xrightarrow{[\![_f} v_b \xrightarrow{(\!|_8} v_t \xrightarrow{]\!]_f} ret_1 \xrightarrow{|\!)_8} v_c \xrightarrow{]\!]_h} v_z$. In the simplified graph G_f (Figure 2(c)), the CFL-reachability algorithm gives an exact solution.
- *SPDS-reachability algorithm.* In Figure 1(b), the SPDS-reachability algorithm computes a non-InterDyck-reachable pair (v_x, v_z). In particular, the field-insensitive pushdown system accepts the path $p_1 = v_x \xrightarrow{[\![_g} v_a \xrightarrow{[\![_f} v_b \xrightarrow{(\!|_8} v_t \xrightarrow{]\!]_f} ret_1 \xrightarrow{|\!)_8} v_c \xrightarrow{]\!]_h} v_z$ and the context-insensitive pushdown system accepts the path $p_2 = v_x \xrightarrow{[\![_g} v_y \xrightarrow{(\!|_{11}} v_s \xrightarrow{]\!]_g} ret_2 \xrightarrow{|\!)_{12}} v_z$. After synchronization, the SPDS system concludes that v_z is InterDyck-reachable from v_x. In G_f (Figure 2(c)), neither path exists, and consequently, the SPDS-reachability algorithm produces a precise solution.
- *LCL-reachability algorithm.* The LCL-reachability algorithm computes the exact solution in this example because the graph is acyclic. In practice, graph simplification allows the LCL-reachability algorithm to run faster and consume less memory. It also eliminates some cycles in the graph, and improves the precision of LCL-reachability. Moreover, the cost is not prohibitive: in the experiments with LCL-reachability, the cost of running the simplification algorithm is recouped—often dramatically—for all examples that require more than seven seconds to run in the original graphs.

3 PRELIMINARIES

This section introduces definitions used in the article. Section 3.1 reviews Dyck languages and the graph-reachability framework. Section 3.2 describes InterDyck-reachability. Section 3.3 defines the graph-simplification problem.

3.1 Dyck Language and L-Reachability

A Dyck language is a context-free language that describes the set of well-balancedparenthesis strings. Let $CFG = (\Sigma, N, P, S)$ be a context-free grammar for the Dyck language with k kinds of parentheses. The CFG has the alphabet $\Sigma = \{(\!|_i, |\!)_i \mid i \in [1..k]\}$, the nonterminal symbol set $N = \{D_k\}$, the start symbol set $S = \{D_k\}$, and the following productions P:

$$D_k \rightarrow D_k\ D_k \mid (\!|_1\, D_k\, |\!)_1 \mid \ldots \mid (\!|_k\, D_k\, |\!)_k \mid \varepsilon. \tag{1}$$

Given a formal language L and a directed graph $G = (V, E)$ with each edge $u \xrightarrow{t} v$ in E labeled by a terminal $t \in \Sigma$, we say that a path $p = v_0 \xrightarrow{t_0} v_1 \xrightarrow{t_1} v_2 \xrightarrow{t_2} \ldots \xrightarrow{t_{m-1}} v_m$ in G *realizes* a string $R(p)$ over the alphabet Σ by concatenating the edge labels in the path in order, i.e., $R(p) = t_0 t_1 t_2 \ldots t_{m-1}$. A path p in G is an L-*path* if the realized string $R(p)$ is a word in the formal language L. Node v

is *L-reachable* from node u iff there exists an L-path from u to v in G. The *L-reachability* problem *Reach*$\langle L, G\rangle$ is to compute *all* L-reachable node pairs in graph G.

3.2 InterDyck-Reachability

This article focuses on the reachability problem related to the InterDyck language. The InterDyck language is a prototypical example of a non-context-free language. Informally, the InterDyck language describes the intersection of multiple Dyck languages, where the parentheses in each Dyck language can be arbitrarily interleaved. For example, consider two Dyck strings "$\llbracket\,\rrbracket$" $\in L_b$ and "$\llparenthesis\,\rrparenthesis$" $\in L_p$. All of "$\llbracket\,\llparenthesis\,\rrbracket\,\rrparenthesis$", "$\llparenthesis\,\llbracket\,\rrparenthesis\,\rrbracket$", and "$\llbracket\,\rrbracket\,\llparenthesis\,\rrparenthesis$" belong to the InterDyck language based on L_b and L_p. We formally define the class of InterDyck languages based on an *interleaving operation* $\odot$. Formally, $\odot : \Sigma^* \times \Sigma^* \to \mathcal{P}(\Sigma^*)$ is a binary operator that takes two strings and returns a set of strings, where $\mathcal{P}(\cdot)$ denotes the power-set operator. The operator $\odot$ is inductively defined as follows: for every $u \in \Sigma^*$, we have $u \odot \epsilon = \epsilon \odot u = \{u\}$. Moreover, for every $\alpha_1, \alpha_2, u_1, u_2 \in \Sigma^*$, $\alpha_1 u_1 \odot \alpha_2 u_2 = \{\alpha_1 w \mid w \in (u_1 \odot \alpha_2 u_2)\} \cup \{\alpha_2 w \mid w \in (\alpha_1 u_1 \odot u_2)\}$. The interleaving operator can be extended to languages with

$$L_1 \odot L_2 = \bigcup_{u_1 \in L_1,\, u_2 \in L_2} u_1 \odot u_2.$$

Note that $\odot$ is associative—i.e., $(L_1 \odot L_2) \odot L_3 = L_1 \odot (L_2 \odot L_3)$—and hence can be extended to k Dyck languages with disjoint alphabets. If $L_1, L_2, \ldots, L_N$ are N Dyck languages with disjoint alphabets, we define InterDyck $:= L_1 \odot L_2 \odot \ldots \odot L_N$. The InterDyck-reachability problem is an L-reachability problem by restricting L to an InterDyck language. In particular,

Definition 3.1 (InterDyck-Reachability). Given an edge-labeled digraph $G = (V, E)$ and an InterDyck language, compute all InterDyck-reachable node pairs in G.

3.3 Problem Formulation

Our technique eliminates graph edges to improve solving InterDyck-reachability. To determine the set of edges to eliminate, we formally define the "usefulness" of each edge.

Definition 3.2 (L-Contributing Edges). Given an instance *Reach*$\langle L, G\rangle$ of L-reachability, an edge $u \to v \in G$ is *contributing* to L-reachability iff it is in an L-path in G, i.e., there exists a path "$p = \ldots \to u \to v \to \ldots$" in G and $R(p) \in L$.

Example 3.3. In the motivating example from Section 2 (Figure 1(b)), the contributing edges are $v_a \xrightarrow{\llbracket_f} v_b$, $v_b \xrightarrow{\llparenthesis_8} v_t$, $v_t \xrightarrow{\rrbracket_f} ret_1$, and $ret_1 \xrightarrow{\rrparenthesis_8} v_c$ that appear in the simplified graph G_f in Figure 2(c).

In this article, we consider the following graph-simplification problem for InterDyck-reachability:

> Given an InterDyck-reachability problem instance *Reach*$\langle$InterDyck$, G\rangle$, simplify graph G by eliminating non-InterDyck-contributing edges.

It is interesting to note that there is a correspondence between the reachability problem in Definition 3.1 and the graph-simplification problem stated above. Intuitively, based on Definition 3.2, the problem of deciding all InterDyck-contributing edges should be as hard as computing InterDyck-reachability. We now establish the undecidability of computing all InterDyck-contribution edges via a reduction from InterDyck-reachability. Note that InterDyck-reachability is undecidable even when restricted to the *single-source-single-sink* variant [17].

Theorem 3.4. *It is undecidable to compute all* InterDyck*-contributing edges in a graph G.*

Proof. We show a reduction from the single-source-single-sink variant of InterDyck-reachability. Given any single-source-single-sink InterDyck-reachability problem instance *Reach*⟨InterDyck, G⟩, we first introduce a new Dyck language L_p with an alphabet $\Sigma_{L_p} = \{(\!|, |\!)\}$ and $\Sigma_{L_p} \cap \Sigma_{\text{InterDyck}} = \emptyset$. Define InterDyck′ = InterDyck $\odot L_p$. Let s and t be the source and sink in graph G, respectively. We construct a new graph G' by inserting two additional edges $s' \xrightarrow{(\!|} s$ and $t \xrightarrow{|\!)} t'$. Based on the reduction, we can see that the edge $s' \xrightarrow{(\!|} s$ is an InterDyck′-contributing edge in G' iff t is InterDyck-reachable from s in G. It is straightforward to verify that the reduction is decidable. □

To side-step the undecidability of graph simplification, we describe two novel relaxations in Section 4. Here we define the notion of *correctness* of graph simplification, which is similar to the concept of soundness in static analysis. Let ϕ be the set of all InterDyck-contributing edges in G. Intuitively, a graph-simplification algorithm computes an over-approximating solution ϕ' (of "apparently contributing" edges). Therefore, if it determines an edge to be non-InterDyck-contributing, the edge can be safely eliminated graph G. To sum up,

Definition 3.5 (Correctness). A graph-simplification algorithm is correct if and only if it computes a solution ϕ' to the contributing-edge problem such that $\phi' \supseteq \phi$.

4 IDENTIFYING CONTRIBUTING EDGES

Central to our graph-simplification approach is the idea of eliminating non-InterDyck-contributing edges in G. Due to Theorem 3.4, identifying non-L-contributing edges is as hard as computing the L-reachability problem, and solving InterDyck-reachability, in general, is undecidable [17].

Our key idea is to cast the undecidable problem (i.e., identifying InterDyck-contributing edges in a digraph G) to an easier problem (i.e., identifying Dyck-contributing edges in a bidirected graph G') that admits an efficient polynomial-time solution. In particular, we give two forms of relaxation:

- *Graph Relaxation.* We first relax the general directed graph G to a bidirected graph G' by introducing reverse edges (Section 4.1); and
- *Formulation Relaxation.* We then relax the InterDyck-reachability problem in the bidirected graph G' to the Dyck-reachability problem in a contracted graph (L_x-graph) derived from G', where L_x represents a Dyck language in InterDyck (Section 4.2).

The benefit of our relaxations is that Dyck-reachability can be efficiently solved in $O(m+n\cdot\alpha(n))$ time on a bidirected L_x-graph with m edges and n nodes [27]. The Dyck-reachability algorithm also identifies an *anchor-node* property in L_x-graph. We utilize the anchor-node property to identify the non-Dyck-contributing edges in the L_x-graph (Section 4.3). Finally, if an edge is not a Dyck-contributing edge in the L_x-graph, its corresponding edge in G is a not an InterDyck-contributing edge. Graph-simplification can be performed safely by eliminating those edges in G.

Figure 3 provides a roadmap to this section: it summarizes the relations among various lemmas. Combining these lemmas together, it provides a criterion for identifying—and removing—non-contributing edges in G.

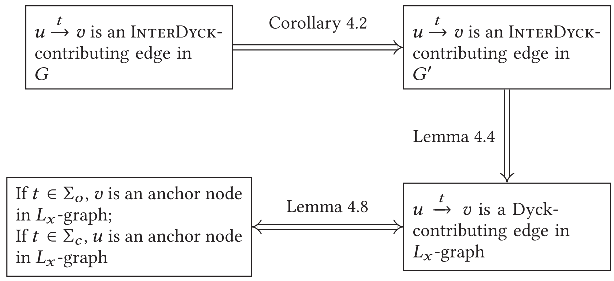


Fig. 3. Summary of lemmas used in Section 4. Let L_x be a Dyck language in InterDyck. The alphabet $\Sigma_{L_x} = \Sigma_o \cup \Sigma_c$ of L_x can be partitioned into Σ_o and Σ_c representing open and close parentheses, respectively. Let $t \in \Sigma_{L_x}$.

4.1 Graph Relaxation: From G to G'

In edge-labeled graphs, we say that two edges of the form $u \xrightarrow{\llparenthesis_i} v$ and $v \xrightarrow{\rrparenthesis_i} u$ are *reverse edges* for each other. Given an edge-labeled input graph $G = (V, E)$, we construct a relaxed graph $G' = (V, E')$ by introducing additional reverse edges. In particular, the node set $V \in G$ remains unmodified. Let $\llparenthesis_i$ and $\rrparenthesis_i$ be two matched open and close parentheses in InterDyck. The edge set $E' \in G'$ is constructed as follows:

— For each edge $u \xrightarrow{\llparenthesis_i} v \in E$, we insert both edges $u \xrightarrow{\llparenthesis_i} v$ and $v \xrightarrow{\rrparenthesis_i} u$ into E';
— For each edge $u \xrightarrow{\rrparenthesis_i} v \in E$, we insert both edges $u \xrightarrow{\rrparenthesis_i} v$ and $v \xrightarrow{\llparenthesis_i} u$ into E'.

Each edge $e \in G$ is mapped to two *corresponding edges* in G', denoted as set $h(e)$. The size of the edge set $|E'| = 2|E|$ and relaxed graph G' can potentially be a *multi-graph*, i.e., between two nodes u and v, there can be more than one edge $u \xrightarrow{t} v$ with a given label t. Based on the construction of G', it follows immediately that InterDyck-reachability in G' over-approximates InterDyck-reachability in G.

Lemma 4.1 (Relaxed Reachability in G'). *Given two nodes u and v, if v is InterDyck-reachable from u in G, node v must be InterDyck-reachable from u in G'.*

Corollary 4.2. *If an edge $e = u \rightarrow v$ is an InterDyck-contributing edge in G, the corresponding edges in $h(e)$ are InterDyck-contributing in G'.*

4.2 Formulation Relaxation: From InterDyck-Reachability to Dyck-Reachability

We now describe how to relax the problem of determining InterDyck-contributing edges to the problem of determining Dyck-contributing edges in the bidirected graph G'.

Let InterDyck be InterDyck $= L_1 \odot L_2 \odot \ldots \odot L_N$. Note that each L_x in InterDyck represents a Dyck language for all $x \in [1, N]$. Let L_x be a Dyck language and $\Sigma_{L_x} = \{\llparenthesis_1, \rrparenthesis_1, \ldots, \llparenthesis_k, \rrparenthesis_k\}$. Given a valid InterDyck string s, we could indeed "extract" a substring s' by concatenating all L_x terminals in s. The resulting string s' is always a valid Dyck string. For example, let s be a valid InterDyck string "$\llparenthesis_1 [\![_2 \llparenthesis_2]\!]_2 \rrparenthesis_2 \rrparenthesis_1$". The "extracted" substring is "$\llparenthesis_1 \llparenthesis_2 \rrparenthesis_2 \rrparenthesis_1$", which is a valid Dyck string. In general, let $\llparenthesis_i$ be a terminal in Σ_{L_x}. It is straightforward to see that if $\llparenthesis_i$ is in a valid InterDyck string, $\llparenthesis_i$ must belong to a valid Dyck (L_x) string as well.

We extend the discussion about InterDyck strings to the InterDyck-reachability problem on graphs. Consider an InterDyck-reachability instance *Reach*$\langle$InterDyck, $G'\rangle$. Rather than "extracting" an L_x substring from an InterDyck string, we build a contracted graph called the L_x-graph from G'. Intuitively, an L_x-graph is derived from G by maintaining only L_x-edges in G', merging the nodes joined by any t-edge, and deleting any t-edges where $t \notin L_x$.

Definition 4.3 (L_x-Graph). Let L_x be a Dyck language. Given an input graph G', the L_x-graph is constructed by replacing labels of $u \xrightarrow{t} v$ edges to ϵ-labels in G' where $t \notin L_x$.

LEMMA 4.4. *Let $L_x \in$ INTERDYCK and $t \in \Sigma_{L_x}$. If an edge $u \xrightarrow{t} v$ is INTERDYCK-contributing in G', it is a Dyck-contributing edge in the L_x-graph.*

4.3 Identifying Dyck-Contributing Edges

According to Definition 3.2, identifying Dyck-contributing edges requires computing Dyck-reachability. The L_x-graph is essentially a bidirected graph with each edge labeled by a terminal t in a Dyck language L_x. Dyck-reachability on bidirected graphs can be solved in $O(m + n \cdot \alpha(n))$ time [1].

4.3.1 Computing Dyck-Reachability in L_x-Graphs. In general, Dyck-reachable node pairs (u, v) in a graph $G = (V, E)$ can be described as a binary relation DYCK over $V \times V$. Specifically, a pair $(u, v) \in$ DYCK iff node v is Dyck-reachable from u in G. The relaxed L_x-graph is a bidirected graph. One property that is special for bidirected Dyck-reachability is that the DYCK relation on a bidirected graph is an *equivalence relation* [27]: (i) it is reflexive and transitive based on the Dyck grammar given in Equation (1) (see Section 3.1); and (ii) it is symmetric based on the G' construction given in Section 4.1.[1] Due to the equivalence property, we can collapse all nodes that belong to the DYCK relation into a single representative node, i.e., node v is Dyck-reachable from u in G' iff u and v belong to the same representative node in the L_x-graph. However, we have only the L_x-graph rather than the DYCK relation itself, so we are not in a position to find and collapse all Dyck-reachable nodes. Instead, the collapsing can be done on-the-fly as Dyck-reachability is computed.

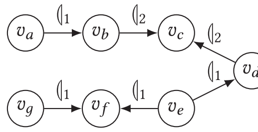
Following the work of Chatterjee et al. [1], we summarize the intuition of the algorithm for solving Dyck-reachability in L_x-graphs. When there are two incoming edges $u \xrightarrow{(\!|_i} w$ and $v \xrightarrow{(\!|_i} w$ for a node w, the algorithm performs two operations:

- *Node collapsing*: The algorithm collapses two nodes u and v into a single representative node $n_{\{u,v\}}$ and updates the edges of node $n_{\{u,v\}}$ based on u and v. Node $n_{\{u,v\}}$ inherits the incoming and outgoing edges of u and v. To avoid any misunderstanding about the original optimal Dyck-reachability algorithm in [1], we clarify that there is no concept of merged nodes in the work of Chatterjee et al. [1]. Nodes with Dyck Relation are unioned in a disjoint-set data structure. In our article, we assume they are merged into a concrete representative node to facilitate our presentation.
- Edge merging: After the node collapsing of the nodes u, v there exists multiple $n_{\{u,v\}} \xrightarrow{(\!|_i} w$ edges. Remove the redundant edges until there is only one such edge remaining in the graph. We regard it as edge merging.

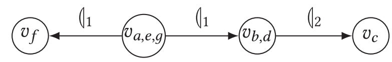
Node collapsing and edge merging may introduce additional Dyck-reachable node pairs in the graph. Therefore, we continue the process until there are no newly introduced Dyck-reachable nodes. We refer to such an algorithm as procedure OPT-DYCK().

LEMMA 4.5 (CORRECTNESS OF OPT-DYCK [1]). *In a bidirected L_x-graph, node v is Dyck-reachable from node u iff u and v are in the same representative node after running OPT-DYCK() on the L_x-graph.*

[1]The DYCK relation in a general digraph is not symmetric. Therefore, it is not an equivalence relation in the general case.



(a) A graph G with a Dyck path over the word "$\llparenthesis_1\llparenthesis_2\rrparenthesis_2\rrparenthesis_1\llparenthesis_1\rrparenthesis_1$".



(b) Constructed L_p-graph, where L_p denotes the Dyck language of parentheses.

Fig. 4. Anchor-k node example.

To facilitate further discussion, let G_f denote the resulting graph after running Opt-Dyck() on G. We define rep_node[·] as a mapping from a node in G to its representative node in G_f. For example, if $u \in V(G)$ is merged to the representative node $u_f \in V(G_f)$, we write rep_node$[u] = u_f$.

4.3.2 Anchor Nodes. Lemma 4.5 indicates that every Dyck-path in the L_x-graph is obtained via edge merging in Opt-Dyck(). To identify Dyck-contributing edges in the L_x-graph, we leverage the *anchor node* w of the two edges $u \xrightarrow{\llparenthesis_k} w$ and $v \xrightarrow{\llparenthesis_k} w$ merged by Opt-Dyck(). Intuitively, every Dyck-path computed by Opt-Dyck() is associated with at least one such anchor node. Formally, we have the following definition:

Definition 4.6 (Anchor Node). Node w is an anchor-$\llparenthesis_i$ node in an L_x-graph iff there exist nodes u, v, and w' in the L_x-graph, such that there are two distinct edges $u \xrightarrow{\llparenthesis_i} w$ and $v \xrightarrow{\llparenthesis_i} w'$ existing in the L_x-graph and rep_node$[w]$ = rep_node$[w']$ after running Opt-Dyck().

Example 4.7 (Anchor-node Example). Figure 4(a) presents an edge-labeled graph with a Dyck path $v_a \xrightarrow{\llparenthesis_1} v_b \xrightarrow{\llparenthesis_2} v_c \xrightarrow{\rrparenthesis_2} v_d \xrightarrow{\rrparenthesis_1} v_e \xrightarrow{\llparenthesis_1} v_f \xrightarrow{\rrparenthesis_1} v_g$. Figure 4(b) is the constructed L_p-graph after node collapsing. For the graph G in Figure 4(a), the anchor-$\llparenthesis_2$ node is v_c, anchor-$\llparenthesis_1$ nodes include v_b, v_d, and v_e. To verify that node v_c is an anchor-$\llparenthesis_2$ node, without loss of generality, it suffices to let $w' = v_c$, $u = v_b$, and $v = v_d$ according to Definition 4.6. The anchor-$\llparenthesis_2$ node (v_b) can also be detected during the execution of Opt-Dyck(). When the node collapsing happens because of the two edges $v_b \xrightarrow{\llparenthesis_2} v_c$ and $v_d \xrightarrow{\llparenthesis_2} v_c$, node v_c will be marked as an anchor-$\llparenthesis_2$ node. Similarly, to verify that node v_b is an anchor-$\llparenthesis_1$ node, according to the definition, we can set $w' = v_d$, $u = v_a$, and $v = v_e$. It will also be marked as an anchor-$\llparenthesis_1$ node when collapsing the nodes v_a and v_e due to the two edges $v_a \xrightarrow{\llparenthesis_1} v_b$ and $v_e \xrightarrow{\llparenthesis_1} v_d$.

Lemma 4.8. *An edge $u \xrightarrow{\llparenthesis_i} v$ is a contributing edge for Dyck-reachability in a bidirected L_x-graph iff v is an anchor-$\llparenthesis_i$ node in the L_x-graph. Similarly, an edge $u \xrightarrow{\rrparenthesis_i} v$ is a contributing edge for Dyck-reachability in an L_x-graph iff u is an anchor-$\llparenthesis_i$ node in the L_x-graph.*

Proof. Without loss of generality, we consider the $u \xrightarrow{\llparenthesis_i} v$ case. We prove the forward direction by induction on the length of the Dyck-path that involves the edge $u \xrightarrow{\llparenthesis_i} v$.

Base case. The contributing edge $u \xrightarrow{\llparenthesis_i} v$ is involved in a Dyck-path of length 2. There must exist another node w such that $v \xrightarrow{\rrparenthesis_i} w$. Because L_x-graph is bidirected, we have $w \xrightarrow{\llparenthesis_i} v \in E$. Therefore, v is an anchor-$\llparenthesis_i$ node.

Inductive step. Assume that the lemma holds for contributing edges involved in a Dyck-path with length less than or equal to $2p$. Suppose that a contributing edge $e = u \xrightarrow{\llparenthesis_i} v$ is involved in a Dyck-path of length $2p + 2$ and not involved in any Dyck-path with length less or equal to $2p$. Consider the Dyck grammar rule $S \to (\!|_i S |\!)_i \mid SS$.

— If the Dyck-path is generated based on the first rule, edge e is the first edge in the Dyck-path. There must exist nodes v', w in the same Dyck-path such that the subpath between v and v' is also a Dyck-path, and $v' \xrightarrow{|\!)_i} w \in E$. By Lemma 4.5, we have rep_node[v] = rep_node[v']. Based on the bidirectedness, we have $u \xrightarrow{(\!|_i} v, w \xrightarrow{(\!|_i} v' \in E$. By the definition of anchor-$(\!|_i$ nodes, we conclude that v is an anchor-$(\!|_i$ node.
— If the Dyck-path is generated by the second rule, edge e is involved in a Dyck-path with length less than or equal to $2p$, thus v is an anchor-$(\!|_i$ node.

Now we prove the backward direction. Suppose that v is an anchor-$(\!|_i$ node in the L_x-graph. According to the definition, there exists a node v' such that rep_node[v] = rep_node[v'], and there exists another node w with $w \neq u$ and edge $w \xrightarrow{(\!|_i} v' \in E(L_x)$. Because rep_node[$v$] = rep_node[$v'$], i.e., v and v' are merged by Opt-Dyck(), there exists a Dyck-path $p = v \to v'$. By utilizing the bidirectedness of the L_x-graph, we have $v' \xrightarrow{|\!)_i} w \in E(L_x)$, as well. Then, $u \xrightarrow{(\!|_i} v$, p, and $v' \xrightarrow{|\!)_i} w \in E(L_x)$ form a new Dyck-path. Therefore, $u \xrightarrow{(\!|_i} v$ is a contributing edge. □

To obtain the main theorem, we revisit Figure 3. In general, if an edge $u \to v$ is InterDyck-contributing in G, it must be an InterDyck-contributing edge in relaxed graph G' (Corollary 4.2). Any InterDyck-contributing edge in G' must be a Dyck-contributing edge in an L_x-graph derived from G' (Lemma 4.4). Finally, the problem of deciding Dyck-contributing edges is equivalent to deciding the corresponding anchor-$(\!|_i$ nodes in the L_x-graph (Lemma 4.8). Putting everything together, we have the Theorem 4.9:

Theorem 4.9. *Let L_x be a Dyck language in InterDyck and $(\!|_i, |\!)_i \in \Sigma_{L_x}$. If either an edge $u \xrightarrow{(\!|_i} v$ or an edge $v \xrightarrow{|\!)_i} u$ is contributing to InterDyck-reachability in G, the node v is an anchor-$(\!|_i$ node in the L_x-graph.*

Corollary 4.10. *If a node v is not an anchor-$(\!|_i$ node in the L_x-graph, both edges $u \xrightarrow{(\!|_i} v$ and $v \xrightarrow{|\!)_i} u$ are non-contributing edges for InterDyck-reachability in G.*

Thus, the proposed graph-simplification algorithm can remove from G all edges that meet the criterion given in Corollary 4.10.

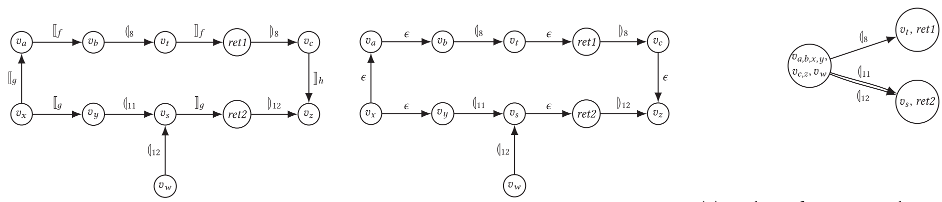
5 GRAPH-SIMPLIFICATION ALGORITHM

This section discusses the graph-simplification algorithm. Section 5.1 describes the key steps in the algorithms. Section 5.2 presents the main algorithm. Section 5.3 discusses the correctness and complexity of the simplification algorithm. Section 5.4 extends our graph-simplification algorithm to a variant that works more effectively when the graph G is already bidirected.

5.1 Key Steps

There are two key steps in the graph simplification: (1) constructing the L_x-graphs and (2) identifying all anchor-$(\!|_i$ nodes.

5.1.1 L_x-Graph Construction. Consider an interleaved Dyck language InterDyck = $L_1 \odot \ldots \odot L_N$. Given a relaxed graph G', to identify the anchor-$(\!|_i$ nodes, our algorithm needs to construct



(a) The graph to be simplified.

(b) Constructed L_p-graph.

(c) Identifying anchor-$(\!|_i$ nodes based on OPT-DYCK-MODIFIED.

Fig. 5. Collection of anchor-$(\!|_i$ node information. Figure 5(a) repeats the graph of our motivating example from Figure 1(b). Graph simplification involves repeated application of Algorithm 2. Figure 5(b) illustrates the L_p-graph construction result based on Section 5.1.1 during the first application of the algorithm. Figure 5(c) gives the corresponding graph after running OPT-DYCK-MODIFIED described in Section 5.1.2. In Figures 5(b) and (c), each $(\!|$, $[\![$ edge has a corresponding reverse $|\!)$, $]\!]$ edge. We omit reverse edges for brevity.

Procedure 1: GetLxGraph(G, L_x)

Input : Edge-labeled relaxed bidirected graph $G = (V, E)$, a Dyck language L_x
Output: An L_x-graph G_x

1 rep_node $\leftarrow$ a disjoint-set of size $|V|$.
2 **foreach** $u \xrightarrow{l} v \in E$ **do**
3   **if** $l \notin \Sigma_{L_x}$ **then**
4     $E \leftarrow E \setminus \{u \xrightarrow{l} v\}$
5     $E \leftarrow E \cup \{u \xrightarrow{\epsilon} v\}$
6 **return** $G_x = (V, E)$

an L_x-graph for each $x \in \{1, \ldots, N\}$. We construct an L_x-graph by replacing non-L_x edge-labels by ϵ-labels in the graph of G'. Procedure 1 gives the L_x-graph-construction algorithm. For each non-L_x edge, Line 4 removes the original non-L_x edge, and line 5 adds the edge back with a new ϵ-label.

We describe the L_x-graph construction of the motivating example in Figure 5(a). Recall that we have INTERDYCK $= L_b \odot L_p$. We present how to construct the L_p-graph for the motivating example. The procedure iterates through non-L_p edges. In Figure 5(a), the first non-L_p edge is $v_x \xrightarrow{[\![_g} v_a$ because $[\![_g \in \Sigma_{\text{INTERDYCK}} \setminus \Sigma_{L_p}$. The edge $v_x \xrightarrow{[\![_g} v_a$ is replaced by $v_x \xrightarrow{\epsilon} v_a$. We continue replacing non-L_p edges until there are no more non-L_p edges. Figure 5(b) depicts the final L_p-graph.

5.1.2 Anchor-$(\!|_i$ Node Identification. The second step in graph simplification is anchor-$(\!|_i$ node identification. We modify the OPT-DYCK() algorithm by Chatterjee et al. [1] to collect the anchor-$(\!|_i$ node information. We denote the modified version as OPT-DYCK-MODIFIED(). Recall that OPT-DYCK() tracks the number of incoming edges with the same open-parenthesis label for each node in the graph. If there are two incoming edges $v_b \xrightarrow{(\!|_i} v_a$ and $v_c \xrightarrow{(\!|_i} v_a$ with the same edge label $(\!|_i$, then the OPT-DYCK algorithm performs a *node-collapsing* between node v_b and v_c.

OPT-DYCK-MODIFIED() leverages the node-collapsing process in OPT-DYCK() to mark anchor-$(\!|_i$ nodes. In particular, when OPT-DYCK() detects two incoming edges $v_b \xrightarrow{(\!|_i} v_a$ and $v_c \xrightarrow{(\!|_i} v_a$ with an open-parenthesis edge label $(\!|_i$, OPT-DYCK-MODIFIED() marks the node v_a as an anchor-$(\!|_i$ node. After the OPT-DYCK-MODIFIED() finishes, we can retrieve the set of anchor-$(\!|_i$ nodes in the L_x-graph based on the marking in the merged graph. For any node v in the L_x-graph, it is an anchor-$(\!|_i$ node iff rep_node(v) is marked as an anchor-$(\!|_i$ node by OPT-DYCK-MODIFIED().

ALGORITHM 2: The graph simplification iteration.

```
   Input  : Edge-labeled directed graph G = (V, E), an InterDyck language L = L_1 ⊙ ... ⊙ L_n;
   Output : A new edge-labeled directed graph G_f

1  contrib_edges ← ∅
2  G' ← RelaxedGraph(G)
3  for i ← 1 to n do
4      G''_i ← GetLxGraph(G', L_i)
5      anchor_nodes ← Opt-Dyck-Modified(G''_i)
6      foreach v_l ∈ anchor_nodes do
7          foreach x ∈ In[v] do
8              foreach edge e = x --t--> v do
9                  if t == l then
10                     contrib_edges ← contrib_edges ∪ {e}
11         foreach x ∈ Out[v] do
12             foreach edge e = v --t--> x do
13                 if t == l̄ then
14                     contrib_edges ← contrib_edges ∪ {e}
15 G_f ← (V, contrib_edges )
16 return G_f
```

Notice that the original Opt-Dyck algorithm runs in time $O(m+n\cdot\alpha(n))$, where α is the inverse Ackermann function [1]. After the modification, the extra running time for each node-merging is $O(1)$, and thus the complexity of Opt-Dyck-Modified is still $O(m+n\cdot\alpha(n))$

Example 5.1. We continue our example using the graph shown in Figure 5(b). We apply Opt-Dyck-Modified() on this L_p-graph. Figure 5(c) gives the resulting graph. There exist two $\llparenthesis_{12}$-edges pointing to node $\{v_s, ret_2\}$ and two $\llparenthesis_8$-edges pointing to the node $\{v_t, ret_1\}$. Therefore, Opt-Dyck-Modified() collects the information that nodes v_t, ret_1 are anchor-$\llparenthesis_8$ nodes and v_s, ret_2 are anchor-$\llparenthesis_{12}$ nodes.

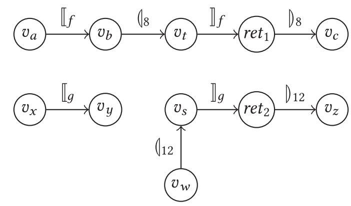
5.2 The Simplification Algorithm

Algorithm 2 gives the graph-simplification algorithm. In lines 1–2, contrib_edges is initialized to an empty set. It contains the set of potential InterDyck-contributing edges in the original graph G when the algorithm terminates. We then obtain the relaxed graph G' defined in Section 4.1. The loop iterates over each Dyck language L_x in InterDyck $= L_1 \odot \ldots \odot L_N$ (lines 3–14). It first builds the L_x-graph based on Procedure 1. After we construct the L_x-graph, the algorithm invokes Opt-Dyck-Modified() described in Section 5.1.2 to collect anchor-$\llparenthesis_i$ nodes in the L_x-graph. The variable anchor_nodes stores a set of anchor nodes of the form v_l, where v is the node in the L_x-graph, l is the open-parenthesis edge label for the corresponding anchor. In lines 6–14, for each anchor node, we add its corresponding contributing edges to the set contrib_edges. After collecting the contributing edges for each L_x-graph, contrib_edges, the union is returned as the new edge set of the graph. It serves as the input for the next iteration (lines 15–16). The graph-simplification algorithm terminates if there are no edges removed in one iteration.

Example 5.2. We illustrate how Algorithm 2 eliminates non-contributing edges in the original graph of our motivating example, i.e., Figures 1(b) and 5(a). After running the L_x-graph construction (Procedure 1) and Opt-Dyck-Modified for both the parenthesis language L_p and the bracket language L_b, Figure 6(a) gives the anchor-$\llparenthesis_i$ nodes identified by the first application of Algorithm 2. It identifies the non-contributing edges based on Lemma 4.8. For instance, provided that v_s is an anchor-12 nodes, all the incoming $\llparenthesis_{12}$ edges to v_s and outgoing $\rrparenthesis_{12}$ from v_s edges are

anchor-12:	v_s, $ret2$
anchor-8:	v_t, $ret1$
anchor-f:	v_b, v_t
anchor-g:	v_y, v_s, v_w

(a) Collected anchor-k node information.



(b) Resulting graph after removing non-contributing edges.

Fig. 6. Elimination of non-contributing edges. Figure 6(a) lists all anchor-k nodes identified in the first application of Algorithm 2. Figure 6(b) gives the simplified graph after the first application. It is the same as Figure 2(a).

contributing edges. By removing the non-contributing ones, we get the resulting graph in Figure 6(b). By applying Algorithm 2 two more times, we obtain the final graph, shown in Figure 2(c).

5.3 Correctness and Complexity

We establish the correctness of Algorithm 2 and analyze its complexity. In lines 6–10, the algorithm collects all the incoming open-parenthesis anchor-labeled edges and outgoing close-parenthesis anchor-labeled edges. Due to Theorem 4.9, all contributing edges are in contrib_edges. Then it suffices to show that the derived L_x-graph in line 4 is correct and the Opt-Dyck-Modified() collects all anchor-$\llparenthesis_i$ nodes.

Lemma 5.3. *Opt-Dyck-Modified() collects all anchor-$\llparenthesis_i$ nodes for each L_x-graph.*

Proof. Opt-Dyck-Modified() tracks the incoming edges incident on the same node with the same open-parenthesis label, performs a node collapsing, and generates an anchor-$\llparenthesis_i$ node. For each anchor-$\llparenthesis_i$ node generated, the previous two (or more) incoming edges incident on the same node become one. Suppose that an anchor-$\llparenthesis_i$ node has not been collected by Opt-Dyck-Modified(); then there will be two incoming edges to the anchor-$\llparenthesis_i$ node with the same open-parenthesis label, which contradicts the fact that Opt-Dyck-Modified() is guaranteed to find every pair of incoming edges with the same open-parenthesis label [1]. □

Theorem 5.4. *For an InterDyck language $L_1 \odot \ldots \odot L_N$ and a graph G, Algorithm 2 computes an over-approximation ϕ' of all InterDyck-contributing edges in G, i.e., $\phi' \supseteq \phi$ where ϕ denotes the exact solution.*

Next, we analyze the complexity of each iteration of the simplification. In Algorithm 2, the loop body in lines 3–14 contains two procedure calls: GetLxGraph and Opt-Dyck-Modified(). Given a graph with m edges, the time complexity of Opt-Dyck-Modified() is $O(m + n \cdot \alpha(n))$ [1]. The GetLxGraph procedure performs a linear traversal of edges; thus, its complexity is $O(m)$. The overall algorithm iterates over all N Dyck languages in InterDyck. N is usually considered as a constant. Therefore, the time complexity of Algorithm 2 is $O(m + n \cdot \alpha(n))$.

5.4 Graph Simplification for Bidirected Input Graphs

In practice, many client analyses work on graphs that are already bidirected: for each edge $u \xrightarrow{\llbracket_a} v$, there is already a corresponding reverse edge $v \xrightarrow{\rrbracket_a} u$, and vice versa. Bidirectedness arises in formulations of alias analyses [16, 23, 27]. This section proposes a variant of our simplification algorithm that works effectively when the graph of the original problem is already bidirected.

Bidirectedness introduces a special challenge for graph simplification because every edge now forms an InterDyck-path with its reverse edge, and therefore every edge in a bidirected graph is "contributing" as defined in Definition 3.2. For example, for an arbitrary edge $u \xrightarrow{[\![_a} v$, there always exists an InterDyck-path: $u \xrightarrow{[\![_a} v \xrightarrow{]\!]_a} u$, where $v \xrightarrow{]\!]_a} u$ is the reverse edge of $u \xrightarrow{[\![_a} v$ in the graph. Obviously, this kind of InterDyck-path never causes there to be an InterDyck-reachable node pair (u, v), where $u \neq v$. We define these InterDyck-paths to be trivial InterDyck-paths.

Definition 5.5 (Trivial InterDyck-Path). An InterDyck-path p is *trivial* if (i) it starts and ends at the same node u, and (ii) in the realized InterDyck path string $R(p)$, each open parenthesis t of an edge $u \xrightarrow{t} v$ is always matched with the close parenthesis t' of the corresponding reverse edge $v \xrightarrow{t'} u$.

Based on Definition 5.5, trivial InterDyck-paths cause *every* edge of a bidirected graph to be a contributing edge. However, trivial InterDyck-paths only identify reflexive InterDyck-reachable node pairs of the form (u, u), which are InterDyck-reachable even without these paths, because the empty string is in the InterDyck language. Therefore, if an edge is only involved in trivial InterDyck-paths, removing the edge does not affect any InterDyck-reachable node pairs. Consequently, for bidirected graphs, we only want to track edges that are involved in non-trivial InterDyck-paths. Next, we extend the original definition of contributing edges (Definition 3.2) to bidirected graphs.

Definition 5.6 (Contributing Edges for Bidirected Graphs). In a bidirected graph G, an edge is considered to be *contributing* in G iff it is involved in a non-trivial InterDyck-path.

To identify contributing edges in bidirected graphs G, our insight is to divide all contributing edges into two categories based on the corresponding "matching edges." Here we define the concept of matching edges to facilitate the discussion.

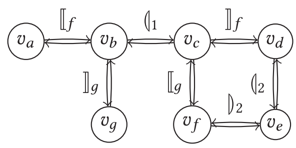
Definition 5.7 (Matching Edges). In an InterDyck-path p, an edge $e = u \xrightarrow{t} v$ is a *matching edge* of another edge $e' = u' \xrightarrow{t'} v'$ in the path p, iff t, and t' form a pair of matching parentheses in the realized string $R(p)$. e' is also a matching edge for e.

The two categories of contributing edges are:

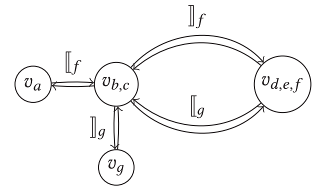
(i) Edges of the form $e = u \xrightarrow{t} v$ such that there exists an InterDyck-path in which the matching edge of e is not its corresponding reverse edge. For example, in Figure 7(a), the edge $v_d \xrightarrow{(\!|_2} v_e$ is in a non-trivial InterDyck-path $v_d \xrightarrow{(\!|_2} v_e \xrightarrow{|\!)_2} v_f$. In this path, the matching edge of $v_d \xrightarrow{(\!|_2} v_e$ is $v_e \xrightarrow{|\!)_2} v_f$, which is not the reverse edge of $v_d \xrightarrow{(\!|_2} v_e$. Therefore, $v_d \xrightarrow{(\!|_2} v_e$ is a contributing edge in category (i).

(ii) Edges of the form $e = u \xrightarrow{t} v$ such that in every non-trivial InterDyck-path that contains e, e's matching edge e' is the reverse edge of e.[2] For example, in Figure 7(a), the edge $v_b \xrightarrow{(\!|_1} v_c$ is in a non-trivial InterDyck-path $v_a \xrightarrow{[\![_f} v_b \xrightarrow{(\!|_1} v_c \xrightarrow{]\!]_f} v_d \xrightarrow{(\!|_2} v_e \xrightarrow{|\!)_2} v_f \xrightarrow{[\![_g} v_c \xrightarrow{|\!)_1} v_b \xrightarrow{]\!]_g} v_g$. And its matching edge can only be its reverse edge $v_c \xrightarrow{|\!)_1} v_b$, because it is the only $|\!)_1$-edge in the graph. Therefore, the edge $v_b \xrightarrow{(\!|_1} v_c$ is a contributing edge in category (ii).

[2]In Section 5.4.2, we show that the concept defined here is too broad, and refine the concept of category (ii) contributing edges to a more desirable subset of them.



(a) A bidirected graph for InterDyck-reachability. For each edge $u \xrightarrow{\llbracket_i} v$, there always exists a reverse edge $v \xrightarrow{\rrbracket_i} u$, and vice versa. Similarly, the "$\llparenthesis_i$"- and "$\rrparenthesis_i$"-labeled edges are accompanied with the corresponding "$\rrparenthesis_i$"- and "$\llparenthesis_i$"-labeled reverse edges, respectively. We omit the labels of reverse edges for brevity.



(b) The L_b-graph for the graph in Figure 7a. When performing the node-collapsing for v_a and $v_{d,e,f}$, we need to collect the $\llparenthesis_1$ edge in the set of contributing edges in the original graph, because it connects the $\llbracket_f$ and $\rrbracket_f$ edges.

Fig. 7. A bidirected input graph.

For graph simplification, we need to collect contributing edges in both categories, and, at the same time, avoid including edges that are only involved in trivial InterDyck-paths as much as possible. For contributing edges in category (i), it suffices to run the anchor-node-identification algorithm of Section 5.2 on the original input graph G. However, anchor-node-identification cannot recognize contributing edges in category (ii). Next, we discuss the challenges and our approach to identifying contributing edges for both categories.

5.4.1 Identifying Category (i) Contributing Edges. Because the input graph G is bidirected, its relaxed graph G' is a multi-graph (see Section 4.1). For example, if $u \xrightarrow{\llparenthesis_i} v$ is an edge of G, G' contains two edges of the form $u \xrightarrow{\llparenthesis_i} v$, and thus there exist at least two edges with the identical open-parenthesis label that point to the same node v. Consequently, the result of running the anchor-node-identification algorithm on G' returns many edges that only contribute to trivial InterDyck-paths. However, thanks to the bidirectedness of the graph G, it is feasible to perform anchor-node-identification directly on the original input graph G instead of the relaxed graph G'. Running on the original graph, the anchor-node-identification algorithm collects all contributing edges whose matching edges are not their reverse edges as described in Section 4.3, i.e., all the contributing edges in category (i). If the algorithm works on the original graph G directly, there is only one $\llparenthesis_i$-labeled edge that points to node v, and consequently, v is not an anchor node. In essence, by working on G, the anchor-node-identification algorithm ignores the trivial InterDyck-paths that contain e_1 and e_2. Thus, by not having the additional edges in the relaxed graph G', the anchor-node-identification algorithm benefits from avoiding node collapsing introduced by trivial InterDyck-paths and it further avoids recognizing trivial contributing edges in the original graph G.

5.4.2 Identifying Category (ii) Contributing Edges. For a contributing edge e in category (ii), its matching edge can only be its reverse edge in non-trivial InterDyck-paths. Identifying contributing edges of a category (ii) requires matching-edge information for other edges in the corresponding InterDyck-paths. The aforementioned anchor-node-identification algorithm is not aware of matching-edge information for any other edges in the InterDyck-paths, thus, it cannot identify the category (ii) contributing edges. Consider the bidirected graph with a non-trivial InterDyck-path $v_a \xrightarrow{\llbracket_f} v_b \xrightarrow{\llparenthesis_1} v_c \xrightarrow{\rrbracket_f} v_d \xrightarrow{\llparenthesis_2} v_e \xrightarrow{\rrparenthesis_2} v_f \xrightarrow{\llbracket_g} v_c \xrightarrow{\rrparenthesis_1} v_b \xrightarrow{\rrbracket_g} v_g$ in Figure 7(a). It is a non-trivial InterDyck-path, because it does not start and end at the same node. Moreover, not all edges have their reverse edges as matching edges. For example, the edge $v_d \xrightarrow{\llparenthesis_2} v_e$ has the matching edge

$v_e \xrightarrow{\rrparenthesis_2} v_f$ instead of its reverse edge $v_e \xrightarrow{\rrparenthesis_2} v_d$. Thus, the edge $v_b \xrightarrow{\llparenthesis_1} v_c$ is a category (ii) contributing edge in this bidirected graph. Note that there is only one $\llparenthesis_1$-edge that points to node v_c. Because v_c is not an anchor node for label "$\llparenthesis_1$" according to Definition 4.6, the anchor-node-identification algorithm does not identify $v_b \xrightarrow{\llparenthesis_1} v_c$ as a contributing edge.

Another observation for category (ii) contributing edges is that these edges can be "trivially" involved in non-trivial InterDyck-paths. We give an example to illustrate such contributing edges. Consider the non-trivial InterDyck-path $v_d \xrightarrow{\llparenthesis_2} v_e \xrightarrow{\rrparenthesis_2} v_f \xrightarrow{\llbracket_g} v_c \xrightarrow{\rrbracket_g} v_f$. Clearly, the $v_f \xrightarrow{\llbracket_g} v_c$ edge is contributing to the path. However, by removing the self-cycle $v_f \xrightarrow{\llbracket_g} v_c \xrightarrow{\rrbracket_g} v_f$, the path is still an InterDyck-path that starts with v_d and ends with v_f. Thus, we only want to collect contributing edges involved in an "irreducible" part for at least one InterDyck-path. An InterDyck-path p is *reducible* if there exists a node u in the path $p = v_s \overset{a}{\rightsquigarrow} u \overset{b}{\rightsquigarrow} u \overset{c}{\rightsquigarrow} v_t$ where each of $v_s \overset{a}{\rightsquigarrow} u, u \overset{b}{\rightsquigarrow} u, u \overset{c}{\rightsquigarrow} v_t$ is a sequence of edges with the realized strings a, b, c, such that the path $p' = v_s \overset{a}{\rightsquigarrow} u \overset{c}{\rightsquigarrow} v_t$ is still an InterDyck-path. We say that p' is the *reduced* InterDyck-path of p. An InterDyck-path is irreducible if it is not a reducible InterDyck-path. The sequence of edges $u \overset{b}{\rightsquigarrow} u$ is called the *reducible part* of the InterDyck-path p. The edges that do not belong to the reducible part are called the *irreducible part* of the path. We denote these edges as strictly contributing edges. We provide a formal definition for these edges.

Definition 5.8 (Strictly Contributing Edges). Consider a contributing edge e in a bidirected edge-labeled graph G. Edge e is *strictly contributing* to InterDyck-reachability if there exists an irreducible InterDyck-path p and the edge e is in the path p.

Suppose a contributing edge is only in the reducible part of an InterDyck-path p connecting nodes v_s and v_t. Removing the edge from the graph G does not make v_s and v_t InterDyck-unreachable, because the reduced path p' of p still exists in the graph. it In light of these observations, we refine our goal: instead of trying to identify all contributing edges in category (ii), the goal becomes to identify the strictly contributing ones. If a contributing edge e is not strictly contributing, removing such an edge does not affect any InterDyck-reachable node pairs. We first present Lemma 5.9, which shows a method to generate a reduced InterDyck-path. Then we introduce Lemma 5.10, which shows a special property of strictly contributing edges in the irreducible part of the InterDyck-path.

Lemma 5.9 (Reducible InterDyck-Path). *In a bidirected graph, consider a non-trivial InterDyck-path $p = x \overset{a}{\rightsquigarrow} u \xrightarrow{t} v \overset{b}{\rightsquigarrow} v \xrightarrow{t'} u \overset{c}{\rightsquigarrow} y$ where each of $x \overset{a}{\rightsquigarrow} u, v \overset{b}{\rightsquigarrow} v, u \overset{c}{\rightsquigarrow} y$ represents a sequence of edges with the realized strings a, b, c. In particular, edge $e = u \xrightarrow{t} v$ is the reverse edge of $e' = v \xrightarrow{t'} u$. For the realized string $R(p) =$ "$atbt'c$", if its sub-string "tbt'" is an InterDyck word, then the path $p' = x \overset{a}{\rightsquigarrow} u \overset{c}{\rightsquigarrow} y$ is also an InterDyck-path connecting the same start node x and end node y in p, i.e., p' is a reduced InterDyck-path of p.*

The lemma describes one possible approach to generate a reduced InterDyck-path. The correctness follows from the definition of InterDyck language: suppose that "ac" is not an InterDyck word and "tbt'" is an InterDyck word, the word "$atbt'c$" cannot be an InterDyck word because the unmatched parentheses in "ac" are still unmatched in "$atbt'c$". With the approach to generate reduced InterDyck-paths, by applying this approach repeatedly, the resulting path containing the strictly contributing edge exhibits a special property. We present this property in Lemma 5.10, and our algorithm exploits this property to identify the set of strictly contributing edges in category (ii).

Lemma 5.10 (Matching-Edge Property for Strictly Contributing Edges). *Let $e = u \xrightarrow{t} v$ be a category (ii) strictly contributing edge. There must exist an* InterDyck*-path p with the following properties: (i) for the edge e and for its reverse edge e', there exists a pair of matching edges e_1, e_2, such that $p = \ldots e_1 \ldots e \ldots e_2 \ldots e' \ldots$ or $p = \ldots e \ldots e_1 \ldots e' \ldots e_2 \ldots$; and (ii) if t is in $\Sigma(L_x)$ of the* InterDyck *language $L_1 \odot L_2 \odot \ldots \odot L_N$, then the edge labels of e_1, e_2 described in property (i) are not in the alphabet $\Sigma(L_x)$ of the language of L_x.*

Proof. Consider a strictly contributing edge $e = u \xrightarrow{t} v$ in category (ii) with an open-parenthesis label t. Because edge e is strictly contributing, removing e from the original graph G will make some node pair (v_s, v_t) InterDyck-unreachable. Then there exists a shortest InterDyck-path of the form $p = v_s \overset{a}{\rightsquigarrow} u \xrightarrow{t} v \overset{b}{\rightsquigarrow} v \xrightarrow{t'} u \overset{c}{\rightsquigarrow} v_t$. According to Lemma 5.9, tbt' cannot form an InterDyck-word, otherwise, the path is not the shortest one. Thus, some edge e_1 in $v \overset{b}{\rightsquigarrow} v$ has a matching edge e_2 in $v_s \overset{a}{\rightsquigarrow} u$ or $u \overset{c}{\rightsquigarrow} v_t$. Therefore, we have proved the property (i) of the lemma.

Assume for the sake of contradiction that the edge labels of e, e_1, and e_2 are all in $\Sigma(L_x)$. According to the property (i) that we just proved, $p = \ldots e_1 \ldots e \ldots e_2 \ldots e' \ldots$ or $p = \ldots e \ldots e_1 \ldots e' \ldots e_2 \ldots$. Note that e is the matching edge of e', and e_1 is the matching edge of e_2. Consider the realized string $R(p)$ if $p = \ldots e_1 \ldots e \ldots e_2 \ldots e' \ldots$; the label of e_1 cannot match with the label of e_2 because the label of e is still unmatched for the L_x language. By a similar argument, if $p = \ldots e \ldots e_1 \ldots e' \ldots e_2 \ldots$, the realized string $R(p)$ cannot be a valid InterDyck-word either. Thus, the assumption that the edge labels of e, e_1, and e_2 are all in $\Sigma(L_x)$ contradicts the fact that p is an InterDyck-path. This completes the proof of the property (ii) of the lemma. □

Consider the example in Figure 7(a). We illustrate how Lemma 5.10 applies to the edge $e = v_b \xrightarrow{(\!|_1} v_c$. It is a strictly contributing edge in category (ii), i.e., in any InterDyck-path, its matching edge is always its reverse edge $e' = v_c \xrightarrow{|\!)_1} v_b$. Recall that we use L_b to denote the Dyck language for brackets and L_p to denote the Dyck language for parentheses. The $L_b \odot L_p$-path $p = v_a \xrightarrow{[\![_f} v_b \xrightarrow{(\!|_1} v_c \xrightarrow{]\!]_f} v_d \xrightarrow{(\!|_2} v_e \xrightarrow{|\!)_2} v_f \xrightarrow{[\![_g} v_c \xrightarrow{|\!)_1} v_b \xrightarrow{]\!]_g} v_g$ has the format of $p = \ldots e_1 \ldots e \ldots e_2 \ldots e' \ldots$ with $e_1 = v_a \xrightarrow{[\![_f} v_b$ and $e_2 = v_c \xrightarrow{]\!]_f} v_d$. The edge $v_b \xrightarrow{(\!|_1} v_c$ also has the label "$(\!|_1$" in the Dyck language L_p, with the labels of the two matching edges e_1, e_2 in the alphabet of L_b.

We say an edge e is *in between* two edges e_1, e_2 in a path p, if the path p has the form $p = \ldots e_1 \ldots e \ldots e_2$. Lemma 5.10 shows that for a strictly contributing edge e in category (ii) and its reverse edge e', one of them must be in between a pair of matching edges in some InterDyck-paths. In our algorithm that identifies the strictly contributing edges in category (ii), we first identify all the pairs of matching edges. Then we collect all the edges between them as a superset of strictly contributing edges. Algorithm 3 describes the approach to identify strictly contributing edges. In Algorithm 3, step 1 builds the L_x-graph for the original input graph. Step 1 is the prerequisite for step 2 to find all matching edge pairs in the original graph. In step 2, the anchor-node-identification algorithm from Section 4.3 finds all pairs of edges with the form $u \xrightarrow{t} v$ and $w \xrightarrow{t} v$. Due to the bidirectedness, the algorithm finds all the matching edges $u \xrightarrow{t} v$ and $v \xrightarrow{t'} w$ in the L_x-graph. Step 2 collects the corresponding nodes in the original graph as a preparation for step 3 to compute

ALGORITHM 3: Strictly Contributing Edge Identification for category (ii)

Step 1: Build the L_x-graphs as described in Section 5.1.1 for each L_x language;
Step 2: When performing anchor-node identification on L_x-graphs, if there is a node collapsing introduced by edges $u \xrightarrow{t} v$ and $w \xrightarrow{t} v$, find the corresponding nodes of v in the original graph G. The corresponding nodes for v in the original graph are denoted as rep_node(v);
Step 3: For every edge $e = (v_1, v_2)$, if $v_1, v_2 \in$ rep_node(v) , add e into the set of contributing edges.

all the edges between them. Step 3 computes all the edges connecting the nodes from step 2, so they are the edges that are between the matching edges in the InterDyck-paths.

Theorem 5.11 (Correctness of Algorithm 3). *Algorithm 3 identifies a set of edges C. If an edge e is in category (ii) and strictly contributing, then $e \in C$.*

Proof. According to Lemma 5.10, if a non-trivial InterDyck-path contains a strictly contributing edge e in category (ii), either e or its reverse edge e' is in between two matching edges e_1 and e_2. At the same time, the labels of e_1, e_2 are not in the same Dyck language as the label of e. Without loss of generality, assume that e is the edge between e_1 and e_2, and the $L_1 \odot \ldots \odot L_k$-path p has the form $p = \ldots e_1 \ldots e \ldots e_2 \ldots$. Let the three edges be $e = u \xrightarrow{l_e} v$, $e_1 = a_1 \xrightarrow{l_1} b_1$, and $e_2 = a_2 \xrightarrow{l_2} b_2$ with $l_e \in L_x$ and $l_1, l_2 \in L_j (i \neq j)$. There is no unmatched L_j-edge between e_1, e_2 in the path p, otherwise, e_1 cannot match with e_2. Thus, the nodes u, v, b_1, a_2 in the original graph will be collapsed into one node, denoted by v_{u,v,b_1,a_2}, in the L_j-graph. In step 1, Algorithm 3 constructs all L_x-graphs. In step 2, in the L_j-graph, e_1 and e_2 will induce a node collapsing , and the algorithm will collect a set of corresponding nodes for node v_{u,v,b_1,a_2} in the original graph, which includes nodes u, v, b_1, a_2. Thus, Algorithm 3 identifies the strictly contributing edge e connecting between these corresponding nodes in the original graph and in step 3. The algorithm safely includes e in the resulting set of edges. We show that Algorithm 3 identifies a superset of strictly contributing category (ii) edges. ☐

We can bound the complexity of Algorithm 3 as follows. Because to check whether rep_node(v_1) = rep_node(v_2) for an edge $e = (v_1, v_2)$ requires only constant time; thus the algorithm for identifying strictly contributing category (ii) edges still takes within $O(m)$ time which does not increase the overall complexity for the simplification algorithm.

Example 5.12. We illustrate how Algorithm 3 identifies the strictly contributing edge $v_b \xrightarrow{(\!|_1} v_c$. Note that the edge $v_b \xrightarrow{(\!|_1} v_c$ is a contributing edge in category (ii), and the edge cannot be removed from the original graph, because it is involved in the only InterDyck-path that connects nodes v_a and v_g. Recall that we denote the Dyck language for parentheses by L_p, and the Dyck language for brackets by L_b. Step 1 builds the L_b-graph for the original input graph. It facilitates finding the pairs of matching edges in the original graph G. Figure 7(b) shows the result of the L_b-graph construction. In step 2, the node collapsing of v_a and $v_{d,e,f}$ via edges $v_a \xrightarrow{[\![_f} v_{b,c}$ and $v_{d,e,f} \xrightarrow{]\!]_f} v_{b,c}$ indicates that in the original graph $v_a \xrightarrow{[\![_f} v_b$ and $v_d \xrightarrow{]\!]_f} v_c$ are matching edges. Step 3 collects all the edges between pairs of matching edges, and thus the algorithm identifies the edge $v_b \xrightarrow{(\!|_1} v_c$ (which is in between $v_a \xrightarrow{[\![_f} v_b$ and $v_d \xrightarrow{]\!]_f} v_c$) as a strictly contributing edge in category (ii). Note that the original anchor-node-identification algorithm in Section 5.1.2 cannot identify the $v_b \xrightarrow{(\!|_1} v_c$ edge as a contributing edge, because there is no other $(\!|_1$-edge in the L_p-graph.

6 EVALUATION

We implemented the graph-simplification algorithm, and—using three different InterDyck-reachability solvers—applied it to the problem of a context- and field-sensitive taint analysis for Android applications [9]. The experiments were performed on a 16 GB memory machine with an Intel Xeon 2.10 GHz CPU, running Ubuntu 18.04.

We compared three InterDyck-reachability algorithms on both the original and simplified graphs. Our evaluation focused on addressing the following three research questions:

- ***RQ1:*** How does graph size influence the size reduction and the efficiency of the simplification algorithm?
- ***RQ2:*** How much can graph simplification improve the performance and precision of various InterDyck-reachability algorithms?
- ***RQ3:*** Do different InterDyck-reachability algorithms benefit similarly from graph simplification? For which of the InterDyck-reachability algorithms is the performance improved the most by graph simplification?

6.1 Experimental Setup

The Client Analysis. The experiment was conducted with a context- and field-sensitive taint analysis for Android applications [9], applied to 95 Google App-store applications. Context-sensitivity is captured by a Dyck language L_p, where each open parenthesis $(\!|_i$ represents a method call, and a matching close parenthesis $|\!)_i$ represents a corresponding return. The analysis uses another Dyck language L_b to encode field sensitivity, where an open bracket $[\![_f$ represents an assignment to field f and a close bracket $]\!]_f$ represents an access on field f. Therefore, the analysis is based on InterDyck-reachability where InterDyck $= L_b \odot L_p$.

We performed taint analysis on both the original and simplified graphs. The set of subject Android applications includes the top 30 free apps, as well as some popular apps in the Editor's Choice list as of January 2015. We extracted the taint-analysis graphs using the tools from the work of Huang et al. [9]. Note that the original taint analysis [9] is demand-driven, while ours is exhaustive. The tool successfully generates graphs from 95 out of the 150 Google store apps provided in the implementation.[3] For each benchmark, we apply graph simplification iteratively until the simplification procedure cannot remove any additional non-contributing edges.

The 95 obtained taint-analysis graphs have various sizes, ranging from a few hundred nodes to more than 100,000 nodes. On average, each graph consists of 40,129 nodes and 147,009 edges. These taint-analysis graphs also contain more call/return edges than field read/write edges. On average, each taint-analysis graph has 21,559 different calls/returns and 2,250 different field accesses.

InterDyck-Reachability Algorithms. We used the following three InterDyck-reachability algorithms as the graph-reachability engine for variants of the taint analysis:

- *CFL-reachability algorithm* [16]. This method is the traditional over-approximation for InterDyck-reachability. To approximate $L_b \odot L_p$, we used a regular language R_p, presented in Figure 8, to over-approximate L_p. The language $L_b \odot R_p$ is still context-free, so one can apply the CFL-reachability algorithm to solve the $(L_b \odot R_p)$-reachability problem.
- *SPDS-reachability algorithm* [21]. In our client analysis, the SPDS separates the analysis into a context-insensitive, field-sensitive analysis and a context-sensitive, field-insensitive analysis. Each problem can be effectively formulated as a CFL-reachability problem. The SPDS algorithm solves them independently and intersects the results.

[3]Both the implementation and the subject apps are publicly available at https://github.com/proganalysis/type-inference.

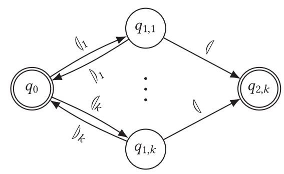
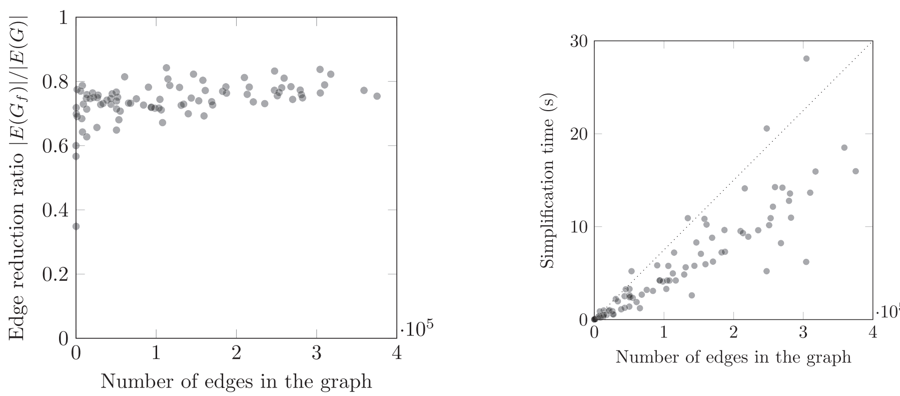


Fig. 8. A finite-state automaton that approximates a Dyck language L_i. We use (|-label to represent an arbitrary open parenthesis label.



(a) Plot of the graph-reduction ratio. In general, the reduction ratio is lower than 0.8. A lower ratio indicates there are fewer edges remaining in the simplified graph.

(b) The relation between graph size and graph-simplification time. The data shows that, in practice, the running time of the graph-simplification algorithm is linear in the size of the graph.

Fig. 9. Amount of graph reduction, and running time of the graph-simplification algorithm as a function of graph size.

— *LCL-reachability algorithm* [28]. The LCLs properly contain the InterDyck languages. Unlike CFL- and SPDS-reachability, LCL-reachability precisely models InterDyck-reachability. The LCL-reachability algorithm, in contrast, is an *over-approximating* algorithm, which means that it may return a superset of the exact result, i.e., there may be pairs of nodes that are connected by an accepting-state summary edge that is not InterDyck-reachable.

We implemented all algorithms in C++. All experiments were repeated three times, and we report the average of the three trials to improve the reliability of the collected results.

6.2 RQ1: Graph-Simplification Efficiency

Our graph-simplification algorithm reduces an original graph G to G_f. We define the graph-reduction ratio as $r = \frac{|E(G_f)|}{|E(G)|}$. Figure 9(a) presents the simplification results *w.r.t.* ratio r. On average, $r = 0.743$, indicating that the other 25.7% edges have been removed from the original graph G by applying the graph-simplification technique.

As graph size increases, Figure 9(a) indicates that there is a very slight trend for ratio r to increase. However, for most graphs, the reduction ratio is below 0.8, even for large graphs with around 400K edges. Thus, simplification can consistently remove a significant number of edges.

In terms of the running time, graph simplification is much faster than the InterDyck-reachability algorithms in most cases. The only exception is when the graph size is very small

Table 2. Timeout Statistics in the Experiments

	Finished on G, G_f	Finished only on G_f	Timed out on G, G_f
LCL	41	13	41
SPDS	9	5	81
CFL	5	2	88

(i.e., when the number of edges is less than 150), the simplification procedure can take time comparable to the InterDyck-reachability algorithms. Figure 9(b) gives the relationship between graph size and running time of the graph-simplification algorithm. It demonstrates that the asymptotic running time of the algorithm is close to linear in the size of the (original) graph.

Summary. On average, after the graph-simplification algorithm, there are 74.3% of the edges remaining in the simplified graphs. The algorithm can consistently remove more than 20% of the edges in large graphs. The running time of the simplification algorithm is almost linear in practice. The linear-time performance allows it to serve as a pre-processing step for an InterDyck-reachability algorithm.

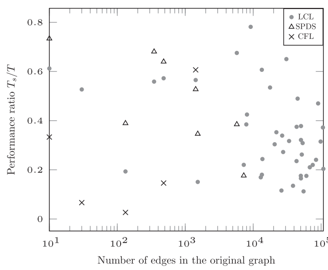
6.3 RQ2: Performance and Precision Improvement of InterDyck-Reachability Algorithms

Performance. We set a time budget of 300 seconds for all InterDyck-reachability algorithms. Table 2 presents the timeout information of different algorithms. Typically, the CFL-reachability algorithm runs out of time for graphs with more than 1 K edges. The SPDS-reachability algorithm runs out of time for graphs with more than 5 K edges. The LCL-reachability algorithm usually finishes processing graphs with fewer than 70 K edges within the time budget.

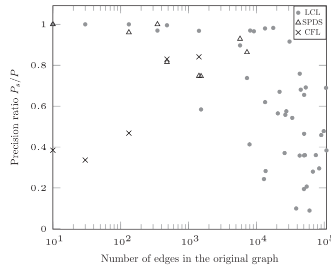
We use the following metric to the measure performance improvement: given the running time T on the original graph G, and the running time T_s on the simplified graph, the performance ratio is defined as $\frac{T_s}{T}$. If an InterDyck-reachability algorithm finishes on both the original and the simplified graphs, we collect the performance ratio; the data is plotted in Figure 10(a). The plot shows that for the majority of graphs, graph simplification reduces the running time of InterDyck-reachability algorithms to less than 40% of the original running time. On all graphs, the running time is reduced by more than 20%.

In practice, it is also necessary to take graph-simplification time into account. We define T' as the time needed for both graph simplification and running the InterDyck-reachability algorithm on the simplified graphs. Figure 11(a) presents the LCL-reachability result. From the plot, we see that, as the running time of the LCL-reachability algorithm increases, the cost of graph simplification becomes less and less significant. If the LCL-reachability algorithm completes on the original graph within 7 seconds, it is not worth performing graph simplification. However, for larger graphs, the time graph simplification is recouped. The observation is consistent for both SPDS- and CFL-reachability algorithm *w.r.t.* Figures 11(b) and (c). Overall, running graph simplification and performing an InterDyck-reachability algorithm on the simplified graph is 2.18× faster than running the same algorithm on the original graph.

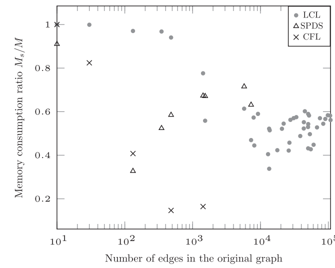
Precision. We define the precision ratio as $\frac{y}{x}$ where x and y denote the number of InterDyck-reachable pairs obtained from running the InterDyck-reachability algorithm on the original and simplified graphs, respectively. Figure 10(b) gives information about the precision improvements. Theoretically, performing graph simplification does not affect the InterDyck-reachability results. In practice, when the set of non-contributing edges in the graph is smaller, the various



(a) Performance improvements. The running time of all benchmark programs is reduced to less than 80% after the graph simplification.

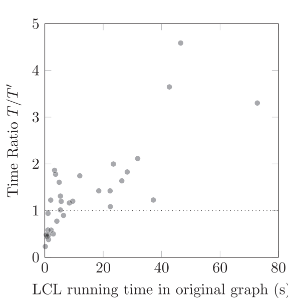


(b) Precision improvements. For about two-thirds of the benchmark programs, the InterDyck algorithms find less than 80% of the reachable pairs in simplified graphs: the discarded pairs are false positives.

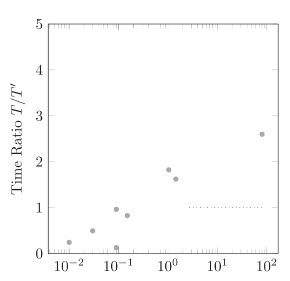


(c) Memory-usage improvements. For most of the graphs, the simplification process reduces the memory consumption of the InterDyck-reachability algorithms by half.

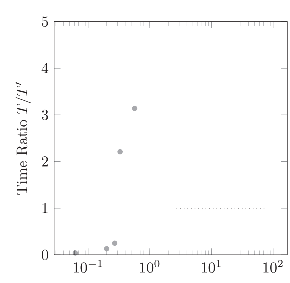
Fig. 10. The effect of the graph simplification on InterDyck-reachability algorithms. Running on a simplified graph helps improve the performance, precision, and memory usage of the algorithm. The x-axis represents the number of edges in the original graphs, and the y-axis indicates the improvement ratio. Each plot shows the improvement of one benchmark program. A lower y-axis value indicates a better degree of improvement from graph simplification.



(a) Improvement on LCL-reachability.



(b) Improvement on SPDS-reachability.



(c) Improvement on CFL-reachability.

Fig. 11. Performance comparison. In these comparisons, we compare two approaches to solve the InterDyck-reachability problem: directly running an InterDyck algorithm on the original graph (with time T), versus performing graph simplification and then running the same InterDyck algorithm on the simplified graph (with time T'). The value on the y-axis indicates the speedup due to graph simplification. $y = 1$ means that the time needed to run InterDyck algorithm on the original graph is the same as first performing graph simplification and then running the algorithm on the simplified graph.

over-approximation algorithms are likely to obtain more precise answers. It is interesting to note that the observed precision improvement is quite significant: on average, graph simplification helps InterDyck-reachability to generate a solution that has only 64.92% of the pairs that are in the solution computed using the original graph. (The discarded pairs are false positives.) For the LCL-reachability algorithm, there are three graphs where graph simplification helps to detect more than 80% of pairs as false positives in the original solution. Moreover, there is a trend that with an increasing number of edges in the graph, the precision improvement from graph simplification is likely to be more significant.

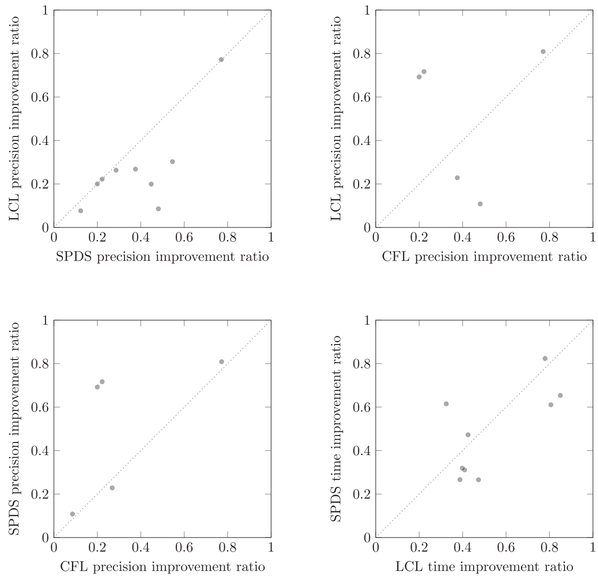


Fig. 12. Comparison of graph-simplification improvement among the algorithms. The x value for a data point is the improvement for the first InterDyck algorithm, the y value is the improvement for the second InterDyck algorithm. A data point in the bottom-right half-space indicates that the algorithm on the x-axis has the greater improvement; a data point in the top-left half-space indicates the algorithm on the y-axis has the greater improvement.

Memory Consumption. As mentioned in Section 6.2, the average number of edges in the simplified graph is 74.3% of the original graphs. However, Figure 10(c) shows that, in most cases, the InterDyck-reachability algorithms consume around half the memory when running on the simplified graph. On average, running InterDyck-reachability algorithms on the simplified graphs consumes only 57.37% of the original memory.

6.4 RQ3: Graph-Simplification Improvements for Different InterDyck-Reachability Algorithms

From a practical perspective, the LCL-reachability algorithm benefits the most from graph simplification. Table 2 shows that, with the graph simplification technique, the LCL-reachability algorithm can handle 13 more graphs within the same time budget, which is significantly more than the other two InterDyck algorithms.

Figure 12 compares the precision improvements based on the graphs that all three algorithms can process. Specifically, the top-left plot of Figure 12 shows that most of the data points occur in the bottom-right half-space, which indicates that for the same graph, SPDS-reachability achieves better precision improvements compared to LCL-reachability. The top-right plot shows that the precision improvements for both LCL- and CFL-reachability are comparable. The bottom-left plot shows that the precision improvement of SPDS-reachability is more significant compared to CFL-reachability. To sum up, the graph simplification technique benefits the SPDS-reachability algorithm the most in terms of precision improvement.

In terms of the performance benefits, we only compare the LCL-reachability algorithm against the SPDS-reachability algorithm. The CFL-reachability algorithm can only terminate successfully on very small graphs. In Figure 12, the bottom-right plot shows that the performance improvement (in terms of running time) is similar for the two InterDyck algorithms.

6.5 Discussion

Our graph-simplification algorithm is based on the bidirected Opt-Dyck algorithm on the relaxed graph G'. A natural extension is to utilize a general Dyck-reachability algorithm on G to identify the InterDyck-contributing edges in G. However, the Dyck-relation on general digraphs is not an equivalence relation. We have to resort to a more expensive (sub)cubic-time Dyck-reachability algorithm to identify Dyck-contributing edges in G. In Table 2, we have seen that the SPDS-reachability algorithm basedon Dyck-reachability does not scale well in practice.

For simplification results, besides node and edge numbers, the proposed graph-simplification algorithm also decreases the number of different calls/returns and the number of different fields significantly. In the simplified graph, the total numbers of different calls/returns and fields are usually below one fifth of the number in the original graph. The number of different calls/returns and fields establishes the size of the alphabets used in the Dyck languages L_b and L_p. The time complexity of InterDyck algorithms also depends on the number of different kinds of parentheses and brackets. Thus, the decrease in these numbers also contributes to the smaller time and space consumption of the different InterDyck algorithms, and contributes to the performance gain from graph simplification as well.

In the work of Späth et al. [21], it has been observed that over-approximation for InterDyck reachability almost never happens in practice. Their conclusion is supported by the empirical study of a typestate analysis for relatively small graphs. In our experiments, we observed significant over-approximation of taint analysis. In general, the degree of over-approximation depends on what kind of information the client analysis is computing.

We implemented the LCL- and CFL-reachability algorithms given in the original references. The original SPDS article presents a *demand-driven* reachability algorithm, which also accepts the prefix of the InterDyck languages, i.e., the algorithm accepts words with unmatched open parentheses/brackets, such as "$\llparenthesis_1 \llbracket_1 \rrparenthesis_1$". Our SPDS implementation is restricted to only the InterDyck language and always computes the *all-pairs* InterDyck-reachability results.

7 RELATED WORK

Many program-analysis problems can be formulated as an InterDyck-reachability problem [3, 20, 22–24, 26]. However, solving InterDyck-reachability is undecidable [17]. Existing approaches use different techniques to over-approximate the exact solution for InterDyck-reachability problems. Traditional approaches include over-approximating some Dyck languages in InterDyck using regular languages [7, 8]. Access-path-based analysis approximates field-sensitivity by restricting the access-paths with a bounded length, and thus also over-approximates InterDyck-reachability [4, 12]. The recent work by Späth et al. [21] over-approximates InterDyck-reachability using synchronized pushdown systems. Zhang and Su [28] propose linear-conjunctive-language reachability to precisely formulate InterDyck-reachability, and provide an over-approximating algorithm for solving the LCL-reachability problem.

The proposed graph-simplification algorithm is based on the Fast-Dyck algorithm proposed by Zhang et al. [27]. Chatterjee et al. [1] give an $O(m + n \cdot \alpha(n))$ worst-time algorithm for solving bidirected Dyck-reachability, which improves the $O(m \log m)$ expected running time by Zhang et al. [27]. In practice, many techniques have been proposed to improve CFL-reachability-based analyses [2, 25, 29]. Our work focuses on simplifying the input graphs for InterDyck-reachability and is applicable to any existing sound InterDyck-reachability-based analysis. Graph simplification techniques are also studied in other program-analysis applications. In pointer analysis, various techniques [5, 6, 19] are applied to reduce the size of the constraint graphs for inclusion-based analysis. For example, the work by Hardekopf and Lin [6] focus on deriving pointer-equivalence and location-equivalence relationships between variables. They simplify the graphs by collapsing the equivalent nodes. Our graph simplification focuses on eliminating irrelevant edges.

8 CONCLUSION

This article has proposed a new graph-simplification algorithm for InterDyck-reachability. Our key insight is to reduce the graph by eliminating graph edges that do not contribute to any InterDyck-paths. We have applied the simplification algorithm to context- and field-sensitive taint analysis for Android. The experimental results demonstrate that graph simplification can significantly speed up existing InterDyck-reachability algorithms. Moreover, graph simplification improves both the precision and the memoryusage of the client analysis.

ACKNOWLEDGMENTS

We thank the PLDI 2020 reviewers for their feedback on Li et al. [13] and the TOPLAS referees for valuable feedback on earlier drafts of this article. Specifically, we would like to thank the TOPLAS referee who suggested an idea that improved the complexity of Algorithm 2 from $O(m \log m)$ time (described in the original PLDI 2020 paper [13]) to $O(m + n \cdot \alpha(n))$ time.

REFERENCES

[1] Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. 2018. Optimal Dyck reachability for data-dependence and alias analysis. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 30:1–30:30.

[2] Swarat Chaudhuri. 2008. Subcubic algorithms for recursive state machines. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 159–169.

[3] Ben-Chung Cheng and Wen-mei W. Hwu. 2000. Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 57–69.

[4] Arnab De and Deepak D'Souza. 2012. Scalable flow-sensitive pointer analysis for Java with strong updates. In *Proceedings of the European Conference on Object-Oriented Programming*. 665–687.

[5] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. 1998. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 85–96.

[6] Ben Hardekopf and Calvin Lin. 2007. Exploiting pointer and location equivalence to optimize pointer analysis. In *International Static Analysis Symposium* H. R. Nielsona and G. Filé (Eds.). Lecture Notes in Computer Science, vol 4634. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-74061-2_17

[7] M. A. Harrison. 1978. *Introduction to Formal Language Theory*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

[8] John E. Hopcroft and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley.

[9] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. 2015. Scalable and precise taint analysis for Android. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*. 106–117.

[10] Vineet Kahlon. 2009. Boundedness vs. unboundedness of lock chains: Characterizing decidability of pairwise CFL-reachability for threads communicating via locks. In *Proceedings of the ACM/IEEE Symposium on Logic in Computer Science*. 27–36.

[11] John Kodumal and Alexander Aiken. 2004. The set constraint/CFL reachability connection in practice. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 207–218.

[12] Johannes Lerch, Johannes Späth, Eric Bodden, and Mira Mezini. 2015. Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths. In *Proceedings of the International Conference on Automated Software Engineering*. 619–629.

[13] Yuanbo Li, Qirun Zhang, and Thomas W. Reps. 2020. Fast graph simplification for interleaved Dyck-reachability. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020*. ACM, 780–793.

[14] Yuanbo Li, Qirun Zhang, and Thomas W. Reps. 2021. On the complexity of bidirected interleaved Dyck-reachability. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–28.

[15] G. Ramalingam. 2000. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems* 22, 2 (2000), 416–430.

[16] Thomas W. Reps. 1998. Program analysis via graph reachability. *Information & Software Technology* 40, 11–12 (1998), 701–726.

[17] Thomas W. Reps. 2000. Undecidability of context-sensitive data-dependence analysis. *ACM Transactions on Programming Languages and Systems* 22, 1 (2000), 162–186.

[18] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 49–61.

[19] Atanas Rountev and Satish Chandra. 2000. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 47–56.

[20] Micha Sharir and Amir Pnueli. 1981. Two approaches to interprocedural data flow analysis. In *Proceedings of the Program Flow Analysis: Theory and Applications*, Steven S. Muchnick and Neil D. Jones (Eds.). Prentice-Hall, 189–234.

[21] Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, flow-, and field-sensitive data-flow analysis using synchronized Pushdown systems. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 48:1–48:29.

[22] Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 387–400.

[23] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 59–76.

[24] Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. 2015. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 83–95.

[25] Guoqing Xu, Atanas Rountev, and Manu Sridharan. 2009. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *Proceedings of the European Conference on Object-Oriented Programming*. 98–122.

[26] Dacong Yan, Guoqing (Harry) Xu, and Atanas Rountev. 2011. Demand-driven context-sensitive alias analysis for Java. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*. 155–165.

[27] Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. 2013. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 435–446.

[28] Qirun Zhang and Zhendong Su. 2017. Context-sensitive data-dependence analysis via linear conjunctive language reachability. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 344–358.

[29] Qirun Zhang, Xiao Xiao, Charles Zhang, Hao Yuan, and Zhendong Su. 2014. Efficient subcubic alias analysis for C. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 829–845.

[30] Xin Zheng and Radu Rugina. 2008. Demand-driven alias analysis for C. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 197–208.

Received April 2021; revised September 2021; accepted October 2021