Fast Graph Simplification for Interleaved Dyck-Reachability

Yuanbo Li
Georgia Institute of Technology
USA
yuanboli@gatech.edu

Qirun Zhang
Georgia Institute of Technology
USA
qrzhang@gatech.edu

Thomas Reps
University of Wisconsin-Madison
USA
reps@cs.wisc.edu

Abstract

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 dependences, 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 paper offers a new perspective on improving both the precision and the scalability of InterDyck-reachability: we aim to simplify the underlying input graph G. Our key insight is based on the observation that if an edge is not contributing to any InterDyck-path, 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 approaches 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 on 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.

PLDI '20, June 15–20, 2020, London, UK

ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3386021

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

Keywords: CFL-Reachability, Static Analysis

ACM Reference Format:
Yuanbo Li, Qirun Zhang, and Thomas 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 '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3385412.3386021

1 Introduction

The L language-reachability (L-reachability) framework is a popular model to formulate many program-analysis problems [14]. 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, 25]. A Dyck language essentially generates well-balanced parentheses, which can be used to capture well-paired program properties, such as function calls/returns [15, 16, 22], pointer references/dereferences [27, 28], locks/unlocks [10, 13], and field reads/writes [9, 24, 25].

A natural generalization of Dyck-reachability is Interleaved Dyck-reachability (InterDyck-reachability) [9, 15, 26]. The Interleaved Dyck language denotes the intersection of multiple Dyck languages based on an interleaving operator $\odot$. For instance, let D_1 and D_2 be two Dyck languages that generate matched parentheses and matched brackets, respectively. The string "$(\![\ [\![\ (\![\ ]\!]\ )\!]\ ]\!]$" belongs to the language InterDyck $= D_1 \odot D_2$, 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, 19, 26].

Unfortunately, solving InterDyck-reachability is computationally hard because the InterDyck-reachability problem

is, in general, undecidable [15]. 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, 21, 24]. For example, the recent work by Späth *et al.* proposed synchronized pushdown systems to compute a sound solution for InterDyck-reachability [19]. The work by Zhang and Su proposed linear-conjunctive-language reachability (LCL-reachability) to precisely describe the InterDyck-reachability formulation [26]. 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 paper, 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 $L_1 \odot L_2 \ldots \odot L_k$, where for each $i \in [1, k]$, L_i is a Dyck language. One enabling insight is to decompose the InterDyck-reachability problem in the input graph G into k Dyck-reachability problems in a graph G', which is a *relaxation* of G that makes G' *bidirected*. It turns out that 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* in G', as well as in G. The problem then becomes one of identifying non-contributing edges in G'.

One natural question to ask is why we identify the non-contributing edges in G' and not in G itself. The reason is that Dyck-reachability in 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 $O(m \log m)$ time with $O(m)$ space. This graph-simplification algorithm is asymptotically faster than the fastest $O(mn)$-time InterDyck-reachability algorithm [26]. The technique is 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 context-free language reachability (CFL-reachability) [14], synchronized pushdown systems reachability (SPDS) [19], and linear conjunctive language reachability (LCL-reachability) [26]. The empirical results are encouraging: graph simplification significantly improves both the performance and the precision of the client analyses.

- We found that, on average, it is 2.18× 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 $O(m \log m)$ time with $O(m)$ space. 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.

The remainder of the paper 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 1b gives the graph G that encodes the taint-analysis problem for the program in Figure 1a 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.*, $(_i$, $)_i$) represent calls and returns. For a path in G to represent the flow of a tainted value, both brackets and parenthesis must be properly matched. Let D_b and D_p be two Dyck languages. Due to the work of Huang *et al.* [9], the taint analysis can be formulated as an InterDyck-reachability problem over G, where InterDyck $= D_b \odot D_p$.

InterDyck-reachability algorithms. The problem of precise InterDyck-reachability is undecidable [15]. We briefly mention three popular over-approximation algorithms for the InterDyck-reachability problem.

- *CFL-reachability algorithm* [14]. 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 D_b. The reachability problem could be solved by a CFL-reachability algorithm based on the $CFL = R_b \cap D_p$.
- *SPDS-reachability algorithm* [19]. The synchronized pushdown systems (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* [26]. The InterDyck languages belong to the class of linear conjunctive languages (LCLs). Therefore, an LCL-reachability formulation can precisely encode a InterDyck-reachability problem. However, the LCL-reachability algorithm only computes an over-approximating solution.

Graph simplification. Recall that our key idea for graph simplification is to eliminate "useless" 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 "useless" 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.

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	$\{(v_a, v_c)\}$	$\{(v_a, v_c)\}$	$\{(v_a, v_c)\}$

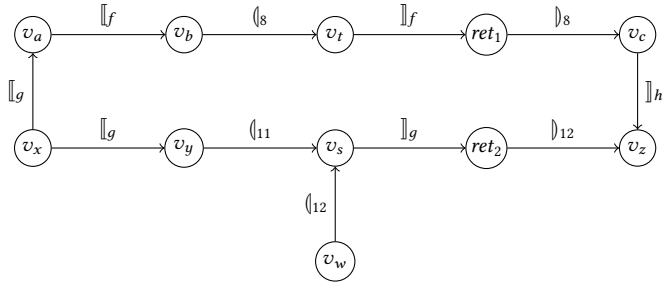
Benefits of graph simplification. The graph simplification is iterative. Figures 2a and 2b give two intermediate steps based on the first and second applications, respectively, of our graph simplification algorithm (Algorithm 2 in Section 5.2). Figure 2c shows the final graph G_f. Compared with the original graph in Figure 1b, 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. We can see that graph simplification improves the precision of the results from various InterDyck-reachability algorithms.

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

- *CFL-reachability algorithm.* The CFL-reachability algorithm in Figure 1b 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 2c), the CFL-reachability algorithm gives an exact solution.
- *SPDS-reachability algorithm.* In Figure 1b, 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 2c), 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.

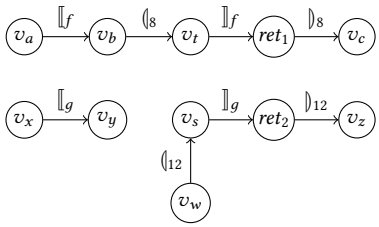
```
1  class T { T f; T g; T h; }
2  T getF(T v_t){ return v_t.f; }
3  T getG(T v_s){ return v_s.g; }
4
5  T v_a, v_b, v_c, v_y, v_z, v_w; ...
6  v_a.g = v_x;
7  v_b.f = v_a;
8  v_c = getF(v_b);
9  v_z = v_c.h
10 v_y.g = v_x;
11 getG(v_y)
12 v_z = getG(v_w)
```

(a) Example Java code.

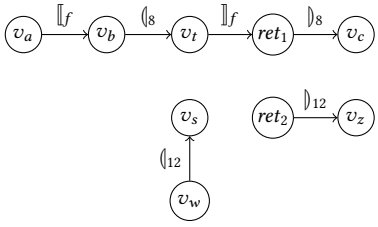


(b) InterDyck-reachability graph for taint analysis. Bracket edges (*i.e.*, $\llbracket_f$ and $\rrbracket_f$) represent field reads and writes *w.r.t.* a field name f. Parenthesis edges (*i.e.*, $(_l$ and $)_l$) represent calls and returns *w.r.t.* the line number l of the call-sites.

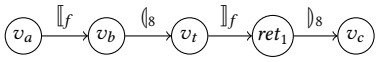
Figure 1. Motivating taint-analysis example.



(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 $v_y \xrightarrow{(_{11}} v_s$ 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) Final graph G_f. This iteration eliminates the call edge $v_w \xrightarrow{(_{12}} v_s$ and the return edge $ret_2 \xrightarrow{)_{12}} v_z$. There are no more edges to eliminate from G_f.

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

3 Preliminaries

This section introduces definitions used in the paper. 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-balanced-parenthesis 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:

$$D_k \rightarrow D_k\ D_k \mid (_1\, D_k\,)_1 \mid \ldots \mid (_k\, D_k\,)_k \mid \varepsilon. \qquad (1)$$

Given a context-free grammar $CFG = (\Sigma, N, P, S)$ 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 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 in G is an *S-path* if the realized string can be derived from the start symbol S in CFG. Node v is *S-reachable* from node u iff there exists an S-path from u to v in G. Because S denotes the start symbol of language L, we also say that node v is L-reachable from u. 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 paper focuses on the reachability problem related to the interleaved Dyck language (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 "$\llbracket\ \rrbracket$" $\in D_b$ and "$(\ )$" $\in D_p$. All of "$\llbracket\ (\ \rrbracket\ )$", "$(\ \llbracket\ )\ \rrbracket$", and "$\llbracket\ \rrbracket\ (\ )$" belong to the InterDyck language based on D_b and D_p.

We formally define the class of InterDyck languages based on an *interleaving operation* $\odot$. Formally, $\odot : \Sigma^* \times \Sigma^* \rightarrow \mathcal{P}(\Sigma^*)$ 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 $L_1, L_2, \ldots, L_k$ are k Dyck languages with disjoint alphabets, we define InterDyck $:= L_1 \odot L_2 \odot \ldots \odot L_k$. The InterDyck-reachability problem is an L-reachability problem by restricting L to InterDyck. 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 1b), the contributing edges are $v_a \xrightarrow{\mathbb{I}_f} v_b$, $v_b \xrightarrow{(_8} v_t$, $v_t \xrightarrow{\mathbb{I}_f} ret_1$, and $ret_1 \xrightarrow{)_8} v_c$ that appear in the simplified graph G_f in Figure 2c.

In this paper, 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 the 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 [15].

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 $Reach\langle$InterDyck$, G\rangle$, we first introduce a new Dyck language D_p with an alphabet $\Sigma_{D_p} = \{(,)\}$ and $\Sigma_{D_p} \cap \Sigma_{\text{InterDyck}} = \emptyset$. Define InterDyck$' =$ InterDyck $\odot D_p$. 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 in polynomial time. □

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 [15].

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 inverse 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 $O(m \log m)$ time on a bidirected L_x-graph with m edges and n nodes [25]. 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.

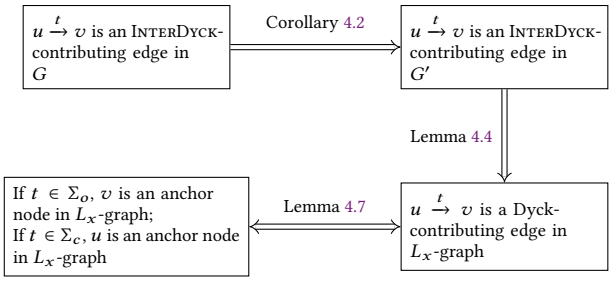


Figure 3. Summary of lemmas used in Section 4. Let L_x be a Dyck language in InterDyck. The alphabet $\Sigma_{L_x} = \Sigma_o \cup \Sigma_c$ of L_x can be partitioned into Σ_o and Σ_c representing open and close parentheses, respectively. Let $t \in \Sigma_{L_x}$.

4.1 Graph Relaxation: From G to G'

Given an edge-labeled input graph $G = (V, E)$, we construct a relaxed graph $G' = (V, E')$ by introducing additional inverse edges. In particular, the node set $V \in G$ remains unmodified. Let $(_i$ and $)_i$ be two matched open and close parentheses in InterDyck. The edge set $E' \in G'$ is constructed as follows:

- For each edge $u \xrightarrow{(_i} v \in E$, we insert both edges $u \xrightarrow{(_i} v$ and $v \xrightarrow{)_i} u$ into E';
- For each edge $u \xrightarrow{)_i} v \in E$, we insert both edges $u \xrightarrow{)_i} v$ and $v \xrightarrow{(_i} u$ into E'.

Each edge $e \in G$ is mapped to two *corresponding edges* in G', denoted as set $h(e)$. 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 $e = u \to v$ 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_i in InterDyck represents a Dyck language for all $i \in [1, n]$. Let L_x be a Dyck language and $\Sigma_{L_x} = \{(_1,)_1, \ldots, (_k,)_k\}$. 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 "$(_1[\![_2(_2]\!]_2)_2)_1$". The "extracted" substring is "$(_1(_2)_2)_1$", which is a valid Dyck string. In general, let $($ be a terminal in Σ_{L_x}. It is straightforward to see that if $($ is in a valid InterDyck string, $($ 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*⟨InterDyck, G'⟩. 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', we construct the L_x-graph by contracting all $u \xrightarrow{t} v$ edges 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 linear-logarithmic time [25].

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* [25]: (i) it is reflexive and transitive based on the Dyck grammar given in Eqn. (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 Zhang *et al.* [25], we summarize the algorithm for solving Dyck-reachability in L_x-graphs. The principal idea is to collapse the nodes in the Dyck relation. There are two major steps.

- *Edge merging*: Identifying two edges $u \xrightarrow{(_i} w$ and $v \xrightarrow{(_i} w$, and merging one edge with the other based on the degrees of u and v;
- *Node collapsing*: Collapsing two nodes u and v into a single representative node $n_{\{u,v\}}$ and updating all adjacency edges of node $n_{\{u,v\}}$ based on u and v.

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

Repeating node collapsing and edge merging introduces 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 Fast-Dyck().

Lemma 4.5 (Correctness of Fast-Dyck [25]). *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 Fast-Dyck() on the L_x-graph.*

To facilitate further discussion, let G_f denote the resulting graph after running Fast-Dyck() on G. We define rep_node[$\cdot$] 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 Fast-Dyck(). To identify Dyck-contributing edges in the L_x-graph, we leverage the *anchor node* w of the two edges $u \xrightarrow{(_k} w$ and $v \xrightarrow{(_k} w$ merged by Fast-Dyck(). Intuitively, every Dyck-path computed by Fast-Dyck() is associated with at least one such anchor node. Formally, we have

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

Lemma 4.7. *An edge $u \xrightarrow{(_k} v$ is a contributing edge for Dyck-reachability in a bidirected L_x-graph iff v is an anchor-k node in the L_x-graph. Similarly, an edge $u \xrightarrow{)_k} v$ is a contributing edge for Dyck-reachability in an L_x-graph iff u is an anchor-k node in the L_x-graph.*

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

Base case. The contributing edge $u \xrightarrow{(_k} v$ is involved in a Dyck-path of length 2. There must exist another node w such that $v \xrightarrow{)_k} w$. Because L_x-graph is bidirected, we have $w \xrightarrow{(_k} v \in E$. Therefore, v is an anchor-k 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 $e = u \xrightarrow{(_k} v$ 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 $v' \xrightarrow{)_k} w \in E$. By Lemma 4.5, we have rep_node[v] = rep_node[v']. Based on the bidirectedness, we have $u \xrightarrow{(_k} v, w \xrightarrow{(_k} v' \in E$. By the definition of anchor-k nodes, we conclude that v is an anchor-k 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-k node.

Now we prove the backward direction. Suppose that v is an anchor-k 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 $w \xrightarrow{(_k} v' \in E(L_x)$. Because rep_node[$v$] = rep_node[$v'$], *i.e.*, v and v' are merged by Fast-Dyck(), there exists a Dyck-path $p = v \to v'$. By utilizing the bidirectedness of the L_x-graph, we have $v' \xrightarrow{)_k} w \in E(L_x)$, as well. Then, $u \xrightarrow{(_k} v$, p, and $v' \xrightarrow{)_k} w \in E(L_x)$ form a new Dyck-path. Therefore, $u \xrightarrow{(_k} v$ 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-k nodes in the L_x-graph (Lemma 4.7). Putting everything together, we have

Theorem 4.8. *Let L_x be a Dyck language in InterDyck and $(_k,)_k \in \Sigma_{L_x}$. If either an edge $u \xrightarrow{(_k} v$ or an edge $v \xrightarrow{)_k} u$ is contributing to InterDyck-reachability in G, the node v is an anchor-k node in the L_x-graph.*

Corollary 4.9. *If a node v is not an anchor-k node in the L_x-graph, both edges $u \xrightarrow{(_k} v$ and $v \xrightarrow{)_k} u$ are non-contributing edges for InterDyck-reachability in G.*

Thus, the graph-simplification algorithm can remove from G all edges that meet the criterion given in Corollary 4.9.

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.

5.1 Key Steps

There are two key steps in the graph simplification: constructing the L_x-graphs and identifying anchor-k nodes.

5.1.1 L_x-Graph Construction Consider an interleaved Dyck language InterDyck = $L_1 \odot \ldots \odot L_n$. Given a relaxed

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 ← a disjoint-set of size |V|.
2  foreach u -l-> v ∈ E do
3      if l ∉ Σ_{L_x} then
4          E ← E \ {u -l-> v}
5          if rep_node[u] == rep_node[v] then continue;
6          rep_node.union(u, v)
7          if degree(rep_node[u]) > degree(rep_node[v]) then
8              w_b ← rep_node[u], w_s ← rep_node[v]
9          else
10             w_b ← rep_node[v], w_s ← rep_node[u]
11         V ← V \ {w_s}.
12         Let internal_edges be the edges between w_s and w_b
13         foreach e ∈ internal_edges do
14             E ← E \ {e}
15             l ← getlabel(e)
16             if l ∈ Σ_{L_x} then
17                 E ← E ∪ {w_b -l-> w_b}
18         foreach x ∈ In[w_s] with x ≠ w_b do
19             foreach edge e = x -l-> w_s do
20                 E ← E \ {e} ∪ {x -l-> w_b}
21         foreach x ∈ Out[w_s] with x ≠ w_b do
22             foreach edge e = w_s -l-> x do
23                 E ← E \ {e} ∪ {w_b -l-> x}
24 return G_x = (V, E)
```

graph G', to identify the anchor-k nodes, our algorithm needs to construct an L_i-graph for each $i \in \{1, \cdots, n\}$. An L_i-graph is essentially a contracted graph of G'. In particular, let t' be a letter in $\Sigma_{\text{InterDyck}} \setminus \Sigma_{L_i}$. The L_i-graph is constructed by contracting all $u \xrightarrow{t'} v$ edges in G'.

We describe the L_x-graph construction of the motivating example in Figure 4a. Recall that we have InterDyck = $D_b \odot D_p$. The procedure iterates through non-L_p edges. In Figure 4a, 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 contracted by collapsing nodes v_x and v_a. Nodes v_x and v_a form a representative node $\{v_x, v_a\}$ and the edge $v_x \xrightarrow{[\![_g} v_a$ is removed. We continue contracting non-L_p edges until there are no more non-L_p edges. Figure 4b depicts the final L_p-graph.

To facilitate node collapsing, we adopt the standard disjoint-set data structure. Procedure 1 gives the L_x-graph-construction algorithm. Line 1 initializes the disjoint-set rep_node. Given a node u, rep_node[u] returns the representative node of u in the graph. The union method always joins the smaller-degree vertex to the larger-degree vertex. Lines 2-3 show that the procedure iterates over all non-L_x edges in the graph. Lines 4-11 perform node collapsing. To collapse the edge $e = u \xrightarrow{l} v$, we merge nodes u, v to a representative set. Line 5 shows that if u, v have been merged together, we skip the rest of the loop body. Lines 6-11 merge the smaller-degree node between rep_node[u] and rep_node[v] into the larger one. Lines 12-17 collapse all the non-L_x edges between nodes rep_node[u] and rep_node[v], preserving only the L_x-edges. Lines 18-23 move all its adjacent edges into the other node for the smaller-degree node of rep_node[u] and rep_node[v].

5.1.2 Anchor-k Node Identification

The second step in graph simplification is anchor-k node identification. We modify the Fast-Dyck() algorithm by Zhang *et al.* [25] to collect the anchor-k node information. We denote the modified version as Fast-Dyck-Modified(). Recall that Fast-Dyck() tracks the number of incoming edges with the same edge label for each node in the graph. If there are two incoming edges $u \xrightarrow{k} v$ and $w \xrightarrow{k} v$ with the same edge label k, and k is an open-parenthesis, then the Fast-Dyck algorithm performs a *node-collapsing* between node u and w.

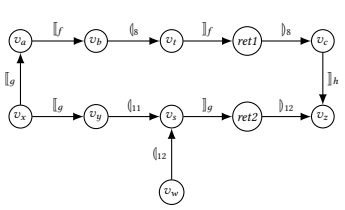
Fast-Dyck-Modified() leverages the node-collapsing process in Fast-Dyck() to mark anchor-k nodes. In particular, when Fast-Dyck() detects two incoming edges $u \xrightarrow{k} v$ and $w \xrightarrow{k} v$ with an open-parenthesis edge label k. Fast-Dyck-Modified() marks v as an anchor-k node according to the anchor-k node definition. Through the marking process, we collect all information to recover anchor-k nodes for the L_x-graph. For any node $v \in V(L_x)$, it is an anchor-k node iff rep_node(v) is marked as an anchor-k node by Fast-Dyck-Modified(). Finally, Fast-Dyck-Modified() returns the set of collected anchor-k nodes in the L_x graph.

Notice that the original Fast-Dyck algorithm runs in $O(m \log m)$ time. After the modification, the extra running time for each node-merging is $O(1)$, and thus the complexity of Fast-Dyck-Modified is still $O(m \log m)$

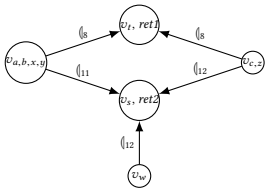
Example 5.1. We continue our example using the graph shown in Figure 4b. We apply Fast-Dyck-Modified() on this L_p-graph. Figure 4c gives the resulting graph. There exist two $(\!|_{12}$-edges pointing to node $\{v_s, ret_2\}$ and two $(\!|_8$-edges pointing to the node $\{v_t, ret_1\}$. Therefore, Fast-Dyck-Modified() collects the information that nodes v_t, ret_1 are anchor-8 nodes and v_s, ret_2 are anchor-12 nodes.

5.2 The Simplification Algorithm

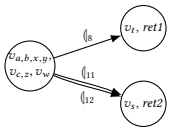
Algorithm 2 gives the graph-simplification algorithm. In lines 1-2, contrib_edges is initialized to an empty set. It contains the set of potential InterDyck-contributing edges in the original graph G when the algorithm terminates. We then obtain the relaxed graph G' defined in Section 4.1. The loop iterates over each Dyck language L_i in InterDyck = $L_1 \odot \cdots \odot L_n$ (lines 3-14). It first builds the L_i-graph based on Procedure 1. After we construct the L_i-graph, the algorithm invokes Fast-Dyck-Modified() described in Section 5.1.2 to collect anchor-k nodes in the L_i-graph. The variable anchor_nodes stores a set of anchor nodes of the form v_l, where v is the node in the L_i-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



(a) The graph to be simplified.



(b) Constructed L_p-graph.

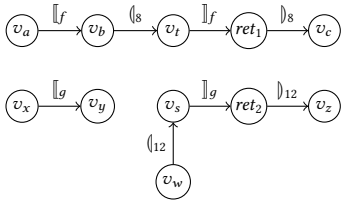


(c) Identifying anchor-k nodes based on Fast-Dyck-Modified.

Figure 4. Collection of anchor-k node information. Figure 4a repeats the graph of our motivating example from Figure 1b. Graph simplification involves repeated application of Algorithm 2. Figure 4b illustrates the L_p-graph construction result based on Section 5.1.1 during the first application of the algorithm. Figure 4c gives the corresponding graph after running Fast-Dyck-Modified described in Section 5.1.2. In Figures 4b and 4c, each $(_k$, $[\![$ edge has a corresponding reverse $)_k$, $]\!]$ edge. We omit reverse edges for brevity.

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.

Figure 5. Elimination of non-contributing edges. Figure 5a lists all anchor-k nodes identified in the first application of Algorithm 2. Figure 5b gives the simplified graph after the first application. It is the same as Figure 2a.

Algorithm 2: The graph simplification iteration.

Input : Edge-labeled directed graph $G = (V, E)$, an InterDyck language $L = L_1 \odot \cdots \odot L_n$;
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 ← Fast-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
```

L_i-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.*, Figure 1b and Figure 4a. After running the L_x-graph construction (Procedure 1) and Fast-Dyck-Modified for both the parenthesis language L_p and the bracket language L_b, Figure 5a gives the anchor-k nodes identified by the first application of Algorithm 2. It identifies the non-contributing edges based on Lemma 4.7. For instance, provided that v_s is an anchor-12 nodes, all the incoming $(_{12}$ edges to v_s and outgoing $)_{12}$ from v_s edges are contributing edges. By removing the non-contributing ones, we get the resulting graph Figure 5b. By applying Algorithm 2 two more times, we obtain the final graph, shown in Figure 2c.

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 4.8, 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 Fast-Dyck-Modified() collects all anchor-k nodes.

Lemma 5.3. *Fast-Dyck-Modified() collects all anchor-k nodes for each L_i-graph.*

Proof. Fast-Dyck-Modified() identifies two incoming edges incident on the same node with the same open-parenthesis label, performs a node collapsing, and generates an anchor-k node. For each anchor-k node generated, it only removes one edge in the graph. Thus, the previous two incoming edges incident to the same node become one. Suppose an anchor-k node has not been collected by Fast-Dyck-Modified(), there will be two incoming edges to the anchor-k node with the same open-parenthesis label. It contradicts the fact that Fast-Dyck-Modified() guarantees to find any pair of incoming edges with same open-parenthesis label [25]. □

Procedure 1 also correctly derives the L_x-graph. Specifically, lines 2 and 3 guarantee that we iterate over all edges with non-L_x labels. For each $u \xrightarrow{l} v$, line 4 contracts this

non-L_x edge based on Definition 4.3. Lines 6-11 merge the smaller-degree node to the larger-degree one. Lines 13-23 move edges incident to smaller-degree node to the other. Based on Section 5.1.1 the disjoint-set rep_node also joins the smaller-degree node to the larger-degree one. All information is updated correctly.

Theorem 5.4. *For an InterDyck language InterDyck = $L_1 \odot \cdots \odot L_n$ 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 line 3-14 contains two procedure calls: GetLxGraph and Fast-Dyck-Modified(). Given a graph with m edges, the time complexity of Fast-Dyck-Modified() is $O(m \log m)$ [25]. When the GetLxGraph procedure contracts an edge, it always moves all smaller-degree node's incident edges to the other. Therefore, for any single edge $u \to v$ during the edge moving, either the degree of u is doubled or the degree of v is doubled. Thus the total number of edge moving is at most $2 \log m$. The time complexity for GetLxGraph is $O(m \log m)$. The complexity of the loop body in lines 3-14 in Algorithm 2 is $O(m \log m)$. The overall algorithm iterates over all k Dyck languages in InterDyck. k is usually considered as a constant. Therefore, the time complexity of Algorithm 2 is $O(m \log m)$.

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 16GB memory machine with an Intel Xeon 2.10GHz 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 two 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?

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 D_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 D_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 = $D_b \odot D_p$.

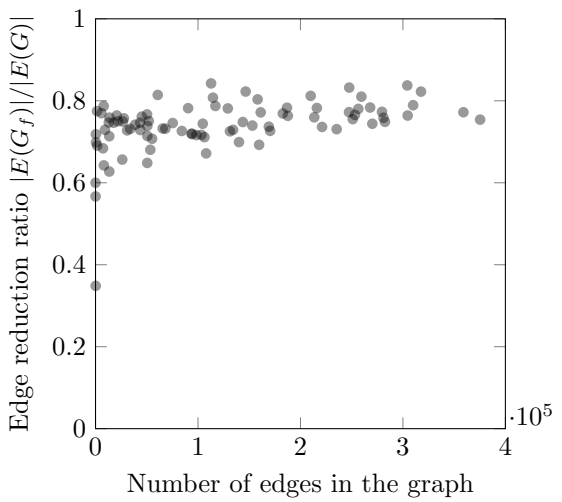
We performed taint analysis on both the original and simplified graphs. The set of subject Android applications includes the top 30 free apps, as well as some popular apps in the Editor's Choice list as of January 2015. We extracted the taint-analysis graphs using the tools from the work of Huang *et 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.[2]

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.

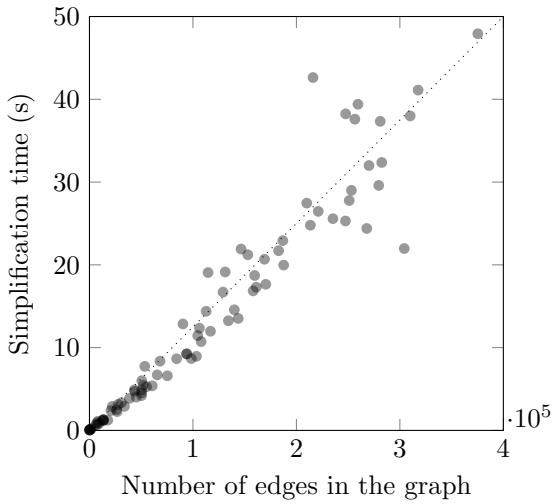
InterDyck-Reachability Algorithms. We used the following three InterDyck-reachability algorithms as the graph-reachability engine for variants of the taint analysis:

- *CFL-reachability algorithm* [14]. This method is the traditional over-approximation for InterDyck-reachability. To approximate $D_b \odot D_p$, we used a regular language R_p to over-approximate D_p. The language $D_b \odot R_p$ is still context-free, so one can apply the CFL-reachability algorithm to solve the $(D_b \odot R_p)$-reachability problem.
- *SPDS-reachability algorithm* [19]. In our client analysis, the synchronized pushdown system (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.
- *LCL-reachability algorithm* [26]. The linear-conjunctive-languages (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 are not InterDyck-reachable.

We implemented all algorithms in C++.[3] All experiments were repeated three times, and we report the average of the three trials to improve the reliability of the collected results.



(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.

Figure 6. Amount of graph reduction, and running time of the graph-simplification algorithm as a function of graph size.

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 6a 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 simplification.

As graph size increases, Figure 6a 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, the graph simplification is much faster than the InterDyck-reachability algorithms in most cases. The only exception is when the graph size is very small (*i.e.*, when the number of edges is less than 200), the simplification procedure can take time comparable to the InterDyck-reachability algorithms. Figure 6b gives the relationship between graph size and running time of the graph-simplification algorithm. It demostrates 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 remains 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.

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

[3]The implementation is available at https://www.cc.gatech.edu/~qrzhang/projects/interdyck/interdyck.html.

Table 2. Timeout statistics in the experiments.

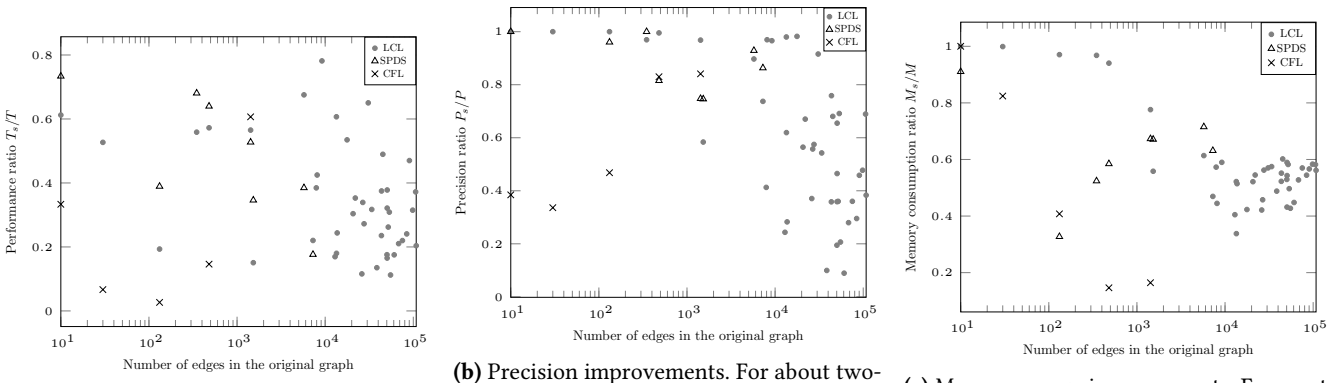
	Finished on G, G'	Finished only on G'	Timed out on G, G'
LCL	41	13	41
SPDS	9	5	81
CFL	5	2	88

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 1K edges. The SPDS-reachability algorithm runs out of time for graphs with more than 5K edges. The LCL-reachability algorithm usually finishes processing graphs with fewer than 70K edges within the time budget.

We use the following metric to the measure performance improvement: given the running time T on 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 7a. 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 8a presents the

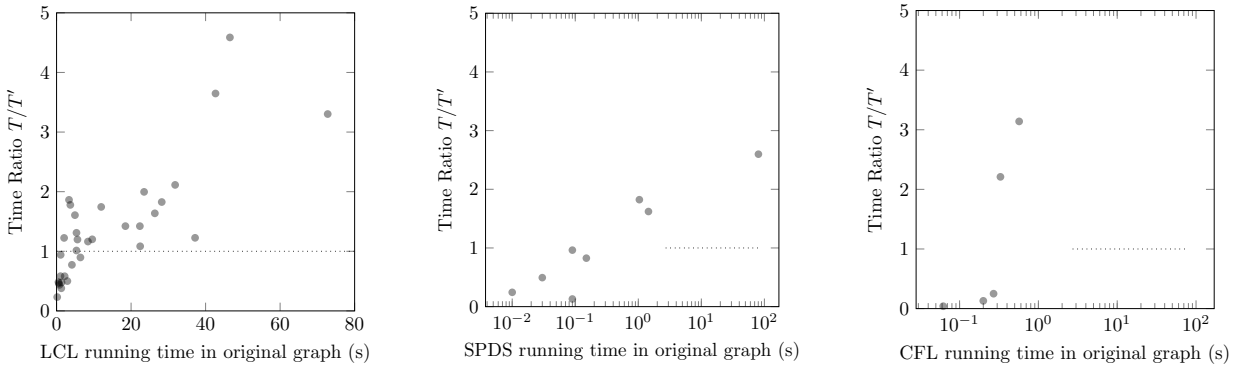


(a) Performance improvements. The running time of all benchmark programs is reduced to less than 80% after the graph simplification.

(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.

Figure 7. 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 on 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.

Figure 8. 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.

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.* Figure 8b and Figure 8c. 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 7b gives information about the precision improvements. It is interesting to note that the precision improvement is quite significant. On average, graph simplification helps InterDyck-reachability to generate a solution with only 64.92% pairs of the solution for the original graph (discarded ones 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 increasing number

of edges in the graph, the precision improvement from graph simplification is likely to be more significant.

Memory Consumption. As mentioned in Section 6.2, the average amount of edges in the simplified graph is 74.3% of the original graphs. However, Figure 7c 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 simplified graphs uses only 57.37% memory.

6.4 Discussion

Our graph-simplification algorithm is based on the bidirected Fast-Dyck algorithm on 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 based-on Dyck-reachability does not scale well in practice. Moreover, a recent result by Chatterjee *et al.* [1] has improved Fast-Dyck from $O(m \log m)$ time to $O(m)$ time. It is interesting to apply their algorithm to further improve the running time of graph simplification. On the other hand, our evaluation (Figure 6b) shows that the Fast-Dyck adopted in our current algorithm, though being in $O(m \log m)$ time, scales almost linearly in practice.

In the work of Späth *et al.* [19], 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 paper 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 "$(_1 [_1$". 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, 18, 20–22, 24]. However, solving InterDyck-reachability is undecidable [15]. 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.* [19] over-approximates InterDyck-reachability using synchronized pushdown systems. Zhang and Su [26] 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.* [25]. Chatterjee *et al.* [1] give an $O(m)$-time algorithm for solving bidirected Dyck-reachability, which improves the $O(m \log m)$ bound by Zhang *et al.* [25]. In practice, many techniques have been proposed to improve CFL-reachability-based analyses [2, 23, 27]. 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, 17] are applied to reduce the size of the constraint graphs for inclusion-based analysis. For example, the work by Hardekopf and Calvin [6] focuses 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 paper 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 memory-usage of the client analysis.

Acknowledgments

We would like to thank our shepherd and the anonymous PLDI reviewers for valuable feedback on earlier drafts of this paper, which helped improve its presentation. We thank Gabriel Eiseman for helpful comments. This work was supported in part by the United States National Science Foundation (NSF) under Grant No. 1917924; by a gift from Rajiv and Ritu Batra; by ONR under grants N00014-17-1-2889 and N00014-19-1-2318. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies.

References

[1] Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. 2018. Optimal Dyck reachability for data-dependence and alias analysis. *PACMPL* 2, POPL (2018), 30:1–30:30.

[2] Swarat Chaudhuri. 2008. Subcubic algorithms for recursive state machines. In *POPL*. 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 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*. 57–69.

[4] Arnab De and Deepak D'Souza. 2012. Scalable Flow-Sensitive Pointer Analysis for Java with Strong Updates. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*. 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 '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*. 85–96.

[6] Ben Hardekopf and Calvin Lin. 2007. Exploiting Pointer and Location Equivalence to Optimize Pointer Analysis. In *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*. 265–280.

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

[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 *ISSTA*. 106–117.

[10] Vineet Kahlon. 2009. Boundedness vs. Unboundedness of Lock Chains: Characterizing Decidability of Pairwise CFL-Reachability for Threads Communicating via Locks. In *LICS*. 27–36.

[11] John Kodumal and Alexander Aiken. 2004. The set constraint/CFL reachability connection in practice. In *PLDI*. 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 (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. 619–629.

[13] G. Ramalingam. 2000. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.* 22, 2 (2000), 416–430.

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

[15] Thomas W. Reps. 2000. Undecidability of context-sensitive data-dependence analysis. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 162–186.

[16] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*. 49–61.

[17] Atanas Rountev and Satish Chandra. 2000. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, Monica S. Lam (Ed.). ACM, 47–56.

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

[19] Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, flow-, and field-sensitive data-flow analysis using synchronized Pushdown systems. *PACMPL* 3, POPL (2019), 48:1–48:29.

[20] Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*. 387–400.

[21] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. In *OOPSLA*. 59–76.

[22] 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 *POPL*. 83–95.

[23] Guoqing Xu, Atanas Rountev, and Manu Sridharan. 2009. Scaling CFL-Reachability-Based Points-To Analysis Using Context-Sensitive Must-Not-Alias Analysis. In *ECOOP*. 98–122.

[24] Dacong Yan, Guoqing (Harry) Xu, and Atanas Rountev. 2011. Demand-driven context-sensitive alias analysis for Java. In *ISSTA*. 155–165.

[25] Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. 2013. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In *PLDI*. 435–446.

[26] Qirun Zhang and Zhendong Su. 2017. Context-sensitive data-dependence analysis via linear conjunctive language reachability. In *POPL*. 344–358.

[27] Qirun Zhang, Xiao Xiao, Charles Zhang, Hao Yuan, and Zhendong Su. 2014. Efficient subcubic alias analysis for C. In *OOPSLA*. 829–845.

[28] Xin Zheng and Radu Rugina. 2008. Demand-driven alias analysis for C. In *POPL*. 197–208.