Optimal Dyck Reachability for Data-Dependence and Alias Analysis

KRISHNENDU CHATTERJEE, Institute of Science and Technology, Austria, Austria
BHAVYA CHOUDHARY, Indian Institute of Technology, Bombay, India
ANDREAS PAVLOGIANNIS, Institute of Science and Technology, Austria, Austria

A fundamental algorithmic problem at the heart of static analysis is Dyck reachability. The input is a graph where the edges are labeled with different types of opening and closing parentheses, and the reachability information is computed via paths whose parentheses are properly matched. We present new results for Dyck reachability problems with applications to alias analysis and data-dependence analysis. Our main contributions, that include improved upper bounds as well as lower bounds that establish optimality guarantees, are as follows.

First, we consider Dyck reachability on bidirected graphs, which is the standard way of performing field-sensitive points-to analysis. Given a bidirected graph with n nodes and m edges, we present: (i) an algorithm with worst-case running time $O(m + n \cdot \alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function, improving the previously known $O(n^2)$ time bound; (ii) a matching lower bound that shows that our algorithm is optimal wrt to worst-case complexity; and (iii) an optimal average-case upper bound of $O(m)$ time, improving the previously known $O(m \cdot \log n)$ bound.

Second, we consider the problem of context-sensitive data-dependence analysis, where the task is to obtain analysis summaries of library code in the presence of callbacks. Our algorithm preprocesses libraries in almost linear time, after which the contribution of the library in the complexity of the client analysis is only linear, and only wrt the number of call sites.

Third, we prove that combinatorial algorithms for Dyck reachability on general graphs with truly sub-cubic bounds cannot be obtained without obtaining sub-cubic combinatorial algorithms for Boolean Matrix Multiplication, which is a long-standing open problem. Thus we establish that the existing combinatorial algorithms for Dyck reachability are (conditionally) optimal for general graphs. We also show that the same hardness holds for graphs of constant treewidth.

Finally, we provide a prototype implementation of our algorithms for both alias analysis and data-dependence analysis. Our experimental evaluation demonstrates that the new algorithms significantly outperform all existing methods on the two problems, over real-world benchmarks.

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

Additional Key Words and Phrases: Data-dependence analysis, CFL reachability, Dyck reachability, Bidirected graphs, treewidth

ACM Reference Format:
Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. 2018. Optimal Dyck Reachability for Data-Dependence and Alias Analysis. *Proc. ACM Program. Lang.* 2, POPL, Article 30 (January 2018), 30 pages.
https://doi.org/10.1145/3158118

Authors' addresses: Krishnendu Chatterjee, Institute of Science and Technology, Austria, Am Campus 1, Klosterneuburg, 3400, Austria, krishnendu.chatterjee@ist.ac.at; Bhavya Choudhary, Indian Institute of Technology, Bombay, IIT Area, Powai, Mumbai, 400076, India, bhavya@cse.iitb.ac.in; Andreas Pavlogiannis, Institute of Science and Technology, Austria, Am Campus 1, Klosterneuburg, 3400, Austria, pavlogiannis@ist.ac.at.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2018 Copyright held by the owner/author(s).
2475-1421/2018/1-ART30
https://doi.org/10.1145/3158118

1 INTRODUCTION

In this work we present improved upper bounds, lower bounds, and experimental results for algorithmic problems related to Dyck reachability, which is a fundamental problem in static analysis. We present the problem description, its main applications, the existing results, and our contributions.

Static analysis and language reachability. Static analysis techniques obtain information about programs without running them on specific inputs. These techniques explore the program behavior for all possible inputs and all possible executions. For non-trivial programs, it is impossible to explore all the possibilities, and hence various approximations are used. A standard way to express a plethora of static analysis problems is via *language reachability* that generalizes graph reachability. The input consists of an underlying graph with labels on its edges from a fixed alphabet, and a language, and reachability paths between two nodes must produce strings that belong to the given language [Reps 1997; Yannakakis 1990].

CFL and Dyck reachability. An extremely important case of language reachability in static analysis is *CFL* reachability, where the input language is context-free, which can be used to model, e.g., context-sensitivity or field-sensitivity. The CFL reachability formulation has applications to a very wide range of static analysis problems, such as interprocedural data-flow analysis [Reps et al. 1995], slicing [Reps et al. 1994], shape analysis [Reps 1995], impact analysis [Arnold 1996], type-based flow analysis [Rehof and Fähndrich 2001] and alias/points-to analysis [Shang et al. 2012; Sridharan and Bodík 2006; Sridharan et al. 2005; Xu et al. 2009; Yan et al. 2011a; Zheng and Rugina 2008], etc. In practice, widely-used large-scale analysis tools, such as Wala [Wal 2003] and Soot [Bodden 2012; Vallée-Rai et al. 1999], equip CFL reachability techniques to perform such analyses. In most of the above cases, the languages used to define the problem are those of properly-matched parenthesis, which are known as Dyck languages, and form a proper subset of context-free languages. Thus Dyck reachability is at the heart of many problems in static analysis.

Alias analysis. Alias analysis has been one of the major types of static analysis and a subject of extensive study [Choi et al. 1993; Hind 2001; Landi and Ryder 1992; Sridharan et al. 2013]. The task is to decide whether two pointer variables may point to the same object during program execution. As the problem is computationally expensive [Horwitz 1997; Ramalingam 1994], practically relevant results are obtained via approximations. One popular way to perform alias analysis is via points-to analysis, where two variables may alias if their points-to sets intersect. Points-to analysis is typically phrased as a Dyck reachability problem on Symbolic Points-to Graphs (SPGs), which contain information about variables, heap objects and parameter passing due to method calls [Xu et al. 2009; Yan et al. 2011a]. In alias analysis there is an important distinction between context and field sensitivity, which we describe below.

- *Context vs field sensitivity.* Typically, the Dyck parenthesis are used in SPGs to specify two types of constraints. *Context sensitivity* refers to the requirement that reachability paths must respect the calling context due to method calls and returns. *Field sensitivity* refers to the requirement that reachability paths must respect field accesses of composite types in Java [Sridharan and Bodík 2006; Sridharan et al. 2005; Xu et al. 2009; Yan et al. 2011a], or references and dereferences of pointers [Zheng and Rugina 2008] in C. Considering both types of sensitivity makes the problem undecidable [Reps 2000]. Although one recent workaround is approximation algorithms [Zhang and Su 2017], the standard approach has been to consider only one type of sensitivity. Field sensitivity has been reported to produce better results, and being more scalable [Lhoták and Hendren 2006]. We focus on context-insensitive, but field-sensitive points-to analysis.

Data-dependence analysis. Data-dependence analysis aims to identify the def-use chains in a program. It has many applications, including slicing [Reps et al. 1994], impact analysis [Arnold 1996] and bloat detection [Xu et al. 2010]. It is also used in compiler optimizations, where data dependencies are used to infer whether it is safe to reorder or parallelize program statements [Kuck et al. 1981]. Here we focus on the distinction between *library vs client analysis* and the challenge of *callbacks*.

- *Library vs Client.* Modern-day software is developed in multiple stages and is interrelated. The vast majority of software development relies on existing libraries and third-party components which are typically huge and complex. At the same time, the analysis of client code is ineffective if not performed in conjunction with the library code. These dynamics give rise to the potential of analyzing library code once, in an offline stage, and creating suitable analysis summaries that are relevant to client behavior only. The benefit of such a process is two-fold. First, library code need only be analyzed once, regardless of the number of clients that link to it. Second, it offers fast client-code analysis, since the expensive cost of analyzing the huge libraries has been spent offline, in an earlier stage. Data-dependence analysis admits a nice separation between library and client code, and has been studied in [Palepu et al. 2017; Tang et al. 2015].
- *The challenge of callbacks.* As pointed out recently in [Tang et al. 2015], one major obstacle to effective library summarization is the presence of callbacks. Callback functions are declared and used by the library, but are implemented by the client. Since these functions are missing when the library code is analyzed, library summarization is ineffective and the whole library needs to be reanalyzed on the client side, when callback functions become available.

Algorithmic formulations and existing results. We describe below the key algorithmic problems in the applications mentioned above and the existing results. We focus on data-dependence and alias analysis via Dyck reachability, which is the standard way for performing such analysis. Recall that the problem of Dyck reachability takes as input a (directed) graph, where some edges are marked with opening and closing parenthesis, and the task is to compute for every pair of nodes whether there exists a path between them such that the parenthesis along its edges are matched.

(1) *Points-to analysis.* Context-insensitive, field-sensitive points-to analysis via Dyck reachability is phrased on an SPG G with n nodes and m edges. Additionally, the graph is *bidirected*, meaning that if G has an edge (u, v) labeled with an opening parenthesis, then it must also have the edge (v, u) labeled with the corresponding closing parenthesis. Bidirected graphs are found in most existing works on on-demand alias analysis via Dyck reachability, and their importance has been remarked in various works [Yuan and Eugster 2009; Zhang et al. 2013].
The best existing algorithms for the problem appear in the recent work of [Zhang et al. 2013], where two algorithms are proposed. The first has $O(n^2)$ *worst-case* time complexity; and the second has $O(m \cdot \log n)$ *average-case* time complexity and $O(m \cdot n \cdot \log n)$ *worst-case* complexity. Note that for dense graphs $m = \Theta(n^2)$, and the first algorithm has better average-case complexity too.

(2) *Library/Client data-dependence analysis.* The standard algorithmic formulation of context-sensitive data-dependence analysis is via Dyck reachability, where the parenthesis are used to properly match method calls and returns in a context-sensitive way [Reps 2000; Tang et al. 2015]. The algorithmic approach to Library/Client Dyck reachability consists of considering two graphs G_1 and G_2, for the library and client code respectively. The computation is split into two phases. In the *preprocessing phase*, the Dyck reachability problem is solved on G_1 (using a CFL/Dyck reachability algorithm), and some summary information is maintained, which is typically in the form of some subgraph G_1' of G_1. In the *query phase*, the Dyck reachability

is solved on the combination of the two graphs G_1' and G_2. Let n_1, n_2 and n_1' be the sizes of G_1, G_2 and G_1' respectively. The algorithm spends $O(n_1^3)$ time in the preprocessing phase, and $O((n_1' + n_2)^3)$ time in the query phase. Hence we have an improvement if $n_1' >> n_1$.
In the presence of callbacks, library summarization via CFL reachability is ineffective, as n_1' can be as large as n_1. To face this challenge, the recent work of [Tang et al. 2015] introduced TAL reachability. This approach spends $O(n_1^6)$ time on the client code (hence more than the CFL reachability algorithm), and is able to produce a summary of size $s < n_1$ even in the presence of callbacks. Afterwards, the client analysis is performed in $O((s + n_2)^6)$ time, and hence the cost due to the library only appears in terms of its summary.

(3) *Dyck reachability on general graphs.* As we have already mentioned, Dyck reachability is a fundamental algorithmic formulation of many types of static analysis. For general graphs (not necessarily bidirected), the existing algorithms require $O(n^3)$ time, and they essentially solve the more general CFL reachability problem [Yannakakis 1990]. The current best algorithm is due to [Chaudhuri 2008], which utilizes the well-knwon Four Russians' Trick to exhibit complexity $O(n^3/\log n)$. The problem has been shown to be 2NPDA-hard [Heintze and McAllester 1997], which yields a conditional cubic lower bound in its complexity.

Our contributions. Our main contributions can be characterized in three parts: (a) improved upper bounds; (b) lower bounds with optimality guarantees; and (c) experimental results. We present the details of each of them below.

Improved upper bounds. Our improved upper bounds are as follows:

(1) For Dyck reachability on bidirected graphs with n nodes and m edges, we present an algorithm with the following bounds: (a) The worst-case complexity bound is $O(m + n \cdot \alpha(n))$ time and $O(m)$ space, where $\alpha(n)$ is the inverse Ackermann function, improving the previously known $O(n^2)$ time bound. Note that $\alpha(n)$ is an extremely slowly growing function, and for all practical purposes, $\alpha(n) \leq 4$, and hence practically the worst-case bound of our algorithm is linear. (b) The average-case complexity is $O(m)$ improving the previously known $O(m \cdot \log n)$ bound. See Table 1 for a summary.
(2) For *Library/Client Dyck reachability* we exploit the fact that the data-dependence graphs that arise in practice have special structure, namely they contain components of small treewidth. We denote by n_1 and n_2 the size of the library graph and client graph, and by k_1 and k_2 the number of call sites in the library graph and client graph, respectively. We present an algorithm that analyzes the library graph in $O(n_1 + k_1 \cdot \log n_1)$ time and $O(n_1)$ space. Afterwards, the library and client graphs are analyzed together only in $O(n_2 + k_1 \cdot \log n_1 + k_2 \cdot \log n_2)$ time and $O(n_1 + n_2)$ space. Hence, since typically $n_1 >> n_2$ and $n_i >> k_i$, the cost of analyzing the large library occurs only in the preprocessing phase. When the client code needs to be analyzed, the cost incurred due to the library code is small. See Table 2 for a summary.

Lower bounds and optimality guarantees. Along with improved upper bounds we present lower bound and conditional lower bound results that imply optimality guarantees. Note that optimal guarantees for graph algorithms are extremely rare, and we show the algorithms we present have certain optimality guarantees.

(1) For Dyck reachability on bidirected graphs we present a matching lower bound of $\Omega(m+n\cdot\alpha(n))$ for the worst-case time complexity. Thus we obtain matching lower and upper bounds for the worst-case complexity, and thus our algorithm is optimal wrt to worst-case complexity. Since the average-case complexity of our algorithm is linear, the algorithm is also optimal wrt the average-case complexity.

Table 1. Comparison of our results with existing work for Dyck reachability on bidirected graphs with n nodes and m edges. We also prove a matching lower-bound for the worst-case analysis.

	Worst-case Time	**Average-case Time**	**Space**	**Reference**
Existing	$O(n^2)$	$O(\min(n^2, m \cdot \log n))$	$O(m)$	[Zhang et al. 2013]
Our Result	$O(m + n \cdot \alpha(n))$	$O(m)$	$O(m)$	Theorem 3.6 , Corollary 3.7

Table 2. Library/Client CFL reachability on the library graph of size n_1 and the client graph of size n_2.
s is the number of library summary nodes, as defined in [Tang et al. 2015].
k_1 is the number of call sites in the library code, with $k_1 < s$.
k_2 is the number of call sites in the client code.

Approach	**Time**		**Space**		Reference
	Library	Client	Library	Client	
CFL	$O(n_1^3)$	$O((n_1 + n_2)^3)$	$O(n_1^2)$	$O((n_1 + n_2)^2)$	[Tang et al. 2015]
TAL	$O(n_1^6)$	$O((s + n_2)^6)$	$O(n_1^4)$	$O((s + n_2)^4)$	[Tang et al. 2015]
Our Result	$O(n_1 + k_1 \cdot \log n_1)$	$O(n_2 + k_1 \cdot \log n_1 + k_2 \cdot \log n_2)$	$O(n_1)$	$O(n_1 + n_2)$	Theorem 5.8

(2) For *Library/Client Dyck reachability* note that $k_1 \leq n_1$ and $k_2 \leq n_2$. Hence our algorithm for analyzing library and client code is almost linear time, and hence optimal wrt polynomial improvements.
(3) For Dyck reachability on general graphs we present a conditional lower bound. In algorithmic study, a standard problem for showing conditional cubic lower bounds is Boolean Matrix Multiplication (BMM) [Abboud and Vassilevska Williams 2014; Henzinger et al. 2015; Lee 2002; Vassilevska Williams and Williams 2010]. While fast matrix multiplication algorithms exist (such as Strassen's algorithm [Strassen 1969]), these algorithms are not "combinatorial"[1]. The standard conjecture (called the BMM conjecture) is that there is no truly sub-cubic[2] combinatorial algorithm for BMM, which has been widely used in algorithmic studies for obtaining various types of hardness results [Abboud and Vassilevska Williams 2014; Henzinger et al. 2015; Lee 2002; Vassilevska Williams and Williams 2010]. We show that Dyck reachability on general graphs even for a single pair is BMM hard. More precisely, we show that for any $\delta > 0$, any algorithm that solves pair Dyck reachability on general graphs in $O(n^{3-\delta})$ time implies an algorithm that solves BMM in $O(n^{3-\delta/3})$ time. Since all algorithms for Dyck reachability are combinatorial, it establishes a conditional hardness result (under the BMM conjecture) for general Dyck reachability. Additionally, we show that the same hardness result holds for Dyck reachability on graphs of constant treewidth. Our hardness shows that the existing cubic algorithms are optimal (modulo logarithmic-factor improvements), under the BMM conjecture. Existing work establishes that Dyck reachability is 2NPDA-hard [Heintze and McAllester 1997], which yields a a conditional lower bound. Our result shows that Dyck reachability is also BMM-hard, and even on constant-treewidth graphs, and thus strengthens the conditional cubic lower bound for the problem.

[1]Not combinatorial means algebraic methods [Le Gall 2014], which are algorithms with large constants. In contrast, combinatorial algorithms are discrete and non-algebraic; for detailed discussion see [Henzinger et al. 2015]
[2]Truly sub-cubic means polynomial improvement, in contrast to improvement by logarithmic factors such as $O(n^3/\log n)$

Experimental results. A key feature of our algorithms are that they are simple to implement. We present experimental results both on alias analysis (see Section 6.1) and library/client data-dependence analysis (see Section 6.2) and show that our algorithms outperform previous approaches for the problems on real-world benchmarks.

Due to lack of space, full proofs can be found in full version of this paper [Chatterjee et al. 2017].

1.1 Other Related Work

We have already discussed in details the most relevant works related to language reachability, alias analysis and data-dependence analysis. We briefly discuss works related to treewidth in program analysis and verification.

Treewidth in algorithms and program analysis. In the context of programming languages, it was shown by [Thorup 1998] that the control-flow graphs for goto-free programs for many programming languages have constant treewidth, which has been followed by practical approaches as well (such as [Gustedt et al. 2002]). The treewidth property has received a lot of attention in algorithm community, for NP-complete problems [Arnborg and Proskurowski 1989; Bern et al. 1987; Bodlaender 1988], combinatorial optimization problems [Bertele and Brioschi 1972], graph problems such as shortest path [Chatterjee et al. 2016b; Chaudhuri and Zaroliagis 1995]. In algorithmic analysis of programming languages and verification the treewidth property has been exploited in interprocedural analysis [Chatterjee et al. 2015b], concurrent intraprocedural analysis [Chatterjee et al. 2016a], quantitative verification of finite-state graphs [Chatterjee et al. 2015a], etc. To the best of our knowledge the constant-treewidth property has not be considered for data-dependence analysis. Our experimental results show that in practice many real-world benchmarks have the constant-treewidth property, and our algorithms for data-dependence analysis exploit this property to present faster algorithms.

2 PRELIMINARIES

Graphs and paths. We denote by $G = (V, E)$ a finite directed graph (henceforth called simply a graph) where V is a set of n nodes and $E \subseteq V \times V$ is an edge relation of m edges. Given a set of nodes $X \subseteq V$, we denote by $G[X] = (X, E \cap (X \times X))$ the subgraph of G induced by X. A *path* P is a sequence of edges $(e_1, \dots, e_r)$ and each $e_i = (x_i, y_i)$ is such that $x_1 = u$, $y_r = v$, and for all $1 \leq i \leq r-1$ we have $y_i = x_{i+1}$. The length of P is $|P| = r$. A path P is *simple* if no node repeats in P (i.e., the path does not contain a cycle). Given two paths $P_1 = (e_1, \dots e_{r_1})$ and $P_2 = (e'_1, \dots e'_\ell)$ with $e_r = (x, y)$ and $e'_1 = (y, z)$, we denote by $P_1 \circ P_2$ the *concatenation* of P_2 on P_1. We use the notation $x \in P$ to denote that a node x appears in P, and $e \in P$ to denote that an edge e appears in P. Given a set $B \subseteq V$, we denote by $P \cap B$ the set of nodes of B that appear in P. We say that a node u is *reachable* from node v if there exists a path $P : u \rightsquigarrow v$.

Dyck Languages. Given a nonnegative integer $k \in \mathbb{N}$, we denote by $\Sigma_k = \{\epsilon\} \cup \{\alpha_i, \overline{\alpha}_i\}_{i=1}^k$ a finite *alphabet* of k parenthesis types, together with a null element ϵ. We denote by $\mathcal{L}_k$ the Dyck language over Σ_k, defined as the language of strings generated by the following context-free grammar $\mathcal{G}_k$:

$$\mathcal{S} \to \mathcal{S}\,\mathcal{S} \mid \mathcal{A}_1\,\overline{\mathcal{A}}_1 \mid \dots \mid \mathcal{A}_k\,\overline{\mathcal{A}}_k \mid \epsilon\,; \qquad \mathcal{A}_i \to \alpha_i\,\mathcal{S}\,; \qquad \overline{\mathcal{A}}_i \to \mathcal{S}\,\overline{\alpha}_i$$

Given a string s and a non-terminal symbol X of the above grammar, we write $X \vdash s$ to denote that X produces s according to the rules of the grammar. In the rest of the document we consider an alphabet Σ_k and the corresponding Dyck language $\mathcal{L}_k$. We also let $\Sigma_k^O = \{\alpha_i\}_{i=1}^k$ and $\Sigma_k^C = \{\overline{\alpha}_i\}_{i=1}^k$ be the subsets of Σ_k of only opening and closing parenthesis, respectively.

Labeled graphs, Dyck reachability, and Dyck SCCs (DSCCs). We denote by $G = (V, E)$ a Σ_k-labeled directed graph where V is the set of nodes and $E \subseteq V \times V \times \Sigma_k$ is the set of edges labeled with symbols from Σ_k. Hence, an edge e is of the form $e = (u, v, \lambda)$ where $u, v \in V$ and $\lambda \in \Sigma_k$. We require that for every $u, v \in V$, there is a unique label λ such that $(u, v, \lambda) \in E$. Often we will be interested only on the endpoints of an edge e, in which case we represent $e = (u, v)$, and will denote by $\lambda(e)$ the label of e. Given a path P, we define the *label* of P as $\lambda(P) = \lambda(e_1) \dots \lambda(e_r)$. Given two nodes u, v, we say that v is Dyck-reachable from u if there exists a path $P : u \rightsquigarrow v$ such that $\lambda(P) \in \mathcal{L}_k$. In that case, P is called a *witness path* of the reachability. A set of nodes $X \subseteq V$ is called a *Dyck SCC* (or *DSCC*) if for every pair of nodes $u, v \in X$, we have that u reaches v and v reaches u. Note that there might exist a DSCC X and a pair of nodes $u, v \in X$ such that every witness path $P : u \rightsquigarrow v$ might be such that $P \nsubseteq X$, i.e., the witness path contains nodes outside the DSCC.

3 DYCK REACHABILITY ON BIDIRECTED GRAPHS

In this section we present an optimal algorithm for solving the Dyck reachability problem on Σ_k-labeled *bidirected* graphs G. First, in Section 3.1, we formally define the problem. Second, in Section 3.2, we describe an algorithm BidirectedReach that solves the problem in time $O(m+n\cdot\alpha(n))$, where n is the number of nodes of G, m is the number of edges of G, and $\alpha(n)$ is the inverse Ackermann function. Finally, in Section 3.3, we present an $\Omega(m + n \cdot \alpha(n))$ lower bound.

3.1 Problem Definition

We start with the problem definition of Dyck reachability on bidirected graphs. For the modeling power of bidirected graphs we refer to [Yuan and Eugster 2009; Zhang et al. 2013] and our Experimental Section 6.1.

Bidirected Graphs. A Σ_k labeled graph $G = (V, E)$ is called *bidirected* if for every pair of nodes $u, v \in V$, the following conditions hold. (1) $(u, v, \epsilon) \in E \quad \text{iff} \quad (v, u, \epsilon) \in E$; and (2) for all $1 \leq i \leq k$ we have that $(u, v, \alpha_i) \in E \quad \text{iff} \quad (v, u, \overline{\alpha}_i) \in E$. Informally, the edge relation is symmetric, and the labels of symmetric edges are complimentary wrt to opening and closing parenthesis. The following remark captures a key property of bidirected graphs that can be exploited to lead to faster algorithms.

Remark 1 ([Zhang et al. 2013]). *For bidirected graphs the Dyck reachability relation forms an equivalence, i.e., for all bidirected graphs G, for every pair of nodes u, v, we have that v is Dyck-reachable from u iff u is Dyck-reachable from v.*

Remark 2. *We consider without loss of generality that a bidirected graph G has no edge (u, v) such that $\lambda(u, v) = \epsilon$, i.e., there are no ϵ-labeled edges. This is because in such a case, u, v form a DSCC, and can be merged into a single node. Merging all nodes that share an ϵ-labeled edge requires only linear time, and hence can be applied as a preprocessing step at (asymptotically) no extra cost.*

Dyck reachability on bidirected graphs. We are given a Σ_k-labeled bidirected graph $G = (V, E)$, and our task is to compute for every pair of nodes u, v whether v is Dyck-reachable from u. As customary, we consider that $k = O(1)$, i.e., k is fixed wrt to the input graph [Chaudhuri 2008]. In view of Remark 1, it suffices that the output is a list of DSCCs. Note that the output has size $\Theta(n)$ instead of $\Theta(n^2)$ that would be required for storing one bit of information per u, v pair. Additionally, the pair query time is $O(1)$, by testing whether the two nodes belong to the same DSCC.

3.2 An Almost Linear-time Algorithm

We present our algorithm BidirectedReach, for Dyck reachability on bidirected graphs, with almost linear-time complexity.

Informal description of BidirectedReach**.** We start by providing a high-level description of BidirectedReach. The main idea is that for any two distinct nodes u, v to belong to some DSCC X, there must exist two (not necessarily distinct) nodes x, y that belong to some DSCC Y (possibly $X = Y$)[3] and a closing parenthesis $\overline{\alpha}_i \in \Sigma_k^C$ such that $(x, u, \overline{\alpha}_i), (y, v, \overline{\alpha}_i) \in E$. See Figure 1 for an illustration. The algorithm uses a Disjoint Sets data structure to maintain DSCCs discovered so far. Each DSCC is represented as a tree T rooted on some node $x \in V$, and x is the only node of T that has outgoing edges. However, any node of T can have incoming edges. See Figure 2 for an illustration. Upon discovering that a root node x of some tree T has two or more outgoing edges $(x, u_1, \overline{\alpha}_i), (x, u_2, \overline{\alpha}_i), \dots (x, u_r, \overline{\alpha}_i)$, for some $\overline{\alpha}_i \in \Sigma_k^C$, the algorithm uses r Find operations of the Disjoint Sets data structure to determine the trees T_i that the nodes u_i belong to. Afterwards, a Union operation is performed between all T_i to form a new tree T, and all the outgoing edges of the root of each T_i are merged to the outgoing edges of the root of T.

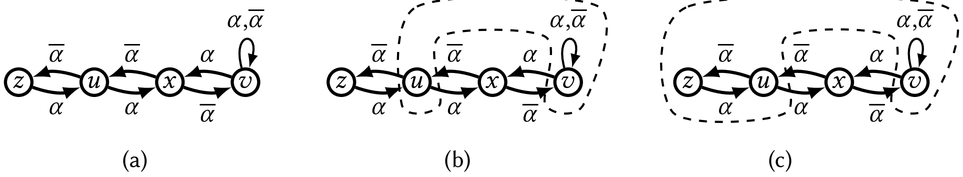


Fig. 1. Illustration of the merging principle of BidirectedReach. **(1a)** The nodes u and v are in the same DSCC since node x has an outgoing edge to each of u and v labeled with the closing parenthesis $\overline{\alpha}$. **(1b)** Similarly, nodes z and v belong to the same DSCC, since there exist two nodes u and v such that (i) u and v belong to the same DSCC, (ii) u has an outgoing edge to z, and v has an outgoing edge to itself, and (iii) both outgoing edges are labeled with the same closing parenthesis symbol. **(1c)** The final DSCC formation.

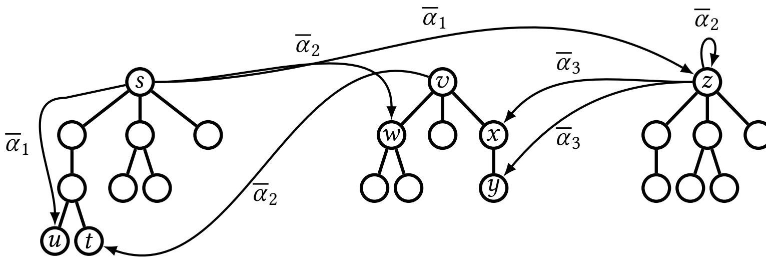


Fig. 2. A state of BidirectedReach consists of a set of trees, with outgoing edges coming only from the root of each tree.

Complexity overview. The cost of every Find and Union operation is bounded by the inverse Ackermann function $\alpha(n)$ (see [Tarjan 1975]), which, for all practical purposes, can be considered constant. Additionally, every edge-merge operation requires constant time, using a linked list

[3]That is, x and y might refer to the same node, and X and Y to the same DSCC.

for storing the outgoing edges. Although list merging in constant time creates the possibility of duplicate edges, such duplicates come at no additional complexity cost. Since every Union of k trees reduces the number of existing edges by $k-1$, the overall complexity of BidirectedReach is $O(m \cdot \alpha(n))$. We later show how to obtain the $O(m + n \cdot \alpha(n))$ complexity.

We are now ready to give the formal description of BidirectedReach. We start with introducing the Union-Find problem, and its solution given by a disjoint sets data structure.

The Union-Find problem. The Union-Find problem is a well-studied problem in the area of algorithms and data structures [Cormen et al. 2001; Galil and Italiano 1991]. The problem is defined over a *universe* X of n elements, and the task is to maintain partitions of X under set union operations. Initially, every element $x \in X$ belongs to a singleton set $\{x\}$. A *union-find* sequence σ is a sequence of m (typically $m \geq n$) operations of the following two types.

(1) Union(x, y), for $x, y \in X$, performs a union of the sets that x and y belong to.
(2) Find(x), for $x \in X$, returns the name of the unique set containing x.

The sequence σ is presented online, i.e., an operation needs to be completed before the next one is revealed. Additionally, a Union(x, y) operation is allowed in the i-th position of σ only if the prefix of σ up to position $i-1$ places x and y on different sets. The output of the problem consists of the answers to Find operations of σ. It is known that the problem can be solved in $O(m \cdot \alpha(n))$ time, by an appropriate Disjoint Sets data structure [Tarjan 1975], and that this complexity is optimal [Banachowski 1980; Tarjan 1979].

The DisjointSets **data structure.** We consider at our disposal a Disjoint Sets data structure DisjointSets which maintains a set of subsets of V under a sequence of set union operations. At all times, the name of each set X is a node $x \in X$ which is considered to be the representative of x. DisjointSets provides the following operations.

(1) For a node u, MakeSet(u) constructs the singleton set $\{u\}$.
(2) For a node u, Find(u) returns the representative of the set that u belongs to.
(3) For a set of nodes $S \subseteq V$ which are pairwise in different sets, and a distinguished node $x \in S$, Union(S, x) performs the union of the sets that the nodes in S belong to, and makes x the representative of the new set.

The DisjointSets data structure can be straightforwardly obtained from the corresponding Disjoint Sets data structures used to solve the Union-Find problem [Tarjan 1975], and has $O(\alpha(n))$ amortized complexity per operation. Typically each set is stored as a rooted tree, and the root node is the representative of the set.

Formal description of BidirectedReach**.** We are now ready to present formally BidirectedReach in Algorithm 1. Recall that, in view of Remark 2, we consider that the input graph has no ϵ-labeled edges. In the initialization phase, the algorithm constructs a map Edges : $V \times \Sigma_k^C \to V^*$. For each node $u \in V$ and closing parenthesis $\overline{\alpha}_i \in \Sigma_k^C$, Edges$[u][\overline{\alpha}_i]$ will store the nodes that are found to be reachable from u via a path P such that $\overline{\mathcal{A}}_i \vdash \lambda(P)$ (i.e., the label of P has matching parenthesis except for the last parenthesis $\overline{\alpha}_i$). Observe that all such nodes must belong to the same DSCC.

The main computation happens in the loop of Line 11. The algorithm maintains a queue $\mathcal{Q}$ that acts as a worklist and stores pairs $(u, \overline{\alpha}_i)$ such that u is a node that has been found to contain at least two outgoing edges labeled with $\overline{\alpha}_i$. Upon extracting an element $(u, \overline{\alpha}_i)$ from the queue, the algorithm obtains the representatives v of the sets of the nodes in Edges$[u][\overline{\alpha}_i]$. Since all such nodes belong to the same DSCC, the algorithm chooses an element x to be the new representative, and performs

a Union operation of the underlying sets. The new representative x gathers the outgoing edges of all other nodes $v \in \mathsf{Edges}[u][\overline{\alpha}_i]$, and afterwards $\mathsf{Edges}[u][\overline{\alpha}_i]$ points only to x.

Algorithm 1: BidirectedReach

```
   Input: A Σ_k-labeled bidirected graph G = (V, E)
   Output: A DisjointSets map of DSCCs
   // Initialization
1  Q ← an empty queue
2  Edges ← a map V × Σ_k^C → V* implemented as a linked list
3  DisjointSets ← a disjoint-sets data structure over V
4  foreach u ∈ V do
5      DisjointSets.MakeSet(u)
6      for i ← 1 to k do
7          Edges[u][ᾱ_i] ← (v : (u, v, ᾱ_i) ∈ E)
8          if |Edges[u][ᾱ_i]| ≥ 2 then Insert (u, ᾱ_i) in Q
9      end
10 end
   // Computation
11 while Q is not empty do
12     Extract (u, ᾱ_i) from Q
13     if u = DisjointSets.Find(u) then
14         Let S ← {DisjointSets.Find(w) : w ∈ Edges[u][ᾱ_i]}
15         if |S| ≥ 2 then
16             Let x ← some arbitrary element of S \ {u}
17             Make DisjointSets.Union(S, x)
18             for j ← 1 to k do
19                 foreach v ∈ S \ {x} do
20                     if u ≠ v or i ≠ j then
21                         Move Edges[v][ᾱ_j] to Edges[x][ᾱ_j]
22                     else
23                         Append (x) to Edges[x][ᾱ_j]
24                     end
25                 end
26                 if |Edges[x][ᾱ_j]| ≥ 2 then Insert (x, ᾱ_j) in Q
27             end
28         else
29             Let x ← the single node in S
30         end
31         if u ∉ S or |S| = 1 then Edges[u][ᾱ_i] ← (x)
32 end
33 return DisjointSets
```

Example. Consider the state of the algorithm given by Figure 2, representing the DSCCs of the Union-Find data structure DisjointSets (i.e., the undirected trees in the figure) as well as the contents of the Edges data structure (i.e., the directed edges in the Figure). There are currently 3 DSCCS, with representatives s, v and z. Recall that the queue Q stores (node, closing parenthesis) pairs with the property that the node has at least two outgoing edges labeled with the respective closing parenthesis. Observe that nodes s and z have at two outgoing edges each that have the same type

of parenthesis, hence they must have been inserted in the queue Q at some point. Assume that $Q = [(s, \overline{\alpha}_1), (z, \overline{\alpha}_3)]$. The algorithm will exhibit the following sequence of steps.

(1) The element $(z, \overline{\alpha}_3)$ is extracted from Q. We have $\text{Edges}[z][\overline{\alpha}_3] = (x, y)$. Observe that x and y belong to the same DSCC rooted at v, hence in Line 14 the algorithm will construct $S = \{v\}$. Since $|S| = 1$, the algorithm will simply set $\text{Edges}[z][\overline{\alpha}_3] = (v)$ in Line 31, and no new DSCC has been formed.
(2) The element $(s, \overline{\alpha}_1)$ is extracted from Q. We have $\text{Edges}[s][\overline{\alpha}_1] = (u, z)$. Since u and z belong to different DSCCs, the algorithm will construct $S = \{s, z\}$, and perform a $\text{DisjointSets.Union}(S, x)$ operation, where $x = z$. Note that union-by-rank will make the tree of z a subtree of the tree of s, i.e., z will become a child of s. Afterwards, the algorithm swaps the names of z and s, as required by the choice of x in Line 16. Finally, in Line 21, the algorithm will move $\text{Edges}[s][\overline{\alpha}_i]$ to $\text{Edges}[z][\overline{\alpha}_i]$ for $i = 1, 2$. Since now $|\text{Edges}[z][\overline{\alpha}_2]| \geq 2$, the algorithm inserts $(z, \overline{\alpha}_2)$ in Q. See Figure 3a.
(3) The element $(z, \overline{\alpha}_2)$ is extracted from Q. We have $\text{Edges}[z][\overline{\alpha}_2] = (v, z)$. Since v and z belong to different DSCCs, the algorithm will construct $S = \{v, z\}$, and perform a $\text{DisjointSets.Union}(S, x)$ operation, where $x = v$. Note that union-by-rank will make the tree of v a subtree of the tree of z, i.e., v will become a child of z. Afterwards, the algorithm swaps the names of v and z, as required by the choice of x in Line 16. Finally, in Line 21, the algorithm will move $\text{Edges}[z][\overline{\alpha}_2]$ to $\text{Edges}[v][\overline{\alpha}_2]$. Since now $|\text{Edges}[v][\overline{\alpha}_2]| \geq 2$, the algorithm inserts $(v, \overline{\alpha}_2)$ in Q. See Figure 3b.
(4) The element $(v, \overline{\alpha}_2)$ is extracted from Q. We have $\text{Edges}[v][\overline{\alpha}_2] = (v, t)$. Observe that v and t belong to the same DSCC rooted at v, hence in Line 14 the algorithm will construct $S = \{v\}$. Since $|S| = 1$, the algorithm will simply set $\text{Edges}[v][\overline{\alpha}_2] = (v)$ in Line 31, and will terminate.

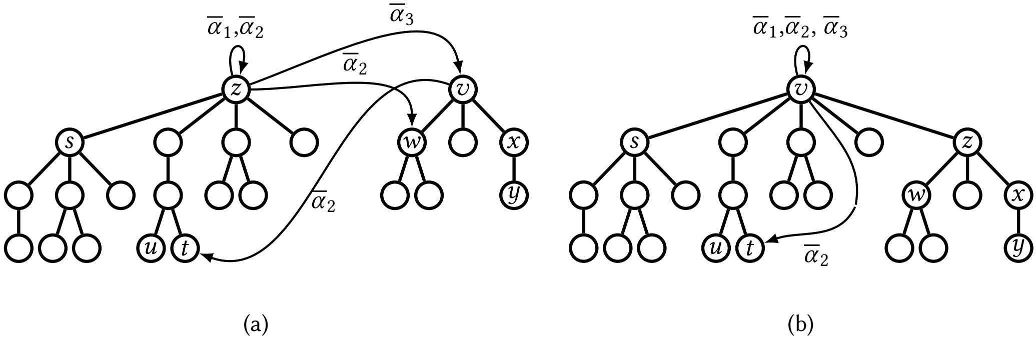


Fig. 3. The intermediate stages of BidirectedReach starting from the stage of Figure 2.

Correctness. We start with the correctness statement of BidirectedReach, which is established in two parts, namely the soundness and completeness, which are shown in the following two lemmas.

LEMMA 3.1 (SOUNDNESS). *At the end of* BidirectedReach, *for every pair of nodes $u, v \in V$, if* $\text{DisjointSets.Find}(u) = \text{DisjointSets.}\mathit{Find}(v)$ *then u and v belong to the same DSCC.*

LEMMA 3.2 (COMPLETENESS). *At the end of* BidirectedReach, *for every pair of nodes $u, v \in V$ in the same DSCC, u and v belong to the same set of* DisjointSets.

Complexity. We now establish the complexity of BidirectedReach, in a sequence of lemmas.

Lemma 3.3. *The main loop of Line 11 will be executed $O(n)$ times.*

Proof. Initially Q is populated by Line 8, which inserts $O(n)$ elements, as $k = O(1)$. Afterwards, for every $\ell \leq k = O(1)$ elements $(u, \overline{\alpha}_j)$ inserted in Q via Line 26, there is at least one node $v \in S$ which stops being a representative of its own set in DisjointSets, and thus will not be in S in further iterations. Hence Q will contain $O(n)$ elements in total, and the result follows. □

The sets S_j and S'_j. Consider an element $(u, \overline{\alpha}_i)$ extracted from Q in the j-th iteration of the algorithm in Line 11. We denote by S'_j the set Edges$[u][\overline{\alpha}_i]$, and by S_j the set S constructed in Line 14. If S was not constructed in that iteration (i.e., the condition in Line 13 does not hold), then we let $S_j = \emptyset$. It is easy to see that $|S_j| \leq |S'_j|$ for all j. The following crucial lemma bounds the total sizes of the sets S'_j constructed throughout the execution of BidirectedReach.

Lemma 3.4. *Let r be the number of iterations of the main loop in Line 11. We have $\sum_{j=1}^{r} |S'_j| = O(m)$.*

Proof. By Lemma 3.3 we have $r = O(n)$. Let $J = \{j : |S'_j| \geq 2\}$, and it suffices to prove that $\sum_{j \in J} S'_j = O(m)$.

We first argue that after a pair $(u, \overline{\alpha}_i)$ has been extracted from Q in some iteration $j \in J$, the number of edges in Edges decreases by at least $|S'_j| - 1$. We consider the following complementary cases depending on the condition of Line 31.

(1) If the condition holds, then we have $|\text{Edges}[u][\overline{\alpha}_i]| = 1$ after Line 31 has been executed.
(2) Otherwise, we must have $u \in S$ and $|S| \geq 2$, hence there exists some $x \in S \setminus \{u\}$ chosen in Line 16, and all edges in Edges$[u]$ will be moved to Edges$[x]$ for some $v = u$ in Line 19. Hence $|\text{Edges}[u][\overline{\alpha}_i]| = 0$.

Note that because of Line 20, the edges in Edges$[u][\overline{\alpha}_i]$ are not moved to Edges$[x][\overline{\alpha}_i]$, hence all Edges$[u][\overline{\alpha}_i]$ (except possibly one) will no longer be present at the end of the iteration. Since $S'_j = \text{Edges}[u][\overline{\alpha}_i]$ at the beginning of the iteration, we obtain that the number of edges in Edges decreases by at least $|S'_j| - 1$.

We define a potential function $\Phi : \mathbb{N} \to \mathbb{N}$, such that $\Phi(j)$ equals the number of elements in the data structure Edges at the beginning of the j-th iteration of the main loop in Line 11. Note that (i) initially $\Phi(1) = m$, (ii) $\Phi(j) \geq 0$ for all j, and (iii) $\Phi(j+1) \leq \Phi(j)$ for all j, as new edges are never added to Edges. Let $(u, \overline{\alpha}_i)$ be an element extracted from Q at the beginning of the j-the iteration, for some $j \in J$. As shown above, at the end of the iteration we have removed at least $|S'_j| - 1$ edges from Edges, and since $|S'_j| \geq 2$, we obtain $\Phi(j+1) \leq \Phi(j) - |S'_j|/2$. Summing over all $j \in J$, we obtain

$$\begin{aligned}
\sum_{j \in J} |S'_j| &\leq 2 \cdot \sum_{j \in J} (\Phi(j) - \Phi(j+1)) && [\text{as } \Phi(j+1) \leq \Phi(j) - |S'_j|/2] \\
&= 2 \cdot \sum_{\ell=1}^{|J|} (\Phi(j_\ell) - \Phi(j_\ell + 1)) && [\text{for } j_\ell < j_{\ell+1}] \\
&\leq 2 \cdot \Phi(j_1) && [\text{as } \Phi \text{ is decreasing and thus } \Phi(j_{\ell+1}) \leq \Phi(j_\ell + 1)] \\
&\leq 2 \cdot m && [\text{as } \Phi(j_1) \leq \Phi(1) = m]
\end{aligned}$$

The desired result follows. □

Finally, we are ready to establish the complexity of BidirectedReach.

Lemma 3.5 (Complexity). BidirectedReach *requires* $O(m \cdot \alpha(n))$ *time and* $O(m)$ *space.*

A speedup for non-sparse graphs. Observe that in the case of sparse graphs $m = O(n)$, and Lemma 3.5 yields the complexity $O(n \cdot \alpha(n))$. Here we describe a modification of BidirectedReach that reduces the complexity from $O(m \cdot \alpha(n))$ to $O(m + n \cdot \alpha(n))$, and thus is faster for graphs where the edges are more than a factor $\alpha(n)$ as many as the nodes (i.e., $m = \omega(n \cdot \alpha(n))$). The key idea is that if a node u has more than k outgoing edges initially, then it has two distinct outgoing edges labeled with the same closing parenthesis $\overline{\alpha}_i \in \Sigma_k^C$, and hence the corresponding neighbors can be merged to a single DSCC in a preprocessing step. Once such a merging has taken place, u only needs to keep a single outgoing edge labeled with $\overline{\alpha}_i$ to that DSCC. This preprocessing phase requires $O(m)$ time for all nodes, after which there are only $O(n)$ edges present, by amortizing at most k edges per node of the original graph (recall that $k = O(1)$). After this preprocessing step has taken place, BidirectedReach is executed with $O(n)$ edges in its input, and by Lemma 3.5 the complexity is $O(n \cdot \alpha(n))$. We conclude the results of this section with the following theorem.

Theorem 3.6 (Worst-case complexity). *Let* $G = (V, E)$ *be a* Σ_k*-labeled bidirected graph of* n *nodes and* $m = \Omega(n)$ *edges.* BidirectedReach *correctly computes the DSCCs of* G *and requires* $O(m + n \cdot \alpha(n))$ *time and* $O(m)$ *space.*

Linear-time considerations. Note that $\alpha(n)$ is an extremely slowly growing function, and for all practical purposes $\alpha(n) \leq 4$. Indeed, the smallest n for which $\alpha(n) = 5$ far exceeds the estimated number of atoms in the observable universe. Additionally, since it is known that a Disjoint Sets data structure operates in amortized constant expected time per operation [Doyle and Rivest 1976; Yao 1985], we obtain the following corollary regarding the expected time complexity of our algorithm.

Corollary 3.7 (Average-case complexity). *For bidirected graphs, the algorithm* BidirectedReach *requires* $O(m)$ *expected time for computing DSCCs.*

3.3 An $\Omega(m + n \cdot \alpha(n))$ Lower Bound

Theorem 3.6 implies that Dyck reachability on bidirected graphs can be solved in almost-linear time. A theoretically interesting question is whether the problem can be solved in linear time in the worst case. We answer this question in the negative by proving that every algorithm for the problem requires $\Omega(m + n \cdot \alpha(n))$ time, and thereby proving that our algorithm BidirectedReach is indeed optimal wrt worst-case complexity.

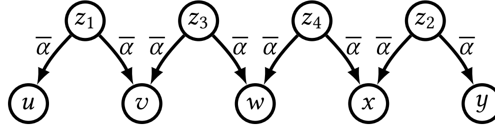
The Separated Union-Find problem. A sequence σ of Union-Find operations is called *separated* if all Find operations occur at the end of σ. Hence $\sigma = \sigma_1 \circ \sigma_2$, where σ_1 contains all Union operations of σ. We call σ_1 a *union sequence* and σ_2 a *find sequence*. The *Separated Union-Find* problem is the regular Union-Find problem over separated union-find sequences. Note that this version of the problem has an *offline* flavor, as, at the time when the algorithm is needed to produce output (i.e. when the suffix of Find operations starts) the input has been fixed (i.e., all Union operations are known). We note that the Separated Union-Find problem is different from the Static Tree Set Union problem [Gabow and Tarjan 1985], which restricts the type of allowed Union operations, and for which a linear time algorithm exists on the RAM model. The following lemma states a lower bound on the worst-case complexity of the problem.

Lemma 3.8. *The Separated Union-Find problem over a universe of size n and sequences of length m has worst-case complexity $\Omega(m \cdot \alpha(n))$.*

Proof. The proof is essentially the proof of [Tarjan 1979, Theorem 4.4], by observing that the sequences constructed there to prove the lower bound are actually separated union-find sequences. □

The union graph G^{σ_1}. Let σ_1 be a union sequence over some universe X. The *union graph* of σ_1 is a Σ_1-labeled bidirected graph $G^{\sigma_1} = (V^{\sigma_1}, E^{\sigma_1})$, defined as follows (see Figure 4 for an illustration).

(1) The node set is $V^{\sigma_1} = X \cup \{z_i\}_{1 \leq i \leq |\sigma_1|}$ where the nodes z_i do not appear in X.
(2) The edge set is $E^{\sigma_1} = \{(z_i, x_i, \overline{\alpha}), (z_i, y_i, \overline{\alpha})\}_{1 \leq i \leq |\sigma_1|}$, where $x_i, y_i \in X$ are the elements such that the i-th operation of σ_1 is Union(x_i, y_i).



σ_1 = Union(u, v), Union(x, y), Union(w, v), Union(w, x)

Fig. 4. A union sequence σ_1 and the corresponding graph G^{σ_1}.

A lower bound for Dyck reachability on bidirected graphs. We are now ready to prove our lower bound. The proof consists in showing that there exists no algorithm that solves the problem in $o(m \cdot \alpha(n))$ time. Assume towards contradiction otherwise, and let A' be an algorithm that solves the problem in time $o(m \cdot \alpha(n))$. We construct an algorithm A that solves the Separated Union-Find problem in the same time.

Let $\sigma = \sigma_1 \circ \sigma_2$ be a separated union-find sequence, where σ_1 is a union sequence and σ_2 is a find sequence. The algorithm A operates as follows. It performs no operations until the whole of σ_1 has been revealed. Then, A' constructs the union graph G^{σ_1}, and uses A' to solve the Dyck reachability problem on G^{σ_1}. Finally, every Find(x) operation encountered in σ_2 is handled by A by using the answer of A' on G^{σ_1}.

It is easy to see that A handles the input sequence σ correctly. Indeed, for any sequence of union operations Union$(x_i, y_i), \dots,$ Union(x_j, y_j) that bring two elements x and y to the same set, the edges $(z_i, x_i, \overline{\alpha}), (z_i, y_i, \overline{\alpha}), \dots, (z_j, x_j, \overline{\alpha}), (z_j, y_j, \overline{\alpha})$ must bring x and y to the same DSCC of G^{Σ_1}. Finally, the algorithm A requires $O(m)$ time for constructing G and answering all queries, plus $o(m \cdot \alpha(n))$ time for running A' on G^{Σ_1}. Hence A operates in $o(m \cdot \alpha(n))$ time, which contradicts Lemma 3.8.

We have thus arrived at the following theorem.

Theorem 3.9 (Lower-bound). *Any Dyck reachability algorithm for bidirected graphs with n nodes and $m = \Omega(n)$ edges requires $\Omega(m + n \cdot \alpha(n))$ time in the worst case.*

Theorem 3.9 together with Theorem 3.6 yield the following corollary.

Corollary 3.10 (Optimality). *The Dyck reachability algorithm* BidirectedReach *for bidirected graphs is optimal wrt to worst-case complexity.*

4 DYCK REACHABILITY ON GENERAL GRAPHS

In this section we present a hardness result regarding the Dyck reachability problem on general graphs, as well as on graphs of constant treewidth.

Complexity of Dyck reachability. Dyck reachability on general graphs is one of the most standard algorithmic formulations of various static analyses. The problem is well-known to admit a cubic-time solution, while the currently best bound is $O(n^3/\log n)$ due to [Chaudhuri 2008]. Dyck reachability is also known to be 2NPDA-hard [Heintze and McAllester 1997], which yields a conditional cubic lower bound wrt polynomial improvements. Here we investigate further the complexity of Dyck reachability. We prove that Dych reachability is Boolean Matrix Multiplication (BMM)-hard. Note that since Dyck reachability is a combinatorial graph problem, techniques such as fast-matrix multiplication (e.g. Strassen's algorithm [Strassen 1969]) are unlikely to be applicable. Hence we consider combinatorial (i.e., discrete, graph-theoretic) algorithms. The standard BMM-conjecture [Abboud and Vassilevska Williams 2014; Henzinger et al. 2015; Lee 2002; Vassilevska Williams and Williams 2010] states that there is no truly sub-cubic ($O(n^{3-\delta})$, for $\delta > 0$) combinatorial algorithm for Boolean Matrix Multiplication. Given this conjecture, various algorithmic works establish conditional hardness results. Here we show that Dyck reachability is BMM-hard on general graphs, which yields a new conditional cubic lower bound for the problem. Additionally, we show that BMM hardness also holds for Dyck reachability on graphs of constant treewidth. We establish this by showing Dyck reachability on general graphs is hard as CFL parsing.

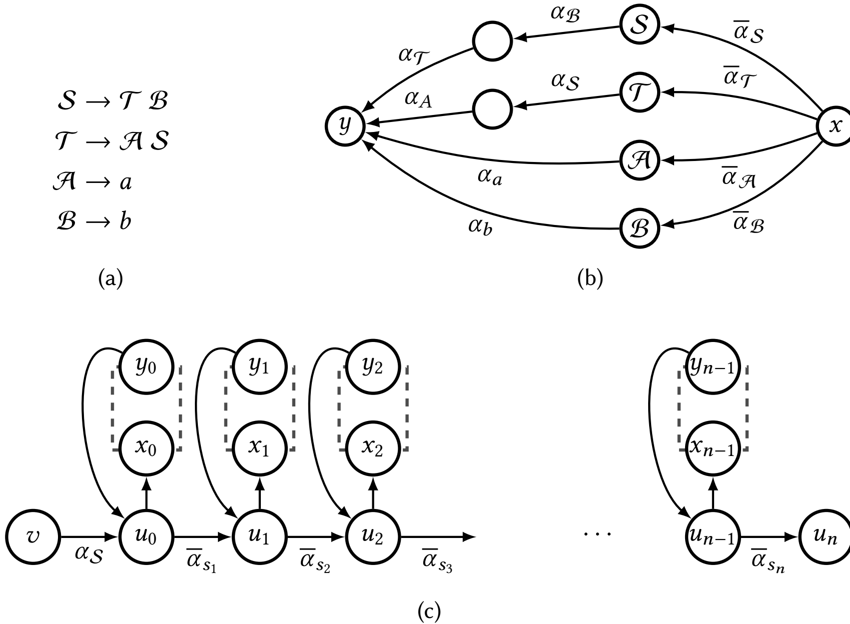


Fig. 5. **(5a)** A grammar $\mathcal{G}$ for the language $a^n b^n$, **(5b)** The gadget graph $G^{\mathcal{G}}$, **(5c)** The parse graph $G_s^{\mathcal{G}}$, given a string $s = s_1, \dots d_n$.

The gadget graph $G^{\mathcal{G}}$. Given a Context-free grammar $\mathcal{G}$ in Chomsky normal form, we construct the *gadget graph* $G^{\mathcal{G}} = (V^{\mathcal{G}}, E^{\mathcal{G}})$ as follows (see Figure 5 (5a), (5b) for an illustration).

(1) The node set $V^{\mathcal{G}}$ contains two distinguished nodes x, y, together with a node x_i for the i-th production rule p_i. Additionally, if p_i is of the form $\mathcal{A} \to \mathcal{B}\ C$, then $V^{\mathcal{G}}$ contains a node y_i.
(2) The edge set $E^{\mathcal{G}}$ contains an edge $(x, x_i, \overline{\alpha}_{\mathcal{A}})$, where $\mathcal{A}$ is the left hand side symbol of the i-th production rule p_i of $\mathcal{G}$. Additionally,
   (a) if p_i is of the form $\mathcal{A} \to a$, then $E^{\mathcal{G}}$ contains an edge (x_i, y, α_a), else
   (b) if p_i is of the form $\mathcal{A} \to \mathcal{B}\ C$, then $E^{\mathcal{G}}$ contains the edges (x_i, y_i, α_C) and $(y_i, y, \alpha_{\mathcal{B}})$.

The parse graph $G_s^{\mathcal{G}}$. Given a grammar $\mathcal{G}$ and an input string $s = s_1, \dots s_n$, we construct the *parse graph* $G_s^{\mathcal{G}} = (V_s^{\mathcal{G}}, E_s^{\mathcal{G}})$ as follows. The graph consists of two parts. The first part is a line graph that contains nodes $v, u_0, u_1, \dots u_n$, with the edges $(v, u_0, \alpha_{\mathcal{S}})$ and $(u_{i-1}, u_i, \overline{\alpha}_{s_i})$ for all $1 \leq i \leq n$. The second part consists of a n copies of the gadget graph $G^{\mathcal{G}}$, counting from 0 to $n-1$. Finally, we have a pair of edges (u_i, x_i, ϵ), (y_i, u_i, ϵ) for every $0 \leq i < n$, where x_i (resp. y_i) is the distinguished x node (resp. y node) of the i-th gadget graph. See Figure 5 (5c) for an illustration.

Lemma 4.1. *The node u_n is Dyck-reachable from node v iff s is generated by $\mathcal{G}$.*

Proof. Given a path P, we denote by $\overline{\lambda}(P)$ the substring of $\lambda(P)$ that consists of all the closing-parenthesis symbols of $\lambda(P)$. The proof follows directly from the following observation: the parse graph $G_s^{\mathcal{G}}$ contains a path $P : v \rightsquigarrow u_n$ with $\lambda(P) \in \mathcal{L}$ if and only if $\overline{\lambda}(P)$ corresponds to a pre-order traversal of a derivation tree of the string s wrt the grammar $\mathcal{G}$. □

Theorem 4.2. *If there exists a combinatorial algorithm that solves the pair Dyck reachability problem in time $\mathcal{T}(n)$, where n is the number of nodes of the input graph, then there exists a combinatorial algorithm that solves the CFL parsing problem in time $O(n + \mathcal{T}(n))$.*

Since CFL-parsing is BMM-hard, by combining Theorem 4.2 with [Lee 2002, Theorem 2] we obtain the following corollary.

Corollary 4.3 (BMM-hardness: Conditional cubic lower bound). *For any fixed $\delta > 0$, if there is a combinatorial algorithm that solves the pair Dyck reachability problem in $O(n^{3-\delta})$ time, then there is a combinatorial algorithm that solves Boolean Matrix Multiplication in $O(n^{3-\delta/3})$ time.*

Remark 3 (**BMM hardness for low-treewidth graphs**). *Note that since the size of the grammar $\mathcal{G}$ is constant, the parse graph $G_s^{\mathcal{G}}$ has constant treewidth. Hence the BMM hardness of Corollary 4.3 also holds if we restrict our attention to Dyck reachability on graphs of constant treewidth.*

5 LIBRARY/CLIENT DYCK REACHABILITY

In this section we present some new results for library/client Dyck reachability with applications to context-sensitive data-dependence analysis. One crucial step to our improvements is the fact that we consider that the underlying graphs are not arbitrary, but have special structure. We start with Section 5.1 which defines formally the graph models we deal with, and their structural properties. Afterwards, in Section 5.2, we present our algorithms.

5.1 Problem Definition

Here we present a formal definition of the input graphs that we will be considering for library/client Dyck reachability with application to context-sensitive data-dependence analysis. Each input graph G is not an arbitrary Σ_k-labeled graph, but has two important structural properties.

(1) G can be naturally partitioned to subgraphs $G_1, \dots G_\ell$, such that every G_i has only ϵ-labeled edges. Each such $G_i = (V_i, E_i)$ corresponds to a method of the input program. There are only few nodes of V_i with *incoming* edges that are non-ϵ-labeled. Similarly, there are only few nodes of V_i with *outgoing* edges that are non-ϵ-labeled. These nodes correspond to the input parameters and return statements of the i-th method of the program, which are almost always only a few.
(2) Each G_i is a graph of *low treewidth*. This is an important graph-theoretic property which, informally, means that G_i is similar to a tree (although G_i is not a tree).

We make the above structural properties formal and precise. We start with the first structural property, we captures the fact that the input graph G consists of many local graphs G_i, one for each method of the input program, and the parenthesis-labeled edges model context sensitivity.

Program-valid partitionings. Let $G = (V, E)$ be a Σ_k-labeled graph. Given some $1 \leq i \leq k$, we define the following sets.

$$V_c(\alpha_i) = \{u : \exists (u, v, \alpha_i) \in E\} \qquad V_e(\alpha_i) = \{v : \exists (u, v, \alpha_i) \in E\}$$
$$V_x(\overline{\alpha}_i) = \{u : \exists (u, v, \overline{\alpha}_i) \in E\} \qquad V_r(\overline{\alpha}_i) = \{v : \exists (u, v, \alpha_i) \in E\}$$

In words, (i) $V_c(\alpha_i)$ contains the nodes that have a α_i-labeled outgoing edge, (ii) $V_e(\alpha_i)$ contains the nodes that have a α_i-labeled incoming edge, (iii) $V_x(\overline{\alpha}_i)$ contains the nodes that have a $\overline{\alpha}_i$-labeled outgoing edge, and (iv) $V_r(\overline{\alpha}_i)$ contains the nodes that have a $\overline{\alpha}_i$-labeled incoming edge. Additionally, we define the following sets.

$$V_c = \bigcup_i V_c(\alpha_i) \quad V_e = \bigcup_i V_e(\alpha_i) \quad V_x = \bigcup_i V_x(\overline{\alpha}_i) \quad V_r = \bigcup_i V_r(\overline{\alpha}_i)$$

Consider a partitioning $\mathcal{V} = \{V_1, \dots, V_\ell\}$ of the node set V, i.e., $\bigcup_i V_i = V$ and $V_i \cap V_j = \emptyset$ for all $1 \leq i, j \leq \ell$. We say that $\mathcal{V}$ is *program-valid* if the following conditions hold: for every $1 \leq i \leq k$, there exist some $1 \leq j_1, j_2 \leq \ell$ such that (i) $V_c(\alpha_i), V_r(\overline{\alpha}_i) \subseteq V_{j_1}$, and (ii) $V_e(\alpha_i), V_x(\overline{\alpha}_i) \subseteq V_{j_2}$. Intuitively, the parenthesis-labeled edges of G correspond to method calls and returns, and thus model context sensitivity. Each parenthesis type models the calling context, and each $G[V_i]$ corresponds to a single method of the program. Since the calling context is tied to two methods (the caller and the callee), conditions (i) and (ii) must hold for the partitioning.

A program-valid partitioning $\mathcal{V} = \{V_1, \dots V_\ell\}$ is called b-bounded if there exists some $b \in \mathbb{N}$ such that for all $1 \leq j \leq \ell$ we have that $|V_e \cap V_j|, |V_x \cap V_j| \leq b$. Note that since $\mathcal{V}$ is program-valid, this condition also yields that for all $1 \leq i \leq k$ we have that $|V_c(\alpha_i)|, |V_r(\overline{\alpha}_i)| \leq b$. In this paper we consider that $b = O(1)$, i.e., b is constant wrt the size of the input graph. This is true since the sets $V_e \cap V_j$ and $V_x \cap V_j$ represent the input parameters and the return statements of the j-th method in the program. Similarly, the sets $V_c(\alpha_i)$, $V_r(\overline{\alpha}_i)$ represent the variables that are passed as input and the variables that capture the return, respectively, of the method that the i-th call site refers to. In all practical cases each of the above sets has constant size (or even size 1, for return variables).

Program-valid graphs. The graph G is called *program-valid* if there exists a constant $b \in \mathbb{N}$ such that G has b-bounded program valid partitioning. Given a such a partitioning $\mathcal{V} = \{V_1, \dots, V_\ell\}$, we call each graph $G_i = (V_i, E_i) = G[V_i]$ a *local graph*. Given a partitioning of V to the library partition V^1 and client partition V^2, $\mathcal{V}$ induces a program-valid partitioning on each of the library subgraph $G^1 = G[V^1]$ and $G^2 = G[V^2]$. See Figure 6 for an example.

We now present the second structural property of input graphs that we exploit in this work. Namely, for a program-valid input graph G with a program-valid partitioning $\mathcal{V} = \{V_1, \dots, V_\ell\}$ the local graphs $G_i = G[V_i]$ have *low treewidth*. It is known that the control-flow graphs (CFGs) of goto-free programs have small treewidth [Thorup 1998]. The local graphs G_i are not CFGs, but rather graphs defined by def-use chains. As we show in this work (see Section 6.2), the local def-use graphs of real-world benchmarks also have small treewidth. Below, we make the above notions precise.

Trees. A (rooted) tree $T = (V_T, E_T)$ is an undirected graph with a distinguished node w which is the root such that there is a unique simple path $P_u^v : u \rightsquigarrow v$ for each pair of nodes u, v. Given a tree T with root w, the *level* $\mathsf{Lv}(u)$ of a node u is the length of the simple path P_u^w from u to the root r. Every node in P_u^w is an *ancestor* of u. If v is an ancestor of u, then u is a *descendant* of v. For a pair of nodes $u, v \in V_T$, the *lowest common ancestor (LCA)* of u and v is the common ancestor of u and v with the largest level. The *parent* u of v is the unique ancestor of v in level $\mathsf{Lv}(v) - 1$, and v is a *child* of u. A *leaf* of T is a node with no children. For a node $u \in V_T$, we denote by $T(u)$ the subtree of T rooted in u (i.e., the tree consisting of all descendants of u). The *height* of T is $\max_u \mathsf{Lv}(u)$.

Tree decompositions and treewidth [Robertson and Seymour 1984]. Given a graph G, a tree-decomposition $\mathsf{Tree}(G) = (V_T, E_T)$ is a tree with the following properties.

C1: $V_T = \{B_1, \dots, B_b : \text{ for all } 1 \leq i \leq b.\ B_i \subseteq V\}$ and $\bigcup_{B_i \in V_T} B_i = V$. That is, each node of $\mathsf{Tree}(G)$ is a subset of nodes of G, and each node of G appears in some node of $\mathsf{Tree}(G)$.
C2: For all $(u, v) \in E$ there exists $B_i \in V_T$ such that $u, v \in B_i$. That is, the endpoints of each edge of G appear together in some node of $\mathsf{Tree}(G)$.
C3: For all B_i, B_j and any bag B_k that appears in the simple path $B_i \rightsquigarrow B_j$ in $\mathsf{Tree}(G)$, we have $B_i \cap B_j \subseteq B_k$. That is, every node of G is contained in a contiguous subtree of $\mathsf{Tree}(G)$.

To distinguish between the nodes of G and the nodes of $\mathsf{Tree}(G)$, the sets B_i are called *bags*. The *width* of a tree-decomposition $\mathsf{Tree}(G)$ is the size of the largest bag minus 1 and the *treewidth* of G is the width of a minimum-width tree decomposition of G. It follows from the definition that if G has constant treewidth, then $m = O(n)$. For a node $u \in V$, we say that a bag B is the *root bag* of u if B is the bag with the smallest level among all bags that contain u, i.e., $B_u = \arg\min_{B \in V_T :\, u \in B} \mathsf{Lv}(B)$. By definition, there is exactly one root bag for each node u. We often write B_u for the root bag of node u, and denote by $\mathsf{Lv}(u) = \mathsf{Lv}(B_u)$. Additionally, we denote by $B_{(u,v)}$ the bag of the largest level that is the root bag of one of u, v. The following well-known theorem states that tree decompositions of constant-treewidth graphs can be constructed efficiently.

Theorem 5.1 ([Bodlaender and Hagerup 1995]). *Given a graph $G = (V, E)$ of n nodes and treewidth $t = O(1)$, a tree decomposition $\mathsf{Tree}(G)$ of $O(n)$ bags, height $O(\log n)$ and width $O(t) = O(1)$ can be constructed in $O(n)$ time.*

The following crucial lemma states the key property of tree decompositions that we exploit in this work towards fast algorithms for Dyck reachability. Intuitively, every bag of a tree decomposition $\mathsf{Tree}(G)$ acts as a separator of the graph G.

Lemma 5.2 ([Bodlaender 1998, Lemma 3]). *Consider a graph $G = (V, E)$, a tree-decomposition $T = \mathsf{Tree}(G)$, and a bag B of T. Let $(C_i)_i$ be the components of T created by removing B from T, and let V_i be the set of nodes that appear in bags of component C_i. For every $i \neq j$, nodes $u \in V_i$, $v \in V_j$ and path $P : u \rightsquigarrow v$, we have that $P \cap B \neq \emptyset$ (i.e., all paths between u and v go through some node in B).*

Program-valid treewidth. Let $G = (V, E)$ be a Σ_k-labeled program-valid graph, and $\mathcal{V} = \{V_1, \dots, V_\ell\}$ a program-valid partitioning of G. For each $1 \leq i \leq \ell$, let $G_i = (V_i, E_i) = G[V_i]$. We define the graph $G'_i = (V_i, E'_i)$ such that

$$E'_i = E_i \bigcup_{1 \leq j \leq k} \left(V_c(\alpha_j) \cap V_i\right) \times \left(V_r(\overline{\alpha}_j) \cap V_i\right)$$

and call G'_i the *maximal* graph of G_i. In words, the graph G'_i is identical to G_i, with the exception that G'_i contains an extra edge for every pair of nodes $u, v \in V_i$ such that u has opening-parenthesis-labeled outgoing edges, and v has closing-parenthesis-labeled incoming edges. We define the treewidth of $\mathcal{V}$ to be the smallest integer t such that the treewidth of each G'_i is at most t. We define the width of the pair $(G, \mathcal{V})$ as the treewidth of $\mathcal{V}$, and the *program-valid treewidth* of G to be the smallest treewidth among its program-valid partitionings.

The Library/Client Dyck reachability problem on program-valid graphs. Here we define the algorithmic problem that we solve in this section. Let $G = (V, E)$ be a Σ_k-labeled, program-valid graph and $\mathcal{V}$ a program-valid partitioning of G that has constant treewidth (k need not be constant). The set $\mathcal{V}$ is further partitioned into two sets, $\mathcal{V}^1$ and $\mathcal{V}^2$ that correspond to the *library* and *client* partitions, respectively. We let $V^1 = \bigcup_{V_i \in \mathcal{V}^1} V_i$ and $V^2 = \bigcup_{V_i \in \mathcal{V}^2} V_i$, and define the *library graph* $G^1 = (V^1, E^1) = G[V^1]$ and the *client graph* $G^2 = (V^2, E^2) = G[V^1]$.

The task is to answer Dyck reachability queries on G, where the queries are either (i) single source queries from some $u \in V^2$, or (ii) pair queries for some pair $u, v \in V^2$. The computation takes place in two phases. In the *preprocessing phase*, only the library graph G^1 is revealed, and we are allowed to some preprocessing to compute reachability summaries. In the *query phase*, the whole graph G is revealed, and our task is to handle queries fast, by utilizing the preprocessing done on G^1.

5.2 Library/Client Dyck Reachability on Program-valid Graphs

We are now ready to present our method for computing library summaries on program-valid graphs in order to speed up the client-side Dyck reachability. The approach is very similar to the work of [Chatterjee et al. 2015b] for data-flow analysis of recursive state machines.

Outline of our approach. Our approach consists of the following conceptual steps. We let the input graph $G = (V, E)$ be any program-valid graph of constant treewidth, with a partitioning of V into the library component V^1 and the client component V^2. Since G is program-valid, it has a constant-treewidth, program-valid partitioning $\mathcal{V}$, and we consider $\mathcal{V}^1$ to be the restriction of $\mathcal{V}$ to the set V^1. Hence we have $\mathcal{V}^1 = \{V_1, \dots V_\ell\}$ be a program-valid partitioning of $G[V^1]$, which also has constant treewidth. Our approach consists of the following steps.

(1) We construct a local graph $G_i = (V_i, E_i)$ and the corresponding maximal local graph $G'_i = (V_i, E'_i)$ for each $V_i \in \mathcal{V}$. Recall that G'_i is a conventional graph, since, by definition, E'_i contains only ϵ-labeled edges. Since $\mathcal{V}$ has constant treewidth, each graph G'_i has constant treewidth, and we construct a tree decomposition $\mathsf{Tree}(G'_i)$.
(2) We exploit the constant-treewidth property of each G'_i to build a data structure $\mathcal{D}$ which supports the following two operations: (i) Querying whether a node v is reachable from a node u in G'_i, and (ii) Updating G_i by inserting a new edge (x, y). Moreover, each such operation is fast, i.e., it is performed in $O(\log n_i)$ time.
(3) Recall that V^1, V^2 are the library and client partitions of G, respectively. In the preprocessing phase, we use the data structure $\mathcal{D}$ to preprocess $G[V^1]$ so that any pair of library nodes that is Dyck-reachable in $G[V^1]$ is discovered and can be queried fast. Hence this library-side reachability information serves as the summary on the library side.

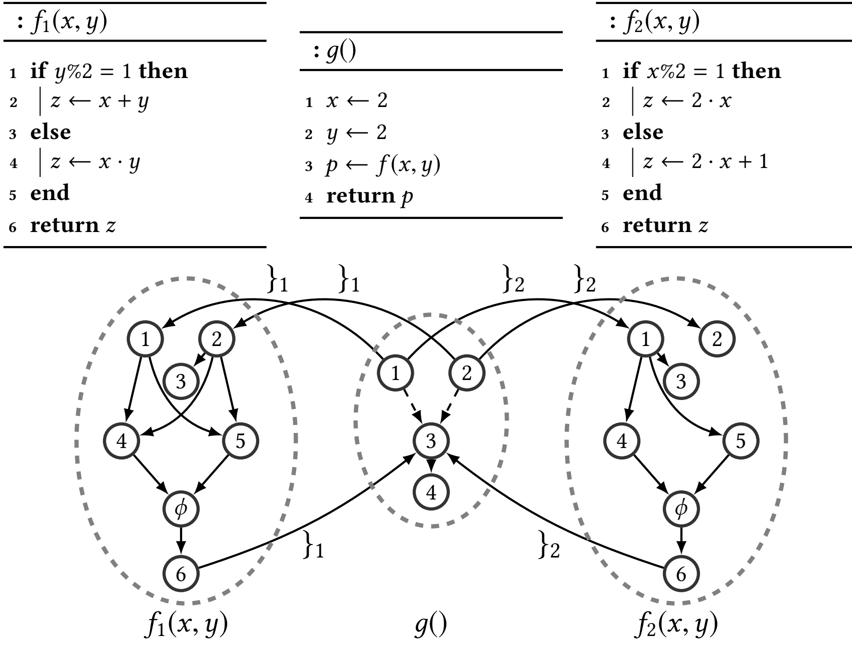


Fig. 6. Example of a library/client program and the corresponding program-valid data-dependence graph. The library consists of method $g()$ which has a callback function $f(x, y)$. The client implements $f(x, y)$ either as $f_1(x, y)$ or $f_2(x, y)$. The parenthesis-labeled edge model context-sensitive dependencies on parameter passing and return. Depending on the implementation of f, there is a data dependence of the variable p on y.

(4) In the query phase, we use $\mathcal{D}$ to process the whole graph G, using the summaries computed in the preprocessing phase.

Step 1. Construction of the local graphs G_i and the tree decompositions. The local graphs G_i are extracted from $G[V^1]$ by means of its program-valid partitioning $\mathcal{V}^1 = \{V_1, \dots V_\ell\}$. We consider this partitioning as part of the input, since every local graph G_i in reality corresponds to a unique method of the input program represented by G. Let $n_i = |V_i|$. The maximal local graphs $G'_i = (V_i, E'_i)$ are constructed as defined in Section 5.1. Each tree decomposition $\mathrm{Tree}(G'_i)$ is constructed in $O(n_i)$ time using Theorem 5.1. Observe that since $E_i \subseteq E'_i$ (i.e., G_i is a subgraph of its maximal counterpart G'_i), $\mathrm{Tree}(G'_i)$ is also a tree decomposition of G_i. We define $\mathrm{Tree}(G_i) = \mathrm{Tree}(G'_i)$ for all $1 \leq i \leq \ell$.

Step 2. Description of the data structure $\mathcal{D}$. Here we describe the data structure $\mathcal{D}$, which is built for a conventional graph $G_i = (V_i, E_i)$ (i.e., E_i has only ϵ-labeled edges) and its tree decomposition $\mathrm{Tree}(G_i)$. The purpose of $\mathcal{D}$ is to handle reachability queries on G_i. The data structure supports three operations, given in Algorithm 2, Algorithm 3 and Algorithm 4.

(1) The $\mathcal{D}$.Build (Algorithm 2) operation builds the data structure for G_i.
(2) The $\mathcal{D}$.Update (Algorithm 3) updates the graph G_i with a new edge (x, y), provided that there exists a bag B such that $x, y \in B$.
(3) The $\mathcal{D}$.Query (Algorithm 4) takes as input a pair of nodes x, y and returns True iff y is reachable from x in G_i, considering all the update operations performed so far.

Algorithm 2: $\mathcal{D}$.Build

```
  Input: A tree-decomposition Tree(G_i)
1 Traverse Tree(G_i) bottom up
2 foreach encountered bag B do
3   Construct the graph G(B) = (B, R(B))
4   Compute the transitive closure G*(B)
5   foreach (u, v) ∈ B do
6     if u ⇝ v in G*(B) then
7       Insert u, v in R
8   end
9 end
```

Algorithm 3: $\mathcal{D}$.Update

```
  Input: A new edge (x, y)
1 Traverse Tree(G) from B_(u,v) to the root
2 foreach encountered bag B do
3   Construct the graph G(B) = (B, R(B))
4   Compute the transitive closure G*(B)
5   foreach u, v ∈ B do
6     if u ⇝ v in G*(B) then
7       Insert (u, v) in R
8   end
9 end
```

Algorithm 4: $\mathcal{D}$.Query

```
   Input: A pair of nodes x, y
 1 Let X ← {x}, Y ← {y}
 2 Traverse Tree(G) from B_x to the root
 3 foreach encountered bag B do
 4   foreach u, v ∈ B do
 5     if u ∈ X and (u, v) ∈ R then
 6       Add v to X
 7   end
 8 end
 9 Traverse Tree(G) from B_y to the root
10 foreach encountered bag B do
11   foreach u, v ∈ B do
12     if v ∈ Y and (u, v) ∈ R then
13       Add u to Y
14   end
15 end
16 return True iff X ∩ Y ≠ ∅
```

Algorithm 5: Process

```
   Input: Method graphs (G_i = (V_i, E_i))_{1≤i≤ℓ}
 1 foreach 1 ≤ i ≤ ℓ do
 2   Construct Tree(G_j)
 3   Run D.Build on Tree(G_i)
 4 end
 5 Pool ← {G_1, ... G_ℓ}
 6 while Pool ≠ ∅ do
 7   Extract G_j from Pool
 8   foreach u ∈ V_j ∩ V_e, v ∈ V_j ∩ V_x do
 9     if D.Query(u, v) then
10       foreach x, y : (x, u, α_i), (v, y, α_i) ∈ E do
11         Let G_r = (V_r, E_r) be the graph s.t.
            x, y ∈ V_r
12         if not D.Query(x, y) then
13           Run D.Update on Tree(G_r) on (x, y)
14           Insert G_r in Pool
15       end
16   end
17 end
```

The reachability set R**.** The data structure $\mathcal{D}$ is built by storing a reachability set R between pairs of nodes. The set R has the crucial property that it stores information only between pairs of nodes that appear in some bag of Tree(G_i) together. That is, $R =\subseteq \bigcup_B B \times B$. Given a bag B, we denote by $R(B)$ the restriction of R to the nodes of B. The reachability set is stored as a collection of $2\sum_i \cdot n_i$ sets $R^F(u)$ and $R^B(u)$, one for every node $u \in V_i$. In turn, the set $R^F(u)$ (resp. $R^B(u)$) will store the nodes in B_u (recall that B_u is the root bag of node u) for which it has been discovered that can be reached from u (resp., that can reach u). It follows directly from the definition of tree decompositions that if $(u, v) \in E_i$ is an edge of G_i then $u \in B_v$ or $v \in B_u$. Given a bag B and nodes $u, v \in B$, querying whether $(u, v) \in R$ reduces to testing whether $v \in R^F(u)$ or $u \in R^B(v)$. Similarly, inserting (u, v) to R reduces to inserting either v to $R^F(u)$ (if $v \in B_u$), or u to $R^B(v)$ (if $u \in B_v$).

Remark 4. *The map R requires $O(n)$ space. Since each G_i is a constant-treewidth graph, every insert and query operation on R requires $O(1)$ time.*

Correctness and complexity of $\mathcal{D}$. Here we establish the correctness and complexity of each operation of $\mathcal{D}$.

It is rather straightforward to see that for every pair of nodes $(u, v) \in R$, we have that v is reachable from u. The following lemma states a kind of weak completeness: if v is reachable from u via a path of specific type, then $(u, v) \in R$. Although this is different from strong completeness, which would require that $(u, v) \in R$ whenever v is reachable from u, it is sufficient for ensuring completeness of the $\mathcal{D}$.Query algorithm.

Left-right-contained paths. We introduce the notion of left-right contained paths, which is crucial for stating the correctness of the data structure $\mathcal{D}$. Given a bag B of Tree(G_i), we say that a path $P : x \rightsquigarrow y$ is *left-contained* in B if for every node $w \in P$, if $w \neq x$, we have that $B_w \in T(B)$. Similarly, P is *right-contained* in B if for every node $w \in P$, if $w \neq y$, we have that $B_w \in T(B)$. Finally, P is *left-right-contained* in B if it is both left-contained and right-contained in B.

Lemma 5.3. *The data structure $\mathcal{D}$ maintains the following invariant. For every bag B and pair of nodes $u, v \in B$, if there is a $P_u^v : u \rightsquigarrow v$ which is left-right contained in B, then after $\mathcal{D}$.*Build *has processed B, we have $(u, v) \in R$.*

It is rather straightforward that at the end of $\mathcal{D}$.Query, for every node $w \in X$ (resp. $w \in Y$) we have that w is reachable from x (resp. y is reachable from w). This guarantees that if $\mathcal{D}$.Query returns True, then y is indeed reachable from x, via some node $w \in X \cap Y$ (recall that the intersection is not empty, due to Line 16). The following two lemmas state completeness, namely that if y is reachable from x, then $\mathcal{D}$.Query will return True, and the complexity of $\mathcal{D}$ operations.

Lemma 5.4. *On input x, y, if y is reachable from x, then $\mathcal{D}$.*Query *returns* True.

Lemma 5.5. $\mathcal{D}$.Build *requires $O(n_i)$ time. Every call to $\mathcal{D}$.*Update *and $\mathcal{D}$.*Query *requires $O(\log n_i)$ time.*

Step 3. Preprocessing the library graph $G[V^1]$**.** Given the library subgraph $G[V^1]$ and one copy of the data structure $\mathcal{D}$ for each local graph G_i of $G[V^1]$, the preprocessing of the library graph is achieved via the algorithm Process, which is presented in Algorithm 5. In high level, Process initially builds the data structure $\mathcal{D}$ for each local graph G_i using $\mathcal{D}$.Build. Afterwards, it iteratively uses $\mathcal{D}$.Query to test whether there exists a local graph G_j and two nodes $u \in V_j \cap V_e$, $v \in V_j \cap V_x$ such that v is reachable from u in G_j. If so, the algorithm iterates over all nodes x, y such that $(x, u, \alpha_i) \in E$ and $(v, y, \overline{\alpha}_i) \in E$, and uses a $\mathcal{D}$.Query operation to test whether y is reachable from x in their respective local graph G_r. If not, then Process uses a $\mathcal{D}$.Update operation to insert the edge x, y in G_r. Since this new edge might affect the reachability relations among other nodes in V_r, the graph G_r is inserted in Pool for further processing. See Algorithm 5 for a formal description. The following two lemmas state the correctness and complexity of Process.

Lemma 5.6. *At the end of* Process*, for every graph $G_i = (V_i, E_i)$ and pair of nodes $u, v \in V_i$, we have that v is reachable from u in $G[V^1]$ iff $\mathcal{D}$.*Query *returns* True.

Lemma 5.7. *Let $n = \sum_i n_i$, and k_1 be the number of labels appearing in $E \subseteq V^1 \times V^1 \times \Sigma_k$ (i.e., k_1 is the number of call sites in $G[V^1]$).* Process *requires $O(n + k_1 \cdot \log n)$ time.*

Step 4. Library/Client analysis. We are ready to describe the library summarization for Library/Client Dyck reachability. Let $G = (V, E)$ be the program-valid graph representing library and client code, and V^1, V^2 a partitioning of V to library and client nodes.

(1) In the preprocessing phase, the algorithm Process is used to preprocess $G[V^1]$. Note that since G is a program valid graph, so is $G[V^1]$, hence Process can execute on $G[V^1]$. The summaries created are in form of $\mathcal{D}$.Update operations performed on edges (x, y).
(2) In the querying phase, the set V^2 is revealed, and thus the whole of G. Hence now Process processes G, without using $\mathcal{D}$.Build on the graphs G_i that correspond to library methods, as they have already been processed in step 1. Note that the graphs G_i that correspond to library methods are used for querying and updating.

It follows immediately from Lemma 5.6 that at the end of the second step, for every local graph $G_i = (V_i, E_i)$ of the client graph, for every pair of nodes $u, v \in V_i$, v is Dyck-reachable from u in the program-valid graph G if and only if $\mathcal{D}$.Query returns True on input u, v.

Now we turn our attention to complexity. Let $n_1 = |V^1|$ and $n_2 = |V^2|$. By Lemma 5.7, the time spent for the first step is, $O(n_1 + k_1 \cdot \log n_1)$, and the time spent for the second step is $O(n_2 + k_1 \cdot \log n_1 + k_2 \cdot \log n_2)$.

Constant-time queries. Recall that our task is to support $O(1)$-time queries about the Dyck reachability of pairs of nodes on the client subgraph $G[V^2]$. As Lemma 5.6 shows, after Process has finished, each such query costs $O(\log n_2)$ time. We use existing results for reachability queries on constant-treewidth graphs [Chatterjee et al. 2016b, Theorem 6] which allow us to reduce the query time to $O(1)$, while spending $O(n_2)$ time in total to process all the graphs.

THEOREM 5.8. *Consider a Σ_k-labeled program-valid graph $G = (V, E)$ of constant program-valid treewidth, and the library and client subgraphs $G^1 = (V^1, E^1)$ and $G^2 = (V^2, E^2)$. For $i \in \{1, 2\}$ let $n_i = |V^i|$ be the number of nodes, and k_i be the number of call sites in each graph G^i, with $k_1 + k_2 = k$. The algorithm* DynamicDyck *requires*

(1) $O(n_1 + k_1 \cdot \log n_1)$ time and $O(n_1)$ space in the preprocessing phase, and
(2) $O(n_2 + k_1 \cdot \log n_1 + k_2 \cdot \log n_2)$ time and $O(n_1 + n_2)$ space in the query phase,

after which pair reachability queries are handled in $O(1)$ time.

6 EXPERIMENTAL RESULTS

In this section we report on experimental results obtained for the problems of (i) alias analysis via points-to analysis on SPGs, and (ii) library/client data-dependence analysis.

6.1 Alias Analysis

Implementation. We have implemented our algorithm BidirectedReach in C++ and evaluated its performance in performing Dyck reachability on bidirected graphs. The algorithm is implemented as presented in Section 3, together with the preprocessing step that handles the ϵ-labeled edges. Besides common coding practices we have performed no engineering optimizations. We have also implemented [Zhang et al. 2013, Algorithm 2], including the Fast-Doubly-Linked-List (FDLL), which was previously shown to be very efficient in practice.

Experimental setup. In our experimental setup we used the DaCapo-2006-10-MR2 suit [Blackburn 2006], which contains 11 real-world benchmarks. We used the tool reported in [Yan et al. 2011b] to extract the Symbolic Points-to Graphs (SPGs), which in turn uses Soot [Vallée-Rai et al. 1999]

to process input Java programs. Our approach is similar to the one reported in [Xu et al. 2009; Yan et al. 2011b; Zhang et al. 2013]. The outputs of the two compared methods were verified to ensure validity of the results. No compiler optimizations were used. All experiments were run on a Windows-based laptop with an Intel Core i7-5500U 2.40 GHz CPU and 16 GB of memory, without any compiler optimizations.

SPGs and points-to analysis. For the sake of completeness, we outline the construction of SPGs and the reachability relation they define. A more detailed exposition can be found in [Xu et al. 2009; Yan et al. 2011b; Zhang et al. 2013]. An SPG is a graph, the node set of which consists of the following three subsets: (i) variable nodes $\mathcal{V}$ that represent variables in the program, (ii) allocation nodes O that represent objects constructed with the *new* expression, and (iii) symbolic nodes $\mathcal{S}$ that represent abstract heap objects. Similarly, there are three types of edges, as follows, where $\text{Fields} = \{f_i\}_{1\leq i\leq k}$ denotes the set of all fields of composite data types.

(1) Edges of the form $\mathcal{V} \times O \times \{\epsilon\}$ represent the objects that variables point to.
(2) Edges of the form $\mathcal{V} \times \mathcal{S} \times \{\epsilon\}$ represent the abstract heap objects that variables point to.
(3) Edges of the form $(O \cup \mathcal{S}) \times (O \cup \mathcal{S}) \times \text{Fields}$ represent the fields of objects that other objects point to.

We note that since we focus on context-insensitive points-to analysis, we have not included edges that model calling context in the definition of the SPG. Additionally, only the forward edges labeled with f_i are defined explicitly, and the backwards edges labeled with $\overline{f}_i$ are implicit, since the SPG is treated as bidirected. Memory aliasing between two objects $o_1, o_2 \in \mathcal{S} \cup O$ occurs when there is a path $o_1 \rightsquigarrow o_2$, such that every opening field access f_i is properly matched by a closing field access $\overline{f}_i$. Hence the Dyck grammar is given by $\mathcal{S} \to \mathcal{S}\ \mathcal{S} \mid f_i\ \mathcal{S}\ \overline{f}_i \mid \epsilon$. This allows to infer the objects that variable nodes can point to via composite paths that go through many field assignments. See Figure 7 for a minimal example.

Fig. 7. A minimal program and its (bidirected) SPG. Circles and squares represent variable nodes and object nodes, respectively. Only forward edges are shown.

Analysis of results. The running times of the compared algorithms are shown in Table 3 We can see that the algorithm proposed in this work is much faster than the existing algorithm of [Zhang et al. 2013] in all benchmarks. The highest speedup is achieved in benchmark *luindex*, where our algorithm is 13x times faster. We also see that all times are overall small.

6.2 Library/Client Data-dependence Analysis

Implementation. We have implemented our algorithm DynamicDyck in Java and evaluated its performance in performing Library/Client data-dependency analysis via Dyck reachability. Our algorithm is built on top of Wala [Wal 2003], and is implemented as presented in Section 5. Besides common coding practices we have performed no engineering optimizations. We used the LibTW library [van Dijk et al. 2006] for computing the tree decompositions of the input graphs, under the *greedy degree* heuristic.

Experimental setup. We have used the tool of [Tang et al. 2015] for obtaining the data-dependence graphs of Java programs. In turn, that tool uses Wala [Wal 2003] to build the graphs, and specifies

Table 3. Comparison between our algorithm and the existing from [Zhang et al. 2013]. The first three columns contain the number of fields (Dyck parenthesis), nodes and edges in the SPG of each benchmark. The last two columns contain the running times, in seconds.

Benchmark	Fields	Nodes	Edges	Our Algorithm	Existing Algorithm
antlr	172	13708	23547	0.428783	1.34152
bloat	316	43671	103361	17.7888	34.6012
chart	711	53500	91869	8.99378	34.9101
eclipse	439	34594	52011	3.62835	12.7697
fop	1064	101507	178338	42.5447	148.034
hsqldb	43	3048	4134	0.012899	0.073863
jython	338	56336	167040	40.239	55.3311
luindex	167	9931	14671	0.068013	0.636346
lusearch	200	12837	21010	0.163561	1.12788
pmd	357	31648	58025	2.21662	8.92306
xalan	41	2342	2979	0.006626	0.045144

the parts of the graph that correspond to library and client code. Java programs are suitable for Library/Code analysis, since the ubiquitous presence of callback functions makes the library and client code interdependent, so that the two sides cannot be analyzed in isolation. Our algorithm was compared with the TAL reachability and CFL reachability approach, as already implemented in [Tang et al. 2015]. The comparison was performed in terms of running time and memory usage, first for the analysis of library code to produce the summaries, and then for the analysis of the library summaries with the client code. The outputs of all three methods were compared to ensure validity of the results. The measurements for our algorithm include the time and memory used for computing the tree decompositions. All experiments were run on a Windows-based laptop with an Intel Core i7-5500U 2.40 GHz CPU and 16 GB of memory, without any compiler optimizations.

Benchmarks. Our benchmark suit is similar to that of [Tang et al. 2015], consisting of 12 Java programs from SPECjvm2008 [SPE 2008], together with 4 randomly chosen programs from GitHub [Git 2008]. We note that as reported in [Tang et al. 2015], they are unable to handle the benchmark *serial* from SPECjvm2008, due to out-of-memory issues when preprocessing the library (recall that the space bound for TAL reachability is $O(n^4)$). In contrast, our algorithm handles *serial* easily, and is thus included in the experiments.

Analysis of results. Our experimental comparison is depicted in Table 4 for running time and Table 5 for memory usage We briefly discuss our findings.

Treewidth. First, we comment on the treewidth of the obtained data-dependence graphs, which is reported on Table 4 and Table 5. Recall that our interest is not on the treewidth of the whole data-dependence graph, but on the treewidth of its program-valid partitioning, which yields a subgraph for each method of the input program. In each line of the tables we report the *maximum* treewidth of each benchmark, i.e. the maximum treewidth over the subgraphs of its program-valid partitioning. We see that the treewidth is typically very small (i.e., in most cases it is 5 or 6) in both library and client code. One exception is the client of mpegaudio, which has large treewidth. Observe that even this corner case of large treewidth was easily handled by our algorithm.

Time. Table 4 shows the time spent by each algorithm for analyzing library and client code separately. We first focus on total time, taken as the sum of the times spent by each algorithm in the library and client graph of each benchmark. We see that in every benchmark, our algorithm significantly outperforms both TAL and CFL reachability, reaching a 10x-speedup compared to TAL (in

mpegaudio), and 5x-speedup compared to CFL reachability (in *helloworld*). Note that the benchmark *serial* is missing from the figure, as TAL reachability runs out of memory. The benchmark is found on Table 4, where our algorithm achieves a 630x-speedup compared to CFL reachability.

We now turn our attention to the trade-off between library preprocessing and client querying times. Here, the advantage of TAL over CFL reachability is present for handling client code. However, even for client code our algorithm is faster than TAL in all cases except one, and reaches even a 30x-speedup over TAL (in *sunflow*). Finally, observe that in all cases, the total running time of our algorithm on library and client code combined is much smaller than each of the other methods on library code alone.

Memory. Table 5 compares the total memory used for analyzing library and client code. We see that our algorithm significantly outperforms both TAL and CFL reachability in all benchmarks. Again, TAL uses more memory that CFL in the preprocessing of libraries, but less memory when analyzing client code. However, our algorithm uses even less memory than TAL in all benchmarks. The best performance gain is achieved in *serial*, where TAL runs out of memory after having consumed more than 12 GB. For the same benchmark, CFL reachability uses more than 4.3 GB. In contrast, our algorithm uses only 130 MB, thus achieving a 33x-improvement over CFL, and at least a 90x-improvement over TAL. We stress that for memory usage, these are tremendous gains. Finally, observe that for each benchmark, the maximum memory used by our algorithm for analyzing library and client code is smaller than the minimum memory used between library and client, by each of the other two methods.

Improvement independent of callbacks. We note that in contrast to TAL reachability, the improvements of our algorithm are not restricted to the presence of callbacks. Indeed, the algorithms introduced here significantly outperform the CFL approach even in the presence of no callbacks. This is evident from Table 4, which shows that our algorithm processes the library graphs much faster than both CFL and TAL reachability.

7 CONCLUSION

In this work we consider Dyck reachability problems for alias and data-dependence analysis. For alias analysis, bidirected graphs are natural, for which we present improved upper bounds, and present matching lower bounds to show our algorithm is optimal. For data-dependence analysis, we exploit constant treewidth property to present almost-linear time algorithm. We also show that for general graphs Dyck reachability bounds cannot be improved without achieving a major breakthrough.

ACKNOWLEDGMENTS

The research was partly supported by Austrian Science Fund (FWF) Grant No P23499- N23, FWF NFN Grant No S11407-N23 (RiSE/SHiNE), and ERC Start grant (279307: Graph Games).

REFERENCES

2003. T. J. Watson Libraries for Analysis (WALA). https://github.com.

2008. GitHub Home. https://github.com.

2008. SPECjvm2008 Benchmark Suit. http://www.spec.org/jvm2008/.

Amir Abboud and Virginia Vassilevska Williams. 2014. Popular Conjectures Imply Strong Lower Bounds for Dynamic Problems. In *FOCS*. 434–443.

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

Robert S. Arnold. 1996. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA.

Table 4. Running time of our algorithm vs the TAL and CFL approach for data-dependence analysis with library summarization. Times are in milliseconds. MEM-OUT indicates that the algorithm run out of memory. The number of nodes and treewidth reflects the average and maximum case, respectively, among all methods in each benchmark.

	Nodes		TW		**Our Algorithm**		**TAL**		**CFL**	
Benchmark	Lib.	Cl.	Lib.	Cl.	Lib.	Cl.	Lib.	Cl.	Li.	Cl.
helloworld	16003	296	5	3	229	5	1044	31	855	578
check	16604	3347	5	4	228	54	1062	72	821	620
compiler	16190	536	5	3	248	11	995	57	876	572
sample	3941	28	4	1	86	1	258	14	368	113
crypto	20094	3216	5	5	273	66	1451	196	961	776
derby	23407	1106	6	3	389	22	1301	83	1003	1100
mpegaudio	28917	27576	5	24	204	177	5358	253	1864	1586
xml	71474	2312	5	3	489	115	5492	100	1891	2570
mushroom	3858	7	4	1	86	1	230	14	349	124
btree	6710	1103	4	4	144	34	583	111	571	197
startup	19312	621	5	3	279	17	1651	110	1087	946
sunflow	15615	85	5	2	217	1	1073	31	811	549
compress	16157	1483	5	3	240	23	1119	112	783	999
parser	7856	112	4	1	172	3	443	21	572	241
scimark	16270	2027	5	5	220	34	1004	70	805	595
serial	69999	468	8	3	440	9	MEM-OUT	MEM-OUT	117147	165958

Table 5. Memory usage of our algorithm vs the TAL and CFL approach for data-dependence analysis with library summarization. Memory usage is in Megabytes. MEM-OUT indicates that the algorithm run out of memory. The number of nodes and treewidth reflects the average and maximum case, respectively, among all methods in each benchmark.

	Nodes		TW		**Our Algorithm**		**TAL**		**CFL**	
Benchmark	Lib.	Cl.	Lib.	Cl.	Lib.	Cl.	Lib.	Cl.	Li.	Cl.
helloworld	16003	296	5	3	31	27	321	44	104	126
check	16604	3347	5	4	34	31	336	89	132	184
compiler	16190	536	5	3	31	28	329	44	108	137
sample	3941	28	4	1	19	16	232	59	59	64
crypto	20094	3216	5	5	45	45	261	61	127	188
derby	23407	1106	6	3	46	41	600	88	204	265
mpegaudio	28917	27576	5	24	96	96	516	219	262	397
xml	71474	2312	5	3	108	108	463	153	373	480
mushroom	3858	7	4	1	19	16	230	59	58	58
btree	6710	1103	4	4	22	19	308	65	72	89
startup	19312	621	5	3	66	66	345	92	178	230
sunflow	15615	85	5	2	30	27	315	43	102	124
compress	16157	1483	5	3	32	29	338	50	105	131
parser	7856	112	4	1	22	19	320	64	73	83
scimark	16270	2027	5	5	32	29	134	49	106	140
serial	69999	468	8	3	130	130	MEM-OUT	MEM-OUT	3964	4314

Lech Banachowski. 1980. A complement to Tarjan's result about the lower bound on the complexity of the set union problem. *Inform. Process. Lett.* 11, 2 (1980), 59 – 65.

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

Umberto Bertele and Francesco Brioschi. 1972. *Nonserial Dynamic Programming.* Academic Press, Inc., Orlando, FL, USA.

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

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

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

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

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

Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. 2017. *Optimal Dyck Reachability for Data-dependence and Alias Analysis*. Technical Report. IST Austria. https://repository.ist.ac.at/id/eprint/870

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

Krishnendu Chatterjee, Rasmus Ibsen-Jensen, Prateesh Goyal, and Andreas Pavlogiannis. 2015b. Faster Algorithms for Algebraic Path Properties in Recursive State Machines with Constant Treewidth. In *POPL*.

Krishnendu Chatterjee, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. 2015a. Faster Algorithms for Quantitative Verification in Constant Treewidth Graphs. In *CAV*.

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

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

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

Jong-Deok Choi, Michael Burke, and Paul Carini. 1993. Efficient Flow-sensitive Interprocedural Computation of Pointer-induced Aliases and Side Effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '93)*. ACM, 232–245.

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

Jon Doyle and Ronald L. Rivest. 1976. Linear expected time of a simple union-find algorithm. *Inform. Process. Lett.* 5, 5 (1976), 146 – 148.

Harold N. Gabow and Robert Endre Tarjan. 1985. A linear-time algorithm for a special case of disjoint set union. *J. Comput. System Sci.* 30, 2 (1985), 209 – 221.

Zvi Galil and Giuseppe F. Italiano. 1991. Data Structures and Algorithms for Disjoint Set Union Problems. *ACM Comput. Surv.* 23, 3 (1991), 319–344.

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

Nevin Heintze and David McAllester. 1997. On the Cubic Bottleneck in Subtyping and Flow Analysis. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science (LICS '97)*. IEEE Computer Society, Washington, DC, USA, 342–. http://dl.acm.org/citation.cfm?id=788019.788876

Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. 2015. Unifying and Strengthening Hardness for Dynamic Problems via the Online Matrix-Vector Multiplication Conjecture. In *STOC*. 21–30.

Michael Hind. 2001. Pointer Analysis: Haven'T We Solved This Problem Yet?. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '01)*. ACM, 54–61.

Susan Horwitz. 1997. Precise Flow-insensitive May-alias Analysis is NP-hard. *ACM Trans. Program. Lang. Syst.* 19, 1 (1997), 1–6.

D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. 1981. Dependence Graphs and Compiler Optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 207–218.

William Landi and Barbara G. Ryder. 1992. A Safe Approximate Algorithm for Interprocedural Aliasing. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI '92)*. ACM, 235–248.

François Le Gall. 2014. Powers of Tensors and Fast Matrix Multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation (ISSAC)*. 296–303.

Lillian Lee. 2002. Fast Context-free Grammar Parsing Requires Fast Boolean Matrix Multiplication. *J. ACM* 49, 1 (2002), 1–15.

Ondřej Lhoták and Laurie Hendren. 2006. Context-Sensitive Points-to Analysis: Is It Worth It?. In *Proceedings of the 15th International Conference on Compiler Construction (CC)*. 47–64.

Vijay Krishna Palepu, Guoqing Xu, and James A. Jones. 2017. Dynamic Dependence Summaries. *ACM Trans. Softw. Eng. Methodol.* 25, 4 (2017), 30:1–30:41.

G. Ramalingam. 1994. The Undecidability of Aliasing. *ACM Trans. Program. Lang. Syst.* 16, 5 (1994), 1467–1471.

Jakob Rehof and Manuel Fähndrich. 2001. Type-base Flow Analysis: From Polymorphic Subtyping to CFL-reachability. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 54–66.

Thomas Reps. 1995. Shape Analysis As a Generalized Path Problem. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '95)*. ACM, 1–11.

Thomas Reps. 1997. Program Analysis via Graph Reachability. In *Proceedings of the 1997 International Symposium on Logic Programming (ILPS)*. 5–19.

Thomas Reps. 2000. Undecidability of Context-sensitive Data-dependence Analysis. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 162–186.

Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. ACM, 49–61.

Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. 1994. Speeding Up Slicing. *SIGSOFT Softw. Eng. Notes* 19, 5 (1994), 11–20.

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

Lei Shang, Xinwei Xie, and Jingling Xue. 2012. On-demand Dynamic Summary-based Points-to Analysis. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. ACM, 264–274.

Manu Sridharan and Rastislav Bodík. 2006. Refinement-based Context-sensitive Points-to Analysis for Java. *SIGPLAN Not.* 41, 6 (2006), 387–400.

Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. 2013. Aliasing in Object-Oriented Programming. Chapter Alias Analysis for Object-oriented Programs, 196–232.

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven Points-to Analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, 59–76.

Volker Strassen. 1969. Gaussian Elimination is Not Optimal. *Numer. Math.* 13, 4 (1969), 354–356.

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 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 83–95.

Robert Endre Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM* 22, 2 (April 1975), 215–225. https://doi.org/10.1145/321879.321884

Robert Endre Tarjan. 1979. A class of algorithms which require nonlinear time to maintain disjoint sets. *J. Comput. System Sci.* 18, 2 (1979), 110 – 127.

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

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

Thomas van Dijk, Jan-Pieter van den Heuvel, and Wouter Slob. 2006. *Computing treewidth with LibTW*. Technical Report. University of Utrecht.

Virginia Vassilevska Williams and Ryan Williams. 2010. Subcubic Equivalences between Path, Matrix and Triangle Problems. In *FOCS*. 645–654.

Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2010. Finding Low-utility Data Structures. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 174–186.

Guoqing Xu, Atanas Rountev, and Manu Sridharan. 2009. *Scaling CFL-Reachability-Based Points-To Analysis Using Context-Sensitive Must-Not-Alias Analysis.* Springer Berlin Heidelberg, 98–122.

Dacong Yan, Guoqing Xu, and Atanas Rountev. 2011a. Demand-driven Context-sensitive Alias Analysis for Java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*. ACM, 155–165.

Dacong Yan, Guoqing Xu, and Atanas Rountev. 2011b. Demand-driven Context-sensitive Alias Analysis for Java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA)*. 155–165.

Mihalis Yannakakis. 1990. Graph-theoretic Methods in Database Theory. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*. 230–242.

Andrew C. Yao. 1985. On the Expected Performance of Path Compression Algorithms. *SIAM J. Comput.* 14, 1 (1985), 129–133.

Hao Yuan and Patrick Eugster. 2009. An Efficient Algorithm for Solving the Dyck-CFL Reachability Problem on Trees. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009 (ESOP)*. 175–189.

Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. 2013. Fast Algorithms for Dyck-CFL-reachability with Applications to Alias Analysis *(PLDI)*. ACM.

Qirun Zhang and Zhendong Su. 2017. Context-sensitive Data-dependence Analysis via Linear Conjunctive Language Reachability. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 344–358.

Xin Zheng and Radu Rugina. 2008. Demand-driven Alias Analysis for C. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, 197–208.