Fast Algorithms for Dyck-CFL-Reachability with Applications to Alias Analysis

Qirun Zhang Michael R. Lyu
The Chinese University of Hong Kong
{qrzhang, lyu}@cse.cuhk.edu.hk

Hao Yuan [*]
BOPU Technologies
hao@bopufund.com

Zhendong Su
University of California, Davis
su@cs.ucdavis.edu

Abstract

The context-free language (CFL) reachability problem is a well-known fundamental formulation in program analysis. In practice, many program analyses, especially pointer analyses, adopt a restricted version of CFL-reachability, Dyck-CFL-reachability, and compute on edge-labeled bidirected graphs. Solving the all-pairs Dyck-CFL-reachability on such bidirected graphs is expensive. For a bidirected graph with n nodes and m edges, the traditional dynamic programming style algorithm exhibits a subcubic time complexity for the Dyck language with k kinds of parentheses. When the underlying graphs are restricted to bidirected trees, an algorithm with $O(n \log n \log k)$ time complexity was proposed recently. This paper studies the Dyck-CFL-reachability problems on bidirected trees and graphs. In particular, it presents two fast algorithms with $O(n)$ and $O(n + m \log m)$ time complexities on trees and graphs respectively. We have implemented and evaluated our algorithms on a state-of-the-art alias analysis for Java. Results on standard benchmarks show that our algorithms achieve orders of magnitude speedup and consume less memory.

Categories and Subject Descriptors F.2.2 [*Nonnumerical Algorithms and Problems*]: Computations on discrete structures; D.3.4 [*Processors*]: Compilers; F.3.2 [*Semantics of Programming Languages*]: Program analysis

General Terms Algorithms, Design, Experimentation, Languages

Keywords Dyck-CFL-reachability; alias analysis

1. Introduction

The context-free language (CFL) reachability problem is a generalization of the traditional graph reachability problem [27]. Many program analyses have been formulated as CFL-reachability problems, such as interprocedural data flow analysis [29], program slicing [28], shape analysis [26], type-based flow analysis [22, 25], and pointer analysis [31–33, 36, 37, 40]. When the underlying CFL is restricted to a Dyck language which generates matched parentheses, the CFL-reachability problem is referred to as Dyck-CFL-reachability. Although a restricted version of CFL-reachability, Dyck-CFL-reachability can express "almost all of the applications of CFL-reachability" in program analysis [19].

Solving Dyck-CFL-reachability of size k (*i.e.*, k kinds of parentheses) is expensive in practice. The traditional dynamic programming style CFL-reachability algorithm [29, 38] runs in $O(k^3 n^3)$ time. Only recently, the first subcubic algorithm was proposed, reducing the cubic time complexity by a factor of $\log n$ [7]. Scaling Dyck-CFL-reachability-based analyses on real-world applications is challenging. Various enhancements have been proposed, such as leveraging demand-driven properties in specific analyses [33, 37, 40], making use of a specialized reduction to set constraints [19], and approximating the client problems [32, 33]. However, all existing Dyck-CFL-reachability algorithms relying on the dynamic programming scheme exhibit a subcubic time complexity. When the underlying graphs are restricted to bidirected trees, Yuan and Eugster proposed an algorithm with $O(n \log n \log k)$ time complexity [39].

In this paper, we focus on the bidirected version of Dyck-CFL-reachability, as detailed in Section 2.2. The bidirected Dyck-CFL-reachability is particularly suitable for pointer analysis. All state-of-the-art demand-driven pointer analyses [31–33, 37, 40] are formulated by extending Dyck-CFL-reachability and compute on edge-labeled bidirected graphs. Specifically, matched parentheses derived from Dyck-CFL-reachability can be used to capture field accesses (*i.e.*, `load`/`store`) in Java [32, 33, 36, 37] and indirections (*i.e.*, references/dereferences) in C [40]. The bidirectness of graphs is also a prerequisite for CFL-reachability formulations of pointer analyses as discussed by Reps [27]. Namely, edges in the original graph need to be augmented with reverse edges (*a.k.a.* barred edges). Otherwise, two nodes may not be reachable even via standard graph reachability.

This paper proposes two fast algorithms for solving the bidirected Dyck-CFL-reachability on trees and graphs respectively. The key insight behind our algorithms is the observation of an equivalence property on bidirected structures that has not been fully utilized in previous work. We exploit this property to obtain asymptotically much faster algorithms by safely collapsing nodes that belong to the same equivalence class. Table 1 compares our new algorithms and some of the existing algorithms for bidirected Dyck-CFL-reachability, where n and m denote the numbers of nodes and edges in the graph respectively. We also present the design and implementation of our algorithms, and apply them to a state-of-the-art alias analysis [37]. Empirical results on the standard benchmarks show that our proposed algorithms achieve orders of magnitude speedup and consume less memory compared to the traditional CFL-reachability algorithm.

The principal contributions of our work are as follows:

- For the case of bidirected trees, we give an algorithm that runs in $O(n)$ time and $O(n)$ space, which answers the Dyck-CFL-

[*] Part of this work was done while the author was working at City University of Hong Kong.

Permission to make digital or hard copies of all or part 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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'13, June 16–19, 2013, Seattle, WA, USA.
Copyright © 2013 ACM 978-1-4503-2014-6/13/06...$15.00

Type	Time	Space	Reference
Tree	$O(n \log n \log k)$	$O(n \log n)$	[39]
Graph	$O(k^3n^3)$	$O(kn^2)$	[29, 38]
Graph	$O(kn^3)$	$O(kn^2)$	[19]
Graph	$O(kn^3/\log n)$	$O(kn^2)$	[7]
Tree	$O(n)$	$O(n)$	Algorithm 4
Graph	$O(n + m \log m)$	$O(n + m)$	Algorithm 5

Table 1. Bidirected Dyck-CFL-reachability algorithms.

reachability queries for any node pair in $O(1)$ time. This result improves the $O(n \log n \log k)$ time and $O(n \log n)$ space algorithm proposed by Yuan and Eugster [39].

- For the case of bidirected graphs, we give an algorithm that runs in $O(n + m \log m)$ time and $O(n + m)$ space, which answers the Dyck-CFL-reachability queries for any node pair in $O(1)$ time. This result improves the traditional subcubic time result in the literature [7].
- We apply our algorithms to a state-of-the-art context-insensitive alias analysis for Java [37]. Our fast algorithms can be directly used in the analysis. Experimental results show that our algorithm achieves orders of magnitude speedup on real-world benchmarks.

The rest of the paper is structured as follows. Section 2 reviews the background material on Dyck-CFL-reachability. Section 3 discusses the equivalence property and a naïve all-pairs Dyck-CFL-reachability algorithm. Sections 4 and 5 present our fast algorithms on bidirected trees and graphs respectively. Section 6 describes an existing state-of-the-art alias analysis for Java as the client application for our algorithms. Section 7 describes our empirical comparison of the performance of our algorithm versus the performance of the CFL-reachability algorithm using the client alias analysis. Section 8 surveys related work, and Section 9 concludes.

2. Preliminaries

This section reviews basic background on Dyck-CFL-reachability and defines its bidirected variants. We also include the traditional subcubic solution for reference and completeness.

2.1 Dyck-CFL-Reachability

Let $CFG = (\Sigma, N, P, S)$ be a context-free grammar with alphabet Σ, nonterminal symbols N, production rules P and start symbol S. Given a context-free grammar $CFG = (\Sigma, N, P, S)$ and a directed graph $G = (V, E)$ with each edge $(u, v) \in E$ labeled by a terminal $\mathcal{L}(u, v)$ from the alphabet Σ or ε, each path $p = v_0, v_1, v_2, \ldots, v_m$ in G *realizes* a string $R(p)$ over the alphabet Σ by concatenating the edge labels in the path in order, *i.e.*, $R(p) = \mathcal{L}(v_0, v_1)\mathcal{L}(v_1, v_2)\mathcal{L}(v_2, v_3) \ldots \mathcal{L}(v_{m-1}, v_m)$. A path in G is an S-path if the *realized* string can be derived from the start symbol S. Node v is S-reachable from node u iff there exists an S-path from u to v.

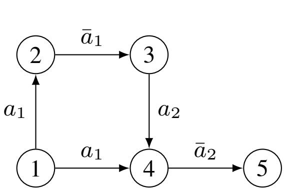
The CFL-reachability problem has four variants:

(1) *The all-pairs S-path problem:* For every pair of nodes u and v, is there an S-path in the graph from u to v?

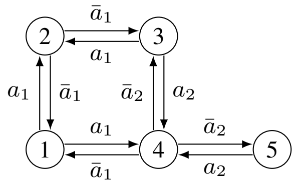
(2) *The single-source S-path problem:* Given a source node u, for all nodes v, is there an S-path in the graph from u to v?

(3) *The single-target S-path problem:* Given a target node v, for all nodes u, is there an S-path in the graph from u to v?

(4) *The single-source-single-target problem:* Given two nodes u and v, is there an S-path in the graph from u to v?



(a) The directed graph case.



(b) The bidirected graph case.

Figure 1. Example graphs illustrating a directed graph and its corresponding bidirected graph.

The Dyck-CFL-reachability is defined similarly by restricting the underlying CFL to a Dyck language, which generates strings of properly balanced parentheses. Consider an alphabet Σ over the set of opening parentheses $A = \{a_1, a_2, \ldots, a_k\}$ and the set of their matching closing parentheses $\bar{A} = \{\bar{a}_1, \bar{a}_2, \ldots, \bar{a}_k\}$. The Dyck language of size k (*i.e.*, k kinds of parentheses) is defined by the following context-free grammar:

$$S \rightarrow SS \mid a_1 S \bar{a}_1 \mid \ldots \mid a_k S \bar{a}_k \mid \varepsilon$$

where S is the start symbol and ε is the empty string. Specially, we say node v is *Dyck-reachable* from node u iff there exists an S-path from u to v, where S is the start symbol in the Dyck grammar above. We call such a path joining nodes u and v a *Dyck-path*.

2.2 Bidirected Dyck-CFL-Reachability

In this paper, we focus on the bidirected Dyck-CFL-reachability problems, which require the underlying graph to be bidirected and edge-labeled. For any directed edge (u, v) in the graph that is not labeled by ε, if it is labeled by an opening parenthesis $a_i \in A$, there must be a reverse edge (v, u) which is labeled by a matching closing parenthesis $\bar{a}_i \in \bar{A}$, and vice versa. Formally, we have the following definition.

DEFINITION 1 (Bidirected Dyck-CFL-Reachability). *Given a bidirected graph $G = (V, E)$ and a Dyck language of size k, the labels of directed edges in the graph must satisfy the following constraints:*

- *$\forall u, v \in V$, if $\mathcal{L}(u, v) = \varepsilon$, $\mathcal{L}(v, u)$ must be ε;*
- *$\forall u, v \in V$, if $\mathcal{L}(u, v) = a_i$, $\mathcal{L}(v, u)$ must be $\bar{a}_i$;*
- *$\forall u, v \in V$, if $\mathcal{L}(u, v) = \bar{a}_i$, $\mathcal{L}(v, u)$ must be a_i.*

The bidirected Dyck-CFL-Reachability and its four variants are defined similarly as Dyck-CFL-Reachability.

The Dyck-CFL-reachable node pairs (u, v) can be defined as a binary relation $\mathbb{D}$.

DEFINITION 2 (Dyck-CFL-Relation). *Given a bidirected graph $G = (V, E)$, we call a binary relation $\mathbb{D}$ on $V \times V$ a Dyck-CFL-relation iff for all $(u, v) \in \mathbb{D}$, v is* Dyck-reachable *from u in G.*

We give an example to illustrate the Dyck-CFL-reachability and the bidirected Dyck-CFL-reachability problems.

EXAMPLE 1. *Consider the two graphs in Figure 1. The graph to the left shows a directed graph for Dyck-CFL-reachability, and the one to the right is its bidirected counterpart. In both graphs, the* realized *string $R(p)$ of the path $p = 1, 2, 3, 4, 5$ is "$a_1\bar{a}_1a_2\bar{a}_2$", with properly matched parentheses. Therefore, node 5 is* Dyck-reachable *from node 1. However, the path $1, 4, 5$ is not a valid* Dyck-path.

The bidirected Dyck-CFL-reachability formulation has wide applications in pointer analysis. For pointer analysis problems, the

Algorithm 1: The standard CFL-reachability algorithm.

```
Input  : Edge-labeled directed graph $G = (V, E)$,
         normalized $CFG = (\Sigma, N, P, S)$
Output: the set $\{(i, j) \mid S\langle i, j\rangle \in G\}$
1  add $E$ to $W$
2  foreach production $X \to \epsilon \in P$ do
3      foreach node $v \in V$ do
4          if $X\langle v, v\rangle \notin E$ then
5              add $X\langle v, v\rangle$ to $E$ and to $W$
6  while $W \neq \emptyset$ do
7      select and remove an edge $Y\langle i, j\rangle$ from $W$
8      foreach production $X \to Y \in P$ do
9          if $X\langle i, j\rangle \notin E$ then
10             add $X\langle i, j\rangle$ to $E$ and to $W$
11     foreach production $X \to YZ \in P$ do
12         foreach outgoing edge $Z\langle j, k\rangle$ from node $j$ do
13             if $X\langle i, k\rangle \notin E$ then
14                 add $X\langle i, k\rangle$ to $E$ and to $W$
15     foreach production $X \to ZY \in P$ do
16         foreach incoming edge $Z\langle k, i\rangle$ to node $i$ do
17             if $X\langle k, j\rangle \notin E$ then
18                 add$X\langle k, j\rangle$ to $E$ and to $W$
```

directed edges in the underlying graph must be augmented with reverse edges (*a.k.a.* barred edges) [27], otherwise, two nodes may not be reachable from each other even by standard graph reachability. All existing CFL-reachability formulations for pointer analysis require the underlying graph to be bidirected. In addition, many pointer analyses employ Dyck-CFL-reachability to match certain properties, such as field accesses (*i.e.*, `load/store`) in Java [32, 33, 36, 37] and indirections (*i.e.*, references/dereferences) in C [40], which naturally satisfy the requirements of bidirected Dyck-CFL-reachability.

2.3 CFL-Reachability Algorithm

In the literature, there is a popular dynamic programming style algorithm [29, 38] for solving the CFL and Dyck-CFL-reachability problems. The algorithm is in Algorithm 1, where W denotes a worklist, and $X\langle u, v\rangle$ denotes the directed edge (u, v) with label $\mathcal{L}(u, v) = X$. The main algorithm has two steps: (1) *CFG Normalization* — The underlying CFG must be converted to a normal form, similar to the Chomsky Normal Form. When the grammar is in the normal form, all production rules are of the form $X \to YZ$, $X \to Y$ or $X \to \varepsilon$, where X is nonterminal, Y and Z are terminals or nonterminals, and ε denotes the empty string; and (2) *"Filling in" New Edges* — In order to compute the S-paths, new edges are added to the graph. For example, lines 11-14 describe that for the production rule $X \to YZ$ and edge $Y\langle i, j\rangle$,all outgoing edges of node j are considered. If there is an outgoing edge $Z\langle j, k\rangle$, a new edge $X\langle i, k\rangle$ is added to E if it is not already in E. The algorithm terminates if there are no more new edges to be inserted.

The complexity of Algorithm 1 is cubic in terms of the number of nodes in the input graph [21]. Chaudhuri [7] shows that the well-known Four Russians' Trick [5] can be employed on lines 12-13 and 16-17 in the CFL-reachability algorithm to immediately yield a subcubic algorithm with $O(n^3/\log n)$ time complexity.

3. Dyck-CFL-Relation

3.1 An Equivalence Property

We first study an equivalence property of Dyck-CFL-relations $\mathbb{D}$ on bidirected trees and graphs, which has not been fully utilized in previous work. Since trees are simply graphs without cycles, we use the more general term "graph" to illustrate the equivalence property. A binary relation $\sim \subseteq B \times B$ on a set B is an equivalence relation iff it is reflexive, symmetric and transitive. Specifically,

- $\sim$ is reflexive if $\forall a \in B,\ a \sim a$;
- $\sim$ is symmetric if $\forall a, b \in B,\ a \sim b \implies b \sim a$; and
- $\sim$ is transitive if $\forall a, b, c \in B,\ a \sim b \wedge b \sim c \implies a \sim c.$

For a given bidirected graph $G = (V, E)$, we consider the Dyck-CFL-relation $\mathbb{D}$ over $V \times V$. Based on the definition of relation $\mathbb{D}$, node $v \in V$ is *Dyck-reachable* from node $u \in V$ iff $(u, v) \in \mathbb{D}$. We list below the properties of relation $\mathbb{D}$ on bidirected graphs:

- *Relation* $\mathbb{D}$ *is reflexive:* This is because the start symbol S in the Dyck grammar is nullable (*i.e.*, it generates the empty string ε). Therefore, $(u, u) \in \mathbb{D}$ for all $u \in V$.
- *Relation* $\mathbb{D}$ *is symmetric:* One can identify a symmetric relation by showing it is equal to its inverse. For the bidirected graphs, the *realized* string $R(p)$ on a path p from node u to v is the reverse of $R(p')$ on the reverse path p' from v to u. It is easy to show $R(p)$ is generated by the Dyck grammar iff $R(p')$ is generated by the Dyck grammar with a simple induction on the path length. As a result, if v is *Dyck-reachable* from u (*i.e.*, $(u, v) \in \mathbb{D}$), u is also *Dyck-reachable* from v (*i.e.*, $(v, u) \in \mathbb{D}$).
- *Relation* $\mathbb{D}$ *is transitive:* That is, for any three nodes $u, v, w \in V$ in graph $G = (V, E)$, if v is *Dyck-reachable* from u (*i.e.*, $(u, v) \in \mathbb{D}$) and w is *Dyck-reachable* from v (*i.e.*, $(v, w) \in \mathbb{D}$), w is *Dyck-reachable* from u (*i.e.*, $(u, w) \in \mathbb{D}$). It is immediate that the *realized* string $R(p_1)$ for any path p_1 connecting node u and v can be derived from the start symbol S in the Dyck grammar. Similarly, the *realized* string $R(p_2)$ for any path p_2 connecting nodes v and w is also generated from the Dyck grammar. Consequently, the concatenated string $R(p_1)R(p_2)$ is generated by the Dyck grammar because of the rule $S \to SS$. Hence, the path p_1p_2 from node u to w is also a *Dyck-path*.

The discussions above lead to the following lemma.

LEMMA 1. *The Dyck-CFL-relation* $\mathbb{D}$ *on a bidirected graph is an equivalence relation.*

The key insight in our algorithms is that the equivalence property can be exploited to obtain asymptotically much faster algorithms. All nodes in the Dyck-CFL-relation $\mathbb{D}$ are equal to the other nodes in the graph, and thus nodes that belong to the same equivalence class can be safely collapsed to a single representative node. For example, in Figure 1(b), node 3 is *Dyck-reachable* from 1, thus, they can be collapsed into a single representative node $\{1, 3\}$ indicating that they are in the same equivalence class. Similarly, node 5 can be collapsed to the representative node $\{1, 3\}$ as well. Finally, we have a representative node $\{1, 3, 5\}$ reflecting the fact that the three nodes are *Dyck-reachable* from each other in the graph.

3.2 A Naïve Approach

We proceed to give a naïve all-pairs Dyck-CFL-reachability algorithm by collapsing the nodes in the graph that are in the Dyck-CFL-relation $\mathbb{D}$. Let $a_i\langle u, v\rangle$ denote the directed edge (u, v) labeled by $a_i \in A$. We note that while collapsing two *Dyck-reachable* nodes x and y in the graph, there always exists a node

z such that $a_i\langle x, z\rangle = a_i\langle y, z\rangle$. For example, in Figure 1(b), we have $a_1\langle 1, 2\rangle = a_1\langle 3, 2\rangle$. Without loss of generality, given a bidirected graph $G(V, E)$, the naïve algorithm can work on a directed graph $G'(V', E')$ by removing all edges labeled by closing parentheses from the original graph, *i.e.*, $V' = V$ and $a_i\langle u, v\rangle \in E'$ iff $a_i\langle u, v\rangle \in E$ for all labeled edges in E'. The basic idea of the naïve approach is to explicitly maintain a list W of nodes. For every item z popped from W, we pick two incoming neighbors x and y whose edges are labeled by the same opening parenthesis *i.e.*, $\exists a_i\langle x, z\rangle = a_i\langle y, z\rangle$, and then collapse x and y since they are *Dyck-reachable* via z. Due to the collapsing between nodes, E' may possibly contain *multiple edges*. The whole algorithm terminates if W is empty.

The naïve algorithm is given in Algorithm 2, where Eq_nodes$[v]$ denotes the equivalence set of node v and Set$[v]$ denotes the equivalence set number that node v belongs to. The procedure HAS-SAME-IN(v) traverses all incoming neighbors of node v, and returns true if there exist two neighbors u_1 and u_2 such that $a_i\langle u_1, v\rangle = a_i\langle u_2, v\rangle$. In Algorithm 2, line 1 transforms the given graph G to G', and lines 2-5 initialize W and Eq_nodes[v]. Lines 10-26 collapse node y to x *w.r.t.* node z, and remove y. The detailed procedure on collapsing y to x is given in Section 5.1.1. Finally, lines 29-31 assign the equivalence set number to each node v, such that any query can be answered in $O(1)$ time.

Complexity Analysis. The time complexity of the naïve algorithm is $O(kn^2)$. We begin by analyzing the maximum number of steps that the "while" loop on line 6 can be executed. We note that Algorithm 2 adds items to W only through lines 5 and 25. On line 25, item x can be added to W for at most $n-1$ times, since line 26 can be executed for at most $n-1$ times. On line 5, W is initialized with n items. Therefore, the worklist W can be filled with at most $2n-1$ items by Algorithm 2. In the "while" loop, only line 28 removes an item from W, thus, the "else" part of the "if" statement can be executed for at most $2n-1$ times. Since the "then" part of the same "if" statement can be executed for at most $n-1$ times, the "while" loop can be executed for at most $(n-1)+(2n-1) = 3n-2 = O(n)$ times. For each item z popped from W in the "while" loop, lines 8-28 take $O(kn)$ time to process. Specifically, the procedure HAS-SAME-IN(v) on lines 8 and 25 takes $O(kn)$ time to traverse all neighbors of node v, and the two "foreach" loops on lines 15 and 16 are bounded by $|A| = k$ and $|V'| = n$ respectively. Therefore, Algorithm 2 takes $O(kn^2)$ time. The space complexity is $O(n+m)$, since the input graph can be stored using FDLL to be introduced in Section 5.1.2 with $O(m)$ space and the worklist W takes $O(n)$ space. Putting everything together, we have the following theorem:

THEOREM 1. *Algorithm 2 pre-processes the input graph in $O(kn^2)$ time and $O(n+m)$ space to answer any online bidirected Dyck-CFL-reachability query in $O(1)$ time.*

In the following two sections, we describe two improved algorithms. They share the same insight with the the naïve approach, which have better time complexities on bidirected trees and graphs respectively. Specifically, our tree algorithm in Section 4 uses a single tree walk to find all equivalence sets because trees do not contain cycles. Our graph algorithm in Section 5 employs improved data structures to track nodes in W and to merge edges on x and y.

Algorithm 2: A naïve Dyck-CFL-reachability algorithm.

```
   Input  : Edge-labeled directed graph $G = (V, E)$
   Output: Set$[v]$ for all $v \in V$
1  transform the input graph $G$ to $G' = (V', E')$
2  initialize $W$ to be empty
3  foreach $v \in V'$ do
4      Eq_nodes$[v]$ = $\{v\}$
5      if HAS-SAME-IN$(v)$ then add $v$ to $W$
6  while $W \neq \emptyset$ and $|V'| > 1$ do
7      let $z$ be the front node from $W$
8      if $z \in V'$ and HAS-SAME-IN$(z)$ then
9          let $x, y$ be two nodes such that $\exists a_i\langle x, z\rangle = a_i\langle y, z\rangle$
10         Eq_nodes$[x]$ = Eq_nodes$[y] \cup$ Eq_nodes$[x]$
11         foreach $a_i \in A$ do
12             if $a_i\langle y, y\rangle \in E'$ then
13                 if $a_i\langle x, x\rangle \notin E'$ then add $a_i\langle x, x\rangle$ to $E'$
14                 remove $a_i\langle y, y\rangle$ from $E'$
15         foreach $a_i \in A$ do
16             foreach $w \in V'$ do
17                 if $a_i\langle w, y\rangle \in E'$ then
18                     if $a_i\langle w, x\rangle \notin E'$ then
19                         add $a_i\langle w, x\rangle$ to $E'$
20                     remove $a_i\langle w, y\rangle$ from $E'$
21                 if $a_i\langle y, w\rangle \in E'$ then
22                     if $a_i\langle x, w\rangle \notin E'$ then
23                         add $a_i\langle x, w\rangle$ to $E'$
24                     remove $a_i\langle y, w\rangle$ from $E'$
25         add $x$ to $W$ if $x \notin W$ and HAS-SAME-IN$(x)$
26         remove $y$ from $V'$
27     else
28         remove $z$ from $W$
29 foreach $v \in V'$ do
30     foreach $u \in$ Eq_nodes$[v]$ do
31         Set$[u]$ = $v$
```

4. Dyck-CFL-Reachability Algorithm on Bidirected Trees

This section presents our algorithm for solving the all-pairs Dyck-CFL-reachability problem on bidirected trees. Its time and space complexities are $O(n)$ and $O(n)$ respectively, and it answers any reachability query in $O(1)$ time. We remind the reader that the previous best result on bidirected trees [39] has $O(n \log n \log k)$ time and $O(n \log n)$ space complexities. First, we describe a linear-sized data structure to store the all-pairs reachability information. We then show how to utilize the equivalence property to solve the all-pairs Dyck-CFL-reachability problem using a single walk on trees.

4.1 The STRATIFIED-SETS Representation

In our algorithm, the all-pairs Dyck-CFL-reachability information is stored in disjoint sets. Two nodes u and v are *Dyck-reachable* from each other in the tree iff they belong to the same set. In other words, each disjoint set C corresponds to an equivalence class described by relation $\mathbb{D}$, *i.e.*, $u, v \in C$ iff $(u, v) \in \mathbb{D}$. We name the disjoint set representation in our main algorithm as STRATIFIED-SETS.

The STRATIFIED-SETS consist of several disjoint sets spanning over different layers. Each disjoint set stores the nodes that are *Dyck-reachable* from each other in the bidirected tree. The layers are indexed by an integer i. Note that the layer information is only used for providing a better explanation. The layer index i grows downward, *i.e.*, layer i is the upper layer in any two adjacent layers

i and $i+1$. The disjoint sets on the same layer i have no edges directly connecting each other. For any two adjacent layers i and $i+1$, there exists at least one edge connecting two disjoint sets C from layer i and C' from layer $i+1$. Specially, the connecting edge is labeled by $\mathcal{L}(u,v) \in A$, respecting the fact that there exist $u \in C$ and $v \in C'$ such that (u,v) is a directed edge in the tree with the same label $\mathcal{L}(u,v) \in A$. Note that there can be at most k edges connecting the set C' with the distinct sets from the upper layer i. However, more than k edges are possible for connecting the set C with the distinct sets from the lower layer $i+1$. Figure 2(b) shows an example STRATIFIED-SETS representation, where there are seven sets spanning four layers.

The STRATIFIED-SETS representation is implemented using three ingredients: one integer variable `curset`, two integer arrays `Set`$[v]$ and `Up`$[$`Set`$[v]][a_i]$. `Set`$[v]$ records the equivalence set number that node v belongs to, and `Up`$[$`Set`$[v]][a_i]$ stores the equivalence set number of the set from the upper layer that is connected to `Set`$[v]$ *w.r.t.* the edge labeled by the opening parenthesis $a_i \in A$. The STRATIFIED-SETS uses the integer variable `curset` to keep track of the *current* total number of disjoint sets. Due to the `Up` array, the tree algorithm does not need the layer information explicitly. The STRATIFIED-SETS implementation also permits three operations: `Init`(v), `Find`(v) and `Add`(v,e) described in Procedure 3. The functioning of procedures `Init`(v) and `Find`(v) is fairly straightforward. The procedure `Init`(v) takes a node v as input, assigns it to a new set indexed by `curset` in STRATIFIED-SETS, and increases the `curset` count. `Find`(v) returns the equivalence set number that node v belongs to.

We detail the description of procedure `Add`(v,e) to illustrate the idea of collapsing nodes in relation $\mathbb{D}$. We use `Add`(v,e) to insert the node v to the STRATIFIED-SETS with regard to the edge $e=(u,v)$ and the edge label $\mathcal{L}(u,v)$ in the tree. Node v is added to STRATIFIED-SETS by either assigning it to an new set (lines 3 and 9) or collapsing it to an existing set (line 13). Consider the example input tree in Figure 2(a), node 3 and edge $(2,3)$ are processed by `Add`(v,e). The resulting STRATIFIED-SETS is in Figure 2(b). Node 3 is assigned to a new set on layer 3. The new set is then linked with the set containing node 2 on layer 2 respecting the fact that $\mathcal{L}(2,3)=a_1$. Then, node 4 and edge $(3,4)$ are processed. Node 4 is collapsed to the set on layer 2 that contains node 2 respecting the facts that $\mathcal{L}(3,4)=\bar{a}_1$ and node 4 is *Dyck-reachable* from node 2 (*i.e.*, $(2,4)\in\mathbb{D}$). Formally, if the edge label $\mathcal{L}(u,v)$ in the tree is an opening parenthesis $a_i \in A$, v is assigned to a new set indexed by `curset` in STRATIFIED-SETS. This new set is then linked with the set returned by `Find`(u) on the upper layer as described by lines 2-5 . If the edge label is a closing parenthesis $\bar{a}_i \in \bar{A}$, we simply collapse node v to the equivalence set that is connected via a matched opening parenthesis $a_i \in A$ from u's upper layer. The equivalence set is indexed by `Up`$[$`Find`$(u)][a_i]$ as described by line 13. Lines 9-11 indicate that, for node u whose link node does not exist, we assign node v to a new set indexed by `curset` and link the set returned by `Find`(u) to the new set from the upper layer.

Note that the `Up` array used in Procedure 3 is indeed a map: $(Num \rightarrow A) \rightarrow Num$, where Num denotes the domain of the set numbers. For each set in STRATIFIED-SETS, line 8 in Procedure 3 needs to find a particular edge a_i from $O(k)$ link edges in `Up`$[s]$, where $s \in Num$. The time taken to search such $O(k)$ edges depends on the actual implementation of the `Up` array. For example, if the `Up` array stores such $O(k)$ edges for each set s using a binary search tree, the lookup for an a_i edge in `Up`$[s]$ takes $O(\log k)$ time as mentioned in Yuan and Eugster's work [39]. In our algorithm, we implement the `Up` array using the FDLL data structure illustrated as Example 3 in Section 5.1.2, thus a lookup takes *expected* $O(1)$ time. The space required is $O(m)$ since there are m edges in a tree,

Procedure 3: Add(v, e) to add a node v to STRATIFIED-SETS according to the directed edge $e=(u,v)$.

```
1  if $\mathcal{L}(u,v) \in A$ then
2      let $a_i = \mathcal{L}(u,v)$
3      Set[$v$] = curset
4      Up[curset][$a_i$] = Find ($u$)
5      curset ++
6  if $\mathcal{L}(u,v) \in \bar{A}$ then
7      let $\bar{a}_i = \mathcal{L}(u,v)$
8      if Up[Find($u$)][$a_i$] does not exist then
9          Set[$v$] = curset
10         Up[Find($u$)][$a_i$] = curset
11         curset ++
12     else
13         Set[$v$] = Up[Find($u$)][$a_i$]
```

where $m=n-1$. Therefore, the time complexity of Procedure 3 is $O(1)$, and the space complexity of the `Up` array is $O(n)$.

4.2 Main Algorithm

This section presents the main algorithm. The key idea is to operate on the linear-sized STRATIFIED-SETS data structure to build the all-pairs Dyck-CFL-reachability information during a single tree walk.

The goal of our algorithm is to assign nodes u and v to the same set in STRATIFIED-SETS, for all $(u,v)\in\mathbb{D}$. The overall algorithm takes two steps:

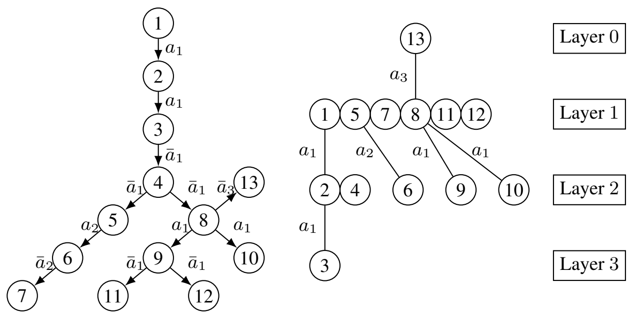
(1) *Initializing a leaf node:* In this step, we pick an arbitrary leaf node v from the tree and invoke the `Init`(v) procedure to initialize the given node v.

(2) *Processing each encountered edge:* For each edge (u,v) with label $\mathcal{L}(u,v)$ encountered during the tree walk, we process the edge *w.r.t.* the edge label and insert the node v to STRATIFIED-SETS according to the `Add`(v,e) procedure.

The complete algorithm is shown as Algorithm 4. In the main algorithm, lines 1-6 initialize the relevant data structures, and lines 7-14 describe a standard depth-first search (DFS) starting at node v. For a given bidirected tree $T=(V,E)$ with n nodes, DFS takes $O(n)$ time. For every node v, the `Add`(v,e) procedure takes $O(1)$ time. The space required by Algorithm 4 depends on the STRATIFIED-SETS representation, which is essentially implemented using the `Up` array. Therefore, the space complexity is $O(n)$.

EXAMPLE 2. *We consider the bidirected tree in Figure 2(a), where reverse edges are omitted for brevity. Algorithm 4 outputs the* STRATIFIED-SETS *in Figure 2(b). The* STRATIFIED-SETS *representation contains seven disjoint sets:* $\{1,5,7,8,11,12\}$, $\{2,4\}$, $\{6\}$, $\{9\}$, $\{10\}$, $\{3\}$ *and* $\{13\}$. *The nodes in the same set are* Dyck-reachable *from each other in the tree. Note that the layer information is only used for providing a better explanation. Algorithm 4 uses the `Up` array for finding a set from the upper layer in* STRATIFIED-SETS.

After constructing the STRATIFIED-SETS, any Dyck-CFL-reachability query (u,v) can be answered in $O(1)$ time by simply checking whether the indices returned by `Find`(u) and `Find`(v) are the same. Putting everything together, we have the following theorem.



(a) Input tree. (b) STRATIFIED-SETS representation.

Figure 2. A running example for Dyck-CFL-reachability on trees.

Algorithm 4: Dyck-CFL-reachability algorithm on trees.

```
Input  : Edge-labeled bidirected tree $T = (V, E)$
Output: the STRATIFIED-SETS
1  initialize the Up array to be empty
2  foreach $v \in V$ do
3      visited[$v$] = false
4      Set[$v$] = 0
5  stack.push(a leaf node $v$)
6  curset = 0
7  Init ($v$)
8  while stack is not empty do
9      $v$ = stack.pop
10     if not visited[$v$] then
11         visited[$v$] = true
12         foreach unvisited neighbor $u$ of $v$ do
13             stack.push($u$)
14             Add ($u$, $e(v, u)$)
```

THEOREM 2. *The bidirected Dyck-CFL-reachability problem on trees can be pre-processed in $O(n)$ time and $O(n)$ space to answer any online query in $O(1)$ time.*

5. Dyck-CFL-Reachability Algorithm on Bidirected Graphs

In this section, we study the Dyck-CFL-reachability problems on bidirected graphs. Computing Dyck-CFL-reachability on graphs is harder than that on trees because graphs may contain cycles. Consequently, the algorithm introduced in Section 4 based on the STRATIFIED-SETS representation cannot be directly applied to graphs.

Although Dyck-CFL-reachability on bidirected graphs is more complicated, the Dyck-CFL-relation $\mathbb{D}$ shares the same equivalence properties as for trees. For bidirected graphs, we utilize the idea of edge merging to collapse the node pairs $(u, v) \in \mathbb{D}$. For a bidirected graph with n nodes and $2m$ edges, our algorithm processes the given graph in $O(n + m \log m)$ time with $O(n + m)$ space, and can answer any Dyck-CFL-reachability query over any pair of nodes (u, v) in $O(1)$ time.

5.1 Basic Idea

As the naïve approach, given a bidirected graph $G(V, E)$, our algorithm works on the same directed graph $G'(V', E')$ by removing all edges labeled by closing parentheses from the original graph, *i.e.*, $V' = V$ and $a_i\langle u, v\rangle \in E'$ iff $a_i\langle u, v\rangle \in E$ for all labeled edges in E'. Therefore, G' has n nodes and m edges. The key idea behind our algorithm is to collapse any node pair (u, v) connected by a *Dyck-path* in the graph because such pairs (u, v) are in the Dyck-CFL-relation $\mathbb{D}$. As in the naïve approach, E' may possibly contain *multiple edges* due to the collapsing between nodes.

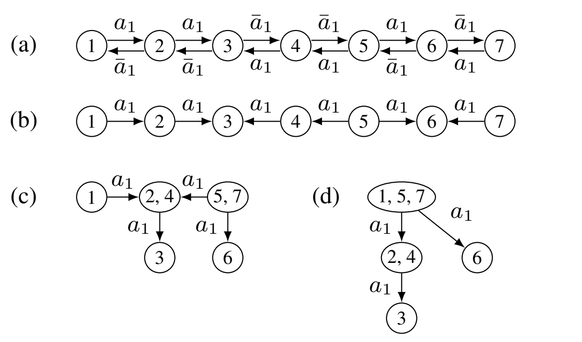


Figure 3. A Dyck-path example.

To make the above idea more concrete, we consider the example in Figure 3. Figure 3(a) shows the original path in G which contains two non-trivial sets of nodes that are *Dyck-reachable* from each other: $\{1, 5, 7\}$ and $\{2, 4\}$. Figure 3(b) shows the corresponding reduced path in G'. In such directed cases, node 5 is "connected" to node 7 via node 6, and $a_1\langle 5, 6\rangle = a_1\langle 7, 6\rangle$. Since nodes 5 and 7 are *Dyck-reachable* from each other, the two edges $a_1\langle 5, 6\rangle$ and $a_1\langle 7, 6\rangle$ should be merged to collapse nodes 5 and 7 into a single representative node $\{5, 7\}$ in Figure 3(c). We define a node like node 6 to be the *merging node*. Formally, we have the following definition.

DEFINITION 3 (Merging Node). *If a node $v \in V$ has at least two incoming edges labeled by $a_i \in A$, we say node v merges a_i edges. We define a node as a* merging node *iff it merges some $a_i \in A$ edges.*

Like the naïve algorithm in Algorithm 2, our main algorithm fulfills the following two tasks:

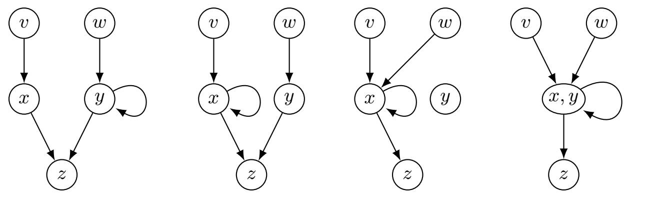
(1) *Merge edges for each merging node:* For any merging node in the graph G', the algorithm finds the incoming nodes with the same labeled edges, and merges the two edges to collapse the nodes. In the path in Figure 3(b), nodes 3 and 6 are two merging nodes. The relevant incoming edges should be merged.

(2) *Track the new merging nodes:* During edge merging, new merging nodes can be introduced into the graph G'. The algorithm tracks all new merging nodes in order to perform another edge merging. For the same example in Figure 3(c), collapsing nodes 2 and 4 generates a new representative node $\{2, 4\}$, which is also a merging node. Its corresponding edges should also be merged. The final output is shown in Figure 3(d).

In the naïve approach, nodes x and y are arbitrarily picked on line 9. Merging edges from y to x by enumerating all neighbors of y exhaustively takes $O(kn)$ time. Moreover, node tracking is achieved by simply traversing all neighbors of node z on line 8, which takes $O(kn)$ time as well. When the given graph is sparse, we can use additional data structures to improve the two tasks of merging edges and tracking nodes. Next, we describe them in detail.

5.1.1 Merging Edges

For the directed graph G', edge merging in G' picks a merging node z that has two incoming edges (x, z) and (y, z) with $a_i\langle x, z\rangle =$



(a) Before edge merging. (b) Collapsing self loops. (c) Collapsing all neighbors. (d) After edge merging.

Figure 4. Steps in edge merging.

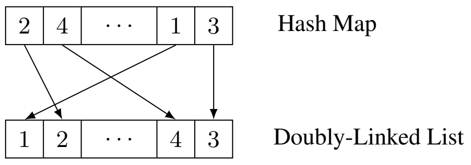


Figure 5. The illustration of the FDLL data structure.

$a_i\langle y, z\rangle$, and collapses nodes x and y. Specifically, if we choose to merge edge (y, z) to edge (x, z), all edges connecting y and its neighbor w should be deleted from G'. Node w is made a neighbor of x by inserting the relevant edges to G'. Finally, node y is removed from G', because it has been collapsed to x.

The order of edge merging is important. The naïve method in Algorithm 2 performs edge merging by collapsing nodes x and y arbitrarily with regard to the merging node z. If one adopts this approach, the time complexity can be $O(kn^2)$. When the graphs are sparse, it is possible to do edge merging faster. To this end, we employ a technique that is similar to the weighted-union heuristic (also known as the "union-by-size" heuristic) used in the disjoint sets data structures [9]. Namely, for each edge merging operation, we always collapse the node with a smaller degree to the node with a larger degree. Our new method bounds the total numbers of edge merging for *each edge* to $O(\log m)$. We provide a detailed complexity analysis in Section 5.3.

Taking the naïve approach in Algorithm 2 as an example, we discuss the process of edge merging. Specifically, our handling of edge merging from node y to node x has three phases, as illustrated in Figure 4. We assume that the degree of node x is larger than that of y, and all irrelevant edges are omitted in Figure 4. Figure 4(a) shows the original graph before edge merging. Figure 4(b) illustrates the handling in the first phase. If y has a self loop, the self loop is removed and added to x if x does not already have one (lines 11-14 in Algorithm 2). Second, we consider all neighbors of node y. As in Figure 4(c), for all shared neighbors z of x and y, those edges between z and y are removed; for those neighbors w that only belong to y, new edges between w and x are inserted (lines 15-24 in Algorithm 2). Finally, Figure 4(d) shows that node y is removed from the graph and degrees for relevant nodes in edge merging are updated.

5.1.2 Tracking Nodes

During edge merging, the in-degrees of merging nodes may change. We need to explicitly maintain a list of merging nodes whose in-degrees *w.r.t.* an opening parenthesis are at least 2. This section describes the design of our data structure to effectively maintain this information.

In the naïve algorithm, tracking nodes is achieved using a worklist W, which is typically implemented using a list. In our improved algorithm, we uses a doubly-linked list (DLL) and a hash map. We name the data structure FAST-DOUBLY-LINKED-LIST (FDLL). It is important to note that traditional list W used by the naïve approach on line 25 takes $O(n)$ time to find an element while FDLL takes *expected* $O(1)$ time. Figure 5 depicts an example FDLL. All elements in the FDLL are stored using a DLL and a hash map. The hash map associates each element with its position in the DLL, represented by arrows in Figure 5, to help quickly locate every element in the DLL. The *insertion* and *pop* operations on FDLL are nearly identical to DLL (or list), both of which take $O(1)$ time. Moreover, the FDLL supports two additional operations in *expected* $O(1)$ time:

- *Query:* Querying the membership of a particular element in FDLL is the same as querying the relevant membership in the hash map, which can be done in *expected* $O(1)$ time.
- *Deletion:* According to the hash map, the position of the element to be deleted can be found in *expected* $O(1)$ time. Thus removing the element at the specified position in the DLL can also be done in $O(1)$ time. Finally, the hash map entry associated with that element is erased in $O(1)$ time.

EXAMPLE 3. *In the tree algorithm, the* `Up` *array can be implemented using FDLL to support the* expected $O(1)$ *time lookup. Recall that the* `Up` *array is indeed a map:* $(Num \rightarrow A) \rightarrow Num$. *For each set* $s \in Num$, *we use an FDLL* $A[s]$ *to store the set of opening parentheses* a_i, *such that there exists an* a_i *labeled edge connecting set* s *and the set from the upper layer. For each* $a_i \in A[s]$, *we use another FDLL* $U[s\_i]$ *to store the corresponding set from the upper layer. For example, to look up an edge* a_3 *of set* 2, *we query whether* $a_3 \in A[2]$. *If such an* a_3 *exists,* $U[2\_3]$ *keeps the corresponding set number. The two lookups both take* expected $O(1)$ *time.*

5.2 Main Algorithm

We now present our algorithm solving bidirected Dyck-CFL-reachability on graphs in Algorithm 5, combining the ideas of merging edges and tracking nodes.

Before delving into the main algorithm, we first illustrate the use of FDLL in our main algorithm. For each node v in the input graph $G' = (V', E')$, the set of opening parentheses of v's incoming edges is represented using an FDLL, denoted as $A_{\text{in}}[v]$. For each $a_i \in A_{\text{in}}[v]$, we use the FDLL `In[`v_i`]` to store all v's incoming neighbors. We denote the size of `In[`v_i`]` as `In[`v_i`].size()`. Node v merges edge a_i iff `In[`v_i`].size()` > 1. We represent `Out[`v_i`]` and $A_{\text{out}}[v]$ similarly. Finally, the worklist FDLL$_w$ is also represented using an FDLL.

Algorithm 5 uses the array $\hat{D}[v]$ to represent the *total degree* of node v, which is initialized to v's degree in the original graph. Notations `eq_nodes`$[v]$ and `Set`$[v]$ are defined in the same way as the naïve approach. The functioning of the algorithm proceeds as follows. On line 1, the original bidirected graph G is pre-processed to obtain the directed graph $G' = (V', E')$. From lines 2-5, the equivalence set `Eq_nodes`$[v]$ is initialized with node v and the FDLL$_w$ is initialized with v_i indicating node v merges a_i. The FDLL$_w$ in the main algorithm is used to implement the idea of node tracking. The main algorithm then proceeds to handle edge merging as follows:

- Lines 7-11: The algorithm pops one z_i from the FDLL$_w$, then chooses its two neighbors x and y for edge merging. Specifically, node y with a smaller *total degree* is collapsed to node x with a larger *total degree*.
- Lines 12-18: The self loop on node y is handled as described in Figure 4(b). During edge merging, if a node v becomes a non-merging node for a_i (*i.e.*, `In[`v_i`].size()` < 2), v_i is removed from the FDLL$_w$. Similarly, when node v becomes a merging node that merges a_i edges, v_i is inserted to the FDLL$_w$. Note

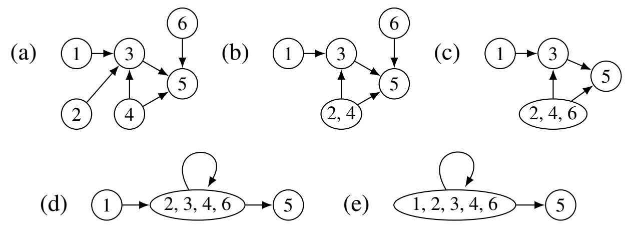


Figure 6. A running example for bidirected Dyck-CFL-reachability on graphs.

Algorithm 5: Dyck-CFL-reachability algorithm on graphs.

```
   Input  : Edge-labeled bidirected graph $G = (V, E)$
   Output: Set[$v$] for all $v \in V$
1  transform the input graph $G$ to $G' = (V', E')$
2  foreach $v \in V'$ do
3      Eq_nodes[$v$] = {$v$}
4      foreach $a_i \in A_{in}[v]$ do
5          add $v\_i$ to FDLL$_w$ if In[$v\_i$].size() $> 1$
6  while $FDLL_w \neq \emptyset$ do
7      let $z\_i$ be the front item from FDLL$_w$
8      let $x$ and $y$ be two front nodes from In[$z\_i$]
9      let $x$ denote the node such that $\hat{D}[x] \geqslant \hat{D}[y]$
10     $\hat{D}[x] = \hat{D}[x] + \hat{D}[y]$
11     Eq_nodes[$x$] = Eq_nodes[$y$] $\cup$ Eq_nodes[$x$]
12     foreach $a_i \in A_{in}[y]$ do
13         if $a_i\langle y, y\rangle \in E'$ then
14             if $a_i\langle x, x\rangle \notin E'$ then
15                 add $a_i\langle x, x\rangle$ to $E'$
16                 add $x\_i$ to FDLL$_w$ if $x\_i \notin$ FDLL$_w$ and
                   In[$x\_i$].size() $> 1$
17             remove $a_i\langle y, y\rangle$ from $E'$
18             remove $y\_i$ from FDLL$_w$ if $y\_i \in$ FDLL$_w$ and
               In[$y\_i$].size() $< 2$
19     foreach $a_i \in A_{in}[y]$ do
20         foreach $w \in$ In[$y\_i$] do
21             if $a_i\langle w, x\rangle \notin E'$ then
22                 add $a_i\langle w, x\rangle$ to $E'$
23                 add $x\_i$ to FDLL$_w$ if $x\_i \notin$ FDLL$_w$ and
                   In[$x\_i$].size() $> 1$
24             remove $a_i\langle w, y\rangle$ from $E'$
25             remove $y\_i$ from FDLL$_w$ if $y\_i \in$ FDLL$_w$ and
               In[$y\_i$].size() $< 2$
26     foreach $a_i \in A_{out}[y]$ do
27         foreach $w \in$ Out[$y\_i$] do
28             if $a_i\langle x, w\rangle \notin E'$ then
29                 add $a_i\langle x, w\rangle$ to $E'$
30             remove $a_i\langle y, w\rangle$ from $E'$
31             remove $w\_i$ from FDLL$_w$ if $w\_i \in$ FDLL$_w$ and
               In[$w\_i$].size() $< 2$
32     remove $y$ from $V'$
33 foreach $v \in V'$ do
34     foreach $u \in$ Eq_nodes[$v$] do
35         Set[$u$] = $v$
```

that when an edge is inserted/removed from E', all of `In`[v_i], $A_{\text{in}}[v]$, `Out`[v_i] and $A_{\text{out}}[v]$ need to be updated accordingly.

- Lines 19-31: All incoming and outgoing neighbors of y are handled as described in Figure 4(c). The update on FDLL$_w$ for merging nodes is similar to the handling of self loop.
- Line 32: Node y is removed from V'. Note that we do not need to remove z_i because it was once w_i.

The edge merging ends when the FDLL$_w$ is empty, *i.e.*, there are no merging nodes in the final graph. Lines 33-35 indicate that all nodes u in the equivalence set `Eq_nodes`[v] are enumerated to associate `Set`[u] with v.

After the main algorithm terminates, `Set`[v] stores the equivalence set number that node v belongs to. Similar to the tree case, any Dyck-CFL-reachability query (u, v) can be answered in $O(1)$ time by simply checking the equivalence set numbers of nodes u and v.

EXAMPLE 4. *Figure 6 shows an example. All edges are labeled by* $a_1 \in A$. *The original edges labeled by* $\bar{a}_1 \in \bar{A}$ *are removed first. Nodes* 3 *and* 5 *are two merging nodes. In the first iteration, nodes* 2 *and* 4 *are collapsed by edge merging. Then node* 6 *is collapsed as well. Finally, node* 1 *is collapsed due to its self loop in the graph. In the final graph, all of the nodes in the original graph are distributed into two disjoint sets.*

5.3 Algorithm Correctness and Complexity Analysis

This section discusses the correctness and complexity of our proposed algorithm. First, we establish its correctness.

THEOREM 3 (Correctness). *Algorithm 5 correctly finds all* Dyck-paths *in the input graph.*

Proof. It is clear that any *Dyck-path* reported by Algorithm 5 is indeed a *Dyck-path* due to the observed equivalence property (Lemma 1). Thus, our proof focuses on the other direction, that is Algorithm 5 finds all *Dyck-path*s in the input graph.

Any trivial *Dyck-path* generated by rule $S \rightarrow \varepsilon$ is handled correctly, because every node v in the graph is marked as *Dyck-reachable* from itself due to line 3 in Algorithm 5. Dyck grammar essentially generates the properly matched parentheses. Therefore, the length of any non-trivial *Dyck-path* in the graph is even. We prove by induction on the length $|p|$ of any non-trivial *Dyck-path*.

Base case. $|p| = 2$. Let the *Dyck-path* be $p = v_1v_2v_3$, with the *realized* string $R(p) = \mathcal{L}(v_1, v_2)\mathcal{L}(v_2, v_3) = a_i\bar{a}_i$. Because the graph is bidirected, we have $\mathcal{L}(v_3, v_2) = \mathcal{L}(v_1, v_2) = a_i$. Nodes v_1 and v_3 are collapsed due to the merging node v_2.

Inductive step. Suppose Algorithm 5 correctly finds all non-trivial *Dyck-paths* of length $|p|$ in the graph. According to the Dyck grammar, any non-trivial *Dyck-path* of length $|p| + 2$ is generated by the following two rules:

- $S \rightarrow a_iS\bar{a}_i$ indicates that the new S-path is generated by prepending an open parenthesis and appending by a matching closing parenthesis. Let the path be $p = v_1v_2 \ldots v_3v_4$, where $\mathcal{L}(v_1, v_2) = a_i$ and $\mathcal{L}(v_3, v_4) = \bar{a}_i$. The *realized* string $R(v_2 \ldots v_3) = S$ indicates that nodes v_2 and v_3 are *Dyck-reachable*, where the *Dyck-path* joining v_2 and v_3 is of length $|p|$. According to the induction hypothesis, they have been collapsed into a single representative node. Such a node is the merging node for v_1 and v_4. Thus, v_1 and v_4 are collapsed according to Algorithm 5.
- $S \rightarrow SS$ indicates that the new S-path is composed of two S-paths. Let the path be $p = v_1 \ldots v_2 \ldots v_3$, where $R(v_1 \ldots v_2) = R(v_2 \ldots v_3) = S$. Note that the length of

any non-trivial *Dyck-path* is at least 2. Since the new S-path is of length $|p|+2$, the lengths of both path $v_1, \ldots, v_2$ and path $v_2, \ldots, v_3$ are less than or equal to $|p|$. According to the induction hypothesis, both v_1, v_2 and v_2, v_3 are collapsed into a single representative node. As a result, v_1 and v_3 are collapsed into the same representation node as well.

□

Next we analyze the complexity of Algorithm 5. Note that the *total degree* $\hat{D}[v]$ used in our algorithm is different from the degree $D[v]$ of node v respecting the fact that the *total degree* $\hat{D}[v]$ admits duplicated edges. For example, in Figure 4(c), $\hat{D}[x] = 4+2 = 6$, but $D[x] = 5$, because edge (y, z) is duplicated with (x, z) according to the final representative node $\{x, y\}$ in Figure 4(d). Therefore, the *total degree* $\hat{D}[v]$ never decreases during edge merging. Our algorithm processes the merging node z and collapses its neighbor y with a smaller $\hat{D}[y]$ to node x with a larger $\hat{D}[x]$. Nodes y and x are collapsed into a single representative node $\{x, y\}$. As a result, the *total degree* of y after merging is the same as the *total degree* of x according to line 10, namely, $\hat{D}[x] = \hat{D}[x] + \hat{D}[y]$. On line 9, we have $\hat{D}[x] \geqslant \hat{D}[y]$. Combining the analysis of the two lines, the following lemma is immediate:

LEMMA 2. *For each edge merging in Algorithm 5, $\hat{D}[y]$ is doubled.*

Let m denote the number of edges in graph G', *i.e.*, $2m = \sum_{v \in V} D[v]$. Due to the duplicated edges, $D[v]$ may decrease during edge merging. Similarly, $\hat{m}$ denotes the *total number* of edges in graph G' *w.r.t.* $\hat{D}[v]$ that admits duplicated edges, *i.e.*, $\hat{m} = \sum_{v \in V} \hat{D}[v]$. For each edge (x, z) in graph G', we say it is "moved" if either the *total degree* $\hat{D}[x]$ or $\hat{D}[z]$ is doubled according to Lemma 2. For all $v \in V$, we have $\hat{D}[v] \leqslant \hat{m} \leqslant 2m$, because in the worst case, all nodes are collapsed into a single representative node (with m duplicated edges) in the final graph. Combined with Lemma 2, an edge is "moved" at most ($\log \hat{D}[x] + \log \hat{D}[z] \leqslant 2 \log 2m$) times. The "while" loop on line 6 in Algorithm 5 takes time proportional to the number of times that an edge is "moved". Therefore, the total running time for the "while" loop is $O(m \log m)$. For lines 33-35, it takes $O(n)$ time to enumerate all nodes u in the equivalence set `Eq_nodes`$[v]$ and to associate `Set`$[u]$ with v. Therefore, the time complexity of our algorithm is $O(n + m \log m)$. At the beginning, each node is associated with two FDLLs `In[`v_i`]` and `Out[`v_i`]`. In each iteration of edge merging, there is one node removed from G', decreasing the number of edges m in G'. As a result, $O(n+m)$ space is required for processing the graph. Putting all these together, we have the following theorem:

THEOREM 4. *Algorithm 5 pre-processes the input graph in $O(n + m \log m)$ time and $O(n+m)$ space to answer any online bidirected Dyck-CFL-reachability query in $O(1)$ time.*

In practice, since the graphs in client analyses are typically sparse, Algorithm 5 performs in $O(n \log n)$ time. When the graphs are dense, we can use the naïve approach in Algorithm 2 for pre-processing. As a result, we have the following theorem:

THEOREM 5. *The bidirected Dyck-CFL-reachability problem on graphs can be pre-processed in $O(min\{kn^2, n + m \log m\})$ time and $O(n+m)$ space to answer any online query in $O(1)$ time.*

6. Application: Scaling an Alias Analysis for Java

This section describes a practical application of our results in speeding up an alias analysis for Java [37]. The goal of alias analysis is to determine if two pointer variables can point to the same memory location during program execution. The aliasing information obtained by an alias analysis is a prerequisite for many compiler optimizations and static analysis tools (*e.g.*, software verifiers [10], data race detectors [23], and static slicers [18]).

Many state-of-the-art alias analyses [31, 36, 37, 40] are formulated using CFL-reachability. Dyck-CFL-reachability plays an important role in these analyses. For instance, it has been used to model the matched references and dereferences in C [40], and has also been employed to describe the field loads and stores in Java [36, 37] to capture the "balanced-parentheses property" observed by Sridharan *et al.* [33]. Yuan and Eugster's recent work [39] further shows that Dyck-CFL-reachability on bidirected trees can be used to solve a simplified *context-insensitive pointers-to analysis* problem.

To demonstrate the practical applicability of our results, we leverage a recent demand-driven *context-sensitive* alias analysis for Java [37] formulated using CFL-reachability. Dyck-CFL-reachability is used to formulate its *context-insensitive* variant. The analysis is demand-driven in the sense that it solves the *single-source-single-target* Dyck-CFL-reachability problem. We show that our fast algorithms for *all-pairs* Dyck-CFL-reachability applies directly to this *context-insensitive* alias analysis.

6.1 Symbolic Points-to Graph

The underlying graph representation of the alias analysis is called the Symbolic Points-to Graph (SPG) [36, 37]. It extends the locally-resolved points-to graph representation [33] by introducing additional symbolic nodes as placeholders for abstract heap objects. The SPG contains three kinds of nodes: *variable nodes* $v \in \mathcal{V}$ representing variables, *allocation nodes* $o \in \mathcal{O}$ representing allocations for `new` expressions, and *symbolic nodes* $s \in \mathcal{S}$ representing abstract heap objects. It also consists of the following three types of edges:

- edges $v \to o_i \in \mathcal{V} \times \mathcal{O}$ to represent that variable v points to object o_i;
- edges $v \to s_i \in \mathcal{V} \times \mathcal{S}$ to represent that variable v points to an abstract heap object.
- edges $o_i \xrightarrow{f} o_j \in (\mathcal{O} \cup \mathcal{S}) \times \text{Fields} \times (\mathcal{O} \cup \mathcal{S})$ to represent that field f of o_i points to o_j.

A Java program's SPG is constructed in three steps. First, symbolic nodes are introduced for each procedure parameter, method invocation and field access. Second, the set of abstract heap locations $\mathcal{O} \cup \mathcal{S}$ that a variable may point to[1] is computed. The relevant points-to edges are inserted to the SPG. Third, the field access edges $o_i \xrightarrow{f} o_j$ are added with regard to field loads and stores. The SPG also includes the barred edges (*i.e.* $o_j \xrightarrow{\bar{f}} o_i$ edges) implicitly.

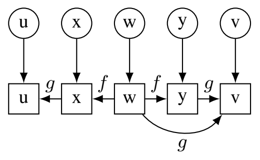
6.2 Context-Insensitive Alias Analysis

The context-insensitive alias analysis computes the aliasing relation over variables within a method. In the analysis, the method invocation edges (*i.e.*, entry and exit edges) are of no interest. Specifically, the *memory aliasing* between the allocation or symbolic nodes that variable nodes x and y may points-to indicates the aliasing relation between x and y. The memory alias relation defined in [37] over $(\mathcal{O} \cup \mathcal{S}) \times (\mathcal{O} \cup \mathcal{S})$ is described by the following context-free

[1] In the original work that using SPG [36, 37], the *flowsTo* edges are used. A *flowsTo* edge is obtained on the flow graph by computing a regular language reachability. An abstract heap object *flowsTo* a variable if it is in the points-to set of that variable.

```
x = w.f;
w.f = y;
u = x.g;
v = y.g;
v = w.g;
```

(a) A code snippet.



(b) Its SPG.

Figure 7. An example of alias analysis with the SPG.

grammar:

$$\begin{aligned} \mathit{memAlias} \quad \rightarrow \quad & \bar{f}_1 \ \mathit{memAlias} \ f_1 \mid \ldots \mid \bar{f}_k \ \mathit{memAlias} \ f_k \\ & \mid \mathit{memAlias} \ \mathit{memAlias} \mid \varepsilon \end{aligned}$$

Note that the alias analysis based on $\mathit{memAlias}$ reachability is a simplification of the alias reachability presented by Sridharan *et al.* [32, 33]. The field edges between abstract symbolic nodes in an SPG approximate the field loads and stores in the flow graph [32, 33]. The approximation may lead spurious aliasing as detailed by Xu *et al.* [36, Section 4]. However, the experimental results show that the overall performance is better than that proposed by Sridharan *et al.* [32, 33] in practice. The precision loss is insignificant enough compared to the performance gains.

EXAMPLE 5. *Consider the example in Figure 7. The Java code snippet (left) and its SPG (right) are shown. In the SPG, the boxes denote symbolic nodes, and the circles denote variable nodes. The reverse edges (*a.k.a. *barred edges) are omitted for brevity. Note that the Dyck-CFL-reachability formulation used in the client alias analysis represents the barred edges as the opening parentheses. There are two pairs of memAlias nodes: (x, y) and (u, v), because the realized strings of the two joining paths are "$\bar{f}f$" and "$\bar{g}\bar{f}fg$" respectively, which can be generated from the memAlias grammar. However, the node pair (x, v) is not memAlias because the realized strings of two possible joining paths are "$\bar{f}g$" and "$\bar{f}fg$", such that the parentheses along the paths are not properly matched.*

6.3 Applying Our Fast Algorithms

Since the CFL used to describe the context-insensitive memory aliasing is a Dyck-CFL with k kinds of parentheses, the two Dyck-CFL-reachability algorithms presented in this paper can be directly applied. Note also that this alias analysis is demand-driven in the sense that the original algorithm solves the "single-source-single-target" Dyck-CFL-reachability problem, because solving "all-pairs" reachability is considered computationally much more expensive in these analyses. Both our algorithms are intended to solve the "all-pairs" Dyck-CFL-reachability problem. Next, we show how our "all-pairs" algorithm performs in practice.

7. Empirical Evaluation

In this section, we compare the traditional CFL-reachability algorithm with our proposed algorithm for solving the *all-pairs* Dyck-CFL-reachability problem on graphs for standard, real-world Java benchmarks. The input graphs are generated from the *context-insensitive* alias analysis for Java described in Section 6. The results show that our algorithm outperforms the traditional CFL-reachability algorithm by several orders of magnitude.

7.1 Experimental Setup

Benchmark Selection. The benchmark suite used in our evaluation is the DaCapo suite [1]. We include the entire DaCapo-2006-10-MR2 suite which consists of 11 benchmarks with five additional large benchmarks form the DaCapo-9.12bach suite. Table 2 describes the benchmarks. For each benchmark, columns 2 and 3 list the numbers of methods and statements in intermediate representations of the underlying analysis infrastructure, respectively.

Graph Collection. We have used the same code as Xu *et al.* [36] and Yan *et al.* [37] to generate the Symbolic Points-to Graphs (SPGs). The analysis is built on top of the Soot program analysis framework for Java [34].

All benchmarks are processed with the nightly-build version[2] of Soot. To measure scalability, we use the latest release of JDK 1.6 (version 1.6u37) as the base analysis library for Soot. The five large benchmarks from DaCapo-9.12bach are processed with the help of Tamiflex [6] for reflection resolution.

Implementation. We implemented the proposed graph algorithm to compare with the traditional CFL-reachability algorithm. Both algorithms are implemented in C++ with extensive use of the Standard Template Library (STL). The FDLL data structure described in Section 5 is implemented using STL `unordered_map` and `list`. The underlying graphs are represented using adjacency lists implemented with FDLL.

Our code is compiled using gcc-4.6.3 with the "-O2" optimization flag. Both algorithms take the same SPG as input. Their outputs are verified to ensure the consistency and correctness . All experiments are conducted on a Dell Optiplex 780 machine with Intel Core2 Quad Q9650 CPU and 8 GB RAM, running Ubuntu-12.04.

7.2 Time and Memory Consumption

Table 2 shows the performance comparison of the two algorithms over our benchmark set. Column 4 and 5 list the numbers of nodes and edges in each SPG respectively. Column 6 lists the aliasing pair counts. Column 7 shows the number of different kinds of parentheses (*i.e.*, the size of each Dyck grammar) in each SPG. The remaining columns list the time and memory consumption of the traditional CFL-reachability algorithm versus that of our algorithm. We denote our algorithm as FAST-DYCK.

The results indicate that our algorithm significantly improves over the traditional CFL-reachability algorithm. We observe that the running time of our algorithm grows very slowly *w.r.t.* the growth of the number of nodes. For example, the running time of the CFL-reachability algorithm on "jython09" is 30 times over that on "xalan06". While, it is only 4 times for our algorithm on the same benchmarks. We also note that our algorithm consumes less memory than the traditional CFL-reachability algorithm.

7.3 Discussion

Understanding the Asymptotic Behavior. The SPGs generated from the benchmarks are very sparse — there are fewer edges than nodes across all SPGs. This is expected for the client alias analysis and is consistent with the information in the original papers [36, 37]. For sparse graphs with $m = O(n)$, the asymptotic complexity of our algorithm is $O(n \log n)$.

Moreover, in the traditional CFL-reachability algorithm, the grammar rules should be scanned for each iteration for inserting new summary edges. Specifically, for the Dyck language of size k, each edge popped from the worklist (line 7) in Algorithm 1 needs to be compared with the $O(k)$ rules in the given grammar. However, in our algorithms, the above is unnecessary. It takes *expected* $O(1)$ time to find a relevant edge labeled by a matched parenthesis in both our tree algorithm and graph algorithm due to the use of the FDLL.

Understanding the Memory Consumption. Both our algorithm and the traditional CFL-reachability algorithm demand moderate amount of memory for the client alias analysis. The memory cost for representing the input graphs in both algorithm is similar. The

[2] As of 2012-10-23.

Benchmark	#Methods	#Statements	SPG				Time		Memory	
			#Nodes	#Edges	#S-pair	#para	CFL	FAST-DYCK	CFL	FAST-DYCK
antlr	9904	170402	16735	13878	19385	1087	37.42	0.041	29.68	20.21
bloat	11818	206857	20320	16224	23080	1197	43.09	0.048	35.09	23.89
chart	25184	448984	44584	36329	50670	2948	253.06	0.119	76.75	52.02
eclipse	10447	181101	17527	14411	20335	1182	42.26	0.042	30.97	21.19
fop	23643	431569	39977	31515	45837	2724	219.53	0.101	67.99	46.08
hsqldb	9177	156265	15015	12693	17615	998	33.39	0.038	27.10	18.22
jython	12802	216068	21615	17381	24487	1240	49.57	0.052	37.20	25.32
luindex	9668	164598	16098	13336	18716	1071	35.15	0.040	28.64	19.45
lusearch	10196	175354	17003	14195	19911	1117	40.22	0.043	30.34	20.73
pmd	11167	193375	18167	14958	20843	1168	40.28	0.046	32.00	21.90
xalan	9181	155180	15030	12645	17608	996	32.93	0.038	26.93	18.21
batik	22938	404097	40273	32052	46225	2565	206.50	0.100	68.77	46.60
eclipse	18741	354818	37531	31889	54471	2221	366.39	0.103	70.82	44.54
jython	41518	642242	63516	49005	85552	2855	947.49	0.163	112.14	72.18
sunflow	22346	385873	39321	31339	45161	2484	196.23	0.096	67.22	45.57
tomcat	25123	441606	45966	37338	63414	3013	622.36	0.124	83.98	53.56

Table 2. Benchmarks and performance comparison: time in seconds and memory in MB.

traditional CFL-reachability algorithm needs more iterations to compute the graph closure than those in our algorithm, therefore, it requires more space as well.

Note that we only used the cubic CFL-reachability algorithm (without applying the Four Russians' Trick) in our comparison. The subcubic CFL-reachability algorithm demands non-trivial memory for storing the input graphs in our client application. For instance, given a medium-sized graph from our client analysis with 15000 nodes and 1000 parentheses, the subcubic algorithm needs about 26.2 GB memory to store the graph. It is an interesting topic to scale the subcubic CFL-reachability algorithm on real-world analysis.

Interpreting the Alias Analysis. In the *field-sensitive, context-insensitive alias analysis* for Java, the aliasing pairs are typically sparse. All benchmarks in our evaluation have $O(n)$ aliasing pairs (the #S-pair column in Table 2). This indicates that for real-world applications, most of the variables are not aliases. We have also observed from the experiments that the length of an aliasing path is small; almost all of the aliasing paths are simple paths without cycles. This observation is consistent with the state-of-the-art demand-driven analyses [33, 37, 40].

Demand-Driven vs. Exhausted. We now discuss perhaps one of the most interesting implications from our study. We have noticed that the performance of our all-pairs algorithm for *field-sensitive, context-insensitive* alias analysis is extremely fast. Such an exhaustive analysis with small time and memory cost is particularly suitable for application scenarios that need client analyses to be able to respond instantly, such as just-in-time (JIT) optimizations and interactive development environments (IDEs). Compared to demand-driven analyses, our exhaustive alias analysis can answer any query in $O(1)$ time.

In practice, the two algorithms introduced in this paper can be combined to achieve better performance. For a connected component of the SPG encountered during analysis, it is straightforward to check whether the component is a graph or a tree by counting the number of nodes and edges. Furthermore, one can design an effective analysis switching between our tree and graph algorithms to achieve even better performance.

8. Related Work

There are two strands of closely related work: CFL-reachability and alias analysis.

8.1 CFL-Reachability

The CFL-reachability framework was initially proposed by Yannakakis [38] for Datalog chain query evaluation. Later, it has been used to formulate interprocedural dataflow analysis [29] and many other program analysis problems [11, 19, 22, 25, 27, 31–33, 36, 40]. The central theme in the CFL formulations is that many program analyses have the balanced-parentheses property that can be captured by Dyck-CFL-reachability. The CFL and Dyck-CFL-reachability problems are also studied in the context of recursive state machines [4], visibly pushdown languages [3] and streaming XML [2]. Specially, when the recursive state machines are restricted to allow a constant number of entry/exit nodes per module, reachability is solvable in linear-time. CFL-reachability-based algorithms have cubic worst-case complexity, commonly known as "the cubic bottleneck in flow analysis" [14]. Finding more efficient algorithms for CFL-reachability is a difficult problem as any breakthrough in CFL-reachability may lead to faster algorithms for CFL parsing [27]. Chaudhuri showed that the well-known Four Russians' Trick [5] could be employed to speed up in the original CFL-reachability algorithm to immediately yield a subcubic algorithm [7]. Similar techniques were used in Rytter's work [30] for CFL parsing. Besides the subcubic result, Kodumal and Aiken [19] described a specialized set constraint reduction for Dyck-CFL-reachability on graphs and Yuan and Eugster [39] proposed an efficient Dyck-CFL-reachability algorithm on bidirected trees. Our paper introduces asymptotically faster algorithms for Dyck-CFL-reachability on both bidirected trees and graphs.

8.2 Alias Analysis

Alias analysis has been extensively studied in the literature. Its goal is to decide if two pointer variables may point to the same memory location during program execution. The problem is first formulated by Choi *et al.* [8] and Landi and Ryder [20]. Points-to analysis is recognized as a natural approach to alias analysis because aliasing relation can be decided by consulting the points-to sets of the two variables. We refer the reader to Hind's survey paper [16] on a large body of points-to analysis work.

Deciding aliasing is a computationally hard problem [17, 24]. Approximations must be made for any practical alias analysis. Various techniques have been proposed to scale alias analyses, such as improving the underlying points-to analysis [12, 13], making the analysis demand-driven [15, 37], and using novel data structures [35]. Specifically, Zheng and Rugina [40] proposed an approach based on CFL-reachability such that the aliasing information can be directed computed without first obtaining the points-to information. The client alias analysis [36, 37] in our evaluation is built on the same insight. We show in this paper that our fast algorithms help dramatically speed up the *context-insensitive* alias analysis.

9. Conclusion

In this paper, we have proposed two fast algorithms for solving Dyck-CFL-reachability on bidirected trees and graphs respectively. We have also applied our graph algorithm to a state-of-the-art alias analysis. The experimental results show that our graph algorithm help bring orders of magnitude speedup on real-world benchmarks. The key insight behind both our algorithms is that the reachability relation is an equivalence relation. However, this property does not hold over general trees or graphs. Some existing analyses (*e.g.*, dataflow analysis) do not seem to be good applications for the bidirected restriction. As a result, fast results for solving Dyck-CFL-reachability on general trees and graphs are still of both theoretical and practical interests.

Acknowledgments

We would like to thank Vu Le, Earl Barr, and the anonymous reviewers for helpful comments on earlier drafts of this paper. Special thanks to Dacong Yan and Guoqing Xu for providing us with their alias analysis implementation. The work described in this paper was significantly supported by the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK 415311 of General Research Fund and Project No. N_CUHK405/11 of the NSFC/RGC Joint Research Scheme), and partially supported by United States NSF grants 0917392 and 1117603.

References

[1] DaCapo benchmark suite. `http://dacapobench.org/`.

[2] R. Alur. Marrying words and trees. In *PODS*, pages 233–242, 2007.

[3] R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*, pages 202–211, 2004.

[4] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. W. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.*, 27(4):786–818, 2005.

[5] V. Arlazarov, E. Dinic, M. Kronrod, and I. Faradzev. On economic construction of the transitive closure of a directed graph. *Soviet Mathematics Doklady*, 11:1209–1210, 1970.

[6] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE*, pages 241–250, 2011.

[7] S. Chaudhuri. Subcubic algorithms for recursive state machines. In *POPL*, pages 159–169, 2008.

[8] J.-D. Choi, M. G. Burke, and P. R. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL*, pages 232–245, 1993.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3rd ed.)*. MIT Press, 2009.

[10] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI*, pages 57–68, 2002.

[11] C. Earl, I. Sergey, M. Might, and D. V. Horn. Introspective pushdown analysis of higher-order programs. In *ICFP*, pages 177–188, 2012.

[12] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *PLDI*, pages 85–96, 1998.

[13] B. Hardekopf and C. Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *PLDI*, pages 290–299, 2007.

[14] N. Heintze and D. A. McAllester. On the cubic bottleneck in subtyping and flow analysis. In *LICS*, pages 342–351, 1997.

[15] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *PLDI*, pages 254–263, 2001.

[16] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE*, pages 54–61, 2001.

[17] S. Horwitz. Precise flow-insensitive may-alias analysis is NP-Hard. *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, 1997.

[18] S. Horwitz, T. W. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[19] J. Kodumal and A. Aiken. The set constraint/CFL reachability connection in practice. In *PLDI*, pages 207–218, 2004.

[20] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *PLDI*, pages 235–248, 1992.

[21] D. Melski and T. W. Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *Theor. Comput. Sci.*, 248(1-2):29–98, 2000.

[22] P. Pratikakis, J. S. Foster, and M. Hicks. Existential label flow inference via CFL reachability. In *SAS*, pages 88–106, 2006.

[23] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Practical static race detection for C. *ACM Trans. Program. Lang. Syst.*, 33(1):3, 2011.

[24] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, 1994.

[25] J. Rehof and M. Fähndrich. Type-base flow analysis: from polymorphic subtyping to CFL-reachability. In *POPL*, pages 54–66, 2001.

[26] T. W. Reps. Shape analysis as a generalized path problem. In *PEPM*, pages 1–11, 1995.

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

[28] T. W. Reps, S. Horwitz, S. Sagiv, and G. Rosay. Speeding up slicing. In *SIGSOFT FSE*, pages 11–20, 1994.

[29] T. W. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[30] W. Rytter. Fast recognition of pushdown automaton and context-free languages. *Information and Control*, 67(1-3):12–22, 1985.

[31] L. Shang, X. Xie, and J. Xue. On-demand dynamic summary-based points-to analysis. In *CGO*, pages 264–274, 2012.

[32] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, pages 387–400, 2006.

[33] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. In *OOPSLA*, pages 59–76, 2005.

[34] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing java bytecode using the Soot framework: Is it feasible? In *CC*, pages 18–34, 2000.

[35] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144, 2004.

[36] G. Xu, A. Rountev, and M. Sridharan. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *ECOOP*, pages 98–122, 2009.

[37] D. Yan, G. H. Xu, and A. Rountev. Demand-driven context-sensitive alias analysis for Java. In *ISSTA*, pages 155–165, 2011.

[38] M. Yannakakis. Graph-theoretic methods in database theory. In *PODS*, pages 230–242, 1990.

[39] H. Yuan and P. T. Eugster. An efficient algorithm for solving the Dyck-CFL reachability problem on trees. In *ESOP*, pages 175–189, 2009.

[40] X. Zheng and R. Rugina. Demand-driven alias analysis for C. In *POPL*, pages 197–208, 2008.