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

Qirun Zhang  Michael R. Lyu  Hao Yuan  Zhendong Su
The Chinese University of Hong Kong  BOPU Technologies  University of California, Davis
{qrzhang, lyu}@cse.cuhk.edu.hk  hao@bopufund.com  su@cs.ucdavis.edu

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

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.

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/stores) 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-
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 Eugsiter [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 completeness and for reference.

### 2.1 Dyck-CFL-Reachability

Let $CFG = (\Sigma, N, P, S)$ be a context-free grammar with alphabet $\Sigma$, 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 $L(u, v)$ from the alphabet $\Sigma$ or $\epsilon$, each path $p = v_0, v_1, v_2, \ldots, v_m$ in $G$ realizes a string $R(p)$ over the alphabet $\Sigma$ by concatenating the edge labels in the path in order, i.e., $R(p) = \prod_{i=1}^{m} L(v_{i-1}, v_{i})$. 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 $v$?

## Table 1. Bidirected Dyck-CFL-reachability algorithms.

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

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 $\Sigma$ 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:

$$ S \to SS \mid a_1S\bar{a}_1 \mid \ldots \mid a_kS\bar{a}_k \mid \varepsilon $$

where $S$ is the start symbol and $\varepsilon$ is the empty string. Specially, we say node $v$ is Dyck-reachable from node $u$ iff there exists an $S\bar{S}$-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 $\varepsilon$, 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 $L(u, v) = \varepsilon$, $\bar{L}(v, u)$ must be $\varepsilon$;
- $\forall u, v \in V$, if $L(u, v) = a_i$, $\bar{L}(v, u)$ must be $\bar{a}_i$;
- $\forall u, v \in V$, if $L(u, v) = \bar{a}_i$, $\bar{L}(v, u)$ 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 $D$.

**Definition 2** (Dyck-CFL-Relation). Given a bidirected graph $G = (V, E)$, we call a binary relation $D$ on $V$ a Dyck-CFL-relation iff for all $(u, v) \in D$, $v$ 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...
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 $X(u, v)$ denotes the directed edge $(u, v)$ with label $L(u, v) = X$. 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 $X \rightarrow Y Z$, $X \rightarrow Y$ or $X \rightarrow \varepsilon$, where $X$ is nonterminal, $Y$ and $Z$ are terminals or nonterminals, and $\varepsilon$ 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 $X \rightarrow Y Z$ and edge $Y(i, j)$ all outgoing edges of node $j$ are considered. If there is an outgoing edge $Z(j, k)$, a new edge $X(i, k)$ 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 $D$ 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 $\forall a, b, c \in B$, $a \sim b \land b \sim c \implies a \sim c$.

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

- **Relation $D$ is reflexive**: This is because the start symbol $S$ in the Dyck grammar is nullable (i.e., it generates the empty string $\varepsilon$). Therefore, $(u, u) \in D$ for all $u \in V$.

- **Relation $D$ 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., $(u, v) \in D$), $u$ is also Dyck-reachable from $v$ (i.e., $(v, u) \in D$).

- **Relation $D$ 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., $(u, v) \in D$) and $w$ is Dyck-reachable from $v$ (i.e., $(v, w) \in D$), $w$ is Dyck-reachable from $u$ (i.e., $(u, w) \in D$). It is immediate that the realized string $R(p_1)$ for any path $p_1$ connecting nodes $u$ 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 $S \rightarrow SS$. Hence, the path $p_1p_2$ from node $u$ to $w$ is also a Dyck-path.

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 $D$. Let $a_i(u, v)$ 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

---

**Algorithm 1:** The standard CFL-reachability algorithm.

**Input**: Edge-labeled directed graph $G = (V, E)$, normalized CFG $G = (\Sigma, N, P, S)$

**Output**: the set $\{(i, j) \mid S(i, j) \in G\}$

1. add $E$ to $W$
2. foreach production $X \rightarrow e \in P$ do
   3. foreach node $v \in V$ do
      4. if $X(v, v) \notin E$ then
         5. add $X(v, v)$ to $E$ and to $W$
   6. while $W \neq \emptyset$ do
      7. select and remove an edge $Y(i, j)$ from $W$
      8. foreach production $X \rightarrow Y \in P$ do
         9. if $X(i, j) \notin E$ then
            10. add $X(i, j)$ to $E$ and to $W$
      11. foreach production $X \rightarrow YZ \in P$ do
         12. foreach outgoing edge $Z(j, k)$ from node $j$ do
            13. if $X(i, k) \notin E$ then
               14. add $X(i, k)$ to $E$ and to $W$
      15. foreach production $X \rightarrow ZY \in P$ do
         16. foreach incoming edge $Z(k, i)$ to node $i$ do
            17. if $X(k, j) \notin E$ then
               18. add $X(k, j)$ to $E$ and to $W$

---

**Lemma 1.** The Dyck-CFL-relation $D$ 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 $D$ 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.
z such that \( a_i(x, z) = a_i(y, z) \). For example, in Figure 1(b), we have \( a_1(1, 2) = a_1(3, 2) \). 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 \( a_i(u, v) \in E' \) iff \( a_i(u, v) \in E \) 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., \( \exists a_i(x, z) = a_i(y, z) \) 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 \( \text{Eq} \_\text{nodes}[v] \) denotes the equivalence set of node \( v \) and \( \text{Set}[v] \) denotes the equivalence set number that node \( v \) belongs to. The procedure \( \text{HAS-SAME-IN}(v) \) traverses all incoming neighbors of node \( v \), and returns true if there exist two nodes \( u_1 \) and \( u_2 \) such that \( a_i(u_1, v) = a_i(u_2, v) \). In Algorithm 2, line 21 transforms the given graph \( G \) to \( G' \), and lines 2-5 initialize \( W \) and \( \text{Eq} \_\text{nodes}[v] \). 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 \( W \) will 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 \( \text{HAS-SAME-IN}(v) \) 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 space \( O(n) \) 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 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 \( y \).

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

We detail the description of procedure \text{Add}(v, e) \) to illustrate the idea of collapsing nodes in relation \( D \). We use \text{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 a 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 \( 2 \) and edge \((2, 3)\) are processed by \text{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 \( \mathcal{L}(3, 4) = a_3 \) and node 4 is Dyck-reachable from node 2 (i.e., \((2, 4) \in \mathcal{D}) \). 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 \( \text{curset} \) in STRATIFIED-SETS. This new set is then linked with the set returned by \text{Find}(u) \) on the upper layer as described by lines 2-5. If the edge label is a closing parenthesis \( a_i \in A \), 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 \( \text{Up}[\text{Find}(u)][a_i] \) 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 \( \text{curset} \) and link the set returned by \text{Find}(u) \) to the new set from the upper layer.

Note that the \( \uparrow \) array used in Procedure 3 is indeed a map: \( \text{Num} \rightarrow A \rightarrow \text{Num} \), where \( \text{Num} \) 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 \( \uparrow\text{Set}[s] \), where \( s \in \text{Num} \). The time taken to search such \( O(k) \) edges depends on the actual implementation of the \( \uparrow \) array. For example, if the \( \uparrow \) array stores such \( O(k) \) edges for each set \( s \) using a binary search tree, the lookup for an \( a_i \) edge in \( \uparrow\text{Set}[s] \) takes \( O(\log k) \) time as mentioned in Yuan and Eugster’s work [39]. In our algorithm, we implement the \( \uparrow \) 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,

<table>
<thead>
<tr>
<th>Procedure 3: Add((v, e)) to add a node ( v ) to STRATIFIED-SETS according to the directed edge ( e = (u, v) ).</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 if ( \mathcal{L}(u, v) \in A ) then | ( \text{Set}[v] = \text{curset} |</td>
</tr>
</tbody>
</table>
| 2 let \( a_i = \mathcal{L}(u, v) \)
| 3 \( \text{Set}[v] = \text{curset} \)
| 4 \( \uparrow\text{Set}[\text{curset}][a_i] = \text{Find}(u) \)
| 5 \( \text{curset}++ \)
| 6 if \( \mathcal{L}(u, v) \in \bar{A} \) then \| \( \text{Set}[\text{curset}][a_i] = \text{Find}(u) \)
| 7 let \( a_i = \mathcal{L}(u, v) \)
| 8 if \( \uparrow\text{Set}[\text{curset}][a_i] \) does not exist then \| \( \text{curset}++ \)
| 9 \( \text{Set}[v] = \text{curset} \)
| 10 \( \uparrow\text{Set}[\text{curset}][a_i] = \text{Find}(u) \)
| 11 \( \text{curset}++ \)
| 12 else \| \( \text{Set}[v] = \uparrow\text{Set}[\text{curset}][a_i] \)

where \( m = n - 1 \). Therefore, the time complexity of Procedure 3 is \( O(1) \), and the space complexity of the \( \uparrow \) 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 \( (u, v) \in D \). The overall algorithm takes two steps:

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

2. (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 \text{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 \text{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 \( \uparrow \) 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 \( \uparrow \) 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 \text{Find}(u) \) and \text{Find}(v) \) are the same. Putting everything together, we have the following theorem.
Algorithm 4: Dyck-CFL-reachability algorithm on trees.

```plaintext
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. \( \text{visited}[v] = \text{false} \)
   4. \( \text{Set}[v] = \emptyset \)
   5. \( \text{stack}.\text{push}(\text{a leaf node } v) \)
   6. \( \text{curset} = \emptyset \)
   7. \( \text{Init}(v) \)
   8. while stack is not empty do
      9. \( v = \text{stack}.\text{pop} \)
      10. if not visited\( [v] \) then
          11. \( \text{visited}[v] = \text{true} \)
          12. foreach unvisited neighbor \( u \) of \( v \) do
              13. \( \text{stack}.\text{push}(u) \)
              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 \( D \) shares the same equivalence properties as for trees. For bidirected graphs, we utilize the idea of edge merging to collapse the node pairs \((u, v) \in D\). 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 \( a_i(u, v) \in E' \) iff \( a_i(u, v) \in E \) 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 \( D \). As in the naïve approach, \( E' \) may possibly contain multiple edges due to the collapsing between nodes.

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 \( a_1(5, 6) = a_1(7, 6) \). Since nodes 5 and 7 are Dyck-reachable from each other, the two edges \( a_1(5, 6) \) and \( a_1(7, 6) \) 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 if 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 \( a_i(x, z) = a_j(y, z) \) and \( a_i \neq a_j \).
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 \( tp \) array can be implemented using FDLL to support the expected \( O(1) \) time lookup. Recall that the \( tp \) array is indeed a map: \( \{\text{Num} \rightarrow A\} \rightarrow \text{Num} \). For each set \( s \in \text{Num} \), 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 \( U[s, z] \) 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 \( A_{\text{in}}[v, z] \) to store all \( v \)'s incoming neighbors. We denote the size of \( A_{\text{in}}[v, z] \) as \( \text{size}(v, z) \). Node \( v \) merges edge \( ai \) iff \( \text{size}(v, z) > 1 \). We represent \( \text{size}(v, z) \) and \( A_{\text{in}}[v, z] \) similarly. Finally, the worklist \( \text{FDLL}_{\text{w}} \) is also represented using an FDLL.

Algorithm 5 uses the array \( D[v] \) to represent the total degree of node \( v \), which is initialized to \( v \)'s degree in the original graph. Notations \( \text{eq}_v[x][v] \) and \( \text{set}_v[v] \) 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 \( \text{eq}_v[x][v] \) is initialized with node \( v \) and the \( \text{FDLL}_{\text{w}} \) is initialized with \( v \), indicating node \( v \) merges \( ai \). The \( \text{FDLL}_{\text{w}} \) 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 \( z, j \) from the \( \text{FDLL}_{\text{w}} \), 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 \( ai \) (i.e., \( \text{size}(v, z) < 2 \), \( v, j \) is removed from the \( \text{FDLL}_{\text{w}} \). Similarly, when node \( v \) becomes a merging node that merges \( ai \) edges, \( v, j \) is inserted into the \( \text{FDLL}_{\text{w}} \). Note
Algorithm 5: Dyck-CFL-reachability algorithm on graphs.

**Input**: Edge-labeled bidirected graph $G = (V, E)$

**Output**: $\text{Set}[u]$ for all $u \in V$

1. transform the input graph $G$ to $G' = (V', E')$

2. foreach $v \in V'$ do

   3. $\text{Eq}_{\text{nodes}}[v] = \{v\}$

   4. foreach $a_i \in A_{\text{in}}[v]$ do

      5. add $v, a_i$ to $\text{FDLL}_w$ if $\text{In}[v, i, \text{size}()] > 1$

3. while $\text{FDLL}_w \neq \emptyset$ do

   6. let $z_i$ be the front item from $\text{FDLL}_w$

   7. let $x$ and $y$ be two front nodes from $\text{In}[z_i, i, \text{size}()]$

   8. let $x$ denote the node such that $\hat{D}[x] > \hat{D}[y]$

   9. $\hat{D}[x] = \hat{D}[x] + \hat{D}[y]$

10. Eq$_{\text{nodes}}[x] = \text{Eq}_{\text{nodes}}[y] \cup \text{Eq}_{\text{nodes}}[x]$

11. foreach $a_i \in A_{\text{in}}[y]$ do

   12. if $a_i(y, y) \in E'$ then

      13. a) if $a_i(x, x) \in E'$ then

         14. add $a_i(x, x)$ to $E'$

         15. add $x, a_i$ to $\text{FDLL}_w$ if $x, a_i \notin \text{FDLL}_w$ and

         16. $\text{In}[x, i, \text{size}()] > 1$

         17. remove $a_i(y, y)$ from $E'$

         18. remove $y, a_i$ from $\text{FDLL}_w$ if $y, a_i \in \text{FDLL}_w$ and

         19. $\text{In}[y, i, \text{size}()] < 2$

      b) add $a_i(x, y)$ to $E'$

      20. remove $y$ from $V'$

   19. foreach $a_i \in A_{\text{in}}[y]$ do

   20. if $a_i(w, x) \notin E'$ then

      21. add $a_i(w, x)$ to $E'$

      22. add $x, a_i$ to $\text{FDLL}_w$ if $x, a_i \notin \text{FDLL}_w$ and

      23. $\text{In}[x, i, \text{size}()] > 1$

      24. remove $a_i(w, y)$ from $E'$

      25. remove $y, a_i$ from $\text{FDLL}_w$ if $y, a_i \in \text{FDLL}_w$ and

      26. $\text{In}[y, i, \text{size}()] < 2$

   26. foreach $a_i \in A_{\text{in}}[w]$ do

   27. if $a_i(w, w) \notin E'$ then

      28. add $a_i(w, w)$ to $E'$

      29. remove $a_i(y, w)$ from $E'$

      30. remove $w, a_i$ from $\text{FDLL}_w$ if $w, a_i \in \text{FDLL}_w$ and

      31. $\text{In}[w, i, \text{size}()] < 2$

   32. remove $y$ from $V'$

33. foreach $v \in V'$ do

34. if $v \in \text{Eq}_{\text{nodes}}[v]$ do

35. $\text{Set}[u] = v$

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-paths 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 $R(p) = L(v_1, v_2) \bar{L}(v_3, v_4) = \bar{a}_iS_{a_i}$. Because the graph is bidirected, we have $L(v_3, v_4) = \bar{L}(v_1, v_2) = a_i$. Nodes $v_2$ 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:

- $S \rightarrow a_iS_{a_i}$ 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 $R(v_2 \ldots v_3) = \bar{S}$ 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 $p = v_1v_2 \ldots v_3$, 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, ..., v_2 and path v_2, ..., 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 \( \hat{D}[y] \) 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 \( \hat{D}[x] \geq \hat{D}[y] \). Combining the analysis of the two lines, the following lemma holds:

**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., \( 2m = \sum_{v \in V} \hat{D}[v] \). Due to the duplicated edges, \( \hat{D}[v] \) may decrease during edge merging. Similarly, \( \hat{n} \) denotes the total number of edges in graph \( G' \) w.r.t. \( \hat{D}[v] \) that admits duplicated edges, i.e., \( \hat{n} = \sum_{v \in V} \hat{D}[v] \). 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 \( \hat{D}[v] \leq \hat{n} \leq 2m \), 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 \( \log \hat{D}[x] + \log \hat{D}[z] \leq 2 \log 2m \) 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 \( Eq\text{-nodes}[v] \) 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 \( \Xi_{v, \text{left}} \) and \( \Xi_{v, \text{right}} \). 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 naive 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 \( v \in V \) representing variables, allocation nodes \( a \in O \) representing allocations for new expressions, and symbolic nodes \( s \in S \) representing abstract heap objects. It also consists of the following three types of edges:

- edges \( v \to o_i \in V \times O \) to represent that variable \( v \) points to object \( o_i \);
- edges \( v \to s_i \in V \times S \) to represent that \( v \) points to an abstract heap object.
- edges \( o_i \to o_j \in (O \cup S) \times Fields \times (O \cup S) \) 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 \( O \cup S \) that a variable may point to \(^1\) is computed. The relevant points-to edges are inserted to the SPG. Third, the field access edges \( o_i \to o_j \) are added with regard to field loads and stores. The SPG also includes the barred edges (i.e., \( o_j \to o_i \)) explicitly.

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 \((O \cup S) \times (O \cup S)\) 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.
grammar:

\[
\text{memAlias} \rightarrow \hat{f}_1 \text{memAlias} f_1 | \ldots | \hat{f}_k \text{memAlias} f_k \\
| \text{memAlias memAlias} | \varepsilon
\]

Note that the alias analysis based on \text{memAlias} reachability is a simplification of the alias reachability presented by Sridharan \textit{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 discussed in Section 5. However, the experimental results show that the overall performance is better than that proposed by Sridharan \textit{et al.} [32, 33] in practice. The precision loss is insignificant enough compared to the performance gains.

\textbf{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 (i.e., the size of each Dyck grammar) in each SPG. The remaining columns list 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 Dyck-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.}

\section{Discussion}
\textbf{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 \textit{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.

\textbf{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.

\section{Empirical Evaluation}
In this section, we compare the traditional CFL-reachability algorithm with our proposed algorithm for solving the \textit{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.

\subsection{Experimental Setup}
\textbf{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.

\textbf{Graph Collection.} We have used the same code as Xu \textit{et al.} [36] and Yan \textit{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.

\textbf{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.

\subsection{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.
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.

---

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#Methods</th>
<th>#Statements</th>
<th>SPG</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr</td>
<td>9904</td>
<td>170402</td>
<td></td>
<td>16735</td>
<td>25939</td>
</tr>
<tr>
<td>blount</td>
<td>11818</td>
<td>206857</td>
<td></td>
<td>20320</td>
<td>16280</td>
</tr>
<tr>
<td>chart</td>
<td>25184</td>
<td>448984</td>
<td></td>
<td>44584</td>
<td>83629</td>
</tr>
<tr>
<td>eclipse</td>
<td>10447</td>
<td>181101</td>
<td></td>
<td>17527</td>
<td>20335</td>
</tr>
<tr>
<td>eup</td>
<td>23643</td>
<td>431569</td>
<td></td>
<td>39977</td>
<td>55151</td>
</tr>
<tr>
<td>hsqldb</td>
<td>9177</td>
<td>156256</td>
<td></td>
<td>15015</td>
<td>12693</td>
</tr>
<tr>
<td>jython</td>
<td>12802</td>
<td>216068</td>
<td></td>
<td>21615</td>
<td>17381</td>
</tr>
<tr>
<td>luindex</td>
<td>9668</td>
<td>164598</td>
<td></td>
<td>16098</td>
<td>13336</td>
</tr>
<tr>
<td>lusearch</td>
<td>10196</td>
<td>175354</td>
<td></td>
<td>17003</td>
<td>14195</td>
</tr>
<tr>
<td>pmd</td>
<td>11167</td>
<td>193375</td>
<td></td>
<td>18167</td>
<td>14958</td>
</tr>
<tr>
<td>xalan</td>
<td>9181</td>
<td>155180</td>
<td></td>
<td>15030</td>
<td>12645</td>
</tr>
<tr>
<td>batik</td>
<td>22938</td>
<td>404097</td>
<td></td>
<td>40273</td>
<td>32052</td>
</tr>
<tr>
<td>eclipse</td>
<td>18741</td>
<td>354818</td>
<td></td>
<td>37531</td>
<td>31889</td>
</tr>
<tr>
<td>jython</td>
<td>41518</td>
<td>642242</td>
<td></td>
<td>63516</td>
<td>49005</td>
</tr>
<tr>
<td>sunflow</td>
<td>22346</td>
<td>385873</td>
<td></td>
<td>39321</td>
<td>31339</td>
</tr>
<tr>
<td>tomcat</td>
<td>25123</td>
<td>441606</td>
<td></td>
<td>45966</td>
<td>37338</td>
</tr>
</tbody>
</table>

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 ap-
prouch 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