An Efficient Algorithm for Solving the Dyck-CFL Reachability Problem on Trees

Hao Yuan and Patrick Eugster

Department of Computer Science, Purdue University
{yuan3,peugster}@cs.purdue.edu

Abstract. The context-free language (CFL) reachability problem is well known and studied in computer science, as a fundamental problem underlying many important static analyses such as points-to-analysis. Solving the CFL reachability problem in the general case is very hard. Popular solutions resorting to a graph traversal exhibit a time complexity of $O(k^3n^3)$ for a grammar of size k. For Dyck CFLs, a particular class of CFLs, this complexity can be reduced to $O(kn^3)$. Only recently the first subcubic algorithm was proposed by Chaudhuri, dividing the complexity of predating solutions by a factor of $\log n$.

In this paper we propose an effective algorithm for solving the CFL reachability problem for Dyck languages when the considered graph is a bidirected tree with specific constraints. Our solution pre-processes the graph in $O(n \log n \log k)$ time in a space of $O(n \log n)$, after which any Dyck-CFL reachability query can be answered in $O(1)$ time, while a naïve online algorithm will require $O(n)$ time to answer a query or require $O(n^2)$ to store the pre-computed results for all pairs of nodes.

1 Introduction

In this paper, we study a well-known problem called the *context-free language reachability* (CFL reachability) problem [1]. This problem is of particular interest in the context of static analyses, such as type-based flow analysis [2] or points-to analysis [3,4]. Consider a directed graph $G = (V, E)$ with n vertices and a context-free grammar, each directed edge $(u, v) \in E$ is labeled by a terminal symbol $\mathcal{L}(u, v)$ from Σ. For any path $p = v_0v_1v_2 \dots v_m$ (which can have loops), we say that this path realizes a string $R(p)$ which is the concatenation of the symbols on the path, i.e., $R(p) = \mathcal{L}(v_0, v_1)\mathcal{L}(v_1, v_2)\mathcal{L}(v_2, v_3) \dots \mathcal{L}(v_{m-1}, v_m)$.

The CFL reachability problem has several facets:

- *Source and destination specified.* Given a source node and a destination node, is there a path p connecting them, whose corresponding string $R(p)$ can be generated by the context-free grammar?
- *Single source.* Given a source node u, answer the questions: for each node v, is there a path p connecting u and v, whose corresponding string $R(p)$ can be generated by the context-free grammar?

G. Castagna (Ed.): ESOP 2009, LNCS 5502, pp. 175–189, 2009.

- *Single destination.* Given a destination node v, answer the questions: for each node u, is there a path p connecting u and v, whose corresponding string $R(p)$ can be generated by the context-free grammar?
- *All pair queries.* Answer for every pair of nodes u and v: is there a path p connecting u and v, whose corresponding string $R(p)$ can be generated by the context-free grammar?

A context-free language is called *Dyck* language if it is used to generate matched parentheses. Basically, it has the following form: a size k (i.e., k kinds of parentheses) Dyck language can be defined by

$$S \longrightarrow \epsilon \mid S\,S \mid (_1\,S\,)_1 \mid (_2\,S\,)_2 \mid \cdots \mid (_k\,S\,)_k$$

where S is the start symbol, and ϵ is the empty string.

When the context-free language is a Dyck language, the CFL reachability problem on that language is referred to as the *Dyck-CFL reachability* problem. In this paper we give an efficient algorithm for solving this problem when the given digraph is in a *specific bidirected tree structure*, as detailed in Sections 3 and 4. A bidirected tree corresponds to some situation in which an object flow (sub-)graph only involves objects of non-recursive types.

In short, our algorithm pre-processes the specific tree graph in $O(n \log n \log k)$ time within $O(n \log n)$ space, which allows for a Dyck-CFL reachability query for any pair of nodes to be performed in $O(1)$ time. Note that a naïve online algorithm will in contrast take $O(n)$ time to answer a query online, or need $O(n^2)$ space to store the pre-computed results for all pairs of nodes.

The speedups in the pre-processing, which are central to the efficiency of our algorithm, are made possible by the following two key ideas:

1. We build linear data structures for a *pivot node* x to answer queries on paths leading through x. To that end, we construct *tries* [5] of size n representing strings of unmatched parentheses for the path from any node to x in a single tree walk using $O(n \log k)$ time.
2. To handle the case where a given path does not lead through x, we apply the above scheme recursively for the subtrees obtained by removing x; x is chosen to be a *centroid node* of the tree [6,7].

Roadmap. This paper is organized as follows. Section 2 covers the related work on the CFL reachability problem. The motivation of studying the Dyck-CFL reachability problem on trees is given in Section 3, which focuses on the application to points-to analysis. Our algorithm to solve the problem efficiently is described in Section 5, after preliminary definitions and lemmas have been provided in Section 4. Finally, Section 6 concludes with final remarks.

2 Background and Related Work

The CFL reachability problem was first formulated by Yannakakis [8] in his work to solve the datalog chain query evaluation problem in the context of database

theory. Since then, it is widely used in the area of program analysis: many program analysis problems can be reduced to it, e.g., interprocedural data flow analysis [9], shape analysis [10], points-to analysis [3,4], alias analysis [11] and type-based flow analysis [2]. For more applications, see the survey paper of [1].

In the work of Yanakakis [8], an $O(k^3n^3)$ algorithm was given to solve the CFL reachability problem, with k the size of the grammar (usually considered to be constant) and n the number of nodes (typically objects in an object graph). Later, Reps gave a very popular iterative algorithm [10], which is still in $O(k^3n^3)$. Since many program analysis problem can be reduced to the CFL reachability problem, it is important to see if we can break the cubic bottleneck.

Recently, Chaudhuri gave the first subcubic time algorithm for the CFL reachability problem [12]. His algorithm runs in $O(k^3n^3/\log n)$ time by using the well-known Four Russians' Trick [13] to speed up set operations under the Random Access Machine model. Similar techniques were used in Rytter's work [14,15]. A closely related problem, the reachability problem on *recursive state machines*, was also studied in [12]. It can be shown that the reachability problem on recursive state machines can be reduced to the CFL reachability problem, and vice versa [16].

It is possible to improve the running time of the CFL reachability algorithm for special cases [1]. One direction is to design algorithm for specific grammars. For example, if the context-free language under consideration is the *Dyck* language, then the general $O(k^3n^3)$ time bound can be reduced to $O(kn^3)$ by a refined analysis [17]; in the type-based flow analysis work of Fähndrich [2], an $O(n^3)$ algorithm is designed to handle the special grammar used in his reduction. The Dyck language captures the nature of the call/return structures of a program execution path, and hence constitutes an important context-free language that is studied within the context of the CFL reachability problem [4,1,12]. In the work of [3], a Dyck language was used to model the `PutField` and `GetField` operations in the field-sensitive flow-insensitive points-to analysis for Java. Dyck languages are also studied in the context of visibly pushdown languages [18] and streaming XML [19].

The other direction is to design algorithms for special graph classes. When the directed graph is a chain, the CFL reachability problem can be viewed as the CFL-*recognition* problem, which has an algorithm running in $O(BM(n))$ time given by Valiant [20], where $BM(n)$ is the upper bound to solve the matrix multiplication problem for $n \times n$ boolean matrices. The best such upper bound known is $O(n^{2.376})$. Yannakakis [8] noted that Valiant's algorithm can also be applied to the case when the graph is a directed acyclic graph. In this work, we will consider the special case when the graph is in the form of a bidirected tree.

3 Motivation: Points-to Analysis

In this section, we present the motivation for the Dyck-CFL reachability problem on trees; it is based on the application of the CFL reachability problem to *field-sensitive flow-insensitive points-to analysis* [3].

3.1 Points-to Analysis via Dyck-CFL Reachability

In points-to analysis, we want to compute for each pointer x, the points-to function

$$pt(x) = \{\text{objects allocated in the heap that are possibly pointed by } x\}.$$

Throughout this paper, we will use the other notation $ft(o)$, the flow-to function, to represent the set of pointers that will possibly point to the object o, i.e.

$$ft(o) = \{x \mid o \in \mathrm{pt}(x)\}.$$

The underlying model discussed is field-sensitive and flow-insensitive. Field-sensitive means that we take the fields of the classes into consideration. Flow-insensitivity entails that we do not consider the execution order of the codes. Figure 1 gives an example illustrating the basic concepts of flow analysis.

The scenario depicted in the figure is as follows. For the statement `x=new Object();` we allocate a new object $\mathtt{o}_1$ in the heap, and then assign it to the pointer `x` by a directed edge labeled with `new`. Similarly, for the second statement, we make a `new` edge from $\mathtt{o}_2$ to pointer `z`. For the assignment statement `w=x;`, we add an `assign` edge to the graph. One can see that object $\mathtt{o}_1$ may flow to pointer `w` through the execution of the first and third statements, this is reflected on the graph by a path from $\mathtt{o}_1$ to `w`. The last two statements demonstrate the field-sensitive analysis, i.e., we add two edges `GetField[f]` and `PutField[f]` accordingly. Object $\mathtt{o}_1$ is only considered possibly flowing to the field `f` of `v` rather than flowing to `v` even if `v` is reachable from $\mathtt{o}_1$ in the graph. The reason is that, the path connecting $\mathtt{o}_1$ and `v` is not closed: there should be a `PutField[f]` before `GetField[f]` to make the object flow to `v` through the field `f`. It is not difficult to see that $\mathtt{o}_2$ indeed can flow to `v` through a path

$$\mathtt{o}_2 \xrightarrow{\texttt{new}} z \xrightarrow{\texttt{PutField[f]}} \mathtt{w} \xrightarrow{\texttt{GetField[f]}} \mathtt{v}$$

If we consider a pair of `PutField[f]` and `GetField[f]` operations as a kind of parenthesis indexed by the field `f`, and consider `assign` and `new` as ϵ, then the points-to analysis can be formulated by the reachability problem under the following Dyck Language:

$$\begin{aligned} S \longrightarrow \epsilon \mid S\,S \mid\ & \texttt{PutField[f]}\, S\, \texttt{GetField[f]} \\ \mid\ & \texttt{PutField[g]}\, S\, \texttt{GetField[g]} \\ \mid\ & \texttt{PutField[h]}\, S\, \texttt{GetField[h]} \\ \mid\ & \cdots \end{aligned}$$

where `f`, `g` and `h` are the available fields. An object `o` can flow to a pointer `x` if and only if there is a path p connecting `o` to `x` such that the corresponding $R(p)$ can be generated by the above grammar. In this points-to definition, we do not consider the "may alias" cases (see [3] for more details).

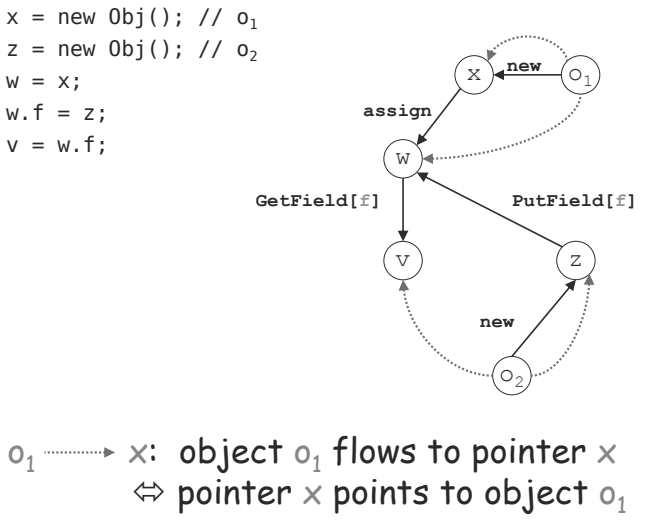

Fig. 1. An example of field-sensitive points-to analysis (modified from the talk slides of [3]). The black edges are generated based on the statements. The dotted blue edges are used to illustrate the flows-to/points-to relationship.

3.2 Special Tree Structure Case

If the corresponding directed graph for a set of program statements forms a tree[1] structure, then we can take advantage of the tree structure to provide a better algorithm for solving the Dyck-CFL reachability problem.

For any two neighbor nodes u and v on the tree, we may have both directed edge (u, v) and (v, u) (i.e., the tree digraph can actually have loops!). In such a case, we restrict the labels on them to satisfy the following constraint: if there are both (u, v) and (v, u) on the tree, then either they are both labeled by ϵ, or they are labeled by a pair of parentheses (or `PutField`/`GetField`) of the same index. For example, if (u, v) is labeled by `PutField[g]`, then (v, u) must be labeled by `GetField[g]`. This constraint will enable us to have a fast algorithm to solve the Dyck-CFL reachability problem on a tree.

The constraint corresponds to a special case of instances of non-recursive types. In the a special scenario, if one has a statement `x=y.f`, then the only other interaction between `x` and `y` about the field f must be `y.f=x`. This constraint ensures that for any path connecting two nodes, there is no reason to go through any loop, because the string labeled by the loop must be well matched. Note that in this case, a special constraint is made: the interaction between `x` and `y` must go trough a single field.

Such a constraint and the tree-structure requirement do not imply that our algorithm is restricted to non-tree graphs and languages which *prohibit* recursive

[1] Throughout this paper, we use the term "tree" to represent the special bidirected tree graph.

types; it is easy to conceive an analysis which switches between our algorithm and a "classic" more complete and more complex one based on the objects and flow graph encountered. Our algorithm is then applied as a "fast path", possibly to a subgraph of an object flow graph only.

4 Preliminaries

Before delving into our algorithm, we present some preliminary definitions and lemmas. Straightforward proofs are omitted, and will be given in the full version of this paper.

4.1 Problem Definition

Given a bidirected tree $T = (V, E)$, for every neighboring node pair u and v, at least one of the edges $\{(u, v), (v, u)\}$ exists. Each directed edge $(u, v) \in E$ is assigned a label $\mathcal{L}(u, v)$, which is a symbol in either $A = \{a_1, a_2, \cdots, a_k\}$ or $\bar{A} = \{\bar{a}_1, \bar{a}_2, \cdots, \bar{a}_k\}$. Here, A represents the set of opening parentheses, and $\bar{A}$ represents the set of closing parentheses. For any $1 \leq i \leq k$, we call the two symbols a_i and $\bar{a}_i$ a pair of *matched* parentheses.

Let $\mathscr{A}$ be $A \cup \bar{A}$. For any $x \in \mathscr{A}$, we define

$$flip(x) = \begin{cases} \bar{a}_i & \text{if } x = a_i \text{ for some } i, \\ a_i & \text{if } x = \bar{a}_i \text{ for some } i. \end{cases}$$

Note that we will also use $\bar{x}$ to denote $flip(x)$. For any directed edge $(u, v) \in E$, we assume that $\mathcal{L}(v, u) = flip(\mathcal{L}(u, v))$ if (v, u) exists.

A Dyck language $L(G)$ of size k is defined by the following context free grammar G:

- The only non-terminal symbol is S, which is also served as the start symbol.
- The set of terminal symbols is $\mathscr{A} = A \cup \bar{A}$.
- The production rules are

$$S \longrightarrow \epsilon \mid S\,S \mid a_1\,S\,\bar{a}_1 \mid a_2\,S\,\bar{a}_2 \mid \cdots \mid a_k\,S\,\bar{a}_k,$$

  where ϵ represents the empty string.

For any path $p = v_0 v_1 v_2 \ldots v_m$ (which can have loops) in the tree, we use $R(p)$ to denote the string that is realized by p. More specifically, we define $R(p)$ to be $R(p) = \mathcal{L}(v_0, v_1)\mathcal{L}(v_1, v_2)\mathcal{L}(v_2, v_3)\mathcal{L}(v_3, v_4)\ldots\mathcal{L}(v_{m-1}, v_m)$, i.e., the concatenation string of the symbols along the path.

The Dyck-CFL reachability problem asks the following query $Q(u, v)$: for any two nodes u and v, is there a path p connecting u and v, such that $R(p)$ can be produced from the grammar G?

4.2 Basic Definitions

For any string s, we call it an S-string if it can be produced from the grammar G by starting from the non-terminal S. Similarly, for any path p, if its realized string $R(p)$ is an *S-string*, then we call the path p an *S-path*. Since S is by default the starting non-terminal symbol in grammar G, therefore, the Dyck-CFL reachability problem can be formulated as: for any two nodes u and v, is there an S-path connecting u and v?

Definition 1. *We define a function $R'(s)$ for a string s to be the string generated by repeatedly eliminating matched parentheses from s. Formally, $R'(s)$ is generated based on the following elimination process: for any substring of s, if it is an S-string, then we remove the substring from s and repeat the process on the resulting string.*

For example, if $s = \bar{a}_3 a_1 a_2 a_1 \bar{a}_1 \bar{a}_2 a_3 a_4 \bar{a}_4 a_2$, then we have $R'(s) = \bar{a}_3 a_1 a_3 a_2$, because $a_2 a_1 \bar{a}_1 \bar{a}_2$ and $a_4 \bar{a}_4$ are the two removed S-substrings. Given a string s, the computation of $R'(s)$ can be done in $O(|s|)$ time using a stack. The definition of $R'(s)$ directly gives the following facts:

Lemma 1. *A string s is an S-string if and only if $R'(s) = \epsilon$, where ε represents the empty string.*

Lemma 2. *For any two strings s_1 and s_2, we have $R'(s_1 s_2) = R'(R'(s_1)R'(s_2))$.*

The following definitions and lemmas are used to test if the concatenation of two strings is an S-string.

Definition 2. *For any string s_1, we call it a valid S-prefix if there exists a string s_2 such that $s_1 s_2$ is an S-string. Similarly, for any string s_2, we call it a valid S-suffix if there exists a string s_1 such that $s_1 s_2$ is an S-string.*

Note that testing whether a string is an S-prefix or S-suffix can be done using a stack in linear time.

Definition 3. *For any two strings s_1 and s_2, we say that s_1 matches s_2 if and only if $s_1 s_2$ is an S-string, or equivalently, $R'(s_1 s_2) = \epsilon$.*

Let $reverse(s)$ represent the reversal of a string s, e.g., $reverse(a_1 \bar{a}_4 a_2 a_3) = a_3 a_2 \bar{a}_4 a_1$.

Lemma 3. *For any two strings s_1 and s_2, we have s_1 matches s_2 if and only if $R'(s_1) = flip(reverse(R'(s_2)))$ and s_1 is a valid S-prefix.*

For any two nodes u and v in the tree, we denote the only loopless directed path from u to v by $P(u, v)$ – if such a path exists. We also use $R(u, v)$ and $R'(u, v)$ to denote $R(P(u, v))$ and $R'(P(u, v))$ respectively for short.

5 Dyck-CFL Reachability Algorithm on Trees

In this section, we describe our algorithm for solving the Dyck-CFL reachability problem on bidirected trees, which can be pre-computed using $O(n \log n)$ space and $O(n \log n \log k)$ time to subsequently answer any online query $Q(u, v)$ in $O(1)$ time.

5.1 Loopless Property

In the problem definition, we have assumed that for any directed edge $(u, v) \in E$, $\mathcal{L}(v, u) = flip(\mathcal{L}(u, v))$ if (v, u) exists. This assumption leads to the following lemmas:

Lemma 4. *Any closed directed path $p = v_0 v_1 v_2 \cdots v_m v_0$ in the tree is an S-path.*

Lemma 5. *For any query $Q(u, v)$, it is only required to consider the loopless directed path $P(u, v)$ to see if it is the S-path. If $P(u, v)$ is not an S-path, then we can conclude that no S-path joining u and v exists.*

5.2 Basic Idea

Let x be a *fixed* tree node. We will call this node x a *pivot* node. Now, consider the following set of queries:

$$Q_x = \left\{ Q(u, v) \;\middle|\; u \text{ and } v \text{ are a query pair such that } P(u, v) \text{ goes through } x \right\}.$$

The goal is to build a linear data structure to answer any query in Q_x efficiently. Later in Section 5.3, we will describe how to handle queries outside Q_x by recursively building the data structures.

For any query $Q(u, v) \in Q_x$, according to Lemma 1 and Definition 3, we know that $P(u, v)$ is an S-path if and only if $R(u, x)$ matches $R(x, v)$. By Lemma 3, this is equivalent to testing whether $R(u, x)$ is a valid S-prefix and $R'(u, x) = flip(reverse(R'(x, v)))$. So we need to build data structures that support the following subqueries

- For any node u, is $R(u, x)$ a valid S-prefix?
- For any nodes u and v, is $R'(u, x) = flip(reverse(R'(x, v)))$?

***S*-prefix test.** If $R'(u, x)$ can be computed efficiently, then we are able to tell whether $R(u, x)$ is a valid S-prefix due to the following Lemma 6.

Lemma 6. *$R(u, x)$ is a valid S-prefix if and only if $R'(u, x)$ does not contain any symbol from $\bar{A}$.*

A naïve algorithm to compute $R'(u, x)$ is to use a stack to cancel matched parentheses in $|R(u, x)|$ time (see the **NaiveStack** procedure in Algorithm 1). If we do that for every u separately, then the total time can be as bad as $\Theta(n^2)$, and the spaces required to store the n realized strings can be as large as $\Theta(n^2)$.

Algorithm 1. Stack-based algorithm to compute $R'(u, x)$ for a single u

Procedure NaiveStack(u)
Input: a tree node u
Output: a string $R'(u, x)$, represented by the stack

1: Initialize an empty stack.
2: $w \leftarrow u$.
3: **while** $w \neq x$ **do**
4: Let w_p be the parent node of w in the tree.
5: If the directed edge (w, w_p) does not exist, then we terminate and report that no directed path from u to x exists.
6: **if** $\mathcal{L}(w, w_p) \in \bar{A}$ **and** the stack is non-empty **and** $\mathcal{L}(w, w_p)$ is the flipped parenthesis of the top of the stack **then**
7: Pop the top symbol from the stack. // Detected a pair of matched parentheses.
8: **else**
9: Push $L(w, w_p)$ to the stack.
10: **end if**
11: $w \leftarrow w_p$. // Move w to the next node on the path to x.
12: **end while**

Constructing a trie. Our idea to speed up the computation is to pre-compute $R'(u, x)$ for all u's in a single tree walk using $O(n \log k)$ time, and represent the realized strings in a trie [7] of size $O(n)$. See Algorithm 2. The trie constructed by **BuildTrie** will have the following properties:

- Denote the trie by TRIE, and its root by r.
- Each edge (z', z) of TRIE will be labeled by a symbol $\mathcal{L}(z', z)$ from $\mathscr{A}$. Here, the edges of the trie are undirected.
- For a trie node z, denote $R(z)$ to be a string that is concatenated by the symbols on the path from z to the root r. Note that this definition is in the bottom-up fashion, in contrast to the traditional top-down reading of the trie. Also, the algorithm processes the path $P(u, x)$ in the order from x to u rather than from u to x (like **NaiveStack**).
- For each tree node $u \in T$, there is a corresponding trie node $z \in$ TRIE such that $R(z) = R'(u, x)$. We store this trie node z in $TriePos(u)$. Note that, if $TriePos(u)$ is not set for some $u \in T$, then it means that there is no directed path from u to x.
- At each node $z \in$ TRIE, we store the set of tree nodes that are associated to z in $TreeNodeSet(z)$, i.e., $TreeNodeSet(z) = \left\{u \mid TriePos(u) = z\right\}$.

The correctness of **BuildTrie** is based on the fact that: the trie simulates a stack. Line 5 simulates "stack pop" by moving z to its parent z_p, and line 8 simulates "stack push" by expanding/walking-down the trie. In this way, a trie node z effectively represents a stack, and the contents of the stack is captured by $R(z)$. The space complexity of this tree-walk style pre-processing algorithm is $O(|T|)$, and time complexity is $O(|T| \log k)$. The $\log k$ factor comes from

Algorithm 2. Trie-based algorithm to compute $R'(u, x)$ for all u's

Make a call to the following recursive procedure by **BuildTrie**(x, r), where r is a pre-allocated root of a trie.

Procedure BuildTrie(u, z)
Input: a tree node u, and a trie node z

1: $TriePos(u) \leftarrow z$ and add u to the set $TreeNodeSet(z)$.
2: **for** each child node u' of u such that the directed edge (u', u) exists **do**
3: Let z_p be the parent node of z in the trie.
4: **if** z_p exists and $\mathcal{L}(u', u) = flip(\mathcal{L}(z, z_p))$ and $\mathcal{L}(u', u) \in A$ **then**
5: Call BuildTrie(u', z_p).
6: **else**
7: Choose a child node z' of z such that $\mathcal{L}(z', z) = \mathcal{L}(u', u)$; if it does not exist, then add a new child node z' to z, and label the edge (z', z) by $\mathcal{L}(u', u)$.
8: Call BuildTrie(u', z').
9: **end if**
10: **end for**

line 7, where a balanced binary tree of size at most $|\mathscr{A}| = 2k$ is used efficiently to search for child nodes in the trie.

Now, we have the trie to compactly represent $R'(u, x)$ for all u's. Getting back to the S-prefix test problem, the validity for $R'(u, x)$ can be easily pre-computed by a top-down tree walk as in Algorithm 3. The correctness is guaranteed from Lemma 6: each time we visit a node z, we have made sure that $R(z)$ contains symbols only from A and the tree walk only passes through edges that are labeled by symbols from A. Therefore, the S-prefix validity test for $R(u, x)$ can be pre-computed in linear time and space, and the subquery can be answered later in $O(1)$ time.

Algorithm 3. Pre-compute the information for S-prefix validity test

Make a call to the following procedure by **MarkValidity**(r), where r is the trie root.

Procedure MarkValidity(z)
Input: a trie node z

1: For each $u \in TreeNodeSet(z)$, we mark down that $R(u, x)$ is a valid S-prefix.
2: **for** each child node z' of z **do**
3: **if** $\mathcal{L}(z', z) \in A$ **then**
4: Call MarkValidity(z').
5: **end if**
6: **end for**

Match test. Now, consider the second subquery, i.e., testing whether $R'(u, x) = flip(reverse(R'(x, v)))$. We will first show that $R'(x, v)$ for all v's can also be represented by a trie, and pre-computed efficiently. The previous Algorithm 2 can be modified to construct a trie such that:

– For each trie node z, if we denote $\hat{R}(z)$ to be the string that is concatenated from the symbols along the path from the trie root to z (this time, it is in the top-down fashion), then we have $\hat{R}(z) = R'(x, v)$ for any tree node v that is associated to the trie node z.

The key modifications of Algorithm 2 to compute such a trie are

– Consider downward edges (u, u') instead of upward edges (u', u).
– When testing to "pop" or not (at line 4), change the condition from $\mathcal{L}(u', u) \in A$ to $\mathcal{L}(u, u') \in \bar{A}$. This is important, because when we compute $R'(x, v)$, we should use a symbol from $\bar{A}$ to initiate the cancelation of matched parentheses.

The time and space complexities after the modifications are still $O(n \log k)$ and $O(n)$ respectively.

Assume that we now have two tries TRIE_1 and TRIE_2, where TRIE_1 is constructed to represent $R'(u, x)$, and TRIE_2 is to represent $R'(x, v)$. A naïve way to test whether $R'(u, x) = flip(reverse(R'(x, v)))$ can be done as follows

1. Find out $z_1 = TriePos(u)$ in TRIE_1.
2. Find out $z_2 = TriePos(v)$ in TRIE_2.
3. Let z_1 and z_2 simultaneously walk up the tries towards their corresponding roots. During the walk, test to see if the edge labels are matched (i.e., one label is the flipped version of the other edge label in the other trie).

The correctness is based on the fact that $R(z_2)$ is the reversed string of $\hat{R}(z_2)$, so we are actually testing whether $R(z_1) = R'(u, x)$ is the flipped version of $R(z_2) = reverse(\hat{R}(z_2)) = reverse(R'(x, v))$. This naïve algorithm would use as much as $\Theta(n)$ time if the tries have very large heights.

To speed up the testing, we adapt the following pre-processing algorithm:

1. Flip all the edge labels of TRIE_2.
2. Merge TRIE_1 and TRIE_2 to be a single trie. This can be done in linear time [5]. Denote the new trie by $\text{TRIE}_{\text{merged}}$.
3. For a tree node $u \in T$, let $z_1 \in \text{TRIE}_1$ be the trie node where $R(z_1) = R'(u, x)$, i.e., z_1 is the $TriePos(u)$ in the context of TRIE_1. Then after the merge, we denote $TriePos_1(u)$ to be the new location of z_1 in the merged trie $\text{TRIE}_{\text{merged}}$.
4. For a tree node $v \in T$, let $z_2 \in \text{TRIE}_2$ be the trie node where $\hat{R}(z_2) = R'(x, v)$, i.e., z_2 is the $TriePos(v)$ in the context of TRIE_2. Then after the the merge, we denote $TriePos_2(v)$ to be the new location of z_2 in the merged trie $\text{TRIE}_{\text{merged}}$.

Step 1 and 2 take linear time. The data structures (i.e., $TriePos_1$ and $TriePos_2$) defined in step 3 and 4 can be computed naturally in linear time during the merging.

Using $\text{TRIE}_{\text{merged}}$, we can tell whether $R'(u, x) = flip(reverse(R'(x, v)))$ by simply checking the equality of $TriePos_1(u)$ and $TriePos_2(v)$. More specifically,

$R'(u,x) = flip(reverse(R'(x,v)))$ if and only if $TriePos_1(u) = TriePos_2(v)$ based on the above analysis.

Therefore, the second subquery can be answered in $O(1)$ time, with an $O(n \log k)$-time and $O(n)$-space pre-processing. Combing the results in this subsection, we have the following theorem.

Theorem 1. *There exists a data structure, which can be preprocessed in $O(n \log k)$ time and $O(n)$ space to answer any query pair whose path goes through a pre-defined separator x of the tree.*

5.3 Divide and Conquer

Now the question is how to efficiently handle the cases when the path does not go through the predefined pivot node x. We can solve those cases by recursively building data structures for the subtrees obtained by removing x from the tree. The recursions are expected to be balanced in order to achieve a good time bound, so we choose the *centroid* node of a tree to be such an x.

A node x in a tree T is called a centroid of T if the removal of x will make the size of each remaining connected component no greater than $|T|/2$. A tree may have at most two centroids, and if there are two then one must be a neighbor of the other [6,5]. Throughout this paper, we specify the centroid of a tree to be the one whose numbering is lexicographically smaller (i.e., we number the nodes from 1 to n). There exists a linear time algorithm to compute the centroid of a tree due to the work of Goldman [21]. We use $\mathrm{CT}(T)$ to denote the centroid of T computed by the linear time algorithm.

Algorithm 4 is the well-known recursive tree centroid decomposition method (see Figure 2 for an example of the tree centroid decomposition). The time complexity for the recursive tree centroid decomposition algorithm is $O(n \log n)$, since no node will participate in the centroid computations for more than $O(\log n)$ times. The stack space for the recursion is bounded by $O(n + n/2 + n/4 + n/8 + \dots) = O(n)$.

Algorithm 4. Tree centroid decomposition

Procedure CentroidDecomposition(T)

1: Find the centroid of T. Denote the set of the remaining connected components by $\mathrm{Remain}(T) = \{T' \mid T' \text{ is a connected component after the removal of } \mathrm{CT}(T)\}$.

2: Let $c = \mathrm{CT}(T)$ be the computed centroid, then for each neighbor x of c in T, we use $T_{c,x}$ to denote the remaining connected component that contains x. Also, we denote the current tree T by T_c.

3: Recursively call CentroidDecomposition(T') for each $T' \in \mathrm{Remain}(T)$.

Define *canonical* subtrees to be all the subtrees considered during the recursive call of Algorithm 4 if we start the recursion at T, i.e., the canonical subtrees are $\{T_c \mid c \in V\}$. Please note that, each node $c \in V$ must be a centroid of some

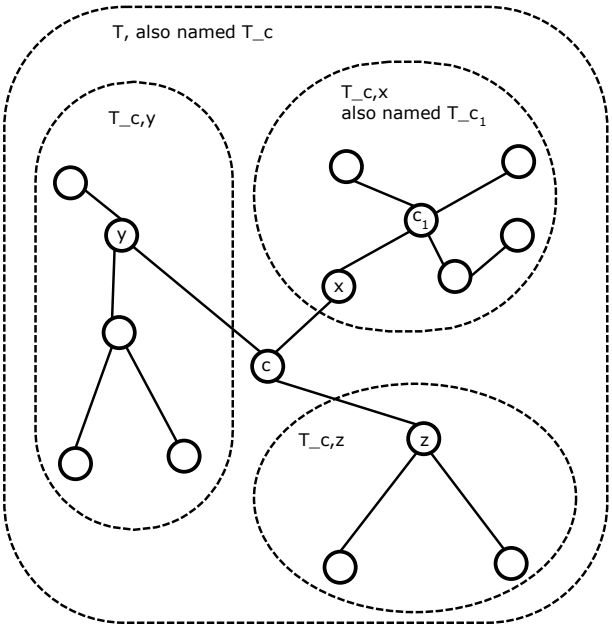

Fig. 2. An example for the tree centroid decomposition. In this example, node c is a centroid of the whole tree, and c_1 is the centroid of the subtree $T_{c,x}$.

canonical subtree; an extreme case is when node c is the centroid of a subtree which only consists of a single node (c itself). Therefore, there are exactly n such canonical subtrees, and one can see that each "remaining connected component" $T_{c,x}$ is just $T_{\mathrm{CT}(T_{c,x})}$. Based on the fact that no node will be in more than $O(\log n)$ canonical subtrees, we have

$$\sum_{c \in V} |T_c| = O(n \log n).$$

For each canonical subtree T_c, we build a trie using the **BuildTrie** algorithm specified in Section 5.2 to preprocess for the following query set

$$Q_c = \left\{ Q(u,v) \;\middle|\; u \text{ and } v \text{ are a query pair such that } P(u,v) \text{ goes through } c \right\}.$$

The total time complexity is

$$\sum_{c \in V} |T_c| \log k = O(n \log n \log k).$$

Once the data structures for each canonical subtree are built, we can answer any query $Q(u,v)$ in the following way:

- Locate a smallest canonical subtree T_c such that both u and v are in that tree. Note that we must have node c on the undirected path from u to v, otherwise T_c can not be the smallest such canonical subtree.
- Query on the data structures that were built for T_c since $Q(u, v) \in Q_c$.

The second step takes $O(1)$ time from the trie-based data structures. For the first step, we can have a linear size data structure to help us locate the T_c efficiently: preprocess the recursion tree of Algorithm 4 in linear time so that the least common ancestor query [22,23] for any two nodes in the recursion tree can be answered in constant time. In the recursion tree, let n_u and n_v be two nodes that correspond to T_u and T_v respectively, then the least common ancestor of n_u and n_v in the recursion tree corresponds to our desired T_c. Therefore, the first step takes $O(1)$ time as well. Combing the analysis, we have the following theorem.

Theorem 2. *The Dyck-CFL reachability problem can be preprocessed in* $O(n \log n \log k)$ *time and* $O(n \log n)$ *space to answer any online query in* $O(1)$ *time.*

6 Conclusions

We considered the CFL reachability problem for the case when the underlying graph is a specific bidirected tree of size n, and the grammar is the Dyck language of size k. We have described an efficient algorithm to build a data structure of size $O(n \log n)$ in $O(n \log n \log k)$ time to handle any online query in $O(1)$ time. Possible future work can be considering dynamic graph updates, i.e., graph nodes and edges are added/deleted dynamically and online CFL reachability queries need to be answered efficiently.

References

1. Reps, T.W.: Program analysis via graph reachability. Information & Software Technology 40(11-12), 701–726 (1998)
2. Rehof, J., Fähndrich, M.: Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In: Proceedings of the 28th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 2001), pp. 54–66 (2001)
3. Sridharan, M., Gopan, D., Shan, L., Bodík, R.: Demand-driven points-to analysis for Java. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005), pp. 59–76 (2005)
4. Sridharan, M., Bodík, R.: Refinement-based context-sensitive points-to analysis for Java. In: Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI 2006), pp. 387–400 (2006)
5. Knuth, D.E.: The art of computer programming, volume III: sorting and searching. Addison-Wesley, Reading (1973)
6. Hakimi, S.: Optimum locations of switching center and the absolute center and medians of a graph. Operations Research 12, 450–459 (1964)

7. Knuth, D.E.: The art of computer programming, volume I: fundamental algorithms. Addison-Wesley, Reading (1973)
8. Yannakakis, M.: Graph-theoretic methods in database theory. In: Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 1990), pp. 230–242 (1990)
9. Reps, T.W., Horwitz, S., Sagiv, S.: Precise interprocedural dataflow analysis via graph reachability. In: Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1995), pp. 49–61 (1995)
10. Reps, T.W.: Shape analysis as a generalized path problem. In: Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 1995), pp. 1–11 (1995)
11. Zheng, X., Rugina, R.: Demand-driven alias analysis for C. In: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008), pp. 197–208 (2008)
12. Chaudhuri, S.: Subcubic algorithms for recursive state machines. In: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008), pp. 159–169 (2008)
13. Arlazarov, V.L., Dinic, E.A., Faradzev, M.A.K.,, I.A.: On economical construction of the transitive closure of an oriented graph. Soviet Mathematics Doklady 11, 1209–1210 (1970)
14. Rytter, W.: Time complexity of loop-free two-way pushdown automata. Inf. Process. Lett. 16(3), 127–129 (1983)
15. Rytter, W.: Fast recognition of pushdown automaton and context-free languages. Inf. Control 67(1-3), 12–22 (1986)
16. Alur, R., Benedikt, M., Etessami, K., Godefroid, P., Reps, T.W., Yannakakis, M.: Analysis of recursive state machines. ACM Transactions on Programming Languags and Systems 27(4), 786–818 (2005)
17. Kodumal, J., Aiken, A.: The set constraint/cfl reachability connection in practice. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2004), pp. 207–218 (2004)
18. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: STOC 2004: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing, pp. 202–211. ACM, New York (2004)
19. Alur, R.: Marrying words and trees. In: PODS 2007: Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pp. 233–242. ACM, New York (2007)
20. Valiant, L.G.: General context-free recognition in less than cubic time. Journal of Computer and System Sciences 10(2), 308–315 (1975)
21. Goldman, A.: Optimal center location in a simple network. Transportation Science 5, 212–221 (1971)
22. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM Journal of Computing 13(2), 338–355 (1984)
23. Bender, M.A., Farach-Colton, M.: The lca problem revisited. In: Proceedings of the 4th Latin American Symposium on Theoretical Informatics, pp. 88–94 (2000)