On the Complexity of Bidirected Interleaved Dyck-Reachability

YUANBO LI, Georgia Institute of Technology, USA
QIRUN ZHANG, Georgia Institute of Technology, USA
THOMAS REPS, University of Wisconsin–Madison, USA

Many program analyses need to reason about pairs of matching actions, such as call/return, lock/unlock, or set-field/get-field. The family of Dyck languages $\{D_k\}$, where D_k has k kinds of parenthesis pairs, can be used to model matching actions as balanced parentheses. Consequently, many program-analysis problems can be formulated as Dyck-reachability problems on edge-labeled digraphs. *Interleaved Dyck-reachability* (InterDyck-reachability), denoted by $D_k \odot D_k$-reachability, is a natural extension of Dyck-reachability that allows one to formulate program-analysis problems that involve *multiple* kinds of matching-action pairs. Unfortunately, the general InterDyck-reachability problem is undecidable.

In this paper, we study variants of InterDyck-reachability on *bidirected graphs*, where for each edge $\langle p, q\rangle$ labeled by an open parenthesis "$(_a$", there is an edge $\langle q, p\rangle$ labeled by the corresponding close parenthesis "$)_a$", and *vice versa*. Language-reachability on a bidirected graph has proven to be useful both (i) in its own right, as a way to formalize many program-analysis problems, such as pointer analysis, and (ii) as a *relaxation* method that uses a fast algorithm to over-approximate language-reachability on a directed graph. However, unlike its directed counterpart, the complexity of bidirected InterDyck-reachability still remains open.

We establish the first decidable variant (*i.e.*, $D_1 \odot D_1$-reachability) of bidirected InterDyck-reachability. In $D_1 \odot D_1$-reachability, each of the two Dyck languages is restricted to have only a single kind of parenthesis pair. In particular, we show that the bidirected $D_1 \odot D_1$-reachability problem is in **PTIME**. We also show that when one extends each Dyck language to involve k different kinds of parentheses (*i.e.*, $D_k \odot D_k$-reachability with $k \geq 2$), the problem is **NP**-hard (and therefore much harder).

We have implemented the polynomial-time algorithm for bidirected $D_1 \odot D_1$-reachability. $D_1 \odot D_1$-reachability provides a new over-approximation method for bidirected $D_k \odot D_k$-reachability in the sense that $D_k \odot D_k$-reachability can first be relaxed to bidirected $D_1 \odot D_1$-reachability, and then the resulting bidirected $D_1 \odot D_1$-reachability problem is solved precisely. We compare this $D_1 \odot D_1$-reachability-based approach against another known over-approximating $D_k \odot D_k$-reachability algorithm. Surprisingly, we found that the over-approximation approach based on bidirected $D_1 \odot D_1$-reachability computes *more precise* solutions, even though the $D_1 \odot D_1$ formalism is inherently less expressive than the $D_k \odot D_k$ formalism.

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

Additional Key Words and Phrases: Formal Language Graph Reachability, Interleaved-Dyck Language, Static Analysis, Complexity

Authors' addresses: Yuanbo Li, School of Computer Science, Georgia Institute of Technology, 266 Ferst Dr NW, Atlanta, GA, 30332, USA, yuanboli@gatech.edu; Qirun Zhang, School of Computer Science, Georgia Institute of Technology, 266 Ferst Dr NW, Atlanta, GA, 30332, USA, qrzhang@gatech.edu; Thomas Reps, Computer Science Department, University of Wisconsin–Madison, 1210 West Dayton St., Madison, WI, 53706, USA, reps@cs.wisc.edu.

2475-1421/2021/1-ART59
https://doi.org/10.1145/3434340

ACM Reference Format:
Yuanbo Li, Qirun Zhang, and Thomas Reps. 2021. On the Complexity of Bidirected Interleaved Dyck-Reachability. *Proc. ACM Program. Lang.* 5, POPL, Article 59 (January 2021), 28 pages. https://doi.org/10.1145/3434340

1 INTRODUCTION

Many static-analysis problems can be formulated as a reachability problem with respect to a formal language L on an edge-labeled graph G. Two nodes in G are L-reachable iff there exists a path between them, and the string spelled out by the path is a word in L. An L-reachability algorithm computes all L-reachable pairs in G. A Dyck language D_k, which consists of all strings of well-balanced parentheses over k kinds of parentheses, is perhaps the most widely used formal language in L-reachability formulations [Reps 1998]. Many practical analyses use Dyck-reachability to express (ordered) matching actions, such as call/return (*i.e.*, context-sensitivity) [Reps 2000; Yan et al. 2011], lock/unlock [Kahlon 2009; Ramalingam 2000], file-open/file-close [Späth et al. 2019], or set-field/get-field (*i.e.*, field-sensitivity) [Arzt et al. 2014; Yan et al. 2011]. Kodumal and Aiken [2004] observed that "almost all of the applications of [L-reachability, where L is a context-free language] in program analysis are based on Dyck languages."

Interleaved Dyck-reachability ("InterDyck-reachability" for short) is a more expressive formalism than Dyck-reachability. An InterDyck-reachability problem is an L-reachability problem in which L involves two *interleaved* Dyck languages. Let D_p and D_b be the Dyck languages of balanced parentheses and balanced brackets, respectively. The InterDyck language $D_p \odot D_b$ is the language in which D_p and D_b are interleaved. For example, the string "$[_1 (_1 [_2)_1 (_2]_2]_1)_2$" in $D_p \odot D_b$ represents the interleaving of the string "$(_1)_1 (_2)_2$" in D_p and the string "$[_1 [_2]_2]_1$" in D_b. The InterDyck-reachability formalism can be applied to many client applications, such as alias analysis [Cheng and Hwu 2000; Sridharan and Bodík 2006], taint analysis [Arzt et al. 2014; Huang et al. 2015], and typestate analysis [Späth et al. 2019]. However, precise InterDyck-reachability is undecidable in general [Reps 2000]. Thus, practical analysis usually over-approximates the exact solution.

In this paper, we study variants of InterDyck-reachability restricted to *bidirected graphs*, where for each edge $\langle p, q\rangle$ labeled by a open parenthesis "$(_a$", there is an edge $\langle q, p\rangle$ labeled by the corresponding close parenthesis "$)_a$", and *vice versa.* Although L-reachability problems on bidirected graphs are a special case of general L-reachability problems, they play an important role in program analysis. For instance, graph bidirectedness allows pointer analysis to be formulated as a CFL-reachability problem [Reps 1998, §4.4]. Many context- and field-sensitive alias analyses are based on bidirected InterDyck-reachability [Xu et al. 2009; Yan et al. 2011]. Moreover, bidirected Dyck-reachability problems can be solved by a linear-time algorithm [Chatterjee et al. 2018], whereas the best algorithm for the directed counterpart runs in $O(n^3/\log n)$ time [Chaudhuri 2008]. Recent work by [Li et al. 2020] shows that graph simplification based on bidirected Dyck-reachability can be used to speed up any InterDyck-reachability algorithm. Bidirected InterDyck-reachability can also be used as a relaxation of directed InterDyck-reachability (*i.e.*, by introducing all necessary labeled reversed edges into the underlying graph): the solution of the bidirected variant over-approximates the solution of the directed variant.

Unfortunately, despite tremendous progress on algorithms for solving bidirected problems, the question of whether bidirectedness makes any InterDyck-reachability variant decidable has remained open. This paper fills this gap by studying the computational complexity of two bidirected

InterDyck-reachability variants (*i.e.*, $D_1 \odot D_1$-reachability and $D_k \odot D_k$-reachability[1]). In particular, the paper makes two main contributions.

- We identify the first decidable variant of InterDyck-reachability, namely, bidirected $D_1 \odot D_1$-reachability. We show that bidirected $D_1 \odot D_1$-reachability is in **PTIME**.
- For the bidirected $D_k \odot D_k$-reachability problem with $k \geq 2$, we establish that it is **NP**-hard, and thus likely to be a much harder problem than bidirected $D_1 \odot D_1$-reachability.

We implemented the bidirected $D_1 \odot D_1$-reachability algorithm, and applied it to a real-world alias analysis [Xu et al. 2009; Yan et al. 2011]. Our empirical evaluation uncovered an interesting phenomenon. We compared the precision of (i) relaxing a context- and field-sensitive alias analysis to a context- and field-insensitive variant—by relaxing bidirected $D_k \odot D_k$-reachability to bidirected $D_1 \odot D_1$-reachability—and applying the polynomial-time $D_1 \odot D_1$-reachability algorithm against (ii) a known over-approximation algorithm for bidirected $D_k \odot D_k$-reachability [Zhang and Su 2017]. Our empirical results show that the *exact solution* from (i) is more precise than the *over-approximating solution* obtained from (ii), even though the $D_1 \odot D_1$ formalism is inherently less expressive than the $D_k \odot D_k$ formalism.

Organization. Section 2 defines several concepts used in the rest of the paper. Section 3 presents the key idea behind our proof of the **PTIME** complexity of bidirected $D_1 \odot D_1$-reachability. Section 4 establishes that $D_1 \odot D_1$-reachability is in **PTIME**. Section 5 establishes the **NP**-hardness result for bidirected $D_k \odot D_k$-reachability. Section 6 presents experimental results. Section 7 discusses related work. Section 8 concludes.

2 PRELIMINARIES

2.1 Formal-Language-Reachability Problems

A large number of program-analysis problems can be formulated as formal-language-reachability problems. A formal-language-reachability problem in an edge-labeled graph extends ordinary graph reachability: for a node to be considered reachable (*e.g.*, from a distinguished start node), it must be connected by a path along which the edge labels spell out a word in a given formal language L. Language L is used to specify potential "flows of information" in the graph. More precisely, the restriction to L-paths provides a mechanism for *excluding* paths in the graph along which information could *not* flow.

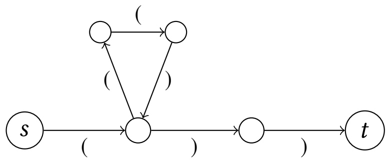
Definition 2.1 (Formal-Language-Reachability Problem). *Let L be a formal language over an alphabet Σ, and G be a graph in which each edge is labeled with a symbol from Σ. Each path corresponds a word formed by concatenating the symbols of the edges of the path (in the same order as the edges). A path is an L-path iff its corresponding word is in the language L. For each L-path from node u to v, we say that v is* L-reachable from u *and the two nodes are an L-reachable pair. The* all-pairs *L-reachability problem is to compute all L-reachable pairs in G.*

Example 2.2. Consider the context-free language D_1 defined in Figure 1a. Figure 1b gives an edge-labeled graph. In this graph, the path from s to t that traverses the cycle exactly once forms the word "((()))" in language D_1. Consequently, t is D_1-reachable from s.

[1]Except in a few cases, we are not explicit about the actual parenthesis symbols used, referring to $D_1 \odot D_1$ when each Dyck language has exactly one kind of parenthesis pair, and to $D_k \odot D_k$ when each language has more than one kind of parenthesis pair. It would be more precise to say $D_{k_1} \odot D_{k_2}$, but we prefer to avoid the double subscripting. In examples, when we wish to distinguish between the two languages, we use $D_p \odot D_b$ ("p" for "parentheses" and "b" for "brackets").

$$S ::= S\ S$$
$$S ::= (\ S\ )$$
$$S ::= \epsilon$$

(a) Dyck language D_1.



(b) Dyck-reachability.

Fig. 1. Dyck language (D_1) and Dyck-reachability (D_1-reachability).

2.2 Dyck Languages and Interleaved Dyck Languages

This section introduces the specific family of languages that we focus on in this paper. An interleaved Dyck language (InterDyck language) is constructed from multiple Dyck languages. A Dyck language generates a set of strings of balanced parentheses:

Definition 2.3 (Dyck Language). *A Dyck language D_k with k different kinds of parenthesis pairs can be defined by the following grammar:*

$$S ::= S\ S \mid (_1\ S\ )_1 \mid (_2\ S\ )_2 \mid \dots \mid (_k\ S\ )_k \mid \epsilon.$$

The language D_1 defined in Figure 1a is a special Dyck language with only one kind of parenthesis pair. In general, program-analysis applications can use the different kinds of parenthesis pairs in D_k to track calls/returns through different call-sites of functions and reads/writes to different fields [Reps 2000; Zhang and Su 2017].

Many practical program analyses need to track multiple matching properties simultaneously [Arzt et al. 2014; Huang et al. 2015; Ramalingam 2000; Späth et al. 2019; Sridharan and Bodík 2006]. For example, a context- and field-sensitive alias analysis [Xu et al. 2009; Yan et al. 2011] needs to consider only the paths in which each method call is matched with the corresponding method return (*i.e.*, context-sensitivity); at the same time, it should consider the paths in which data assigned to a field of an object can only be passed via an access on the same field (*i.e.*, field-sensitivity). One Dyck language can be used to capture context-sensitivity, and a second Dyck language can be used to capture field-sensitivity. Consequently, when an analysis needs to keep track of two properties simultaneously, we formulate the problem as an InterDyck-reachability problem: an InterDyck language accepts all words that consist of the *interleaving* of two Dyck words. We define the interleaving operator and the InterDyck languages as follows:

Definition 2.4 (Interleaving Operator). *Suppose that two languages L_1 and L_2 are defined over alphabets Σ_1 and Σ_2, respectively. The interleaving operator $\odot : L_1 \times L_2 \to (\Sigma_1 \cup \Sigma_2)^*$ on words is defined as follows:*

$$\begin{aligned} a \odot \epsilon &= \{a\} \text{ for } a \in L_1 \\ \epsilon \odot b &= \{b\} \text{ for } b \in L_2 \\ c_1 a \odot c_2 b &= \{c_1 w \mid w \in (a \odot c_2 b)\} \cup \{c_2 w \mid w \in (c_1 a \odot b)\} \\ &\qquad \text{for } c_1 \in \Sigma_1, c_2 \in \Sigma_2, a \in L_1, b \in L_2. \end{aligned}$$

The interleaving operator can also be overloaded as a binary operator on languages:

$$L_1 \odot L_2 = \bigcup_{a \in L_1, b \in L_2} a \odot b.$$

Definition 2.5 (Interleaved Dyck language). *Given two Dyck languages D_m and D_n with disjoint alphabets, the* interleaved Dyck language *("InterDyck language") constructed from D_m and D_n is defined as the language $D_m \odot D_n$.*

```
1  class T { T f; T g; }
2  T getF(T a){ return a.f; }
3  void setF(T b, T val){ b.f = val; }
4  ...
5  T x,y;
6  setF(y,x);
7  z = getF(y);
```

(a) Code snippet.

$$\begin{tikzcd} x \arrow[r, "(_6"] & val \arrow[r, "{[_f}"] & b \arrow[r, ")_6"] & y \arrow[r, "(_7"] & a \arrow[r, "{]_f}"] & ret \arrow[r, ")_7"] & z \end{tikzcd}$$

(b) Graph representation.

Fig. 2. Alias analysis via InterDyck-language reachability.

Example 2.6. Consider the two Dyck languages D_p and D_b defined by the following grammars:

$$D_p : S ::= S\,S \mid (\ S\ ) \mid \epsilon$$
$$D_b : S ::= S\,S \mid [\ S\ ] \mid \epsilon.$$

The interleaved Dyck language $D_p \odot D_b$ contains such words as "([)]", "([)(])", etc.

2.3 Bidirected InterDyck-Reachability

We proceed to define the InterDyck-reachability problem on *bidirected* graphs.

Definition 2.7 (Bidirected InterDyck-Reachability). *Consider an L-reachability problem on a digraph $G = (V, E)$, where L is the InterDyck language $D_p \odot D_b$. Let O and C denote the sets of all open- and close-parenthesis symbols from $D_p \odot D_b$, respectively. For each edge $e = \langle u, v\rangle \in E$, if the edge label is an open parenthesis "$(_i$"$\in O$, there must exists a reverse edge $\overline{e} = \langle v, u\rangle$ with edge label "$)_i$"$\in C$, and* vice versa. *We say that G is a* bidirected graph *(for the InterDyck language $D_p \odot D_b$) and the reachability problem is a* bidirected InterDyck-reachability problem.

2.4 Applying InterDyck-Reachability in Program Analysis

We illustrate how InterDyck-reachability is used in program analysis by sketching how it can be used in a context- and field-sensitive alias analysis for a Java-style language [Yan et al. 2011] shown in Figure 2. We use one Dyck language D'_p to capture context-sensitivity, and another Dyck language D'_b to capture field-sensitivity. In D'_p, an open parenthesis "$(_i$" represents a method call at line i; a close parenthesis "$)_i$" models a method return to the call-site at line i. In D'_b, an open bracket "$[_f$" represent a write to the field f; a close bracket "$]_f$" represents a read to the field f. Consider the program in Figure 2a. The alias analysis checks whether two variables can point to the same object. In the program, the f field of variable y is set to the value of variable x, and then variable z is set to the value of the f field of y. Thus, variables x and z can potentially (and in this case, must) point to the same object.

Figure 2b gives the corresponding graph. The graph is augmented with inverse edges. Specifically, for each edge $\langle p, q\rangle$ labeled with "$[_f$", there always exists an inverse edge $\langle q, p\rangle$ labeled with "$]_f$", and vice versa. Similarly, each "$(_i$"-labeled edges is accompanied with the corresponding "$)_i$"-labeled edges in the inverse direction. The bidirectedness is a prerequisite for CFL-reachability-based pointer/alias analysis [Reps 1998; Sridharan et al. 2005], otherwise, two nodes may not be reachable from each other even by standard graph reachability. To check whether variables x and z can be aliases, it suffices to decide whether there exists an InterDyck-path from node x to z. In graph G, there is a valid InterDyck-path "$(_6[_f)_6(_7]_f)_7$", indicating that the variable x and z can be aliases in the original program. Note that the alias analysis based on InterDyck-reachability is a simplification of the *alias* reachability presented by Sridharan and Bodík [2006]. The field edges approximate the field loads and stores in the flow graph [Sridharan and Bodík 2006; Sridharan et al. 2005].

The approximation may lead to spurious aliasing, as discussed by Xu et al. [2009, §4]. However, experimental results show that, in practice, the overall performance is better than the algorithms proposed by Sridharan et al. [2005] and Sridharan and Bodík [2006].

3 OVERVIEW OF THE BIDIRECTED $D_1 \odot D_1$-REACHABILITY PROOF

This section presents the key ideas behind the proof that bidirected $D_1 \odot D_1$-reachability is polynomial-time solvable. As in most graph-reachability problems, in $D_1 \odot D_1$-reachability it is infeasible to collect up all $D_1 \odot D_1$-paths because the number of such paths can be infinite, and the lengths of such paths can be unbounded as well. Instead, the goal is to identify all $D_1 \odot D_1$-reachable node pairs (Definition 2.1) . The argument for identifying all $D_1 \odot D_1$-reachable pairs is achievable in **PTIME** is structured into two parts:

- We first show that every bidirected $D_1 \odot D_1$-reachable pair (u, v) is also connected by a $D_1 \odot D_1$-path of a special form, which we call a *shallow $D_1 \odot D_1$-path* (Definition 3.2).
- We then give a polynomial-time tabulation algorithm to find all (u, v) pairs that are connected by at least one shallow $D_1 \odot D_1$-path.

3.1 Bidirectedness and Shallow Paths

Bidirectedness plays a pivotal role in the proof. In particular, bidirectedness enables a path to use the reverse edges, which allows a path to go back and forth along a sequence of edges in the graph. As we illustrate below, for a given path P that contains some cycles, it is possible to construct a new path P' in which the cycles are arranged differently. *Cycle rearrangement* is the key insight that enables us to prove the existence of shallow paths, which we now define.

A shallow path is characterized by the number of unmatched open parentheses in each prefix of the path. To specify the property that a shallow path must satisfy, we first define a quantity to characterize the unmatched open parentheses in a given path prefix.

Definition 3.1 (Tuple of Unmatched Open Parentheses). *Let $G = (V, E)$ be a bidirected graph for a bidirected $D_1 \odot D_1$-reachability problem, and let $P = v_0 v_1 \dots v_l$ be a $D_1 \odot D_1$-path in G. We define the* tuple of unmatched open parentheses *at the i^{th} node of P, denoted by $unmatch(P, i)$, to be the tuple (a, b), where*

- *a is the number of unmatched open parentheses in the first Dyck language in the length-i prefix $v_0 v_1 \dots v_i$ of the path P,*
- *b is the number of unmatched open parentheses in the second Dyck language in the length-i prefix $v_0 v_1 \dots v_i$ of the path P.*

Definition 3.2 (Shallow Path). *Let $G = (V, E)$ be a bidirected graph for a $D_1 \odot D_1$-reachability problem. A $D_1 \odot D_1$-path P in G is* shallow *if, for each i between 1 and $|P|$, the tuple $(a, b) = unmatch(P, i)$ satisfies $a \leq 6n$ or $b \leq 6n$, where $n = |V|$ is the number of nodes in G.*

Figure 3 illustrates the state space of unmatched open parentheses in $D_1 \odot D_1$-paths, along with the trajectory in that space of a specific shallow $D_1 \odot D_1$-path. In particular, because the number of unmatched open parentheses for the two Dyck languages must never *simultaneously* exceed $6n$ in a shallow path, the trajectory can never enter the $(6n, \infty) \times (6n, \infty)$ region of Figure 3.

Given a $D_1 \odot D_1$-path between two nodes u and v in bidirected graph G, there is always a shallow $D_1 \odot D_1$-path that connects u and v. This property is captured by the following theorem:

Theorem 3.3. *In a bidirected $D_1 \odot D_1$-reachability problem, if there exists a $D_1 \odot D_1$-path from node u to node v, then there exists a shallow $D_1 \odot D_1$-path from u to v.*

We defer the proof of Theorem 3.3 to Section 4.1, but note here that to establish the existence of the shallow path, the key observations used in the proof are as follows:

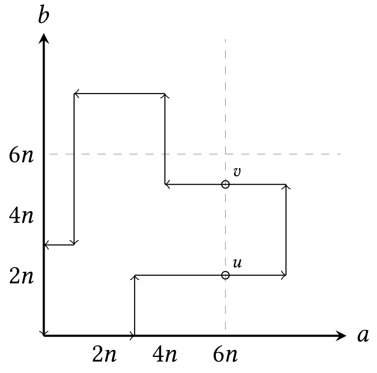
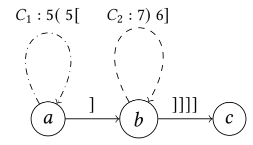


Fig. 3. State space for unmatched open parentheses in $D_1 \odot D_1$-paths, and the trajectory of a shallow path.



Original path: $\underset{(0,0)}{a} \overset{7C_1}{\dashrightarrow} \underset{(35,35)}{a} \xrightarrow{]} \underset{(35,34)}{b} \overset{5C_2}{\dashrightarrow} \underset{(0,4)}{b} \xrightarrow{]]]]} \underset{(0,0)}{c}$

Rearranged path: $\underset{(0,0)}{a} \overset{6C_1}{\dashrightarrow} \underset{(30,30)}{a} \xrightarrow{]} b \overset{C_2}{\dashrightarrow} \underset{(23,23)}{b} \xrightarrow{[} a \overset{C_1}{\dashrightarrow} \underset{(28,29)}{a} \xrightarrow{]} b \overset{4C_2}{\dashrightarrow} b \xrightarrow{]]]]} \underset{(0,0)}{c}$

Fig. 4. Cycle rearrangement in a bidirected graph. In this graph, we assume the bound for the number of the unmatched parentheses is 30. Cycle rearrangement moves one cycle C_2 from a later position to the current position to decrease the current number of unmatched parentheses in the path.

- The bidirectedness of the graph allows a path to go back and forth along a sequence of edges.
- This flexibility always allows us to transform a non-shallow path into a shallow one, by rearranging cycles in the original $D_1 \odot D_1$-path.

The following example provides some intuition about these points, particularly about how cycle rearrangement is used to transform a non-shallow $D_1 \odot D_1$-path into a shallow one.

Example 3.4 (Cycle Rearrangement). Consider the graph shown in Figure 4. There exists a $D_1 \odot D_1$-path from node a to node c that first goes through cycle C_1 seven times and then cycle C_2 five times. As we follow the path and reach node b for the first time, the tuple of unmatched open parentheses/brackets is $(35, 34)$. From this $D_1 \odot D_1$-path, we wish to construct a shallow(er) path, which we will do by exploiting bidirectedness.

To keep the example simple, let us assume that the bound is 30 (rather than $6n$). In general, our construction makes use of bidirectedness of the graph to go back and forth along a sequence of edges that connect two cycles in the path.

In our example, we need to rearrange the positions of the cycles so that the number of unmatched open parentheses never goes higher than 30. When the original path reaches node b for the first time, the number of unmatched parentheses/brackets for D_1 and D_1 both exceed 30. We can construct an alternative path in which the number of unmatched symbols for at least one of the two languages is no larger than 30; this path is labeled as the "rearranged path" in Figure 4. The rearranged path is based on the original path; the difference is that one occurrence of cycle C_2 is shifted earlier in the path. The rearranged path begins by going around cycle C_1 six times (instead of seven); then uses the edge $\langle a, b\rangle$ to reach node b; goes around cycle C_2 once; then uses the (reversed) edge $\langle b, a\rangle$ to return to node a; goes around cycle C_1 a final time; and finishes up by returning to b, going around C_2 four times, and going to node c. Because the first occurrence of cycle C_2 cancels some open parentheses and open brackets, the number of unmatched symbols for at least one of the two languages—in this case, both languages—is no larger than 30 at each point in the rearranged path. In general, it turns out that we can always transform a non-shallow $D_1 \odot D_1$-path into a shallow one via such cycle rearrangements (Section 4.1).

3.2 From Shallow Paths to a Tabulation Algorithm

Theorem 3.3 reduces the problem of bidirected $D_1 \odot D_1$-reachability to the problem of finding all pairs of nodes that are connected by a shallow $D_1 \odot D_1$-path. In this section, we sketch why shallow $D_1 \odot D_1$-reachability can be solved in **PTIME**. At a high level, the reason is that shallowness—*i.e.*, at each position in the path, the number of unmatched parentheses for at least one of the languages is bounded—enables us to use a summary-based approach to store information about certain segments of shallow paths. These summary edges can be tabulated in polynomial time, as can information about all node pairs that are connected by shallow $D_1 \odot D_1$-paths.

In Section 4.2, we give a tabulation algorithm for finding summary edges and all node pairs that are connected by shallow $D_1 \odot D_1$-paths (Algorithm 2). Figure 3 can be used to illustrate the concept behind the tabulation strategy used in the algorithm.

- For a shallow $D_1 \odot D_1$ path, because it never steps into the $(6n, \infty) \times (6n, \infty)$ region, once it steps out of the $[0, 6n] \times [0, 6n]$ region from the $a = 6n$ line, it always stays in the $(6n, \infty) \times [0, 6n]$ region until it comes back to the line $a = 6n$. It indicates that even though the path segments in the $(6n, \infty) \times [0, 6n]$ region can have an unbounded number of open parentheses in one Dyck language, eventually, these parentheses are all matched with the close parentheses. An analogous observation holds for the $[0, 6n] \times (6n, \infty)$ region.
- Such segments can be represented by *summary edges*, of which there are only a polynomial number, and we can compute all the summary edges for path segments in the $(6n, \infty) \times [0, 6n]$ and $[0, 6n] \times (6n, \infty)$ regions in polynomial time.
- We can explore all paths within the $[0, 6n] \times [0, 6n]$ region—including all summary edges on the $[0, 6n] \times \{6n\}$ and $\{6n\} \times [0, 6n]$ borders—in polynomial time.
- From the information tabulated in this fashion, we can enumerate all shallow $D_1 \odot D_1$-reachable pairs in polynomial time.

Theorem 3.5. *For a given bidirected $D_1 \odot D_1$-reachability problem, if there is a shallow $D_1 \odot D_1$-path between nodes u and v, Algorithm 2 will identify u, v are shallow $D_1 \odot D_1$-reachability.*

Together with Theorem 3.3, we have the following corollary:

Corollary 3.6. *Algorithm 2 solves the bidirected $D_1 \odot D_1$-reachability problem precisely.*

4 PROOF OF THE PTIME SOLVABILITY OF BIDIRECTED $D_1 \odot D_1$-REACHABILITY

This section presents the details of the **PTIME** proof for bidirected $D_1 \odot D_1$-reachability. In Section 4.1, we prove Theorem 3.3, demonstrating how to construct a shallow path, given a $D_1 \odot D_1$-reachable path. In Section 4.2, we prove Theorem 3.5, establishing that our tabulation algorithm identifies all shallow $D_1 \odot D_1$-reachable pairs, and analyze the time complexity of the algorithm. Our result shows that the bidirected $D_1 \odot D_1$-reachability problem is in **PTIME**.

4.1 Shallow-Path Property

This section proves that between any $D_1 \odot D_1$-reachable nodes u, v, there always exists a shallow path that connects u and v. We prove this property by showing that, given any $D_1 \odot D_1$-path P in a bidirected graph, we can construct a shallow $D_1 \odot D_1$-path P' based on P. In the shallow-path construction, we exploit the bidirectedness of the graph. The bidirected edges enable the constructed path P' to go back and forth over segments of P, which provides the flexibility to rearrange in P'

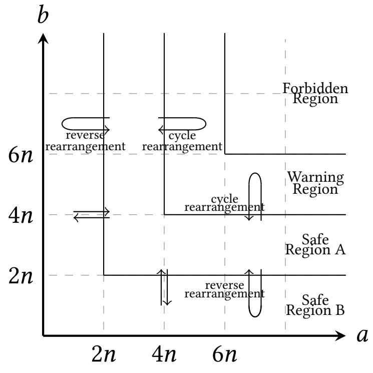


Fig. 5. Regions and region transitions during shallow-path construction.

the position of cycles from the original path P. We divide the tuples of unmatched open parentheses for a given path position into four regions as follows (see Figure 5):

Definition 4.1 (Unmatched Number Regions). *We classify a $D_1 \odot D_1$-path P at each node position as being in one of four states. In particular, for the path P at the i^{th} node position, we define four regions according to the values in the tuple (a, b) of unmatched open parentheses at the i^{th} node, as depicted in Figure 5:*

- ***Forbidden Region**: When the tuple (a, b) satisfies $a > 6n$ and $b > 6n$, path P is in the* forbidden region *at the i^{th} node. A shallow path never steps into the forbidden region.*
- ***Warning Region:** When the tuple (a, b) satisfies $a \geq 4n$ and $b \geq 4n$, but is not in the forbidden region, path P is in the* warning region *at the i^{th} node.*
- ***Safe Region A**: When the tuple (a, b) satisfies $a \geq 2n$ and $b \geq 2n$, but is not in the warning region or the forbidden region, path P is in* safe region A *at the i^{th} node.*
- ***Safe Region B**: If at the i^{th} node the path P is not in any of the regions defined above, it is in* safe region B *at the i^{th} node.*

We devise a method to construct a shallow path P' based on the original path P. The starting point for the construction is the observation that if the constructed path P' never steps into the forbidden region, P' is a shallow $D_1 \odot D_1$-path. The construction therefore "steers" P' away from the forbidden region. The key insight behind the method is that when we find path P' stepping into the *warning region*, we can move a cycle from a later position to the current position so that the constructed path P' is forced out of the warning region after the rearrangement. Because the warning region acts as a buffer around the forbidden region, P' never has a chance to step into the forbidden region. The proof is constructive, so we describe the proof in the form of algorithms.

Shallow Path Construction Algorithm. Algorithm 1 describes our approach to shallow-path construction. In the algorithm, we use a tuple (s_1, s_2) with $s_1, s_2 \in \{+, -, 0, ?\}$ to characterize the *type of a cycle*. When the cycle has more open-parentheses edges than close-parenthesis edges in the first Dyck language, s_1 is '+'. If it has more close parentheses than open parentheses, s_1 is '−'. For cycles with equal number of open and close parentheses, s_1 is '0'. When the relation between the numbers of open and close parentheses of the first Dyck language is unimportant, s_1 is '?'. The value of s_2 characterizes the cycle's type with respect to the second Dyck language in an analogous fashion. In the algorithm, we treat the nodes v_i, v_j, $i \neq j$, in a path $P = v_0 v_1 \ldots v_{|P|}$ as different nodes due to their different indexes, even though they may correspond to the same node in the actual graph. This treatment helps the algorithm distinguish cycles according to the

Algorithm 1: Shallow-Path Construction

```
Input  : Edge-labeled bidirected graph $G = (V, E)$,
         An $D_1 \odot D_1$-path $P = v_0 v_1 \cdots v_{|P|}$
Output : A shallow $D_1 \odot D_1$-path $P'$
1  $i \leftarrow 0$
2  $L_1, L_2 \leftarrow \varnothing$
3  $P' \leftarrow$ empty path
4  while $i \neq |P|$ do
5      prev $\leftarrow$ get_region $(P')$
6      $P' \leftarrow P' \cdot v_i$
7      $i \leftarrow i + 1$
8      cur $\leftarrow$ get_region $(P')$
9      $(a, b) \leftarrow$ unmatch $(P', |P'|)$
10     if prev is safe region $A$, cur is warning region then
11         if $a = 4n$ then
12             cycle rearrangement for cycle type $(-, ?)$
13         if $b = 4n$ then
14             cycle rearrangement for cycle type $(?, -)$
15     if prev is safe region $A$ and cur is safe region $B$ then
16         if there exists a $(-, -)$ cycle $C$ in $L_1 \cup L_2$ then
17             reverse rearrangement for $C$
18     if $v_i$ is in some cycle $C$ in the lists $L_1, L_2$ then
19         $i = i + |C|$
20         $L_1 \leftarrow L_1 \setminus \{C\}$,  $L_2 \leftarrow L_2 \setminus \{C\}$
21 return $P'$
```

sequence of (indexed) nodes that they consist of. For example, two cycles $C_1 = v_i v_{i+1} v_{i+2} v_{i+3}$ and $C_2 = v_j v_{j+1} v_{j+2} v_{j+3}$ are considered as two different cycles, even though they may actually represent the same cyclic sequence of edges in the graph.

In Algorithm 1, the variable i tracks the index in the original path P that corresponds to the position in the new path P'. We also introduce two *rearrangement lists* L_1 and L_2. In particular, L_1 records the cycles that are moved earlier in P' to lower the number of unmatched parentheses for the first Dyck language; L_2 records the moved cycles for the second Dyck language. The cycles in the lists L_1 and L_2 have been already used in the new path P', but not yet in the current prefix of the original path P. Algorithm 1 constructs the shallow path as follows:

- **Lines 5-9.** In most cases, the newly constructed path P' simply follows the original path P. In lines 6-7, the algorithm appends node v_i to P', i is incremented to indicate that node v_i has already been used in P'. In lines 5 and 8-9, the algorithm collects the information of unmatched tuple (a, b) of P' and the region of P' to help decide whether P' need to deviate from path P in the construction. The construction deviates from path P base on three cases.
- **Lines 10-14.** The algorithm detects that the constructed path P' steps into the warning region from safe region A. To prevent the path going further and possibly stepping into the forbidden region, the algorithm "moves" a cycle that occurs later in path P to the end of path P'.[2] As shown in lines 11-12, if the path steps into the warning region with $a = 4n$—*i.e.*, the

[2]The algorithm "moves" the cycle in the sense previously explained in Example 3.4.

number of unmatched open parentheses for the first Dyck language is $4n$—the algorithm uses a $(-,?)$ cycle C, which will bring down the number of unmatched parentheses for the first Dyck language, thereby forcing P' to leave the warning region. Then the rearrangement also adds cycle C to L_1, to record that this cycle has already been used in P', but has not yet been reached in the original path P. Lines 13-14 are the analogous steps for paths stepping into the warning region with $4n$ unmatched open parentheses of the second Dyck language. We call this cycle-moving behavior "**cycle rearrangement**." The details of cycle rearrangement are discussed in Procedure 1 later in the section, where we also demonstrate that (i) it is always feasible to perform cycle rearrangement, and (ii) during the rearrangement, path P' cannot step into the forbidden region.

- **Lines 15-17.** The algorithm detects that the path P' steps into the safe region B from safe region A. If there is a $(-,-)$ cycle C recorded in either L_1 or L_2, the algorithm moves the reverse cycle of C, denoted by $\overline{C}$, to the end of P', thereby cancelling the rearrangement effects for cycle C. It also removes cycle C from its rearrangement lists L_1 and L_2. We call this reverse-cycle movement "**reverse rearrangement**." We present the details of reverse rearrangement in Procedure 2 later in the section. Reverse rearrangement guarantees the validity of the constructed path P', *i.e.*, during the construction of P', there are always more open parentheses than close parentheses in both D_1 languages.
- **Lines 18-20.** The algorithm finds that the original path P encounters a cycle that has already been used in the constructed path P'. It skips over the cycle in P, and removes the cycle from the rearrangement lists L_1 and L_2.

Figure 5 illustrates all possible transitions between regions in shallow-path construction. Due to the invariants of the construction, a shallow path P' can never step into the forbidden region. Consequently, P' is a shallow $D_1 \odot D_1$-path. In the remainder of the section, we describe the details of cycle rearrangement and reverse rearrangement.

Cycle Rearrangement. When the constructed path P' steps into the warning region, Algorithm 1 performs a cycle rearrangement. The goal of cycle rearrangement is to lower the number of unmatched open parentheses in the constructed path P' by moving a $(-,?)$ or $(?,-)$ cycle that occurs in a latter position of P to the current position of P', thereby decreasing the number of unmatched open parentheses for the appropriate Dyck language, and forcing path P' to step out of the warning region after the rearrangement.

Procedure 1 performs the cycle rearrangement. The rearrangement is for a cycle of some specified type, held in the input parameter type. The cycle-rearrangement algorithm attaches the closest simple cycle C of the required type to the end of path P'. Suppose at the beginning of Procedure 1, a path P' ends with a node v_i. Procedure 1 finds the path in P from v_i to cycle C. This connecting path is accumulated in variable bridge, and the path fragment appended to P' is the sequence $\text{bridge} \cdot C \cdot \overline{\text{bridge}}$. Note that the effects of bridge and its reverse $\overline{\text{bridge}}$ cancel each other, in the sense that the net change in the tuple of unmatched open parentheses at the end of P' versus at the end of $P' \cdot \text{bridge} \cdot C \cdot \overline{\text{bridge}}$ is due to the unmatched parentheses in C. In addition, Procedure 1 also records the information that cycle C has been used in P' by adding cycle C to the appropriate rearrangement list, L_1 or L_2, according to the sought-for cycle type: if type is $(-,?)$, C is added to L_1; if type is $(?,-)$, C is added to L_2.

Specifically, in lines 3-17, Procedure 1 considers simple cycles that are successively further and further away from v_i, while gathering up a cycle-free connecting path bridge between node v_i and the current cycle under consideration. The test in lines 5-6 causes all cycles that are already in the rearrangement lists L_1 and L_2 to be skipped over because they have already been used in path P'. To maintain bridge as a cycle-free path, whenever the algorithm identifies a cycle in lines 7-16,

```
Procedure 1: Cycle Rearrangement
Input  : Edge-labeled bidirected graph $G = (V, E)$,
         The original $D_1 \odot D_1$ path $P = v_0 v_1 \cdots v_{|P|}$
         Currently constructed shallow path $P'$
         Rearrangement lists $L_1, L_2$
         Current corresponding node $v_i$
         The type of cycle to rearrange: type $\in \{(?, -), (-, ?)\}$
Output : Updated shallow path $P'$
         Modified rearrangement lists $L_1, L_2$
1  bridge $\leftarrow v_i$
2  pos $\leftarrow i$
3  while true do
4      pos $\leftarrow$ pos $+1$
5      if $v_{\text{pos}}$ is in any of the cycles of $L_1$ or $L_2$ then
6          continue
7      if $v_{\text{pos}}$ as a node already exists in bridge then
8          There exists a cycle $C$ in bridge
9          bridge $\leftarrow$ bridge $- C$
10         if $C$ belongs to the desired cycle type type then
11             $P' \leftarrow P' +$ bridge $+ C + \overline{\text{bridge}}$
12             if type is $(-, ?)$ then
13                 $L_1 \leftarrow L_1 \cup \{C\}$
14             if type is $(?, -)$ then
15                 $L_2 \leftarrow L_2 \cup \{C\}$
16             break
17     bridge $\leftarrow$ bridge $+ v_{\text{pos}}$
18 return $P', L_1, L_2$
```

the cycle is deleted from connecting path bridge. If the cycle satisfies the cycle-type requirement, bridge $\cdot C \cdot \overline{\text{bridge}}$ is appended to P' (line 11) , and C is recorded in the appropriate rearrangement list (lines 12-15). Otherwise, the algorithm looks for the next cycle.

Note that it is always feasible and valid to carry out the cycle-rearrangement operation when it is invoked by Algorithm 1. Consider the tuple (a, b) of unmatched open parentheses for node v_i at the end of path P'. We know that $a > 4n$ and $b > 4n$ because v_i is in the warning region. Then the cycle-rearrangement algorithm can always find a cycle with more close parentheses than open parentheses, otherwise the original path cannot be a valid $D_1 \odot D_1$-path. Then we show the validity of cycle rearrangement in terms of the shallow-path construction, *i.e.*, during cycle rearrangement, path P' cannot step into the forbidden region. When Algorithm 1 invokes cycle rearrangement, path P' has just stepped into the warning region. Without loss of generality, let us assume that it was the first Dyck language that just crossed the $4n$ threshold of unmatched open parentheses; i.e., $a = 4n + 1$. We observe that

- because bridge is acyclic, $|\text{bridge}| \leq n - 1$, and
- for the simple cycle C, $|C| \leq n$.

Consequently, the total number of unmatched open parentheses at any point in the fragment bridge $\cdot C \cdot \overline{\text{bridge}}$ is at most $2n - 1$. When bridge $\cdot C \cdot \overline{\text{bridge}}$ is attached to P', the total number of

```
Procedure 2: Reverse Rearrangement
Input  : Edge-labeled bidirected graph $G = (V, E)$,
         The original $D_1 \odot D_1$ path $P = v_0 v_1 \cdots v_{|P|}$
         Currently constructed shallow path $P'$
         Rearrangement list $L_1, L_2$
         Current corresponding node $v_i$
         The simple cycle $C$ to be reversed
Output : Updated shallow path $P'$
         Modified Rearrangement List $L_1, L_2$
1  bridge $\leftarrow v_i$
2  pos $\leftarrow i$
3  while true do
4      pos $\leftarrow$ pos +1
5      if $v_{\text{pos}}$ is in $C$ then
6          bridge $\leftarrow$ bridge + $v_{\text{pos}}$
7          $P' \leftarrow$ bridge + $\overline{C}$ + $\overline{\text{bridge}}$
8          if $C \in L_1$ then
9              Remove $C$ from $L_1$
10         else
11             Remove $C$ from $L_2$
12         break
13     bridge $\leftarrow$ bridge + $v_{\text{pos}}$
14     if $v_{\text{pos}}$ as a node already exists in bridge then
15         There exists a cycle $C$ in bridge
16         bridge $\leftarrow$ bridge − $C$
17 return $P', L_1, L_2$
```

unmatched parentheses for the first Dyck language cannot exceed $6n$ at any point, meaning that the path $P' \cdot \text{bridge} \cdot C \cdot \overline{\text{bridge}}$ never enters the forbidden region.

Reverse Rearrangement. In Algorithm 1, some simple cycles of P of type $(-,?)$ or $(?,-)$ are moved to earlier relative positions in P'. P' could use up all of the unmatched open parentheses, in which case one or both components of the tuple of unmatched open parentheses at some position of P' could become negative; this indicates that P' becomes an invalid $D_1 \odot D_1$-path. The shallow path construction uses reverse rearrangement to ensure the validity of the constructed path P'. In particular, reverse rearrangement attaches to P' the reverse cycle $\overline{C}$ of a moved $(-,-)$ cycle C that is already part of P'. The net effect of reverse rearrangement is to raise the number of open parentheses at the end of P': $\overline{C}$ is a $(+,+)$ cycle.

Procedure 2 presents the reverse-rearrangement method. The loop body (lines 4-16) carries out the search for the cycle C. Note that the search starts from position i (line 2), which represents the corresponding position in the original path P' of the current position in the constructed path P'. If the algorithm finds C, it appends $\text{bridge} \cdot \overline{C} \cdot \overline{\text{bridge}}$ to P'. The net effect is equivalent to appending the reverse cycle $\overline{C}$ at the end of P'. In lines 14-16, if the algorithm encounters an irrelevant cycle, it is removed from bridge. This guarantees bridge is always acyclic.

Note that the reverse-rearrangement operation is always feasible when Algorithm 1 invokes it. The connecting path bridge is acyclic, so we have $|\text{bridge}| \leq n - 1$. C is a simple cycle, so $|C| \leq n$. Consequently, the total number of close parentheses in $\text{bridge} \cdot \overline{C} \cdot \overline{\text{bridge}}$ is at most $2n - 1$. When

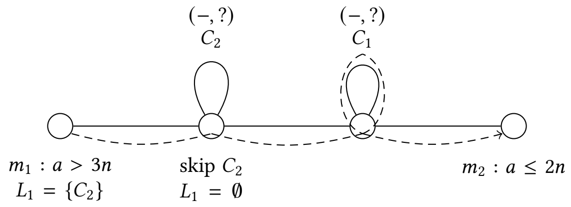


Fig. 6. The reason why $L_1 = \emptyset$ by the time P' encounters a $(-,?)$ cycle. The dashed arrow represents the path P' from the moment m_1 when it finishes the last cycle rearrangement, to the moment m_2 when the value of a becomes no larger than $2n$.

Algorithm 1 invokes reverse rearrangement, path P' has just stepped into safe region B. Thus, for the tuple of unmatched open parentheses (a, b), it has $a \geq 2n$ and $b \geq 2n$. Without loss of generality, let us assume that it was the first Dyck language that just touched the $2n$ threshold of unmatched open parentheses; i.e., $a = 2n$ and $b > 2n$. When $\text{bridge} \cdot \overline{C} \cdot \overline{\text{bridge}}$ is attached to P', the total number of unmatched open parentheses for the first Dyck language cannot reach 0 at any point, meaning that the path $P' \cdot \text{bridge} \cdot \overline{C} \cdot \overline{\text{bridge}}$ never uses up all the unmatched parentheses.

Correctness of Shallow Path Construction. To present the correctness of the shallow path construction, we first verify the constructed path is indeed shallow. Then we show that the constructed path is a valid InterDyck path. We verify the constructed path P' is shallow by showing that it never steps into the forbidden region. Algorithm 1 invokes cycle rearrangement to steer path P' from the warning region back to safe region A in lines 10-14. Similarly, for each invocation of reverse rearrangement in lines 15-17 of Algorithm 1, it steers P' from safe region B back to safe region A. The reason is that the reversal of a $(-,-)$ cycle is a $(+,+)$ cycle, and appending a $(+,+)$ cycle directs the trajectory of P' in the north-east direction in Figure 5. As we described above, during the construction, P' never steps into the forbidden region, thus it is a shallow path.

Then we show that during the construction, the path P' will never use up unmatched open parentheses/brackets, thus it is a valid InterDyck path. We start with a Lemma.

Lemma 4.2. *Let (a, b) = unmatch(P', i). If $L_1 \neq \emptyset$, then $a > 2n$; if $L_2 \neq \emptyset$, then $b > 2n$.*

Proof. By the symmetry of a and b, it suffices to prove just one of the two cases—if $L_1 \neq \emptyset$, then $a > 2n$. We prove the lemma by contradiction. Suppose that the path P' constructed so far satisfies $L_1 \neq \emptyset$ and $a \leq 2n$. Let us compare two moments m_1 and m_2 in the construction of P': (i) m_1, earlier in the construction of P', just after the most-recent cycle rearrangement, and (ii) m_2, the (hypothetical) current moment when we have $L_1 \neq \emptyset$ and $a \leq 2n$. At m_1, we must have $a > 3n$. The reason is because a cycle rearrangement happens when path P' steps into the warning region (and hence $a > 4n$), and a cycle rearrangement always attaches a simple cycle with length less than n. Thus, as the construction proceeded from m_1 to m_2, the number a of unmatched open symbols for the first Dyck language changed from $a > 3n$ to $a \leq 2n$. In other words, a dropped in value by more than n from moment m_1 to moment m_2.

We first demonstrate that such a drop can happen only if P' passes through some $(-,?)$ cycle C_1. To establish this claim, suppose that P' does not go through a $(-,?)$ cycle; then the path appended to P' as the construction proceeded from m_1 to m_2 can be decomposed into several $(+,?)$ cycles and a set of path segments S_{chains}, where there are no repetitions of nodes among any of the path segments in S_{chains}. Each $(+,?)$ cycle can only increase a, so only the path segments in S_{chains} can decrease the value of a. Because S_{chains} has no repeated nodes, the total number of nodes is less than n, which contradicts the fact that the decrease of a needs to be larger than n. Consequently, as the construction proceeded from m_1 to m_2, P' must have been extended with some $(-, ?)$ cycle C_1.

All cycles recorded in L_1 have the type $(-, ?)$. Because of the greedy search carried out by Procedure 1, cycle rearrangement is always performed with respect to the $(-, ?)$ cycle that is *closest* to the position in path P at the time Procedure 1 is invoked. Consequently, at moment m_1 every cycle recorded in L_1 comes earlier in path P than cycle C_1. Figure 6 depicts this situation, where cycle C_2 is an arbitrary member of L_1. The dashed arrow indicates how as the construction of P' proceeds from m_1 to m_2, each cycle in L_1 (such as C_2) is skipped over and removed from L_1—see lines 18-20 of Algorithm 1. This aspect of Algorithm 1 will drain L_1 of all its members before the algorithm gets to append cycle C_1 to P'. As a consequence, $L_1 = \emptyset$ at moment m_2, which contradicts the assumption that $L_1 \neq \emptyset$. We have shown that it is impossible to have $L_1 = \emptyset \wedge a \leq 2n$, and thus if $L_1 \neq \emptyset$, then $a > 2n$ as was to be shown. □

The following theorem establishes that the constructed path P' is a valid $D_1 \odot D_1$-path.

Theorem 4.3. *In the newly constructed path P', at each node position j, let (a, b) = unmatch(P', j). Then $a \geq 0 \wedge b \geq 0$. Consequently, P' is a valid $D_1 \odot D_1$-path.*

Proof. We prove this theorem by contradiction, showing that it is impossible to have $a < 0$ or $b < 0$. Without loss of generality, suppose that at some moment during the construction of P', $a < 0$ at the end of P'—*i.e.*, P' moved from safe region B to a situation in which $a < 0$. In this case, by Lemma 4.2, we must have $L_1 = \emptyset$—otherwise, a must be greater than $2n$.

Now consider the other rearrangement list, L_2. First, suppose that L_2 is also $\emptyset$—i.e., both $L_1 = \emptyset$ and $L_2 = \emptyset$. In this situation, either (i) no cycle rearrangements have been performed so far in creating P', (ii) for each rearranged cycle C that appears in P', a reverse rearrangement with respect to C has also been performed in P', or (iii) for each rearranged cycle C that appears in P', the while-loop in lines 4–20 of Algorithm 1 has also encountered C and removed it from L_1 or L_2 in line 20. Note that in Algorithm 1, the value of variable i establishes the correspondence between the current position in P' and a prefix $v_0v_1 \ldots v_i$ of the original path P. When $L_1 = \emptyset \wedge L_2 = \emptyset$, if P' contains a cycle C from a rearrangement, it either contains the cycle $\overline{C}$ from a subsequent reverse rearrangement, or the prefix $v_0v_1 \ldots v_i$ of path P also contains the cycle C. The prefix $v_0v_1 \ldots v_i$ of P and P' are "synchronized" in the weak sense that $\mathtt{unmatch}(P, i) = \mathtt{unmatch}(P', |P'|)$. In particular, the number of unmatched open parentheses at the ends of $v_0v_1 \ldots v_i$ and P' must be equal. Consequently, it is impossible to have $a < 0$, otherwise at location i in P, the number of unmatched open parenthesis would also be negative. It means that the original path P is not a valid $D_1 \odot D_1$-path, which contradicts the underlying assumption of the shallow-path construction.

Now suppose that $L_2 \neq \emptyset$. Except during cycle rearrangement, path P' is either in safe region B or safe region A throughout the shallow-path construction. Cycles can only be inserted in the rearrangement list L_2 by cycle rearrangements. After a cycle rearrangement, the path P' is in the safe region A. Note that if P' ever enters safe region A, it can only re-enter safe region B after all $(-, -)$ cycles have been drained from L_2 and L_1. The reason is because whenever a reverse rearrangement is performed in lines 15–17 of Algorithm 1, P' is steered back to safe region A. Thus, P' can only re-enter safe region B if there are no more $(-, -)$ cycles in $L_1 \cup L_2$ Consequently, whenever P' is in safe region B, if there are cycles recorded in L_2 they must have the type $(\{0, +\}, -)$. Now consider path P': it contains each of the $(\{0, +\}, -)$ cycles that are recorded in L_2. We now compare the number of unmatched open parentheses at the end of prefix $v_0v_1 \ldots v_i$ of P and at the end of P' (as constructed so far). Because all the cycles in L_2 have type $(\{0, +\}, -)$, path P' can have some additional unmatched open parentheses compared to path P at position i. If at the end of P' we have $a < 0$, then the number of unmatched open parentheses at position i of P must also be negative. That would mean that path P is not a valid $D_1 \odot D_1$-path, which again contradicts the underlying assumption of the shallow-path construction. □

Theorem 4.3 guarantees that the constructed path is always a valid $D_1 \odot D_1$-path. The shallow-path construction also ensures that P' cannot step into the forbidden region. Consequently, the path P' is a shallow $D_1 \odot D_1$-path.

4.2 Bidirected $D_1 \odot D_1$-Reachability Algorithm

This section presents our algorithm to compute all pairs connected by shallow $D_1 \odot D_1$-paths for bidirected $D_1 \odot D_1$-reachability problem, which—by the results of Section 4.1—provides the solution to the bidirected $D_1 \odot D_1$-reachability problem. As shown in Figure 3, for a shallow path, we can enumerate all the intermediate states of the path in the $[0, 6n] \times [0, 6n]$ region. For the path segments that pass out of the square region $[0, 6n] \times [0, 6n]$, we track them using the "CO summary", which is defined as follows:

Definition 4.4 (Close-Open (CO) summary). *A CO summary counts a bounded number of unmatched close parentheses and unmatched open parentheses, while allowing any number of matched brackets. The CO summary of unbalanced parentheses, denoted by $\mathrm{CO}_{)(}$, is defined by the grammar:*

$$C_0O_0 ::= \epsilon$$
$$C_iO_j ::= \text{`['}\, C_iO_j \,\text{`]'} \quad \text{for } 0 \le i, j \le 6n$$
$$C_iO_j ::= \text{`)'}\, C_{i-1}O_j \quad \text{for } 1 \le i \le 6n$$
$$C_iO_j ::= C_iO_{j-1}\, \text{`('} \quad \text{for } 1 \le j \le 6n$$
$$C_iO_j ::= C_aO_b\ C_cO_d \quad \text{where } i = a + \max(c - b, 0), j = d + \max(b - c, 0), 0 \le i, j, a, b, c, d \le 6n.$$

The language of CO summary involves the alphabets of two Dyck languages. Above, we used parentheses to represent the symbols of the first Dyck language and brackets for the symbols of the second Dyck language. n denotes the number of nodes in the graph G.

The CO summary of the unbalanced brackets is also defined in a similar manner, denoted by $\mathrm{CO}_{][}$, in which the roles of parentheses and brackets are interchanged. Both $\mathrm{CO}_{)(}$ and $\mathrm{CO}_{][}$ are context-free languages, and hence the all-pairs $\mathrm{CO}_{)(}$ and all-pairs $\mathrm{CO}_{][}$-reachability problems can be computed via standard *CFL*-reachability algorithms in polynomial time [Reps 1998]. We refer to such information as "CO summary information."

We then argue that the CO summary information can summarize all path segments that pass out of, and then return to, the $[0, 6n] \times [0, 6n]$ region in Figure 3. The path segment $u \rightsquigarrow v$ in Figure 3 is an example of the kind of path segment that will be summarized by CO summary information. In a $\mathrm{CO}_{)(}$-path, there can be no more than $6n$ unmatched close and open *parentheses*, whereas the *brackets* are always matched. Similarly, in a $\mathrm{CO}_{][}$-path, there can be no more than $6n$ unmatched close and open *brackets*, whereas the *parentheses* are always matched. Consider again the path segment $u \rightsquigarrow v$, it has no more than $6n$ unmatched close and open brackets, but the parentheses are always matched. In other words, $u \rightsquigarrow v$ is a $\mathrm{CO}_{][}$-path. A dual kind of path segment that passes out of the central $[0, 6n] \times [0, 6n]$ region, traverses the $[0, 6n] \times (6n, \infty)$ region, and then returns to the $[0, 6n] \times [0, 6n]$ region has no more than $6n$ unmatched close and open parentheses, but the brackets are always matched. Consequently, it is a $\mathrm{CO}_{)(}$-path. Then, the finite sets of pairs obtained by computing all-pairs $\mathrm{CO}_{)(}$ and all-pairs $\mathrm{CO}_{][}$-reachability can summarize all possible path segments of shallow $D_1 \odot D_1$-paths in G that pass out of, and return to, the central $[0, 6n] \times [0, 6n]$ region.

Assume that we have the access to the CO summary information $S_1 = \mathrm{CO}_{][}$ and $S_2 = \mathrm{CO}_{)(}$, which can be computed in polynomial-time, Algorithm 2 identifies all $D_1 \odot D_1$ shallow-path-reachable pairs. Algorithm 2 constructs a new undirected graph G'. Each node in G' has a label (v, a, b). The

Algorithm 2: Precise $D_1 \odot D_1$ Algorithm

```
Input  : Edge-labeled bidirected graph $G = (V, E)$,
         CO summary information $S_1 = CO_{][}$
         CO summary information $S_2 = CO_{)(}$
Output : A set of $D_1 \odot D_1$ reachable pairs Sol
1  Sol ← ∅.
2  $G'$ ← a new graph with $n * 6n * 6n$ nodes and no edges.
3  In graph $G'$, give each node a label $(i, j, k)$ where $1 \le i \le n$, $1 \le j, k \le 6n$
4  foreach $u \xrightarrow{(} v \in E$ do
5      for j = 1 to 6n-1 do
6          for k = 1 to 6n do
7              add an undirected edge between node $(u, j, k)$ and $(v, j + 1, k)$ in $G'$
8  foreach $u \xrightarrow{[} v \in E$ do
9      for j = 1 to 6n do
10         for k = 1 to 6n-1 do
11             add an undirected edge between node $(u, j, k)$ and $(v, j, k + 1)$ in $G'$
12 foreach summary $u \xrightarrow{C_a O_b} v \in S_1$ do
13     for $j = a$ to $\min(6n - b + a, 6n)$ do
14         for k = 1 to 6n do
15             add an undirected edge between node $(u, j, k)$ and $(v, j - a + b, k)$ in $G'$.
16 foreach summary $u \xrightarrow{C_a O_b} v \in S_2$ do
17     for j = 1 to 6n do
18         for $k = a$ to $\min(6n - b + a, 6n)$ do
19             add an undirected edge between node $(u, j, k)$ and $(v, j, k - a + b)$ in $G'$
20 Run an undirected-graph-reachability algorithm on $G'$
21 foreach node pair $((u, 0, 0), (v, 0, 0)) \in G'$ do
22     if $(u, 0, 0)$ and $(v, 0, 0)$ are connected $\in G'$ then
23         Sol ← Sol ∪ {(u, v)}
24 return Sol
```

first entry, v, is a node in the original graph G, whereas a and b are counts of unmatched open parentheses and unmatched open brackets, respectively. We only create nodes in G' for which $0 \le a, b \le 6n$, and thus G' has $36n^3 + 12n^2 + n$ nodes in total.

- **Lines 4-11.** For every open-parenthesis-edge between v_1, v_2 in the original graph, we create an undirected edge between (v_1, i, j) and $(v_2, i + 1, j)$ as described in lines 4-7. Similarly, in lines 8-11, for an open-bracket-edge between v_1, v_2, we create an undirected edge between (v_1, i, j) and $(v_2, i, j + 1)$. Thus, all path segments of shallow $D_1 \odot D_1$-paths in G that stay within the central $[0, 6n] \times [0, 6n]$ region will be connected in G'. For example, if between nodes u, v in G there is a path with two unmatched open parentheses and three unmatched open brackets, then there will be a path in G' between the nodes $(u, 0, 0)$ and $(v, 2, 3)$.
- **Lines 12-19.** To capture the path segments that pass out of and return to the $[0, 6n] \times [0, 6n]$ square, we use the *CO* summary information to guide the edge construction. In lines 12-15, if

nodes u, v are connected by a $C_p O_q$ summary in $S_1 = \mathrm{CO}_{][}$, then we connect node (u, i, j) and $(v, i, j - p + q)$. Similarly in lines 16-19, the algorithm performs the similar construction for CO summary information in $S_2 = \mathrm{CO}_{)(}$. By this means, all path segments of shallow $D_1 \odot D_1$-paths in G that pass out of and return to the $[0, 6n] \times [0, 6n]$ square are represented in the new undirected graph G'.

- **Lines 20-23.** Bidirected $D_1 \odot D_1$-reachability is now performed by a standard algorithm for undirected graph reachability on G'. As shown in lines 21-23, if nodes $(u, 0, 0)$ and $(v, 0, 0)$ are connected, nodes u, v are $D_1 \odot D_1$-reachable in the bidirected $D_1 \odot D_1$-reachability problem.

Complexity Analysis of the Bidirected $D_1 \odot D_1$-reachability Algorithm. Here we analyze the time complexity of our algorithm for bidirected $D_1 \odot D_1$-reachability. The algorithm has two steps: (i) compute the CO summary information, and (ii) perform Algorithm 2. To compute the CO summary information in step (i), we can use any standard *CFL*-reachability algorithm. One standard algorithm has $O(n^3)$ time complexity when the grammar size is considered to be a constant [Reps 1998]. However, in our case the size of the grammar is not a constant: the CO summary grammar has n^2 non-terminals and $O(n^4)$ productions. A variation of the time-complexity argument for the standard algorithm can be used to show that the running time for the *CFL*-reachability step is $O(n^7)$. For step (ii), the for loops in lines 4-7 and 8-11 of Algorithm 2 have time complexity $O(n^3)$. Lines 12-15 and 16-19 insert edges into graph G'. G' has $O(n^3)$ nodes, and in the worst case the number of edges is $O(n^6)$; thus, in the worst case, the number of insertions is $O(n^6)$. Finally, running a standard algorithm for undirected-graph reachability on a graph with m edges takes $O(m)$ time, and thus the time complexity of line 20 is $O(n^6)$. To sum up, the total running time for the precise $D_1 \odot D_1$-reachability algorithm is $O(n^7)$; therefore, the $D_1 \odot D_1$-reachability problem is in **PTIME**.

5 BIDIRECTED $D_K \odot D_K$-REACHABILITY COMPLEXITY

The InterDyck-reachability problem on directed graphs has been known to be undecidable [Reps 2000]. Note that the graph construction used in the undecidability proof of Reps [2000] cannot be applied to the bidirected case. This section shows that even with the bidirected relaxation, the problem is still at least **NP**-hard. We establish the hardness by reducing an **NP**-complete problem, paths avoiding forbidden pairs, to the bidirected $D_k \odot D_k$-reachability problem.

5.1 Overview

We first present an overview of the bidirected $D_k \odot D_k$-reachability proof. In our reduction from the paths avoiding forbidden pairs (PAFP) problem to bidirected $D_k \odot D_k$-reachability, we construct a new graph such that $D_k \odot D_k$-reachability between nodes of the new graph provides the solution for the corresponding PAFP instance. The graph for $D_k \odot D_k$-reachability consists of three major parts: the transformed graph, the preprocessing graph, and the checker graph. To facilitate the discussion of the reduction, we first reduce the PAFP problem to another variant of bidirected InterDyck-reachability: the N-fold D_1-reachability problem (Section 5.2). The reduction to N-fold D_1-reachability serves as a bridge to the reduction to bidirected $D_k \odot D_k$-reachability. In particular, it contains the same three parts as the bidirected $D_k \odot D_k$-reachability reduction. However, one of the parts, the checker graph, is much simpler and more intuitive, thus paving the way for the illustration of the reduction to bidirected $D_k \odot D_k$-reachability (Section 5.3). We start by introducing the PAFP problem, which is defined as follows:

Definition 5.1 (Paths Avoiding Forbidden Pairs). *Given a graph $G = (V, E)$ and two nodes s, $t \in V$ and a set of node pairs $F = \{f_1, \ldots, f_{|F|}\} \subseteq (V \times V)$, the paths avoiding forbidden pairs problem is to decide whether there exists a path from s to t such that the path contains at most one node in each node pair in F. Such paths are called F-paths.*

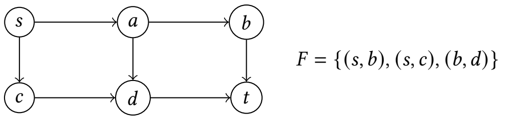


Fig. 7. A path avoiding forbidden pairs instance.

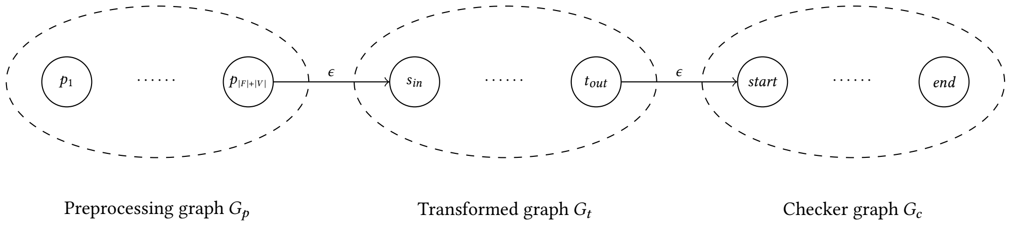


Fig. 8. Overview for the N-fold D_1 reachability reduction.

The complexity of PAFP problems has been extensively studied in the literature [Gabow et al. 1976; Kovác 2014]. It is known that the complexity for many variants of the problem are **NP**-complete, and here we focus on the variant on directed acyclic graphs for our reduction.

Theorem 5.2 ([Gabow et al. 1976]). *The paths with forbidden pairs problem on directed acyclic graphs is* **NP**-*complete.*

Example 5.3 (Paths Avoiding Forbidden Pairs). Figure 7 provides an instance of the PAFP problem. Consider the directed acyclic graph in Figure 7. It is associated with a forbidden-pair set $F = \{(s,b),(s,c),(b,d)\}$. The instance is to decide whether there exists a path from node s to node t such that the path only involves at most one of the nodes in each forbidden pair in the set F. This instance has a positive answer because the path $s \longrightarrow a \longrightarrow d \longrightarrow t$ is an F-path, *i.e.*, it involves at most one node in each forbidden pair.

5.2 Complexity of Bidirected N-Fold D_1-Reachability

This section establishes the **NP**-hardness of the N-fold D_1-reachability problem on bidirected graphs. It serves as an warm-up construction for $D_k \odot D_k$-reachability on bidirected graphs. We define the bidirected N-fold D_1-reachability problem as follows.

Definition 5.4 (Bidirected N-fold D_1-reachability). *An instance of the N-fold D_1-reachability problem is an instance of L-reachability where L is an interleaved Dyck language $L ::= \underbrace{D_1 \odot D_1 \odot \cdots \odot D_1}_{N\ times}$. An instance of the bidirected N-fold D_1-reachability problem is a problem instance on a graph that is bidirected for the alphabet symbols in the N Dyck languages.*

We then present the polynomial time reduction from PAFP problem to N-fold D_1-reachability problem. In this case, the upper bound for N is $O(|V|^2)$, where V is the set of nodes of the graph $G = (V, E)$ in the PAFP problem. In the reduction, we construct a new graph with three major parts: the transformed graph G_t, the preprocessing graph G_p, and the checker graph G_c, illustrated in Figure 8. In the graph constructed for N-fold D_1-reachability, the preprocessing and transformed graphs are connected by an edge labeled by the ϵ symbol from the last node in G_p to the s_{in} node in G_t. The transformed graph and the checker graph are connected by another "ϵ"-edge from the t_{out} node in G_t to the start node in G_c. We design the graphs with the following properties in mind.

- **Transformed Graph.** The transformed graph G_t is based on the original graph in the PAFP instance. There exists an F-path between nodes s and t in the PAFP problem iff there exists a path in the transformed graph from s_{in} to t_{out} that spells out a word consisting of only non-repeated open parentheses.

- **Preprocessing Graph.** The preprocessing graph G_p serves to append an arbitrary prefix of up to $|F| + |V|$ open parentheses. F is the set of the forbidden pairs and V is the node set of the PAFP graph. The constructed graph has $|F| + |V|$ different kinds of open parentheses. The preprocessing graph and the transformed graph can produce a path that uses every kind of open parenthesis exactly once iff the PAFP instance has a solution. This path could start from an arbitrary node in the preprocessing graph and end at node t_{out} in G_t.
- **Checker Graph.** The checker graph G_c will consume exactly one instance of every kind of open parenthesis, thereby checking whether G_p and G_t can produce such a path.

This construction reduces a PAFP instance to a corresponding instance of the N-fold D_1-reachability problem. If any node in the preprocessing graph is N-fold D_1-reachable to the end node of the checker graph, then in the original PAFP problem, there is an F-path from s to t. If the PAFP problem has a solution, then at least one node in the preprocessing graph is N-fold D_1-reachable to the end node of the checker graph. The rest of the section describes the three major parts of the constructed graph for N-fold D_1-reachability.

Transformed-Graph Construction. To construct the transformed graph G_t, for every node v in the original graph, we introduce two nodes v_{in} and v_{out} in the transformed graph. In the PAFP instance, the forbidden pair set $F = \{f_1, f_2, \dots, f_{|F|}\}$ has a subset $F_v = \{f_{i_1}, \dots, f_{i_{|F_v|}}\}$ that contains all the forbidden pairs that involve node v. Then we connect v_{in} to v_{out} with a path that spells out the word that consists of all the open parentheses that have an index corresponding to a forbidden pair in F_v, followed by a final open parenthesis that represents node v itself, e.g., "$(_{i_1}(_{i_2}\dots(_{i_{|f_v|}}(_{node_v}$". If there is no forbidden pair involving node v, *i.e.*, $F_v = \emptyset$, then we connect v_{in} with v_{out} with a single edge labeled "$(_{node_v}$". The edges in the transformed graph are bidirected. Figure 9 provides an example of the construction, for a node a in the original graph. Suppose that in the forbidden-pair set F, the subset of all forbidden pairs involving node a is $F_a = \{f_i, f_j, f_k\}$. We then add a path from a_{in} to a_{out} that spells out the word "$(_i(_j(_k(_{node_a}$". For every edge $e = (u, v)$ in the original PAFP graph, we insert a bidirected ϵ edge from u_{out} to v_{in} in the transformed graph G_t. The transformed graph G_t ensures the property in the following lemma.

Lemma 5.5. *In the transformed graph, there exists a path from s to t with only non-repeated open parentheses if and only if there exists an F-path in the original graph from s to t.*

Proof. *The "if" direction.* Suppose that in the original paths avoiding forbidden pair problem, there exists an F-path from node s to t. Then we can always find a simple F-path in the graph. Because the F-path involves at most one node in each node pair, for a simple F-path, due to the construction its corresponding path in the transformed graph contains at most one instance of each kind of open-parenthesis symbol.

The "only if" direction. Suppose that we have a path whose word contains at most one instance of each open-parenthesis symbol. Then there is always a simple path satisfying this "at most one instance" property as well. For the simple path, the construction forbids the possibility to exploit the bidirected edges in the transformed graph. Between every pair of nodes v_{in} and v_{out}, to use the reversed edges from v_{out} to v_{in}, the $)_{node_v}$ parenthesis edge must be matched with the corresponding open parenthesis, which only exists in the edge from v_{in} to v_{out}. If there exists such an open parenthesis in the path, the path is not a simple path, because both of the nodes v_{in} and v_{out} are used twice. Thus, the simple path with the word containing each open parenthesis at most once corresponds to an F-path in the PAFP instance. □

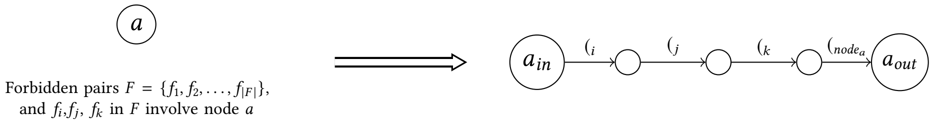

a
Forbidden pairs $F = \{f_1, f_2, \ldots, f_{|F|}\}$, and f_i, f_j, f_k in F involve node a
a_{in}
$(_i$
$(_j$
$(_k$
$(_{node_a}$
a_{out}

Fig. 9. Node construction in transformed graph.

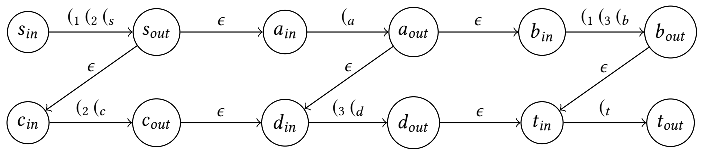


Fig. 10. Transformed graph construction example for the PAFP instance in Example 5.3.

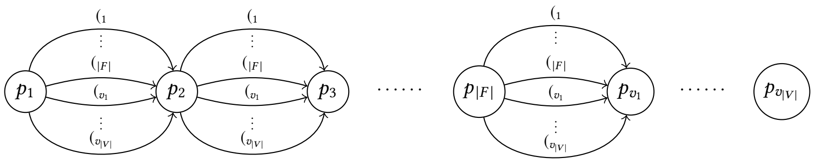


Fig. 11. Preprocessing graph.

Example 5.6 (Transformed-Graph Construction). Here we present a concrete example of the transformed-graph construction, based on the PAFP from Figure 7. In Figure 10, given the original graph and the forbidden-pair set F, we can construct the corresponding transformed graph following the construction described above.

This example illustrates the property of the transformed graph. For an F-path $s \longrightarrow a \longrightarrow d \longrightarrow t$ in the original graph, there is a corresponding path $s_{\text{in}} \longrightarrow s_{\text{out}} \longrightarrow a_{\text{in}} \longrightarrow a_{\text{out}} \longrightarrow d_{\text{in}} \longrightarrow d_{\text{out}} \longrightarrow t_{\text{in}} \longrightarrow t_{\text{out}}$ spells out the word "$(_1(_2(_s(_a(_3(_d(_t$", which contains each open parenthesis at most once.

Preprocessing-Graph Construction. The transformed graph only produces paths with *at most one* edge for each kind of open parenthesis. However, the checker graph verifies whether there exists a path that contains every kind of open parenthesis *exactly once.* Preprocessing graph G_p appends the other missing open parentheses that are not included in the transformed graph. For convenience, we denote the node set $V = \{v_1, \ldots, v_{|V|}\}$ and their corresponding parentheses as $(_{v_1}, \ldots (_{v_{|V|}}$. In the preprocessing graph G_p, we have $|F| + |V|$ nodes $p_1, \ldots, p_{|F|+|V|}$. Each pair of nodes p_i and p_{i+1} are connected by $|F| + |V|$ parallel edges, with the labels "$(_1$", "$(_2$", $\ldots$, "$(_{|F|}$", "$(_{v_1}$", $\ldots$ "$(_{v_{|V|}}$", as shown in Figure 11. The edges constructed are bidirected. In the preprocessing graph, a path from node p_i to p_n can append $n - i$ arbitrary open parenthesis to a path that starts at p_n. Due to the construction, we obtain the following lemma:

Lemma 5.7. *In the constructed graph, s_{in} is N-fold D_1-reachable to t_{out} if and only if there exists a path from some node p_i in the preprocessing graph to the node start in the checker graph with exactly one open parenthesis for each kind.*

Checker-Graph Construction. In the checker graph, there is one path from node start to node end that consumes every kind of open parenthesis exactly once. To construct such a checker graph for N-fold D_1-reachability, we only need to create a path that spells out "$)_1)_2 \ldots)_{|F|})_{v_1} \ldots)_{v_{|V|}}$" from node start to node end (see Figure 12). Note that each constructed edge should be bidirected. In the checker graph, each close parenthesis consumes one corresponding open parenthesis. The checker

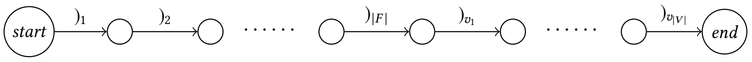


Fig. 12. Checker graph for N-fold D_1-reachability.

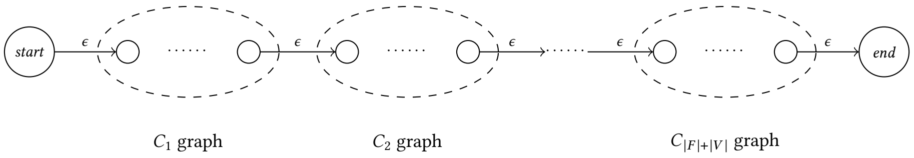


Fig. 13. Overview of checker graph G_c for bidirected $D_k \odot D_k$-reachability.

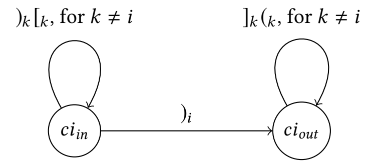


Fig. 14. Checker component C_i for bidrected $D_k \odot D_k$-reachability. The ci_{in} node contains $|F|+|V|-1$ parallel loops. The purpose is to transform the open parenthesis "$(_k$" to "$[_k$" in the other language. The "$)_i$"-edge between the ci_{in} node and ci_{out} node consumes exactly one "$(_i$" parenthesis. The parallel loops at the ci_{out} node can transform the "$[_k$"-edges back to parenthesis edges. The total effect of the C_i component is to consume an "$(_i$" parenthesis regardless its position in the path.

graph contains exactly one occurrence of each kind of close parenthesis, thus matching with a path that has exactly one occurrence of each kind of open parenthesis.

When we put the three component graphs together, if there exists an F-path, then there exists an path from some node p_i in the preprocessing graph to the node start of the checker graph with exactly one occurrence of each kind of open parenthesis. The continuation of the path to the end node of the checker graph causes the path to be accepted as an N-fold D_1-path. Conversely, if we find any N-fold D_1-path between any node in the preprocessing graph and the end node of the checker graph, there must exist a path in the transformed graph that contains every open parenthesis at most once. We can interpret that path as a solution in the paths avoiding forbidden pairs problem. We have reduced the paths avoiding forbidden pairs problem to the bidirected N-fold D_1-reachability problem, and thus the bidirected N-fold D_1-reachability is **NP**-hard.

Theorem 5.8. *The bidirected N-fold D_1-reachability problem is* **NP**-*hard.*

5.3 Complexity of Bidirected $D_k \odot D_k$-Reachability

The reduction from a PAFP instance to an instance of bidirected $D_k \odot D_k$-reachability again involves constructing a graph with three major parts. The preprocessing graph and the transformed graph used in the reduction remain the same as for N-fold D_1-reachability. The difference in the construction for the bidirected $D_k \odot D_k$-reachability problem is a more complicated checker graph.

We first provide an overview of the structure of the checker graph G_c in Figure 13. There are $|F| + |V|$ components between the start and end nodes, each constructed to consume one kind of open parenthesis exactly once. In the reduction for the bidirected $D_k \odot D_k$-reachability problem, all the open parentheses that appear in the preprocessing graph and the transformed graph are symbols in the first Dyck language. The use of interleaving with symbols from the second Dyck language only comes into play in the checker graph. In contrast, in the construction for N-fold D_1-reachability, the open parentheses in the preprocessing graph and the transformed graph came from separate D_1 languages. As we will see, for the bidirected $D_k \odot D_k$-reachability problem, in the checker graph, we also need to handle some issues related to the ordering of the open parentheses along the path through the preprocessing graph and the transformer graph.

We denote the i^{th} graph component of the checker graph by C_i. In graph C_i, the goal is to consume one $(_i$ parenthesis from the path that came through the preprocessing graph and the transformer graph. Overall, the goal of the checker graph is to consume every open-parenthesis symbol (of the first Dyck language) exactly once. For each checker component C_i, as illustrated in Figure 14, the checker needs to consume one occurrence of the symbol "$(_i$". We illustrate the functionality of such a component with a concrete example. Suppose that there is a path "$(_1(_2(_3$" before the checker C_2. The purpose of the loop in the ci_{in} node (in this case $i = 2$) in Figure 14 is to transform the open parentheses after "$(_2$" to correspondingly numbered bracket symbols. After the loop, the path can be "$(_1(_2(_3)_3[_3$". It then "consumes" the "$(_2$" symbol, and the path becomes "$(_1(_2(_3)_3[_3)_2$". Finally, the loop on the second node in Figure 14 will transform the bracket edges back to the corresponding parenthesis edges: the path "$(_1(_2(_3$" becomes "$(_1(_2(_3)_3[_3)_2]_3(_3$" after the checker component C_2. Notice that because the path labeled "$(_1(_2(_3)_3[_3)_2]_3(_3$" can always be matched by the same close-parenthesis string as the path labeled "$(_1(_3$", we can regard checker component C_2 as having "consumed" exactly one open-parenthesis symbol from the middle of the string, namely "$(_2$". The checker graph for the construction checks whether a path contains each open-parenthesis symbol exactly once. By an argument similar to the one given in Section 5.2, we can show that this construction reduces the PAFP problem to bidirected $D_k \odot D_k$-reachability.

Theorem 5.9. *The bidirected $D_k \odot D_k$-reachability problem is* **NP***-hard.*

In addition, we can use a D_2 language to simulate the behavior of a D_k language in polynomial time. For example, a naïve approach is to use "$(_1\underbrace{(_2 \dots (_2}_{i\ times}$" to represent "$(_i$." Thus, we have:

Corollary 5.10. *The bidirected $D_2 \odot D_2$-reachability problem is* **NP***-hard.*

6 EVALUATION

We have implemented the precise bidirected $D_1 \odot D_1$-reachability algorithm in Algorithm 2 and applied it to a context- and field-sensitive alias analysis for Java. The bidirected $D_1 \odot D_1$-reachability algorithm provides a new over-approximation approach for bidirected $D_k \odot D_k$-reachability. Specifically, it first relaxes bidirected $D_k \odot D_k$-reachability to bidirected $D_1 \odot D_1$-reachability and then solves the $D_1 \odot D_1$-reachability problem precisely. To explore the precision of the bidirected $D_1 \odot D_1$-reachability algorithm, our evaluation addresses the following research questions:

- As an over-approximation method for $D_k \odot D_k$-reachability, what is the precision of bidirected $D_1 \odot D_1$-reachability compared with other approximation algorithms?
- How much over-approximation is there when bidirected $D_1 \odot D_1$-reachability is used to over-approximate bidirected $D_k \odot D_k$-reachability?

6.1 Experimental Setup

Context-sensitive field-sensitive alias analysis. Our experiments adopt an existing context-sensitive field-sensitive alias analysis for Java [Xu et al. 2009; Yan et al. 2011]. We apply it to the standard Dacapo benchmark [Blackburn et al. 2006]. Section 2.4 describes the alias analysis with an example. Recall that the analysis is formulated as a bidirected InterDyck-reachability problem.

Over-approximating $D_k \odot D_k$-reachability algorithm. We compare the precision of our $D_1 \odot D_1$-reachability algorithm with a $D_k \odot D_k$-reachability algorithm based on linear conjunctive language reachability (LCL-reachability) [Zhang and Su 2017]. The $D_k \odot D_k$ languages are a subclass of the

Table 1. Precision and performance results for bidirected $D_1 \odot D_1$-reachability, bidirected LCL-reachability, and bidirected D'-reachability algorithms. Note that D'-reachability under-approximates $D_k \odot D_k$-reachability. The precision column reports the numbers of $D_k \odot D_k$-reachable pairs.

	Precision			Time (s)		
	$D_1 \odot D_1$	LCL	D'	$D_1 \odot D_1$	LCL	D'
antlr	1526931	1548761	1516159	22584.9	0.59	0.74
bloat	1593965	1614797	1585031	19249.7	0.62	0.83
chart	-	4866011	4711077	-	1.17	1.65
eclipse	1528643	1561457	1512137	32858.5	0.87	0.77
fop	3503778	3667004	3427676	44784.1	2.69	1.49
hsqldb	1515720	1536380	1507386	29122.4	0.56	0.69
jython	1568236	1592280	1557182	18782.9	0.73	0.89
luindex	1518813	1537773	1510299	16472.0	0.54	0.71
lusearch	1520970	1541082	1511882	27845.5	0.56	0.75
pmd	1574309	1593967	1566059	30241.3	0.56	0.76
xalan	1515312	1534172	1507196	21930.4	0.54	0.72

LCL languages. The LCL-reachability algorithm is inherently an over-approximation algorithm. We evaluate the LCL-reachability algorithm on the bidirected graphs.

An under-approximating $D_k \odot D_k$-reachability algorithm. To understand the false positives in both the $D_1 \odot D_1$-reachability and the LCL-reachability algorithms, we compare the solution set with that of an under-approximating D'-reachability algorithm. Let the InterDyck language be $D_p \odot D_b$. We can treat it as a Dyck language D' with alphabet $\Sigma_{D'} = \Sigma_{D_p} \cup \Sigma_{D_b}$. The new Dyck language D' is a subset of the $D_p \odot D_b$ language: only well-balanced words are accepted; interleaved words are rejected. Therefore, we can run the D'_k-reachability algorithm to obtain a lower bound on the number of pairs for both $D_k \odot D_k$ over-approximations.

Implementation We use the implementation described in the work of Yan et al. [2011] to derive the alias-analysis graphs from the DaCapo suite [Blackburn et al. 2006]. All graphs are pre-processed by a graph-simplification algorithm for InterDyck-reachability [Li et al. 2020]. We have implemented both the LCL-reachability algorithm and the precise bidirected $D_1 \odot D_1$-reachability algorithm in C++. All executables are compiled by g++-5.4 with -O2 optimization. The experiments were conducted on a machine with a Xeon E5-2650 CPU with 96 GB memory, running Ubuntu 16.04.

6.2 Precision Comparison with the LCL-Reachability Algorithm

To evaluate the precision, we compared the numbers of reported reachable pairs between the precise $D_1 \odot D_1$-reachability algorithm and the LCL-reachability algorithm. Both algorithms over-approximate $D_k \odot D_k$-reachability. Table 1 presents the precision and performance results for both algorithms. For the $D_1 \odot D_1$-reachability algorithm, we are able to obtain precise solutions to the relaxed problems, except for the `chart` benchmark, which consumes too much memory. In Table 1, we observe that the numbers of reachable pairs reported by the bidirected $D_1 \odot D_1$-reachability algorithm is consistently smaller than those reported by LCL-reachability. On average, bidirected $D_1 \odot D_1$-reachability obtains a solution that is about 2.0% smaller than the one obtained via the LCL-reachability algorithm. Table 1 indicates an interesting result: at least for context-sensitive field-sensitive alias analysis, the benefit of having a precise $D_1 \odot D_1$-reachability algorithm can outweigh its disadvantage of being based on a less expressive formalism.

6.3 Precision Comparison with the D'-Reachability Algorithm

To understand the degree of over-approximation introduced by using $D_1 \odot D_1$-reachability to solve a relaxation of $D_k \odot D_k$-reachability, we also compared the precision of the $D_1 \odot D_1$-reachability algorithm with the under-approximating D'-reachability algorithm. Table 1 shows that the gap between the numbers of reachable pairs found in the $D_1 \odot D_1$-reachability algorithm and the numbers found by the under-approximating D'-reachability algorithm is relatively small. On average, the precise $D_1 \odot D_1$-reachability algorithm produced solutions that were 0.89% larger than the ones obtained via D'-reachability. This result indicates that in the simplified $D_k \odot D_k$ graphs, the over-approximation introduced by ignoring distinctions between different parenthesis symbols in each Dyck language is insignificant.

6.4 Discussion

From Table 1, we can also observe that the bidirected $D_1 \odot D_1$-reachability algorithm is very slow compared to the LCL-reachability algorithm. Recall that the time complexity of $D_1 \odot D_1$-reachability algorithm is $O(n^7)$, which poses significant challenges for any practical use. Our work focuses on establishing the **PTIME** result for $D_1 \odot D_1$-reachability. In particular, the $D_1 \odot D_1$-reachability algorithm used in the experiments is directly adopted from the **PTIME** proof. On the other hand, our experimental study suggests that the $D_1 \odot D_1$-reachability algorithm offers a precise over-approximation for $D_k \odot D_k$-reachability, after some pre-processing [Li et al. 2020]. Designing an efficient bidirected $D_1 \odot D_1$-reachability algorithm remains an interesting future direction.

It is perhaps surprising to see that the bidirected $D_1 \odot D_1$-reachability algorithm can achieve better precision than the LCL-reachability algorithm for $D_k \odot D_k$-reachability. Note that LCL-reachability is perhaps the most precise $D_k \odot D_k$-reachability algorithm based on empirical evaluations [Zhang and Su 2017]. In our experiments, the input graphs of all algorithms are pre-processed by a graph-simplification algorithm [Li et al. 2020]. The graph simplification itself can also contribute to the precision improvements. Specifically, graph simplification utilizes the parenthesis-index information in $D_k \odot D_k$ to remove irrelevant edges. For the original bidirected graphs where n is significantly larger, our $O(n^7)$-time $D_1 \odot D_1$-reachability algorithm is too expensive to evaluate.

7 RELATED WORK

The computational complexity of InterDyck-reachability problem has been extensively studied. It has been shown that the InterDyck-reachability problem on directed graphs is undecidable [Ramalingam 2000; Reps 2000]. The undecidability proof by Reps [2000] reduces the Post Correspondence Problem (PCP) to InterDyck-reachability. Given two lists of words $L_1 = \{x_1, \dots, x_k\}.L_2 = \{y_1, \dots, y_k\}$, the PCP problem is to decide whether there exists a list of indices $\{i_1, \dots, i_l\}$ such that the string $x_{i_1}x_{i_2}\dots x_{i_l}$ is the same as $y_{i_1}y_{i_2}\dots y_{i_l}$. The proof use a Dyck language D_2 to encode the strings, with '$(_1$' parenthesis representing '0' and the '$(_2$' parenthesis representing '1'. The other Dyck language keeps track of the indices of the chosen words. A valid index list exists if and only if there exists an InterDyck-path in the graph representing $x_{i_1}x_{i_2}\dots x_{i_l}\overline{y_{i_l}y_{i_{l-1}}}\dots\overline{y_{i_1}}$. However, in the bidirected variants, the same proof technique does not work. In a directed graph, graphs can always force a path to first finish all x_i words and then go through y_i words. The bidirectedness of the graph introduces additional paths. A valid InterDyck-reachable path may represent the following string: $x_{i_1}x_{i_2}\overline{y_{i_2}}x_{i_3}\overline{y_{i_3}y_{i_1}}$. For these InterDyck-paths, we cannot reconstruct solutions of the PCP instance. Thus, the reduction technique does not apply to the bidirected variant, and the complexity for bidirected $D_k \odot D_k$-reachability still remains open.

InterDyck-reachability is the natural formalism when an analysis needs to keep track of two balanced-parenthesis properties. Various practical approaches have been proposed to over-approximate InterDyck-reachability solutions. The traditional approach includes using a context-free language (CFL) to approximate the InterDyck language, and then solving the CFL-reachability problem [Sridharan and Bodík 2006; Sridharan et al. 2005; Yan et al. 2011]. Recently, the conjunctive-language-reachability approach was proposed to approximate InterDyck-reachability [Zhang and

Su 2017]. It uses a conjunctive language to describe an InterDyck language precisely, and provides an approximation algorithm for the reachability problem for linear conjunctive languages. The synchronized push-down systems (SPDSs) of Späth et al. [2019] also provides an approach to solve InterDyck-reachability. An SPDS uses a push-down automaton with two stacks. Each stack can handle the parenthesis matching for one Dyck language. To over-approximate $D_k \odot D_k$-reachability, SPDSs computes the $D_k \odot D_k$-reachability twice, each time ignoring one of the two stacks in the system. SPDSs collect the intersection of the two solution sets as the final result. SPDSs can also recognize incomplete InterDyck-paths, *i.e.*, prefixes of InterDyck-paths.

In contrast to over-approximation algorithms, Madhusudan and Parlato [2011] propose a bounded tree-width approach, which can provide an under-approximate solution for InterDyck-reachability problems. InterDyck-reachability can be considered to be a variant of the state-reachability problem for multi-stack pushdown automata. In the context of concurrent program analysis, a natural semantic restriction for under-approximation is to consider only the states reachable within a bounded number of context switches [Qadeer and Rehof 2005]. The emptiness problem for multi-stack pushdown automata with bounded context-switches can be further reduced to the emptiness problem for graph automata for a special class of graphs (the so-called multiply nested words) with bounded tree-width. The bounded tree-width approach decides the emptiness problem for such graph automata. It provides a technique to explore state reachability for multi-stack pushdown automata with a bounded number of context switches, thus providing an under-approximation for the InterDyck-reachability problem.

8 CONCLUSION

This paper identifies an important class of InterDyck-reachability problems. Even though there are many practical applications of bidirected InterDyck-reachability, the computational complexity had been unaddressed. Our results in this paper start to fill in the picture. To the best of our knowledge, we have identified the first decidable variant of any InterDyck-reachability problem—namely, bidirected $D_1 \odot D_1$-reachability—and showed that it is solvable in **PTIME**. We also showed that bidirected $D_k \odot D_k$-reachability is **NP**-hard.

We experimented with an implementation of the algorithm for precise bidirected $D_1 \odot D_1$-reachability. We applied it to a context- and field-sensitive alias analysis. The experiment compared the precision of (i) relaxing the problem to bidirected $D_1 \odot D_1$-reachability and solving the relaxed problem precisely, with (ii) a different over-approximation algorithm for bidirected $D_k \odot D_k$-reachability. Surprisingly, the $D_1 \odot D_1$-reachability approach produced more precise results than the $D_k \odot D_k$-reachability approach. This experiment may show the benefit of obtaining a precise solution, even though the bidirected $D_1 \odot D_1$-reachability formalism is inherently less expressive than the bidirected $D_k \odot D_k$-reachability formalism.

ACKNOWLEDGEMENTS

We would like to thank the anonymous POPL reviewers for valuable feedback on earlier drafts of this paper, which helped improve its presentation. This work was supported in part by the United States National Science Foundation (NSF) under Grants No. 1917924 and No. 1420866; by a gift from Rajiv and Ritu Batra; by ONR under grants N00014-17-1-2889 and N00014-19-1-2318. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies.

REFERENCES

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 259–269.

S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. 169–190.

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

Swarat Chaudhuri. 2008. Subcubic algorithms for recursive state machines. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 159–169.

Ben-Chung Cheng and Wen-mei W. Hwu. 2000. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2000)*. ACM, 57–69.

Harold N. Gabow, Shachindra N. Maheswari, and Leon J. Osterweil. 1976. On Two Problems in the Generation of Program Test Paths. *IEEE Trans. Software Eng.* 2, 3 (1976), 227–231.

Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. 2015. Scalable and precise taint analysis for Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, 106–117.

Vineet Kahlon. 2009. Boundedness vs. Unboundedness of Lock Chains: Characterizing Decidability of Pairwise CFL-Reachability for Threads Communicating via Locks. In *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009*. IEEE Computer Society, 27–36.

John Kodumal and Alexander Aiken. 2004. The set constraint/CFL reachability connection in practice. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*. ACM, 207–218.

Jakub Kovác. 2014. On the Complexity of Rearrangement Problems under the Breakpoint Distance. *J. Comput. Biol.* 21, 1 (2014), 1–15.

Yuanbo Li, Qirun Zhang, and Thomas W. Reps. 2020. Fast graph simplification for interleaved Dyck-reachability. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 780–793.

P. Madhusudan and Gennaro Parlato. 2011. The tree width of auxiliary storage. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. ACM, 283–294.

Shaz Qadeer and Jakob Rehof. 2005. Context-Bounded Model Checking of Concurrent Software. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3440)*. Springer, 93–107.

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

Thomas W. Reps. 1998. Program analysis via graph reachability. *Inf. Softw. Technol.* 40, 11-12 (1998), 701–726. https://doi.org/10.1016/S0950-5849(98)00093-7

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

Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, flow-, and field-sensitive data-flow analysis using synchronized Pushdown systems. *Proc. ACM Program. Lang.* 3, POPL (2019), 48:1–48:29.

Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*. ACM, 387–400.

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*. ACM, 59–76.

Guoqing (Harry) Xu, Atanas Rountev, and Manu Sridharan. 2009. Scaling CFL-Reachability-Based Points-To Analysis Using Context-Sensitive Must-Not-Alias Analysis. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference,* *Genoa, Italy, July 6-10, 2009. Proceedings*, Sophia Drossopoulou (Ed.). 98–122.

Dacong Yan, Guoqing (Harry) Xu, and Atanas Rountev. 2011. Demand-driven context-sensitive alias analysis for Java. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, Matthew B. Dwyer and Frank Tip (Eds.). ACM, 155–165.

Qirun Zhang and Zhendong Su. 2017. Context-sensitive data-dependence analysis via linear conjunctive language reachability. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 344–358.