Efficient Algorithms for Dynamic Bidirected Dyck-Reachability*

YUANBO LI, Georgia Institute of Technology, USA
KRIS SATYA, Georgia Institute of Technology, USA
QIRUN ZHANG, Georgia Institute of Technology, USA

Dyck-reachability is a fundamental formulation for program analysis, which has been widely used to capture properly-matched-parenthesis program properties such as function calls/returns and field writes/reads. Bidirected Dyck-reachability is a relaxation of Dyck-reachability on *bidirected graphs* where each edge $u \xrightarrow{(_i} v$ labeled by an open parenthesis "$(_i$" is accompanied with an inverse edge $v \xrightarrow{)_i} u$ labeled by the corresponding close parenthesis "$)_i$", and vice versa. In practice, many client analyses such as alias analysis adopt the bidirected Dyck-reachability formulation. Bidirected Dyck-reachability admits an optimal reachability algorithm. Specifically, given a graph with n nodes and m edges, the optimal bidirected Dyck-reachability algorithm computes *all-pairs* reachability information in $O(m)$ time.

This paper focuses on the dynamic version of bidirected Dyck-reachability. In particular, we consider the problem of maintaining all-pairs Dyck-reachability information in bidirected graphs under a sequence of edge insertions and deletions. Dynamic bidirected Dyck-reachability can formulate many program analysis problems in the presence of code changes. Unfortunately, solving dynamic graph reachability problems is challenging. For example, even for maintaining transitive closure, the fastest deterministic dynamic algorithm requires $O(n^2)$ update time to achieve $O(1)$ query time. All-pairs Dyck-reachability is a generalization of transitive closure. Despite extensive research on incremental computation, there is no algorithmic development on dynamic graph algorithms for program analysis with worst-case guarantees.

Our work fills the gap and proposes the first dynamic algorithm for Dyck reachability on bidirected graphs. Our dynamic algorithms can handle each graph update (*i.e.*, edge insertion and deletion) in $O(n \cdot \alpha(n))$ time and support any all-pairs reachability query in $O(1)$ time, where $\alpha(n)$ is the inverse Ackermann function. We have implemented and evaluated our dynamic algorithm on an alias analysis and a context-sensitive data-dependence analysis for Java. We compare our dynamic algorithms against a straightforward approach based on the $O(m)$-time optimal bidirected Dyck-reachability algorithm and a recent incremental Datalog solver. Experimental results show that our algorithm achieves orders of magnitude speedup over both approaches.

CCS Concepts: • **Theory of computation → Dynamic graph algorithms**; • **Software and its engineering → Automated static analysis**.

Additional Key Words and Phrases: Dynamic Graph Algorithms, Dyck-Reachability, Bidirected Graphs, Incremental Analysis

*As noted by Krishna et al. [2023], the worst-case behavior of Procedure 4 is quadratic. We have posted a related note [Zhang 2024] to discuss both the complexity issue and the handling of the cycle case.

Authors' addresses: Yuanbo Li, Georgia Institute of Technology, Atlanta, USA, yuanboli@gatech.edu; Kris Satya, Georgia Institute of Technology, Atlanta, USA, ksatya3@gatech.edu; Qirun Zhang, Georgia Institute of Technology, Atlanta, USA, qrzhang@gatech.edu.

2475-1421/2022/1-ART62
https://doi.org/10.1145/3498724

ACM Reference Format:
Yuanbo Li, Kris Satya, and Qirun Zhang. 2022. Efficient Algorithms for Dynamic Bidirected Dyck-Reachability. *Proc. ACM Program. Lang.* 6, POPL, Article 62 (January 2022), 29 pages. https://doi.org/10.1145/3498724

1 INTRODUCTION

Many program analysis problems can be formulated as graph reachability problems [Reps 1998]. Dyck-reachability is perhaps the most popular graph reachability formulation for program analysis [Zhang et al. 2013]. A Dyck language D_k consists of all strings of properly matched parentheses over k kinds of parentheses. Given an edge-labeled graph, Dyck-reachability computes whether two nodes can be connected by a path with labels that spell out a Dyck word. Many practical analyses use Dyck-reachability to express properly-matched-parenthesis program properties, such as call/return [Reps 2000; Yan et al. 2011], lock/unlock [Kahlon 2009; Ramalingam 2000], file-open/file-close [Späth et al. 2019], and set-field/get-field [Arzt et al. 2014; Yan et al. 2011]. Kodumal and Aiken [2004] observed that "almost all of the applications of context-free language reachability in program analysis are based on Dyck languages."

Bidirected Dyck-reachability is a variant of Dyck-reachability restricted to *bidirected graphs* [Chatterjee et al. 2018; Zhang et al. 2013]. In bidirected graphs, each "$(_i$"-labeled edge $u \xrightarrow{(_i} v$ is accompanied with a corresponding "$)_i$"-labeled edge $v \xrightarrow{)_i} u$ in the inverse direction, and vice versa. Despite being a restricted variant, bidirected Dyck-reachability has been widely used in the literature. It is particularly useful for two reasons: (1) many problems such as pointer analysis are inherently bidirected [Reps 1998]. Indeed, alias analysis for Java has been formulated using bidirected Dyck-reachability [Yan et al. 2011]; (2) bidirected Dyck-reachability can over-approximate the directed counterpart. Because running bidirected Dyck-reachability is relatively cheap, it can serve as a pre-processing step for improving the expensive directed reachability [Li et al. 2020]. Due to its importance, Dyck-reachability has been extensively studied in the literature [Chatterjee et al. 2018; Reps 1998; Zhang et al. 2013]. However, to our best knowledge, all existing developments of Dyck-reachability have been focused on "static" graph reachability, where the entire input graph is known to the reachability algorithm.

This paper focuses on the dynamic problem of bidirected Dyck-reachability, a much less studied algorithmic topic in program analysis. In general, a dynamic graph algorithm allows changes (*i.e.* edge insertions and deletions) to the input graphs [King and Sagert 1999; Roditty 2003]. It typically allows three operations: (1) *pre-processing*, which is called for the initial graph; (2) *update*, which is called for every input update; and (3) *query*, which is used to answer reachability queries. Efficient dynamic algorithms can establish a solid algorithmic foundation for incremental program analysis. Specifically, with efficient dynamic reachability algorithms, the underlying client analysis can promptly respond to code changes (*i.e.*, code insertions and deletions). For example, modern IDEs run incremental analyses such as type checkers, code smell detectors, and dead code analyses to provide instant feedback to developers [Pacak et al. 2020; Szabó et al. 2021].

Designing asymptotically fast dynamic reachability algorithms is challenging. Recall that the static linear-time algorithm for bidirected Dyck-reachability is optimal [Chatterjee et al. 2018]. In the dynamic setting for an edge-labeled graph with m edges and n nodes, a straightforward way is to run the static algorithm for each update, which leads to a naïve dynamic algorithm for bidirected Dyck-reachability in $O(m)$ pre-processing time, $O(m)$ update time and $O(1)$ query time. In terms of the complexity, an $O(m)$ time algorithm is arguably reasonable. However, it is apparently not efficient because it needs to re-compute the reachability information from scratch for each update.

This paper proposes efficient dynamic algorithms for bidirected Dyck-reachability. In particular, our algorithms can pre-process an initial graph in $O(m)$ time, handle each graph update (*i.e.*, edge

insertion and deletion) in *worst case* $O(n \cdot \alpha(n))$ time, and answer any all-pairs reachability query in $O(1)$ time where α denotes the inverse Ackermann function. Our key insight is to maintain the equivalence property of Dyck-reachability on bidirected graphs [Chatterjee et al. 2018; Zhang et al. 2013] dynamically. In particular, we augment the key data structure used in the $O(m)$-time optimal bidirected reachability algorithm [Chatterjee et al. 2018] with weights. Our algorithm can efficiently update weights and maintain the reachability information in $O(n \cdot \alpha(n))$ time for any edge insertion and deletion.

Unlike existing incremental computation work [Arzt and Bodden 2014; Sreedhar et al. 1997; Szabó et al. 2021, 2016], our approach can guarantee that our dynamic algorithms do not involve any redundant computation. With the weights introduced in our algorithm, we can formally prove the property. Our work is closely related to incremental Datalog evaluation [Ryzhyk and Budiu 2019; Szabó et al. 2021] because Dyck-reachability can be directly expressed using Datalog rules. However, Datalog frameworks are general, and do not leverage the equivalence property exploited in optimal Dyck-reachability [Chatterjee et al. 2018] for bidirected graphs. Improving general dynamic graph reachability is quite challenging. Even for maintaining dynamic transitive closure, the best deterministic algorithm requires an $O(n^2)$ update time to achieve the $O(1)$ query time [Roditty 2003]. Indeed, on general graphs, there is a conditional lower bound result for dynamic transitive closure [Henzinger et al. 2015]. Specifically, unless the Online Boolean Matrix-Vector Multiplication (OMv) conjecture [Henzinger et al. 2015] is false, there is no combinatorial algorithm that can achieve polynomial pre-processing time, $O(m^{1/2-\delta})$ update time, and $O(m^{1-\delta})$ query time *simultaneously*, for any small constant $\delta > 0$. Dynamic graph reachability for program analysis is a less explored area. In particular, there is no asymptotically faster algorithm for bidirected Dyck-reachability than the straightforward solution with an $O(m)$ update time.

We have implemented the dynamic algorithms for bidirected Dyck-reachability and applied it to two practical client analyses, an alias analysis for Java formulated by bidirected Dyck-reachability [Yan et al. 2011] and a context-sensitive data-dependence analysis formulated by Dyck-reachability [Tang et al. 2015]. In addition, we compared our dynamic algorithms with an $O(n \cdot \alpha(n))$ update time against the straightforward algorithm with an $O(m)$ update time [Chatterjee et al. 2018] and an incremental Datalog solver [Ryzhyk and Budiu 2019]. The empirical results for our proposed dynamic bidirected Dyck-reachability algorithm are encouraging.

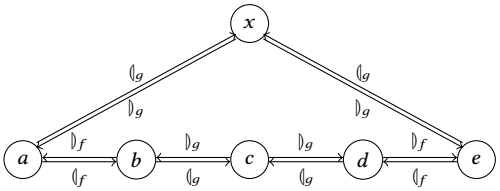
- Compared to the straightforward approach that runs the optimal $O(m)$-time bidirected Dyck-reachability algorithm for each update, our dynamic algorithms have achieved a 534x speedup in the alias analysis and a 331x speedup in the context-sensitive data-dependence analysis.
- Compared to a recent incremental Datalog solver that only recomputes the necessary changes upon each update, our dynamic algorithms have achieved a 496x speedup in the alias analysis and a 20x times speedup in the context-sensitive data-dependence analysis.
- In practice, we have observed that our algorithm scales linearly with the graph size.

We make the following main contributions in this paper:

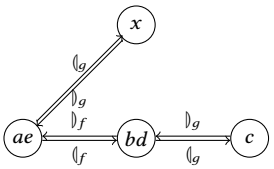
- We present the first algorithmic study on dynamic bidirected Dyck-reachability for program analysis. Specifically, we propose an efficient dynamic reachability algorithm with an $O(m)$ pre-processing time, an $O(n \cdot \alpha(n))$ update time, and an $O(1)$ query time.
- We present a formal analysis of our dynamic algorithms. Our analysis shows that our algorithm is asymptotically faster than the straightforward approach that runs the optimal $O(m)$-time Dyck-reachability algorithm for each update. Moreover, it shows that our algorithm does not incur any redundant computation.
- We conduct extensive evaluations on two client analyses with insertions only, deletions only, and mixed graph update sequences. The empirical results show that our algorithm

```
1  class A {
2      A f;
3      A g;
4  };
5  int main(){
6      A a,b,c,d,e,x,y;
7      ......
8      b.f = a;
9      c.g = b;
10     d = c.g;
11     e = d.f
12     ......
13     x.g = a;
14     e = x.g;
15 }
```

(a) Input program.



(b) Alias analysis graph.



(c) Merged graph for Figure 1b.

Fig. 1. Context-insensitive alias analysis for Java via bidirected Dyck-reachability.

can achieve orders-of-magnitude speedup over the $O(m)$ time approach and an incremental Datalog solver. Moreover, our algorithm scales linearly with the graph size.

The rest of the paper is structured as follows. Section 2 motivates dynamic Dyck-reachability on bidirected graphs. Section 3 presents preliminaries. Section 4 discusses our dynamic algorithms for bidirected Dyck-reachability. Section 5 describes the evaluation setup and results. Finally, Section 6 surveys related work, and Section 7 concludes.

2 MOTIVATING EXAMPLE

We motivate dynamic bidirected Dyck-reachability using a practical context-insensitive alias analysis for Java [Yan et al. 2011].

2.1 Context-Insensitive Alias Analysis

Alias analysis checks whether two variables can point to the same memory object. We consider a context-insensitive alias analysis for a Java-like program in Figure 1a. Context-insensitive alias analysis can be formulated as bidirected Dyck-reachability [Yan et al. 2011; Zhang et al. 2013]. Figure 1b gives the corresponding graph representation G for alias analysis. The labeled graph is bidirected, *i.e.*, for each open-parenthesis edge $u \xrightarrow{(_l} v$, there exists an inverse edge $v \xrightarrow{)_l} u$ with the corresponding close-parenthesis label.

In graph G, nodes represent variables in the program; edges represent variable reads and writes. The alias analysis utilizes Dyck-reachability to capture the matching of reads and writes of the same field of an object. Specifically, a "$(_f$"-labeled edge represents a write to the field f and a "$)_f$"-labeled edge represents a read of the field f. For example, the statement `b.f = a` in Figure 1a is modeled as an edge $a \xrightarrow{(_f} b$ in Figure 1b. Similarly, the statement `d = c.g` is modeled as an edge $c \xrightarrow{)_g} d$. Note that in the bidirected graph G, there exist two inverse edges $b \xrightarrow{)_f} a$ and $d \xrightarrow{(_g} c$ for the above edges. The bidirectedness achieves an over-approximation, which could lead to spurious aliasing, as discussed by Xu et al. [2009, §4]. However, experimental results demonstrate that, in

practice, the overall performance is better than the algorithms proposed by Sridharan et al. [2005] and Sridharan and Bodík [2006].

If two variables may point to the same object at runtime, there exists a Dyck-path (a path with its labels forming a Dyck-word) between the corresponding nodes. For instance, variables a and e may point to the same object in Figure 1a. In Figure 1b, there exists a Dyck-path $a \xrightarrow{(_g} x \xrightarrow{)_g} e$ connecting nodes a and e. The edge labels form a Dyck-word "$(_g)_g$".

2.2 Bidirected Dyck-Reachability

Dyck-reachability can be solved by the general context-free language (CFL) reachability algorithms [Reps 1998]. It is well-known that CFL-reachability exhibits a (sub)cubic time complexity [Chaudhuri 2008]. For bidirected Dyck-reachability, Zhang et al. [2013] observed an interesting equivalence property on bidirected graphs, which leads to an average $O(m \log m)$-time reachability algorithm. Chatterjee et al. [2018] further proposed an $O(m)$-time algorithm and proved that the algorithm is optimal for bidirected Dyck-reachability.

In fast bidirected Dyck-reachability algorithms [Chatterjee et al. 2018; Zhang et al. 2013], Dyck-reachable node pairs (u, v) form a binary relation $\text{Dyck}(u, v)$. On bidirected graphs, the Dyck relation is an equivalence relation. Therefore, the main idea of the fast bidirected Dyck-reachability algorithms [Chatterjee et al. 2018; Zhang et al. 2013] is to merge nodes u and v for all $(u, v) \in \text{Dyck}$. The algorithms eventually generate a merged graph G_m where each node in G_m represents a set of Dyck-reachable nodes in G. Figure 1c presents the merged graph G_m for the graph G in Figure 1b. In Figure 1c, nodes a, e and nodes b, d are merged together, indicating that the nodes a, e and nodes b, d are Dyck-reachable from each other in Figure 1b.

2.3 Dynamic Bidirected Dyck-Reachability

In software development, it is useful that static analysis can respond to code changes. Moreover, an efficient analysis algorithm can update the analysis results incrementally without re-computing everything from scratch for each code update. Therefore, this paper considers the dynamic version of bidirected Dyck-reachability and proposes an efficient algorithm with an $O(m)$ pre-processing time, an $O(n \cdot \alpha(n))$ query time, and an $O(1)$ query time.

We motivate our dynamic bidirected Dyck-reachability using the concrete example in Figure 1a. Given an initial graph G in Figure 1b, we first call a modified version of the optimal Dyck-reachability algorithm to obtain the merged graph G_m in Figure 1c. Suppose that we have a code change of "removing the statement `x.g = a` on line 13". It corresponds to deleting the edge $a \xrightarrow{(_g} x$ (and its inverse edge $x \xrightarrow{)_g} a$) in G (Figure 1b). Therefore, our algorithm needs to update the merged G_m.

Naïve algorithm. Upon an edge deletion, the naïve approach to update the bidirected Dyck-reachability result is to apply the optimal bidirected Dyck-reachability algorithm [Chatterjee et al. 2018] on the new graph in $O(m)$ time. Specifically, it merges nodes a, e and nodes b, d again. To merge these nodes together, it needs to traverse four edges $a \xrightarrow{(_f} b$, $b \xrightarrow{(_g} c$, $d \xrightarrow{(_g} c$ and $e \xrightarrow{(_f} d$ to find the Dyck-paths between them. Clearly, these nodes have already been merged based on the original graph in Figure 1b. The edge traversals and node merging are redundant.

Our algorithm. Our dynamic algorithm is much more efficient. Instead of working on the input graph G from scratch, our algorithm works directly on the merged graph G_m. Recall that every node n' in G_m represents a set of Dyck-reachable nodes in G. Our algorithm only splits nodes n' in G_m whose representing nodes contain the node pairs in the graph G which become unreachable after the edge deletion.

The main challenge is to decide if the nodes in G_m should be split upon an edge deletion. Our key insight is to maintain a `weight` attribute for each edge in the merged graph G_m. Specifically, if there are k edges in the input graph G merged into the same edge e in the merged graph G_m, the `weight` associated with edge e is k. In Figure 1c, the `weight` of edge $\{ae\} \xrightarrow{(_f} \{bd\}$ is 2 because it is the result of merging the two edges $a \xrightarrow{(_f} b$ and $e \xrightarrow{(_f} d$. Similarily, the `weight` of edge $\{ae\} \xrightarrow{(_g} \{x\}$ is also 2. These edges indicate that there is more than one Dyck-path connecting the two nodes a, e in the original graph. When the edge $a \xrightarrow{(_g} x$ is deleted, the `weight` of $\{ae\} \xrightarrow{(_g} \{x\}$ becomes 1. Our algorithm checks whether the node $\{ae\}$ needs to be split. The `weight` associated with the outgoing edge $\{ae\} \xrightarrow{(_f} \{bd\}$ is greater than 1, which means that nodes a, e can still be merged by another Dyck-path without the deleted edge $a \xrightarrow{(_g} x$. Therefore, our dynamic algorithm terminates and does not split node $\{ae\}$ in G_m.

Compared to the four edge traversals in the naïve approach, our dynamic algorithm only needs to traverse one edge $\{ae\} \xrightarrow{(_f} \{bd\}$ in G_m. Moreover, our algorithm does not need to re-compute the all-pairs reachability information.

3 PRELIMINARIES

This section introduces bidirected Dyck-reachability and its dynamic variant.

3.1 Bidirected Dyck-Reachability

A Dyck language generates a set of strings of properly matched parentheses. It can be used to model pairs of matching actions in program analysis such as function calls/returns, pointer references/dereferences, and field writes/reads. A Dyck language D with k different kinds of parentheses can be defined by the following context-free grammar:

$$S ::= SS \mid (_i\ S\ )_i \mid \epsilon, \quad \text{for } i = 1, \ldots, k.$$

This paper considers the bidirected variant of Dyck-reachability. We first define bidirected graphs:

Definition 3.1 (Bidirected graphs). Consider a directed graph $G = (V, E)$ with each edge labeled by a symbol from the alphabet Σ of a Dyck language. The graph is bidirected iff

- for each open-parenthesis edge $u \xrightarrow{(_i} v$, there exists an inverse edge $v \xrightarrow{)_i} u$ with the corresponding close-parenthesis label, and
- for each close-parenthesis edge $u \xrightarrow{)_i} v$, there exists an inverse edge $v \xrightarrow{(_i} u$ with the corresponding open-parenthesis label.

Consider an edge-labeled graph G with each edge labeled by a symbol from the alphabet Σ of a Dyck language D. Each path p in G can realize a word $R(p)$ by concatenating all edge symbols in that path. We say a path is a *Dyck-path* if the corresponding realized word $R(p)$ is a Dyck-word. Moreover, a node pair (u, v) is *Dyck-reachable* iff there exists a Dyck-path from node u to node v in G. We also say node v is Dyck-reachable from u.

Definition 3.2 (Bidirected Dyck-reachability). Given a bidirected graph G defined in Definition 3.1, compute all Dyck-reachable node pairs in G.

Example 3.3. Consider the bidirected graph in Figure 1b. Node e is Dyck-reachable from node a due to the path $a \xrightarrow{(_g} x \xrightarrow{)_g} e$. Based on Definition 3.1, there also exists a corresponding Dyck-path $e \xrightarrow{(_g} x \xrightarrow{)_g} a$. Therefore, node a is Dyck-reachable from e as well. Let $\text{Dyck}(u, v)$ denote a binary

relation that represents Dyck-reachable node pairs. On bidirected graphs, the Dyck(u, v) relation is reflexive, symmetric, and transitive. Therefore, it is an equivalence relation [Zhang et al. 2013].

3.2 Dynamic Bidirected Dyck-Reachability

Dynamic bidirected Dyck-reachability extends Definition 3.2 such that the graph G is modified under a sequence of edge insertions and deletions. Dynamic graph reachability can express incremental program analysis problems in the presence of code changes.

Definition 3.4 (Dynamic bidirected Dyck-reachability). Dynamic bidirected Dyck-reachability maintains all-pairs Dyck-reachability results in a bidirected graph G under a sequence of graph updates including the following opeartions.

- Edge insertion: insert an edge e and its corresponding inverse edge e' to G.
- Edge deletion: delete an edge e and its corresponding inverse edge e' from G.

After a graph update, we can query any Dyck-reachable pair based on the updated graph G.

Different from static graph algorithms, the complexity description of dynamic graph algorithms contains pre-processing time, update time, and query time.

Definition 3.5 (Complexity of dynamic graph reachability [Abboud and Williams 2014; Henzinger et al. 2015]). The complexity description of dynamic graph reachability algorithms consists of the following three components:

- The pre-processing time complexity p: The time complexity for computing the reachability information on the initial graph. An initial graph can be empty.
- The update time complexity u: The time complexity for maintaining all-pairs reachability after each graph update.
- The query time complexity q: The time complexity for answering a reachability query.

Therefore, the complexity of a dynamic graph reachability algorithm can be described as a tuple (p, u, q). Note that in this work, we focus on dynamic bidirected Dyck-reachability algorithms with constant query time $q = O(1)$. For example, the naïve approach that re-runs the optimal bidirected Dyck-reachability algorithm [Chatterjee et al. 2018] for each graph update exhibits a complexity tuple of $(O(m), O(m), O(1))$.

4 DYNAMIC ALGORITHM FOR BIDIRECTED DYCK-REACHABILITY

This section presents our dynamic algorithm for bidirected Dyck-reachability. Our dynamic algorithm can achieve $O(m)$ pre-processing time, $O(n \cdot \alpha(n))$ update time and $O(1)$ query time. Section 4.1 provides an overview of our dynamic algorithms and briefly discusses the key insights. Section 4.2 presents the dynamic algorithm for edge insertions. Section 4.3 describes the dynamic algorithm for edge deletions. Finally, Section 4.4 analyzes the correctness and the complexity of the proposed algorithms.

4.1 Overview

The dynamic algorithms take as input an initial bidirected graph and a sequence of edge insertions and deletions that update the (possibly empty) initial graph. The algorithms maintain a set of Dyck-reachable pairs to answer any reachability query in constant time after the graph updates. The algorithms consist of three major components: pre-processing, update, and query. The pre-processing step generates the initial Dyck-reachability results based on the initial graph. The update component reads an edge insertion or deletion from the given sequence and updates the maintained Dyck-reachability result accordingly. Finally, the query component answers reachability queries between any nodes.

Algorithm 1: Main Algorithm for Dynamic Bidirected Dyck-Reachability.

Input : Initial graph $G = (V, E)$ and an update sequence S
Output: Maintained merged graph G_m

```
1 Initialize the results to get the merged graph G_m from the initial graph G
2 for operation op(e) in the sequence S do
3     if op is an insert oeperation then
4         Insert e into the graph G
5         G_m ← DynamicInsertion (e, G, G_m)
6     if op is a delete operation then
7         Delete e from the graph G
8         G_m ← DynamicDeletion (e, G, G_m)
9 return G_m
```

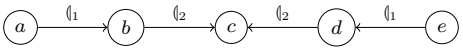


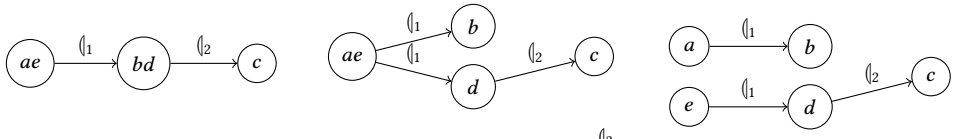
Fig. 2. An input graph before deleting edge $b \xrightarrow{(_2} c$. The close-parenthesis edges are omitted for simplicity.

Our dynamic algorithms work directly on the *merged* graph G_m. In the merged graph G_m, each node represents a set of nodes that are reachable to each other in the input graph G [Chatterjee et al. 2018; Zhang et al. 2013]. The key challenge arises when deleting an edge. Upon an edge deletion, the algorithm needs to split nodes that are no longer reachable in the merged graph G_m. In particular, we need to find a termination condition for the recursive node splitting. We provide an overview of our dynamic algorithms and present our idea for addressing the key challenge.

Pre-processing. The pre-processing step takes input graphs to generate the initial Dyck-reachability results. The algorithm for this step is similar to a static algorithm for bidirected Dyck-reachability. We use a *modified* version of the optimal bidirected Dyck-reachability algorithm [Chatterjee et al. 2018] for pre-processing. We denote this modified optimal Dyck-reachability as procedure `Opt-Dyck'`. The modification for procedure `Opt-Dyck'` is to maintain an extra attribute `weight` for edges in the merged graph G_m. In the merged graph G_m, each edge e represents a set of edges E_e in the input graph due to the merged nodes. We define the weight of e as $\texttt{weight}(e) = |E_e|$. The details of maintaining the `weight` are discussed in Section 4.2.

Dynamic update. To update the reachability results efficiently, our dynamic update algorithms work directly on the merged graph G_m. We use the notation `resp_node` to represent the mapping from nodes in the input graph G to their representative merged nodes in G_m. To maintain the bidirected Dyck-reachability results, the dynamic update algorithms maintain the merged graph G_m. The dynamic algorithm handles two types of graph updates: edge insertion and edge deletion. For an edge insertion, the intuition is to insert the corresponding edge into the merged graph G_m and then treat the newly merged graph G_m as an input graph for the bidirected Dyck-reachability algorithm. Section 4.2 discusses the details for handling edge insertions.

The key challenge is to handle dynamic updates for edge deletions. For each edge deletion, our dynamic deletion algorithm splits the nodes in the graph that just became unreachable. The node splitting is recursive on node predecessors.



(a) The merged graph. (b) After removing edge $b \xrightarrow{(_2} c$, split node b, d. (c) Recursively split node a, e.

Fig. 3. Node splitting on merged graph.

Example 4.1 (Recursive node splitting). Consider an update to delete the edge $b \xrightarrow{(_2} c$ in the graph in Figure 2.[1] Before the edge deletion, node pairs (b, d) and (a, e) are Dyck-reachable. Thus, in the corresponding merged graph in Figure 3a, the nodes b, d and nodes a, e are merged together, respectively. After deleting the edge $b \xrightarrow{(_2} c$ in the graph, there does not exist any Dyck-path. Therefore, the node pairs (b, d) and (a, e) are no longer Dyck-reachable. As shown in Figure 3, the dynamic algorithm first splits node $\{bd\}$ and recursively splits its predecessor node $\{ae\}$.

However, two nodes can still be Dyck-reachable after the edge deletion. In this case, if the dynamic algorithm splits them into two representative nodes, it has to merge them back later. We consider this to be a *redundant computation*. To avoid any redundant computations, we should not split the nodes in the graph with unchanged reachability results. As shown in Example 9, the algorithm recursively splits the predecessors after splitting the current node. The key challenge is to decide when to stop the recursive splitting in the dynamic algorithm for edge deletion. Our insight for addressing the challenge is to use an attribute *weight* of the edges in G_m to help determine when the splitting should terminate. The first observation is that, after an edge deletion, if a predecessor node does not have an outgoing edge e with $\texttt{weight}(e) \geq 2$, then there exists no other Dyck-path to merge the nodes in the predecessor together. Therefore, the dynamic algorithm should recursively split this predecessor node (Section 4.3).

Consider Example 9 again. After splitting node $\{bd\}$, node $\{ae\}$ has outgoing edges $\{ae\} \xrightarrow{(_1} b$ and $\{ae\} \xrightarrow{(_1} d$. The `weight` of edge $\{ae\} \xrightarrow{(_1} b$ is 1 because only one $a \xrightarrow{(_1} b$ edge in the original graph is merged onto it. Similarly, the `weight` of the other edge $\{ae\} \xrightarrow{(_1} d$ is also 1. Because there is no outgoing edge with weight greater than 1, the algorithm continues the node splitting. Thus, the node $\{ae\}$ needs to be split recursively.

Query. Given the merged graph G_m and the mapping `resp_node`, answering reachability queries is straightforward. For a reachability query of two nodes u and v, we return whether $\texttt{resp\_node}(u)$ and $\texttt{resp\_node}(v)$ are the same node in merged graph G_m.

Main algorithm. Algorithm 1 presents the main algorithm for dynamic bidirected Dyck-reachability. The algorithm takes as input an initial graph, an update sequence for the graph, and maintains the merged graph for each edge update. Line 1 describes the pre-processing step, which calls the optimal bidirected Dyck-reachability algorithm [Chatterjee et al. 2018]. Lines 2-8 perform the dynamic updates based on the update sequence S. The update is either an insertion or a deletion with respect to an edge e, representing the changes to be made in the input graph G. The algorithm calls the corresponding dynamic algorithm for edge insertion and edge deletion accordingly.

Algorithm 2: Dynamic Insertion for Bidirected Dyck-Reachability.

Input : Updated edge-labeled bidirected graph $G = (V, E)$,
The inserted edge $e = u \xrightarrow{l} v$ in G,
The merged graph $G_m = (V', E')$
Output: Updated G_m

```
1 if u is a new node in G then
2     V' ← V' ∪ {u}
3     resp_node (u) = u
4 if v is a new node in G then
5     V' ← V' ∪ {v}
6     resp_node (v) = v
7 add edge e' = resp_node (u) -l-> resp_node (v) in E'
8 Opt-Dyck' (G, G_m)
9 return G_m
```

4.2 Dynamic Insertion Algorithm

The dynamic insertion algorithm maintains the updated merged graph G_m after inserting the edge e. In particular, it needs to insert a corresponding edge in the merged graph G_m and invokes the optimal bidirected Dyck-reachability algorithm variant `Opt-Dyck'` on the merged graph G_m directly.

Optimal Dyck-reachability algorithm variant. The difference between the variant `Opt-Dyck'` procedure and the original optimal bidirected Dyck-reachability algorithm [Chatterjee et al. 2018] is that `Opt-Dyck'` also needs to maintain an extra attribute `weight` for edges in the merged graph G_m. The `weight` for each edge in the graph should be initialized to be 1 before the Dyck-reachability algorithm starts. Procedure `Opt-Dyck'` needs to update attribute `weight` whenever node merging happens. When two nodes x, y are merged together, the incoming and outgoing edges of node x are moved and become the incoming and outgoing edges of node y. The `weights` of the edges are moved accordingly as well. In addition, for some incoming edge $e_1 = u \rightarrow x$ or outgoing edge $e_2 = x \rightarrow v$ of node x, if the corresponding edges $e_3 = u \rightarrow y, e_4 = y \rightarrow v$ already exist, the updated weights $\texttt{weight}(e_3)$, $\texttt{weight}(e_4)$ after the merging should be $\texttt{weight}(e_1)$+ $\texttt{weight}(e_3)$ and $\texttt{weight}(e_2)$+ $\texttt{weight}(e_4)$ respectively. This modification does not affect the complexity of the optimal algorithm.

Algorithm 2 describes the dynamic insertion algorithm. The algorithm takes an input graph G, the inserted edge $u \xrightarrow{l} v$ and the merged graph G_m as inputs and updates the merged graph G_m.

- Lines 1-6 check whether the inserted edge e involves any new nodes that do not exist in the current graph G. If there are such new nodes, the algorithm inserts the corresponding nodes into the merged graph G_m on lines 2 and 5. Then the algorithm updates the mapping `resp_node` on lines 3 and 6.
- Lines 7-8 update the merged graph G_m. In particular, line 7 adds the corresponding edge e' for the inserted edge in the merged graph G_m. The inserted edge in the merged graph can possibly cause more merging in the graph. Line 8 uses the optimal bidirected Dyck-reachability algorithm [Chatterjee et al. 2018] variant `Opt-Dyck'` to perform the node merging in G_m. `Opt-Dyck'` updates merged graph G_m, which is the output of the edge insertion.

[1]For brevity, we omit close-parenthesis edges in the bidirected graphs used in Section 4.

Algorithm 3: Dynamic Deletion for Bidirected Dyck-Reachability.

Input : Updated edge-labeled bidirected graph $G = (V, E)$,
The deleted edge $e = u \xrightarrow{l} v$ in G,
The merged graph $G_m = (V', E')$
Output: Updated G_m

```
1 Denote e = u -l-> v
2 u' ← resp_node (u)
3 v' ← resp_node (v)
4 weight (u' -l-> v') ← weight (u' -l-> v') − 1
5 if weight (u' -l-> v') == 0 then
6     remove edge (u' -l-> v') in G_m
7 split_further (u, G, G_m)
8 split_further (v, G, G_m)
9 return G_m
```

4.3 Dynamic Deletion Algorithm

The dynamic algorithm for edge deletion updates the merged graph G_m by node splitting because some of the reachable node pairs may become unreachable after the edge deletion. As discussed in Section 4.1, these nodes need further split.

Challenge. As discussed in Example 9, nodes in the merged graph G_m need to split if their successor node got split. A naïve approach is to split all nodes in the merged graph, equivalent to recomputing everything from scratch. The key challenge of dynamic updates for edge deletion is to determine the termination condition of the recursive node splitting. The insight of our algorithm is to use the `weight` of the edges to decide when to terminate the node splitting to avoid redundant computation. The `weight` of an edge e' in the merged graph G_m represents the number of edges in the input graph G that are merged into e' in G_m. For an edge e', we denote $S_{e'} = \{e_i = u_i \xrightarrow{l} v_i\}_{i=1,\ldots,k}$ as the set of edges in G that are merged into e' in G_m. If the weight satisfies the condition of $\texttt{weight}(e') = |S_{e'}| > 1$, there exists at least one Dyck-path that merges the set of nodes $\{u_i\}_{i=1,\ldots,k}$ together in G_m. We describe this condition on edges as a weight condition and further develop a splitting condition to decide when to split nodes.

Definition 4.2 (Weight condition). For an edge e' in G_m and the set of edges $S_{e'} = \{e_i = u_i \xrightarrow{l} v_i\}_{i=1,\ldots,k}$ merged into e', if $\texttt{weight}(e') > 1$, we say the edge e' satisfies the weight condition. If e' satisfies the weight condition, after running the dynamic Dyck-reachability algorithm, the nodes $\{u_i\}_{i=1\ldots k}$ should be in the same node in the maintained G_m.

Definition 4.3 (Split condition and splitting restriction relation). For a node u' in the merged graph G_m, each of its outgoing edges $\{e'_i\}_i$ satisfying the weight condition generates a set of nodes $\{u_{ij}\}_j$ described in Definition 4.2. The node set $\{u_{ij}\}_j$ should not get split by the algorithm. We define these sets as *splitting restriction sets*. Two nodes v_1, v_2 in the same splitting restriction sets satisfy the *splitting restriction relation*. After an edge deletion, the node u' should be split into two nodes u'_1 and u'_2 if satisfying the following *split condition*: (1) Nodes u'_1, u'_2 are the merging nodes for two disjoint non-empty node sets V_1, V_2 in G, respectively; and (2) there exists no node pair (v_1, v_2) with $v_1 \in V_1, v_2 \in V_2$ such that v_1, v_2 satisfy the splitting restriction relation.

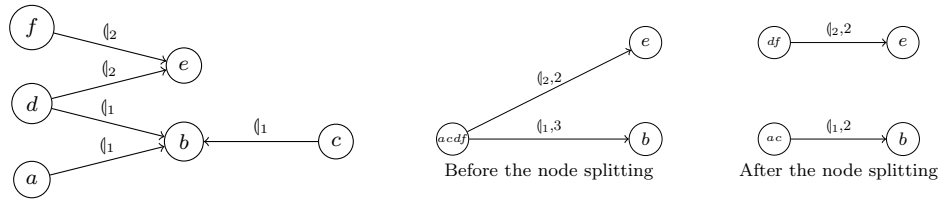
Example 4.4 (Node splitting using weight condition). Consider the input graph G in Figure 4a and an update of deleting the edge $d \xrightarrow{(_1} b$ in G. The first graph in Figure 4b presents the merged

Procedure 4: `split_further`(u, G, G_m)

Input : u The node in G' to split on
G: the input graph after the edge deletion
G_m: the merged graph
Output: G_m: the merged graph after the node splitting on u

```
1  orig_respnode ← resp_node (u)
2  nodes ← {v | resp_node(v) = orig_respnode}
3  groups ← make-disjoint-set(nodes)
4  for outgoing edge e' from resp_node (u) with weight (e') = k > 1 do
5      let {e_i = s_i -l-> t_i}_{i=1...k} be the set of edges merged into e'
6      groups.join({s_i}_{i=1...k})
7  if groups only has one partition then
8      return G_m
9  split resp_node (u) into groups.partitions
10 update the resp_node mapping accordingly
11 V' = V' ∪ groups.partitions \{resp_node(u)}
12 for e' = w -l-> orig_respnode in E' do
13     weight (resp_node(w) -l-> orig_respnode) = 0
14     remove edge resp_node(w) -l-> orig_respnode in G_m
15     let {e_i = s_i -l-> t_i}_{i=1...k} be the set of edges merged into e'
16     for node v in partitions P of groups with the multiset P ∩ {s_i}_i ≠ ∅ do
17         add resp_node (w) -l-> v in E'
18         weight (resp_node (w) -l-> v) = |P ∩ {s_i}_i|
19 for e' = orig_respnode -l-> w in E' do
20     weight (orig_respnode -l-> resp_node(w)) = 0
21     remove edge orig_respnode -l-> resp_node(w) in G_m
22     let {e_i = s_i -l-> t_i}_{i=1...k} be the set of edges merged into e'
23     for node v in partitions P of groups with the multiset P ∩ {s_i}_i ≠ ∅ do
24         add v -l-> resp_node (w) in E'
25         weight (v -l-> resp_node (w)) = |P ∩ {s_i}_i|)
26 for node w in predecessors of u in G' before the splitting do
27     split_further (w, G, G_m)
28 return G_m
```

graph G_m for Figure 4a before the deletion of edge $d \xrightarrow{\mathbb{1}} b$. After the edge deletion, in the merged graph G_m, node $\{acdf\}$ still has two outgoing edges $\{acdf\} \xrightarrow{\mathbb{1}} b$ and $\{acdf\} \xrightarrow{\mathbb{2}} e$ with `weight` 2. As described in Definition 4.3, both edges satisfy the weight condition and each edge generates a splitting restriction set. The splitting restriction set of edge $\{acdf\} \xrightarrow{\mathbb{1}} b$ is $\{a, c\}$ and the splitting restriction set of $\{acdf\} \xrightarrow{\mathbb{2}} e$ is $\{d, f\}$. By splitting the node $\{acdf\}$ into two nodes $\{ac\}$ and $\{df\}$, the split condition in Definition 4.3 is satisfied because $\{a, c\}$ and $\{d, f\}$ are non-empty disjoint sets, and there exists no node pair $(v_1, v_2) \in \{a, c\} \times \{d, f\}$ with v_1, v_2 in the same splitting restriction set. Thus the algorithm splits the node $\{acdf\}$ into $\{ac\}$ and $\{df\}$ as shown in Figure 4b.



(a) The input graph before the edge deletion.

(b) The merged graph for node splitting.

Fig. 4. The illustrative example for node splitting condition.

Algorithm 3 presents the details of the dynamic algorithm for edge deletions. The algorithm takes the original graph G, the deleted edge $u \xrightarrow{l} v$ and the merged graph G_m as input, and then generates the updated merged graph G_m. The algorithm consists of two major components: the edge deletion part and the splitting part. We describe each step in detail.

The splitting algorithm. Procedure 4 presents the splitting algorithm, which takes as input a specified node u to split in G_m, the input graph G, and the merged graph G_m. It splits the node u and performs the subsequent splitting in G_m. If the splitting condition is satisfied, Procedure 4 splits the nodes and updates the edges and their `weight` accordingly. The splitting procedure calls itself on the predecessor nodes recursively. Specifically,

- Lines 1-11 split the node u in the merged graph. The algorithm first decides the result nodes of the splitting in lines 1-6. The variable `groups` represents a disjoint-set of nodes in the merged node. The nodes in the same partition in `groups` should satisfy the splitting restriction relation and remain merged together during the node splitting. Then in lines 4-6 the algorithm iterates over the edges e' in G_m satisfying the weight condition. Each edge e' satisfying the weight condition generates a splitting restriction set. We keep the splitting restriction set in the variable `group`, line 6 joins the set of nodes in the same splitting restriction set in `groups`. The splitting condition is not satisfied if all the nodes in the `groups` are in the same splitting restriction set. In this case, the `split_further` function stops at line 8. Otherwise the algorithm splits the node u. Lines 9-11 split the node according to the splitting restriction set in `groups` and update the representative node mapping `resp_node`.
- Lines 12-25 update the edges in the merged graph after the node splitting. Specifically, lines 12-18 update the incoming edges to the new node and modify their weights accordingly in the graph. Lines 19-25 update the outgoing edges and their weights in the merged graph.
- Lines 26-27 recursively split the predecessor nodes of u.

The deletion algorithm. Algorithm 3 presents the dynamic deletion algorithm. The deletion algorithm deletes an edge according to the input sequence and calls the `split_further` procedure to split the nodes. In Algorithm 3, lines 2-3 find the corresponding nodes $u' =$`resp_node`$(u), v' =$ `resp_node`(v) that nodes u, v are merged into in graph G_m. The `weight` of the corresponding e' in G_m of the deleted edge e is decremented in line 4. If the `weight` reaches 0, the edge is deleted. Lines 7-8 split nodes u, v from the merged nodes u', v' and also perform the subsequent splitting.

Example 4.5. We illustrate the algorithm on the graph in Example 4.4. Consider the case where the dynamic algorithm handles an edge deletion for edge $e = d \xrightarrow{(1} b$. In lines 2-3 of the Algorithm 3,

the dynamic algorithm identifies the corresponding edge $e' = \{acdf\} \xrightarrow{(_1} b$. The `weight` of e' is decremented by 1 in line 4. Then, the `split_further` procedure runs on the node $\{acdf\}$.

In lines 4-6 of Procedure 4, node $\{acdf\}$ has two outgoing edges satisfying the weight condition. After the edge deletion, the first outgoing edges $\{acdf\} \xrightarrow{(_1} b$ is merged by $a \xrightarrow{(_1} b$ and $c \xrightarrow{(_1} b$. Thus it generates a splitting restriction set $\{a, c\}$ in line 6. Similarly, the outgoing edge $\{acdf\} \xrightarrow{(_2} e$ is merged by $d \xrightarrow{(_2} e$ and $f \xrightarrow{(_2} e$. The splitting restriction set for $\{acdf\} \xrightarrow{(_2} e$ is $\{d, f\}$. The splitting condition is satisfied, and the node $\{acdf\}$ is split into two nodes $\{ac\}$ and $\{df\}$. Because $\{acdf\}$ node has no incoming edges, lines 12-18 are skipped. Then, lines 19-25 update the edge $\{acdf\} \xrightarrow{(_2} e$ to $\{df\} \xrightarrow{(_2} e$ and all the edge weights are updated as well. Finally, lines 26-28 recursively split the predecessor nodes of $\{acdf\}$. Because the node $\{acdf\}$ does not have any predecessors, the recursive node splitting terminates.

4.4 Algorithm Analysis

4.4.1 Correctness. In this section, we show the correctness of our dynamic bidirected Dyck-reachability algorithm. The correctness of the pre-processing and the query algorithm is immediate. Thus, we focus on establishing the correctness of the dynamic updates.

Recall that there are two types of graph updates: edge insertions and edge deletions. We first show the correctness of the dynamic update for edge insertions.

Lemma 4.6. *For an edge insertion, the dynamic algorithm maintains the bidirected Dyck-reachability result correctly,* i.e.*, two nodes are Dyck-reachable in G iff they are in the same merged node in the maintained merged graph G_m,*

Proof. Algorithm 2 handles the update of edge insertion for the edge e in the dynamic algorithm. Algorithm 2 only inserts the corresponding edge e' of e in the merged graph and then calls the procedure `Opt-Dyck'`. The soundness and completeness of the maintained merged graph is guaranteed due to the Lemmas 3.1 and 3.2 in the work of Chatterjee et al. [2018].

□

Algorithm 3 handles the graph updates for edge deletions. To prove the correctness of Algorithm 3, we first show that after deleting an edge from graph G, the deletion algorithm correctly maintains the merged graph G_m. We discuss the correctness of the reachability between an arbitrary node pair (u, v) in the graph G based on three cases.

- Before the edge deletion, nodes u, v are Dyck-reachable. After the edge deletion, u, v are still Dyck-reachable. This case is discussed in Lemma 4.7.
- Before the edge deletion, nodes u, v are Dyck-reachable.After the edge deletion, u, v are no longer Dyck-reachable. Lemma 4.8 covers the correctness for this case.
- Before the edge deletion, nodes u, v are not Dyck-reachable. After the edge deletion, u, v are still not Dyck-reachable. This case is discussed in Lemma 4.9.

Notice that edge deletions do not introduce new Dyck-paths in the graph, thus it is impossible that two nodes that are initially not Dyck-reachable become Dyck-reachable after the edge deletion.

Lemma 4.7. *For any two distinct nodes u, v that are Dyck-reachable, and still Dyck-reachable after the deletion of an arbitrary edge, Algorithm 3 maintains the merged graph G_m such that G_m satisfies* `resp_node`(u) == `resp_node`(v).

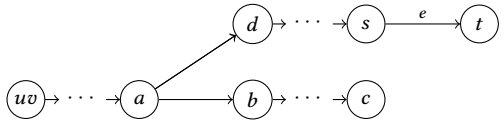


Fig. 5. Illustration for Lemma 4.7: Algorithm 3 does not split reachable node pair (u, v).

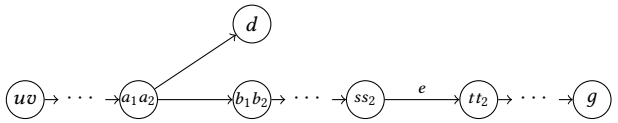


Fig. 6. Illustration for Lemma 4.8: Procedure 4 works on node pair (u, v) which is Dyck-reachable before the edge insertion but no longer reachable after the deletion.

Proof. Because nodes u, v are Dyck-reachable before and after the deletion of edge $e = s \xrightarrow{l} t$ there exists a Dyck-path P between u, v such that P does not involve e. Figure 5 depicts this situation. Suppose the path $\{uv\} \to \cdots \to a \to b \to \cdots \to c \to \cdots \to b \to a \to \cdots \to \{uv\}$ is the corresponding Dyck-path P' in the merged graph G_m. Algorithm 3 applies the node splitting algorithm recursively along the predecessor nodes. Suppose that node a in the path P' is the first predecessor of nodes s, t and Algorithm 3 tries to split node a. Because $\texttt{weight}(a \to b) > 1$, which satisfies the weight condition , the corresponding node of a in path P do not get split. This guarantees that the nodes u, v remain in the same node in G_m. □

Lemma 4.8. *For two distinct nodes u, v that are Dyck-reachable before the deletion of edge $e = s \xrightarrow{l} t$, but no longer Dyck-reachable after the edge deletion. Algorithm 3 splits these two nodes u, v such that* $\texttt{resp\_node}(u) \neq \texttt{resp\_node}(v)$.

Proof. We prove the lemma by contradiction. Because u, v are Dyck-reachable before the deletion of $e = s \xrightarrow{l} t$ but no longer reachable after the edge deletion, all Dyck-paths between u, v must include the edge $s \xrightarrow{l} t$. For an arbitrary Dyck-path P between u, v, the path P must contain the edge e. Figure 6 depicts the situation. We denote an arbitrary Dyck-path $P = u \to \cdots \to a_1 \to b_1 \to \cdots \to s \to t \to \cdots \to g \to \cdots \to t_2 \to s_2 \to \cdots \to b_2 \to a_2 \to \cdots \to v$ in G. Without loss of generality, we omit the edge labels. Algorithm 3 tries to split nodes recursively from the node $\{ss_2\}$ to $\{uv\}$ along the path. For the sake of contradiction, suppose that the nodes u, v are still in the same node,*i.e.*$\texttt{resp\_node}(u) = \texttt{resp\_node}(v)$, after the update. This indicates the node splitting terminates early along the path to $\{uv\}$. Suppose that the splitting terminates at node $\{a_1a_2\}$. Then, there must exist an outgoing edge $\{a_1a_2\} \to d$ with `weight` greater than 1 to keep the nodes a_1a_2 merged together. This means there exists a Dyck-path using nodes in the merged node d, which is a new Dyck-path that does not involve e. Notice that if the new Dyck-path using nodes in the merged node d still involves e, we can apply the same argument repetitively until we find the Dyck-path that does not involve e. Because we have a Dyck-path between u, v that does not involve e, it contradicts the hypothesis. □

Lemma 4.9. *Let u, v be two distinct nodes that are not Dyck-reachable before and after the edge deletion. When Algorithm 3 terminates,* $\texttt{resp\_node}(u) \neq \texttt{resp\_node}(v)$.

This lemma is straightforward because Algorithm 3 does not merge nodes.

Theorem 4.10. *After an edge insertion or edge deletion, two nodes u, v in the input graph G are Dyck-reachable, if and only if* $\texttt{resp\_node}(u) = \texttt{resp\_node}(v)$ *after applying the dynamic bidirected Dyck-reachability algorithm.*

Proof. The theorem follows immediately from Lemmas 4.7, 4.8 and 4.9. □

As the merged graph G_m is updated correctly, the correctness of the algorithm follows.

Corollary 4.11 (Correctness of dynamic algorithm). *Algorithm 1 maintains the bidirected Dyck-reachability result correctly for given graph update sequences.*

4.4.2 Complexity. Before presenting the running time complexity for our algorithm, we first show that our dynamic algorithm does not have any redundant computation.

Definition 4.12 (Redundant computation). There are two types of redundant computation in dynamic Dyck-reachability algorithm:

- Handling edge insertions. Consider two nodes u, v that are Dyck-reachable before the insertion. If the dynamic algorithm merges the two nodes u, v again in the merged graph G_m, the merging between u, v is considered as redundant computation.
- Handling edge deletions. If two Dyck-reachable node are split in the merged graph in the algorithm and later merged back together. The splitting and the merging for these nodes are considered redundant computation in the algorithm.

Theorem 4.13 (No redundant computation). *There is no redundant computation defined in Definition 4.12 in Algorithm 1.*

Proof. To illustrate there exists no redundant computation in Algorithm 1, it suffices to show that the there is no redundant computation in the dynamic insertion and deletion algorithms, *i.e.*, Algorithm 2 and Algorithm 3. We first consider Algorithm 2, this dynamic insertion algorithm uses the merged graph G_m with an inserted edge as inputs for the `Opt-Dyck'` procedure. If two nodes u, v are Dyck-reachable in the graph G, they are in the same node in G_m before the insertion. Because `Opt-Dyck'` does not split nodes, thus nodes u, v do not get merged again. Therefore, there is no redundant computation in Algorithm 2 for edge insertions. Algorithm 3 handles edge deletions. When two nodes u, v are merged together, the weight always increases for at least one edge. Specifically, the weight of the edge that introduces the merging increases. In the dynamic update for edge deletions, edge weights only decrease except for lines 18 and 25. Because the sum of the weights of the new edges in lines 18 and 25 is the weight of the edge e' in line 12 and 19, respectively, which gets deleted in lines 14 and 21. The weights associated with these two lines still decrease. Thus the dynamic algorithm does not merge any nodes. As Corollary 4.11 guarantees the correctness of the algorithm, there is no redundant computation for edge deletion updates. In conclusion, there is no redundant work in the dynamic bidirected Dyck-reachability algorithm. □

Next, we focus on the update complexity of the dynamic algorithm. We first introduce a key quantitative relation between the input graph G and the merged graph G_m.

Lemma 4.14. *The edge number $|E'|$ of the merged graph G_m is $O(n)$ where $n = |V|$ is the node number in the input graph G.*

Proof. We partition the edge set E' into k distinct sets $\{E'_i\}_{i=1...k}$. For the directed merge graph G_m, we have that $\sum_{v \in V} d_{in}(v) = |E'|_i$ for each type of parentheses. For each type of parentheses, the in-degree of an arbitrary node $d_{in}(v) \leq 1$. Therefore, it follows that $\sum_{v \in V} d_{in}(v)$ is $O(|V|)$. With k sets of edges, $|E'|$ is still $O(|V|)$. □

Lemma 4.15. *The dynamic algorithm for edge insertion has a complexity of $O(n)$.*

Proof. This follows from the complexity of the `Opt-Dyck'` algorithm. Its complexity has been shown to be linear to the edge number in the merged graph G_m [Chatterjee et al. 2018]. According to the Lemma 4.14, the complexity is linear to the node number in G. □

Lemma 4.16. *The time complexity of the dynamic algorithm for edge deletion is $O(n \cdot \alpha(n))$ where n is the number of nodes in the input graph.*

Proof. We first analyze the complexity for procedure `split_further`. The procedure `split_further` recursively runs on the predecessors of the nodes. We first claim that the number of the recursive runs of `split_further` is $O(|V'|)$ where V' is the node set of the updated merged graph G_m. It is due to the property that if the `split_further` procedure does not split nodes, as shown in lines 7-8, then the procedure terminates early and stops the recursive runs. Thus, the number of runs of procedure `split_further` is bounded by the node number changes in the merged graph, which is $O(|V'|)$. In lines 4-6, the number of iteration depends on the number of outgoing edges for v', so the complexity is bounded by the out-degree $d_{out}(v')$ in G_m. This means the total execution time of lines 4-6 is $O(|E'|)$. Because the join operation of disjoint-set is $O(\alpha(n))$ [Tarjan 1975], the total running time for these lines is $O(n \cdot \alpha(n))$. For the weight update, lines 12-18 and lines 19-25 iterate on the incoming and outgoing edges in the merged graph G_m. Thus, lines 12-18 and lines 19-25 execute $O(|E'|)$ times during all executions of `split_further`, so the running time for these lines is bounded by $O(n)$ as well. Finally, as we discussed the total number of executions of `split_further` is $O(n)$, the total running time of lines 26-28 is also bounded by $O(n)$. In conclusion, the total running time for `split_further` during all executions is $O(n \cdot \alpha(n))$. With the constant running time of the deletion algorithm in lines 1-9, the overall running time of the dynamic algorithm for edge deletion is $O(n \cdot \alpha(n))$ □

Theorem 4.17. *The time complexity of the dynamic insertion and dynamic deletion algorithm of bidirected Dyck-reachability is $O(n \cdot \alpha(n))$ where n is the number of nodes in the input graph and $\alpha(n)$ is the inverse Ackermann function.*

Proof. It follows from Lemmas 4.15 and 4.16 immediately. □

5 EVALUATION

This section evaluates the performance of our dynamic bidirected Dyck-reachability algorithms. We compare our dynamic algorithms against an incremental Datalog engine (DDlog) [Ryzhyk and Budiu 2019]. Incremental computation based on Datalog is perhaps the most popular approach for incremental program analysis. DDlog and LogiQL (a commercial product of LogicBlox) are two popular incremental Datalog solvers. We choose DDlog because it is more recent and it is publicly available.[2] We describe four evaluated algorithms as follows.

- DynDyck. Our dynamic algorithm described in Algorithm 1. The algorithm processes each graph update in $O(n \cdot \alpha(n))$ time.
- DynDyck$_n$. The naïve dynamic algorithm by running the optimal Dyck-reachability [Chatterjee et al. 2018] for each update. It processes each graph update in $O(m)$ time.
- DDlog. The incremental Datalog solver which only performs the minimum computation necessary to handle each graph update.
- DDlog$_n$. The naïve Datalog approach that recomputes everything (by running DDlog) from scratch upon each update.

[2] https://github.com/vmware/differential-datalog.

Table 1. Benchmark statistics and running time for alias analysis.

Subject	# Node	# Edges	# Fields	T_{Dyck} (s)	T_{DDlog} (s)
antlr	23031	21353	1246	0.42	215.63
bloat	26656	23598	1360	0.45	221.50
chart	51356	44501	3132	0.85	1193.37
eclipse	24004	21943	2857	0.49	216.40
fop	46253	39125	2857	0.79	809.11
hspldb	21646	20271	1160	0.36	214.41
jython	28033	24889	1398	0.52	215.46
luindx	22631	20915	1228	0.41	214.75
lusearch	23344	21569	1275	0.43	215.02
pmd	24586	22522	1322	0.51	218.17
xalan	21574	20186	1152	0.49	221.26

Table 2. Benchmark statistics and running time for data-dependence analysis.

Subject	# Node	# Edges	# Client Edges	# Parens	T_{Dyck} (s)	T_{DDlog} (s)
btree	1811	1757	455	801	0.01	0.03
sample	931	834	11	389	0.02	0.02
mushroom	899	809	1	376	0.02	0.02
parser	1686	1561	27	690	0.02	0.03
check	5240	5267	1304	2167	0.07	0.07
compiler	4189	4101	184	1646	0.05	0.06
compress	4375	4238	341	1721	0.05	0.06
crypto	6202	6300	1464	2540	0.08	0.09
derby	6116	5948	371	2358	0.08	0.08
helloworld	4074	3969	89	1596	0.05	0.06
mpegaudio	9650	9391	1978	3564	0.11	0.12
scimark	4583	4429	477	1782	0.05	0.06
startup	5493	5367	280	2165	0.07	0.07
sunflow	3891	3792	31	1520	0.05	0.05
xml	23922	24391	806	9128	0.34	0.31

In the experiments, we focus on evaluating two aspects of the dynamic algorithms: efficiency and scalability. For efficiency, we focus on demonstrating the benefits of dynamic algorithms, *i.e.*, the speedups of DynDyck and DDlog over DynDyck$_n$ and DDlog$_n$, respectively. For scalability, we focus on illustrating the running time increase based on the length of update operations. Finally, we evaluate all algorithms based on three settings: (1) *Incremental setting* (Section 5.2). We randomly insert all edges to initially empty graphs; (2) *Decremental setting* (Section 5.3). We randomly delete edges from the original graphs; and (3) *Mixed setting* (Section 5.4). We interleave edge insertions and edge deletions randomly in the graph.

5.1 Experimental Setup

Benchmarks. We use two sets of benchmark programs in our evaluation.

- *Alias analysis.* We first consider a context-insensitive alias analysis for Java in the standard DaCapo suite [Blackburn et al. 2006]. The analysis problem has been formulated as a bidirected Dyck-reachability problem [Yan et al. 2011; Zhang et al. 2013]. Specifically, the analysis utilizes Dyck-reachability to model field-sensitivity. The "$(_i$"-labeled edges depict field writes and the "$)_i$"-labeled edges field reads in Java. We obtain the Symbolic Points-to Graphs using the tool described in the work of Yan et al. [2011]. Table 1 presents the basic statistics of the graphs in this benchmark, including the node number, edge number and the total number of different fields for each benchmark. We also apply the optimal bidirected Dyck-reachability algorithm [Chatterjee et al. 2018] and the Datalog solver DDlog on the benchmark graphs. The performance of the two algorithms are reported in T_{Dyck} and T_{DDlog} columns of Table 1. For this benchmark, the Dyck-reachability is on average 617× faster than the Datalog approach.

- *Data-dependence analysis.* In addition, we evaluate the dynamic algorithms using a context-sensitive data-dependence analysis described in the work of Tang et al. [2015]. The analysis utilizes Dyck-reachability to model context-sensitivity. Specifically, the "$(_i$"-labeled edges depict function calls and the "$)_i$"-labeled edges function returns. We apply the analysis to the same benchmark programs given in the reference paper [Tang et al. 2015], which consists of 11 Java programs from SPECjvm2008[3] and four randomly selected programs from GitHub. Note that the input graphs considered in the original work are directed graphs. In our experiment, we use the relaxed bidirected graphs as an over-approximation for the analysis. The programs considered in this benchmark are relatively small. We choose the second benchmark because the Datalog approach ran out of time in the first benchmark. In this benchmark, the edges in the input graphs fall into two categories. The first category contains edges that encode information about the library code. The second category contains edges that encode information about client code and interactions between libraries and clients. Table 2 presents the statistics of the graphs extracted from the benchmark, including the node number, edge number, the number of client-related edges and the number of parenthesis types. Similarily, we evaluated the static algorithm of Dyck-reachability and Datalog on the benchmark graphs. The T_{Dyck} and T_{DDlog} columns report the performance result. In this benchmark, the two (static) algorithms have a similar performance. The bidirected Dyck-reachability algorithm is 1.23× faster than the Datalog solver on average.

Datalog approach for Dyck-reachability. In our experiments, we compare the performance of our dynamic bidirected Dyck-reachability algorithm against Differential Datalog [Ryzhyk and Budiu 2019], a recent incremental Datalog engine. Logic programming has widely been applied to static analysis problems [Bravenboer and Smaragdakis 2009; Madsen et al. 2016; Reps 1993] and Datalog is one of the popular declarative logic programming languages. We translate the bidirected Dyck-reachability problem to a corresponding Datalog program. Specifically, the labeled edges in the input graphs can be translated to relations in Datalog programs. For example, we use the relations $open(u, i, v)$ and $close(u, i, v)$ to present edges $u \xrightarrow{(_i} v$ and $u \xrightarrow{)_i} v$ in the input graph, respectively. The Dyck grammar can be represented as three Datalog rules "$S(u, v)$:– $open(u, i, a)\ S(a, b)\ close(b, i, v)$", "$S(u, v)$:– $S(u, a)\ S(a, v)$", and "$S(u, v)$:– $epsilon(u, v)$". The DDlog solver offers a `commit` command for expressing incremental changes in the inputs. To capture the dynamic updates in the input graphs, we add the `commit` command after each edge insertion and deletion. Finally, the edge insertions and deletions are encoded as commands `insert` and `delete` in DDlog, respectively, according to its documentation.[4] As our proposed algorithms are sequential algorithm, we compare against the sequential version of DDlog.

Sequence generation. Both the Dyck-reachability algorithm and Datalog implementation take as input an initial graph and a sequence of edge insertion and deletion operations. We generate three different types of operation sequences: incremental, decremental, and mixed sequences.

- *Incremental Sequences.* The goal of executing the incremental sequences is to explore the performance of the dynamic algorithm for edge insertions. For each input graph in the two benchmarks, we generate an incremental sequence by inserting all edges into an initially empty graph in a random order.
- *Decremental Sequences.* We design the decremental sequences to evaluate the performance of the dynamic algorithm for edge deletions. The decremental sequences simulate the situation of code deletions. In the first benchmark, algorithms take the complete input graphs from

[3] https://www.spec.org/jvm2008/.
[4] https://github.com/vmware/differential-datalog/blob/master/doc/command_reference/command_reference.md.

Table 3. Running time of DynDyck and DDlog over incremental and decremental sequences for programs in the alias analysis benchmark.

Subject	DynDyck Time (s)		DDlog Time (s)	
	Incremental	Decremental	Incremental	Decremental
antlr	1.28	2.28	542.15	542.47
bloat	1.64	2.41	554.34	551.05
chart	5.75	8.42	4920.39	5044.22
eclipse	1.40	2.34	538.94	532.53
fop	4.52	4.39	2582.05	2782.81
hspldb	1.18	2.29	536.61	533.48
jython	1.83	2.52	544.61	559.69
luindx	1.27	2.22	536.74	534.27
lusearch	1.35	2.21	537.95	558.32
pmd	1.45	2.20	548.81	563.62
xalan	1.17	2.12	537.07	558.48

Table 4. Running time of DynDyck and DDlog for programs in the data-dependence analysis benchmark. Each "$*$" denotes that the running time is less than 0.01 seconds.

Subject	DynDyck Time (s)		DDlog Time (s)	
	Incremental	Decremental	Incremental	Decremental
btree	0.02	0.01	0.69	0.17
sample	0.01	0.00*	0.29	0.00*
mushroom	0.01	0.00*	0.30	0.00*
parser	0.02	0.00*	0.62	0.01
check	0.09	0.03	2.10	0.53
compiler	0.06	0.00*	1.56	0.09
compress	0.06	0.00*	1.61	0.13
crypto	0.12	0.07	2.60	0.64
derby	0.11	0.01	2.35	0.16
helloworld	0.06	0.00*	1.51	0.04
mpegaudio	0.25	0.01	3.66	0.72
scimark	0.07	0.01	1.76	0.18
startup	0.09	0.01	2.09	0.13
sunflow	0.05	0.00*	1.50	0.02
xml	1.48	0.04	10.09	0.33

the benchmark as initial graphs. All edges are deleted from the graphs in a random order. In the second benchmark, because the library codebase is usually untouched by developers, our decremental sequences only delete all client edges based on a random order.

- *Mixed Sequences.* The mixed sequences focus on investigating the performance of the dynamic algorithm in the presence of both edge insertion and deletion operations. In mixed sequences, to ensure the validity of edge insertions, we only insert edges that exist in the original input graphs. We do not insert purely random edges because those edges might be invalid according to the program semantics. To achieve this, before generating the mixed sequences, we delete a set of edges and collect these edges in a candidate insertion pool. The insertion operations randomly choose edges from the candidate insertion pool. When deleting an edge, we also put the deleted edge back to the pool. Similar to the decremental setting, in the second benchmark, we only add client edges to the candidate insertion pool. The mixed sequences consists of 50% insertion operations and 50% deletion operations. We evaluate the performance of different approaches over sequences of different lengths, ranging over 10%, 20%, 30%, 40%, 50% of all graph edges in each alias analysis benchmark program and all client edges in each data-dependence analysis benchmark program.

Implementation. We implement all algorithms in C++. Both the optimal static Dyck-reachability algorithm [Chatterjee et al. 2018] and our dynamic Dyck-reachability algorithm share similar data structures. All executables are compiled using GCC with the "`-O2`" optimization flag. For DDlog, we use the latest stable release `v0.41.0` as of June 2021. All experiments were conducted on a server with two AMD EPYC 7402 CPUs and 512GB RAM, running Ubuntu 20.04.

5.2 Performance Evaluation on Incremental Sequences

We report the performance comparisons based on dynamic insertion operations. As described above, we evaluate the four different algorithms DynDyck, DynDyck$_n$, DDlog and DDlog$_n$ over the incremental sequences on two benchmarks. Recall that we also collect the running time for the corresponding static approaches on the original graphs in Table 1 and Table 2, *i.e.*, the final graphs after processing the entire incremental sequences. The running time of the static approaches can be treated as a lower bound for the running time of any dynamic insertion algorithms. Tables 3 and 4 present the detailed results for our experiments on incremental sequences for two benchmarks. We discussed the following observations for the experimental results.

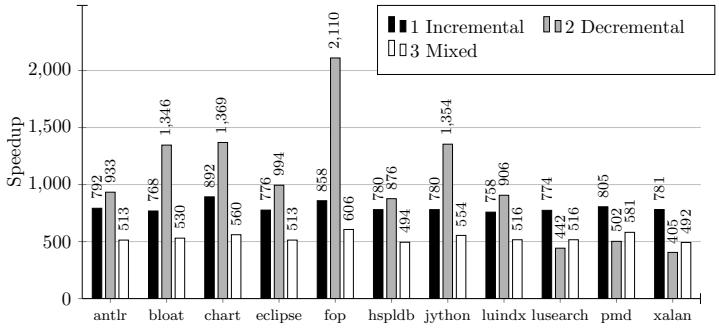


Fig. 7. Speedup of DynDyck over DynDyck$_n$ for alias analysis.

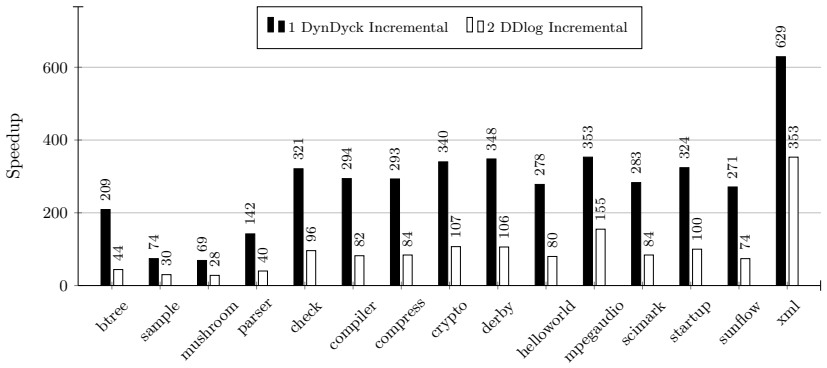
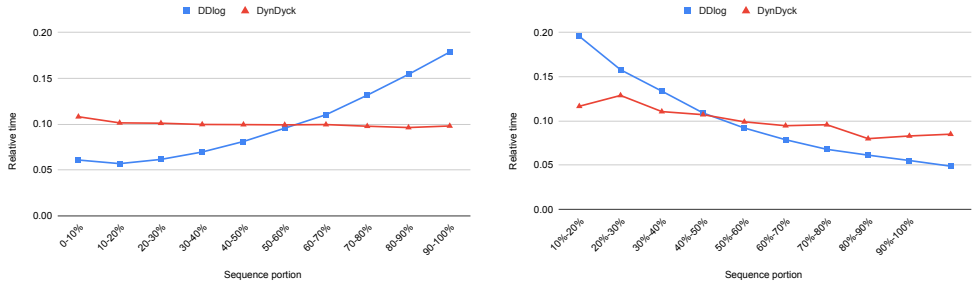


Fig. 8. Speedup Comparison of DynDyck and DDlog over naïve updates on incremental sequences for data-dependence analysis.

Efficiency. The "Incremental" columns in Tables 3 and 4 describe the execution time of DynDyck and DDlog based on incremental sequences. The "Running Time" columns in Tables 1 and 2 describe the execution time of the algorithms in the static setting. The performance of DynDyck is close to the optimal (static) solution in Tables 1 and 2. In the first benchmark, the running time of DynDyck is on average 3.66× more than the running time of the optimal (static) solution. In the second benchmark, the running time is 1.46× more. It demonstrates that our dynamic DynDyck algorithm is efficient because the running time is not far away from the static solution. The running time of dynamic DDlog is on average 2.71× and 26.48× more than that of the static DDlog on two benchmarks, respectively. We notice that the slow down ratio of DynDyck is much smaller than that of DDlog in the second benchmark. The slow down ratio for the first benchmark is slightly larger. We note that the length of the incremental sequence of the first benchmark is significant longer than that of the second benchmark because the graphs in the first benchmark contain 4× more edges on average. At the same time, DynDyck is 541× faster than DDlog in the first benchmark. Therefore, the bookkeeping cost in DynDyck is relatively more significant than that in DDlog, which leads to the slightly larger slow down.

Figure 7 presents the speedup of the running time for DynDyck over DynDyck$_n$. The first bar represents the speedups for the incremental sequences. DynDyck has achieve 796.58× speedup compared to DynDyck$_n$. It shows that DynDyck is extremely efficient. We do not compare the speedup of DynDyck and DDlog over naives updates from the first benchmark as DDlog runs out



(a) Incremental sequences. (b) Decremental sequences.

Fig. 9. Running time for each 10% of the sequences. The line with triangle data points is for dynamic bidirected Dyck-reachability algorithm. The line with square data points is for DDlog

of the time budget of 30 minutes. For the second benchmark, Figure 8 illustrate the comparison of speedups for DynDyck and DDlog. The DDlog has achieved only a 97.4× speedup while DynDyck a 282.4× speedup over the naïve counterpart. Even though the naïve DynDyck$_n$ algorithm itself is more efficient than DDlog$_n$, DynDyck achieves more significant speedups than DDlog in practice.

Scalability. Figure 9a illustrates the average relative running time of executing each 10% of the entire incremental sequences. For each program, we define the relative running time as T/T_{inc} where T is the fractional running time and T_{inc} is the total running time of DynDyck and DDlog for both benchmarks. Figure 9a shows the increase of the running time compared to the sequence size increase. DynDyck does not demonstrate a increase of running time with the increase graph size. The running time of DDlog grows more and more dramatically as the graph size gets larger. It clearly shows that DynDyck is more scalable than the incremental DDlog.

Summary:

- *Efficiency.* In the incremental setting, DynDyck achieves 796.58× and 282.4× speedup over DynDyck$_n$ on two benchmarks, respectively. It also achieves better speedup compared to DDlog because DDlog has only 97.4× speedup over DDlog$_n$ on the second benchmark. In addition, starting with an empty graph, DynDyck is only 3.66× and 1.46× slower than the optimal Dyck-reachability algorithm in the static setting on two benchmarks.
- *Scalability.* As shown in Figure 9a, DynDyck scales linearly in terms of sequence size increases. It is more scalable than DDlog.

5.3 Performance Evaluation on Decremental Sequences

We report the performance comparisons based on dynamic deletion operations. Specifically, we evaluate the four algorithms over decremental sequences. When executing the decremental sequences, we record the running time for every 10% portion of entire sequences.

Efficiency. The "Decremental" columns in Tables 3 and 4 present the running time of the four evaluated algorithms over decremental sequences. For the absolute running time, DynDyck is on average 382× faster than DDlog, which demonstrates that DynDyck is very efficient for decremental sequences. As DDlog$_n$ runs out of time budget for the first benchmark, we only compare the speedups for DynDyck and DDlog over naive updates for the second benchmark. Figure 10 demonstrates the speedup comparison for two appraoches in the data-dependence analysis benchmark. Compared to the naïve DynDyck$_n$, DynDyck achieves on average a 1021.5× speedup for

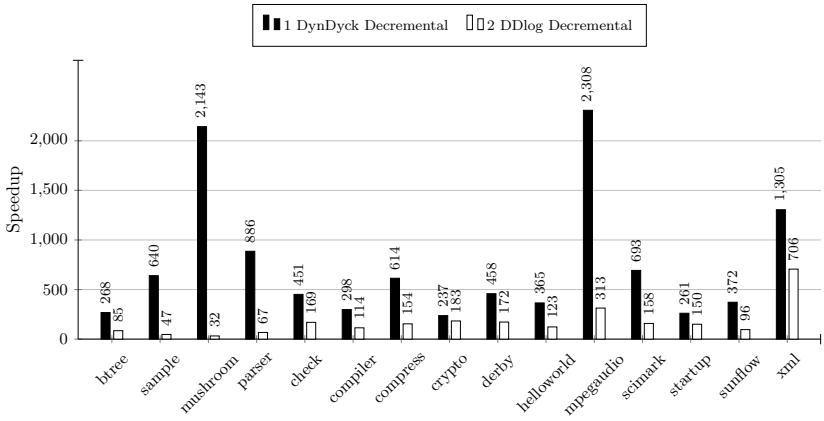


Fig. 10. The speedup comparison of DynDyck and DDlog on the decremental sequences for the data-dependence analysis benchmarks.

the first benchmark and a 753.3× speedup for the second benchmark. It shows DynDyck achieves significant improvements over DynDyck$_n$.

Compared to DDlog, DynDyck achieves more significant speedup over the naïve approach. Specifically, the speedup of DynDyck in the second benchmark is 753.3× on average. However, the average speedup for DDlog is 171.2×. In addition, DynDyck is 381× and 20× faster than DDlog on the two analysis benchmarks.

Scalability. Figure 9b demonstrates the average relative running time of executing every 10% portion of the decremental sequences. We define the relative running time as T/T_{dec} where T is the fractional running time and T_{dec} is the running time of executing the entire decremental sequences. Figure 9b shows how the running time changes when the graph size decreases. In decremental sequences, when executing the first a few 10% sequences, the graph is relatives larger. The figure demonstrates that when the graph size decreases, the relative running time decrease of DynDyck algorithm is less significant than that of DDlog. It further shows that the running time for DDlog on decremental sequences for larger graphs increases more significantly. Therefore, DynDyck is more scalable than DDlog on large graphs.

Summary:

- *Efficiency.* In the decremental setting, DynDyck achieves 1021.5× and 753.3× speedup over DynDyck$_n$ on two benchmarks. Compared to DDlog, DynDyck achieves better speedup because DDlog has achieved only only a 171.2× speedup on the second benchmark.
- *Scalability.* As shown in Figure 9b, DynDyck more scalability because the running time increase compared to DDlog is less significant.

5.4 Performance Evaluation on Mixed Sequences

The last experiment reports the performance of four algorithms on the mixed sequences, which reflects a more practical setting. This experiment demonstrates the overall performance of the dynamic algorithms for bidirected Dyck-reachability.

Recall that in the mixed setting, we choose the sequences that contain 10%, 20%, 30%, 40% and 50% of all graph edges in each benchmark. Figure 7 demonstrates the speedups of DynDyck over DynDyck$_n$ on mixed sequences for first benchmark. We do not include the speedup of DDlog

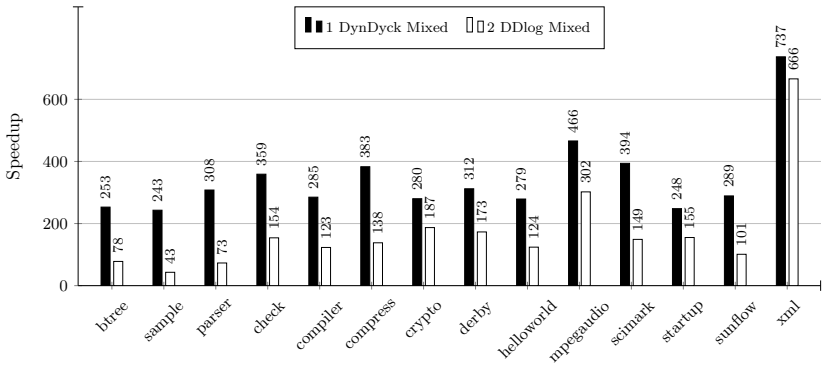


Fig. 11. Speedup comparison for DynDyck and DDlog over mixed sequences for data-dependence analysis.

for this benchmark as it runs out of the time budget of 24 hours. Figure 11 demonstrates the speedup comparison between DynDyck and DDlog over DynDyck$_n$ and DDlog$_n$ for the second benchmark. For each benchmark, we compare the average speedup of the 5 mixed sequences with different length. On average, DynDyck achieves 534× and 331× speedup compared to DynDyck$_n$ in two benchmarks, respectively. It shows that DynDyck is quite efficient compared to DynDyck$_n$. We also compare the speedup achieved by DDlog with DynDyck in the second benchmark. On average, DDlog achieves a 176× speedup while DynDyck achieves a 331× speedup. Even though DynDyck$_n$ is more efficient than DDlog$_n$, DynDyck can achieve more speedup than the DDlog.

Summary:

- *Efficiency.* In the mixed setting, DynDyck achieves 534× and 331× speedups over DynDyck$_n$ in two benchmarks, respectively. It is more efficient than DDlog because DDlog can only achieve a 176× speedup over DDlog$_n$ on the second benchmark.

6 RELATED WORK

6.1 Dyck-Reachability

Dyck reachability is central to program analysis as many program analyses have to match properly matched parentheses property [Chatterjee et al. 2018; Zhang et al. 2013]. Dyck-reachability is expensive to solve. The traditional CFL-reachability algorithm exhibits a cubic time complexity [Reps 1998]. Chaudhuri [2008] proposes a subcubic CFL-reachability algorithm, which improves the cubic complexity by a logarithmic factor. Chatterjee et al. [2018] establish a cubic conditional lower for Dyck-reachability in general case. For bidirected graphs, Zhang et al. [2013] present an $O(m \log m)$-time algorithm by exploiting an equivalence Dyck-relation. The result has been improved to $O(m)$-time by Chatterjee et al. [2018]. Moreover, Chatterjee et al. [2018] also prove that the $O(m)$-time algorithm is optimal. However, all existing Dyck-reachability and CFL-reachability algorithms handle only static graphs. Our work fills the gap and presents the first dynamic bidirected Dyck-reachability algorithm.

6.2 Dynamic Graph Algorithms

Dyck-reachability is a generalization of standard graph reachability (*i.e.*, transitive closure). The problem of dynamic transitive closure has been extensively studied [Demetrescu and Italiano 2000; King and Sagert 1999; Roditty 2003; Roditty and Zwick 2004; Sankowski 2004]. Henzinger et al. [2015] give a conditional lower bound of $(poly, m^{1/2-\delta}, m^{1-\delta})$ on dynamic transitive closure for any

small constant $\delta > 0$ based on the Online Boolean Matrix-Vector Multiplication (OMv) conjecture. This bound even holds for the s-t reachability problem, where both s and t are fixed for all queries. Note that for bidirected Dyck-reachability, the naïve approach of running the $O(m)$-time optimal algorithm for each update gives a dynamic algorithm with a complexity tuple $(O(m), O(m), O(1))$. For directed graphs with n vertices and m edges, the straightforward dynamic transitive closure algorithm based on BFS or DFS exhibits a complexity tuple $(O(1), O(1), O(m+n))$ [Frigioni et al. 2001]. Roditty [2003] gives a deterministic dynamic transitive closure algorithm with an $O(n^2)$ update time and an $O(1)$ query time. Roditty and Zwick [2004] propose a dynamic transitive closure algorithm with an $O(m + n \log n)$ update time and an $O(n)$ query time. Ramalingam and Reps [1996] study the complexity of of dynamic graph algorithms and discuss various upper-bound and lower-bound results. Existing fast dynamic transitive closure algorithms cannot be directly applied to bidirected Dyck-reachability.

6.3 Incremental Program Analysis

Incremental program analysis has been extensively studied in the literature, such as dataflow analysis [Burke and Ryder 1990; Carroll and Ryder 1988; Ryder and Paull 1988] and incremental alias analysis [Liu et al. 2019; Lu et al. 2013; Saha and Ramakrishnan 2005a; Yur et al. 1999]. Incremental algorithms selectively update analysis results after a program change instead of recomputing everything from scratch. Liu [2003]; Ramalingam and Reps [1993] conduct surveys on incremental computation related to program analysis. Sreedhar et al. [1997] propose an incremental algorithm for maintaining dominator trees in flow graphs. Saha and Ramakrishnan [2005b, 2006] propose incremental evaluation methods for logic programs. Szabó et al. [2016] propose a DSL for incremental program analysis based on incremental graph pattern matching. Szabó et al. [2018] propose a framework that supports incremental maintenance of recursive lattice-value aggregation in Datalog. Pacak et al. [2020] compile incremental type checkers to Datalog programs. Szabó et al. [2021] propose a Datalog-based approach for supporting whole-program lattice-based dataflow analyses. Eichberg et al. [2007] propose incremental static analyses specified in Prolog using incremental tabled evaluation. Infer [Distefano et al. 2019] leverages incremental compositional analysis to improve performance. Reviser [Arzt and Bodden 2014] is an incremental framework for IFDS/IDE-based program analyses. Self-adjusting computation [Acar 2009; Acar et al. 2009; Hammer et al. 2015] involves incremental computation. Specifically, given a set of input changes, self-adjusting computation performs change propagation, where it reuses the memorized intermediate results for all sub-computations. Adapton [Hammer et al. 2014] is an incremental computation library based on demand-driven semantics. Moreover, there exist automatic incrementalizers for object-oriented languages [Liu et al. 2005] and dynamic data structures [Shankar and Bodík 2007]. Different from existing work, our work focus on the algorithmic problem of bidirected Dyck-reachability. Our dynamic algorithms are asymptotically faster than the straightforward approach based on the optimal bidirected Dyck-reachability algorithm.

7 CONCLUSION

This paper has presented efficient dynamic algorithms for bidirected Dyck-reachability. Our algorithms can handle dynamic graph updates in $O(m)$ pre-processing time, $O(n \cdot \alpha(n))$ update time, and $O(1)$ query time. To the best of our knowledge, this is the first algorithmic result for dynamic Dyck-reachability. We have applied the dynamic bidirected Dyck-reachability algorithms to two practical client analyses. The evaluation results show that our dynamic algorithms can achieve orders-of-magnitude speedup over a straightforward approach and an incremental Datalog solver.

ACKNOWLEDGMENTS

We thank Shuo Ding, Benjamin Mikek, and the anonymous reviewers for their feedback on earlier drafts of this paper. Specifically, we would like to thank Leonid Ryzhyk for discussions on DDlog and for suggesting a better set of Datalog rules for Dyck-reachability. This work was supported, in part, by Amazon under an Amazon Research Award in automated reasoning; by the United States National Science Foundation (NSF) under grants No. 1917924 and No. 2114627; and by the Defense Advanced Research Projects Agency (DARPA) under grant N66001-21-C-4024. The first author was partially supported by the Facebook Graduate Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the above sponsoring entities.

REFERENCES

Amir Abboud and Virginia Vassilevska Williams. 2014. Popular Conjectures Imply Strong Lower Bounds for Dynamic Problems. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*. IEEE Computer Society, 434–443. https://doi.org/10.1109/FOCS.2014.53

Umut A. Acar. 2009. Self-adjusting computation: (an overview). In *Proceedings of the 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2009, Savannah, GA, USA, January 19-20, 2009*, Germán Puebla and Germán Vidal (Eds.). ACM, 1–6. https://doi.org/10.1145/1480945.1480946

Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. 2009. An experimental analysis of self-adjusting computation. *ACM Trans. Program. Lang. Syst.* 32, 1 (2009), 3:1–3:53. https://doi.org/10.1145/1596527.1596530

Steven Arzt and Eric Bodden. 2014. Reviser: efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 288–298. https://doi.org/10.1145/2568225.2568243

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. https://doi.org/10.1145/2594291.2594299

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* (Portland, OR, USA). ACM Press, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, Shail Arora and Gary T. Leavens (Eds.). ACM, 243–262. https://doi.org/10.1145/1640089.1640108

Michael G. Burke and Barbara G. Ryder. 1990. A Critical Analysis of Incremental Iterative Data Flow Analysis Algorithms. *IEEE Trans. Software Eng.* 16, 7 (1990), 723–728. https://doi.org/10.1109/32.56098

Martin D. Carroll and Barbara G. Ryder. 1988. Incremental Data Flow Analysis via Dominator and Attribute Updates. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, Jeanne Ferrante and P. Mager (Eds.). ACM Press, 274–284. https://doi.org/10.1145/73560.73584

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. https://doi.org/10.1145/3158118

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. https://doi.org/10.1145/1328438.1328460

Camil Demetrescu and Giuseppe F. Italiano. 2000. Fully Dynamic Transitive Closure: Breaking Through the $O(n^2)$ Barrier. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*. IEEE Computer Society, 381–389. https://doi.org/10.1109/SFCS.2000.892126

Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70. https://doi.org/10.1145/3338112

Michael Eichberg, Matthias Kahl, Diptikalyan Saha, Mira Mezini, and Klaus Ostermann. 2007. Automatic Incrementalization of Prolog Based Static Analyses. In *Practical Aspects of Declarative Languages, 9th International Symposium, PADL 2007,*

Nice, France, January 14-15, 2007 (Lecture Notes in Computer Science, Vol. 4354), Michael Hanus (Ed.). Springer, 109–123. https://doi.org/10.1007/978-3-540-69611-7_7

Daniele Frigioni, Tobias Miller, Umberto Nanni, and Christos D. Zaroliagis. 2001. An Experimental Study of Dynamic Algorithms for Transitive Closure. *ACM J. Exp. Algorithmics* 6 (2001), 9. https://doi.org/10.1145/945394.945403

Matthew A. Hammer, Jana Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael W. Hicks, and David Van Horn. 2015. Incremental computation with names. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 748–766. https://doi.org/10.1145/2814270.2814305

Matthew A. Hammer, Yit Phang Khoo, Michael Hicks, and Jeffrey S. Foster. 2014. Adapton: composable, demand-driven incremental computation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 156–166. https://doi.org/10.1145/2594291.2594324

Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. 2015. Unifying and Strengthening Hardness for Dynamic Problems via the Online Matrix-Vector Multiplication Conjecture. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, Rocco A. Servedio and Ronitt Rubinfeld (Eds.). ACM, 21–30. https://doi.org/10.1145/2746539.2746609

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. https://doi.org/10.1109/LICS.2009.45

Valerie King and Garry Sagert. 1999. A Fully Dynamic Algorithm for Maintaining the Transitive Closure. In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing, May 1-4, 1999, Atlanta, Georgia, USA*, Jeffrey Scott Vitter, Lawrence L. Larmore, and Frank Thomson Leighton (Eds.). ACM, 492–498. https://doi.org/10.1145/301250.301380

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. https://doi.org/10.1145/996841.996867

Shankaranarayanan Krishna, Aniket Lal, Andreas Pavlogiannis, and Omkar Tuppe. 2023. On-The-Fly Static Analysis via Dynamic Bidirected Dyck Reachability. arXiv:2311.04319 [cs.PL]

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. https://doi.org/10.1145/3385412.3386021

Bozhen Liu, Jeff Huang, and Lawrence Rauchwerger. 2019. Rethinking Incremental and Parallel Pointer Analysis. *ACM Trans. Program. Lang. Syst.* 41, 1 (2019), 6:1–6:31. https://doi.org/10.1145/3293606

Yanhong A. Liu. 2003. Iterate, Incrementalize, and Implement: A systematic approach to efficiency improvement and guarantees. *Electron. Notes Theor. Comput. Sci.* 90 (2003), 45–47. https://doi.org/10.1016/S1571-0661(03)00007-0

Yanhong A. Liu, Scott D. Stoller, Michael Gorbovitski, Tom Rothamel, and Yanni Ellen Liu. 2005. Incrementalization across object abstraction. 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*, Ralph E. Johnson and Richard P. Gabriel (Eds.). ACM, 473–486. https://doi.org/10.1145/1094811.1094848

Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. 2013. An Incremental Points-to Analysis with CFL-Reachability. In *Compiler Construction - 22nd International Conference, CC 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7791)*, Ranjit Jhala and Koen De Bosschere (Eds.). Springer, 61–81. https://doi.org/10.1007/978-3-642-37051-9_4

Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. 2016. From Datalog to flix: a declarative language for fixed points on lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 194–208. https://doi.org/10.1145/2908080.2908096

André Pacak, Sebastian Erdweg, and Tamás Szabó. 2020. A systematic approach to deriving incremental type checkers. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 127:1–127:28. https://doi.org/10.1145/3428195

G. Ramalingam. 2000. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.* 22, 2 (2000), 416–430. https://doi.org/10.1145/349214.349241

G. Ramalingam and Thomas W. Reps. 1993. A Categorized Bibliography on Incremental Computation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, Mary S. Van Deusen and Bernard Lang (Eds.). ACM Press, 502–510. https://doi.org/10.1145/158511.158710

G. Ramalingam and Thomas W. Reps. 1996. On the Computational Complexity of Dynamic Graph Problems. *Theor. Comput. Sci.* 158, 1&2 (1996), 233–277. https://doi.org/10.1016/0304-3975(95)00079-8

Thomas W. Reps. 1993. Demand Interprocedural Program Analysis Using Logic Databases. In *Applications of Logic Databases (The Kluwer International Series in Engineering and Computer Science 296)*, Raghu Ramakrishnan (Ed.). Kluwer, 163–196.

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. https://doi.org/10.1145/345099.345137

Liam Roditty. 2003. A faster and simpler fully dynamic transitive closure. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA*. ACM/SIAM, 404–412. http://dl.acm.org/citation.cfm?id=644108.644172

Liam Roditty and Uri Zwick. 2004. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, László Babai (Ed.). ACM, 184–191. https://doi.org/10.1145/1007352.1007387

Barbara G. Ryder and Marvin C. Paull. 1988. Incremental Data-Flow Analysis. *ACM Trans. Program. Lang. Syst.* 10, 1 (1988), 1–50. https://doi.org/10.1145/42192.42193

Leonid Ryzhyk and Mihai Budiu. 2019. Differential Datalog. In *Datalog 2.0 2019 - 3rd International Workshop on the Resurgence of Datalog in Academia and Industry (CEUR Workshop Proceedings, Vol. 2368)*. 56–67. http://ceur-ws.org/Vol-2368/paper6.pdf

Diptikalyan Saha and C. R. Ramakrishnan. 2005a. Incremental and demand-driven points-to analysis using logic programming. In *Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 11-13 2005, Lisbon, Portugal*, Pedro Barahona and Amy P. Felty (Eds.). ACM, 117–128. https://doi.org/10.1145/1069774.1069785

Diptikalyan Saha and C. R. Ramakrishnan. 2005b. Symbolic Support Graph: A Space Efficient Data Structure for Incremental Tabled Evaluation. In *Logic Programming, 21st International Conference, ICLP 2005, Sitges, Spain, October 2-5, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3668)*, Maurizio Gabbrielli and Gopal Gupta (Eds.). Springer, 235–249. https://doi.org/10.1007/11562931_19

Diptikalyan Saha and C. R. Ramakrishnan. 2006. A Local Algorithm for Incremental Evaluation of Tabled Logic Programs. In *Logic Programming, 22nd International Conference, ICLP 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4079)*, Sandro Etalle and Miroslaw Truszczynski (Eds.). Springer, 56–71. https://doi.org/10.1007/11799573_7

Piotr Sankowski. 2004. Dynamic Transitive Closure via Dynamic Matrix Inverse (Extended Abstract). In *45th Symposium on Foundations of Computer Science (FOCS 2004), 17-19 October 2004, Rome, Italy, Proceedings*. IEEE Computer Society, 509–517. https://doi.org/10.1109/FOCS.2004.25

Ajeet Shankar and Rastislav Bodík. 2007. DITTO: automatic incrementalization of data structure invariant checks (in Java). In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 310–319. https://doi.org/10.1145/1250734.1250770

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. https://doi.org/10.1145/3290361

Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. 1997. Incremental Computation of Dominator Trees. *ACM Trans. Program. Lang. Syst.* 19, 2 (1997), 239–252. https://doi.org/10.1145/244795.244799

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. https://doi.org/10.1145/1133981.1134027

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. https://doi.org/10.1145/1094811.1094817

Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. 2018. Incrementalizing lattice-based program analyses in Datalog. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 139:1–139:29. https://doi.org/10.1145/3276509

Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. 2021. Incremental whole-program analysis in Datalog with lattices. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1–15. https://doi.org/10.1145/3453483.3454026

Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: a DSL for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 320–331. https://doi.org/10.1145/2970276.2970298

Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. 2015. Summary-Based Context-Sensitive Data-Dependence Analysis in Presence of Callbacks. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium*

on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015, Sriram K. Rajamani and David Walker (Eds.). ACM, 83–95. https://doi.org/10.1145/2676726.2676997

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

Guoqing 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 (Lecture Notes in Computer Science, Vol. 5653)*, Sophia Drossopoulou (Ed.). Springer, 98–122. https://doi.org/10.1007/978-3-642-03013-0_6

Dacong Yan, Guoqing 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. https://doi.org/10.1145/2001420.2001440

Jyh-Shiarn Yur, Barbara G. Ryder, and William Landi. 1999. An Incremental Flow- and Context-Sensitive Pointer Aliasing Analysis. In *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999*, Barry W. Boehm, David Garlan, and Jeff Kramer (Eds.). ACM, 442–451. https://doi.org/10.1145/302405.302676

Qirun Zhang. 2024. A Note on Dynamic Bidirected Dyck-Reachability with Cycles. *CoRR* abs/2401.03570 (2024). arXiv:2401.03570 https://arxiv.org/abs/2401.03570

Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. 2013. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 435–446. https://doi.org/10.1145/2491956.2462159