On-the-Fly Static Analysis via Dynamic Bidirected Dyck Reachability

SHANKARANARAYANAN KRISHNA, IIT Bombay, India
ANIKET LAL, IIT Bombay, India
ANDREAS PAVLOGIANNIS, Aarhus University, Denmark
OMKAR TUPPE, IIT Bombay, India

Dyck reachability is a principled, graph-based formulation of a plethora of static analyses. Bidirected graphs are used for capturing dataflow through mutable heap data, and are usual formalisms of demand-driven points-to and alias analyses. The best (offline) algorithm runs in $O(m + n \cdot \alpha(n))$ time, where n is the number of nodes and m is the number of edges in the flow graph, which becomes $O(n^2)$ in the worst case.

In the everyday practice of program analysis, the analyzed code is subject to continuous change, with source code being added and removed. On-the-fly static analysis under such continuous updates gives rise to *dynamic Dyck reachability*, where reachability queries run on a dynamically changing graph, following program updates. Naturally, executing the *offline* algorithm in this *online* setting is inadequate, as the time required to process a single update is prohibitively large.

In this work we develop a novel dynamic algorithm for bidirected Dyck reachability that has $O(n \cdot \alpha(n))$ worst-case performance per update, thus beating the $O(n^2)$ bound, and is also optimal in certain settings. We also implement our algorithm and evaluate its performance on on-the-fly data-dependence and alias analyses, and compare it with two best known alternatives, namely (i) the optimal offline algorithm, and (ii) a fully dynamic Datalog solver. Our experiments show that our dynamic algorithm is consistently, and by far, the top performing algorithm, exhibiting speedups in the order of 1000X. The running time of each update is almost always unnoticeable to the human eye, making it ideal for the on-the-fly analysis setting.

CCS Concepts: • **Software and its engineering** → **Software verification and validation**; • **Theory of computation** → *Theory and algorithms for application domains*; *Program analysis.*

Additional Key Words and Phrases: CFL reachability, static analysis, dynamic algorithms

ACM Reference Format:

Shankaranarayanan Krishna, Aniket Lal, Andreas Pavlogiannis, and Omkar Tuppe. 2024. On-the-Fly Static Analysis via Dynamic Bidirected Dyck Reachability. *Proc. ACM Program. Lang.* 8, POPL, Article 42 (January 2024), 30 pages. https://doi.org/10.1145/3632884

Authors' addresses: Shankaranarayanan Krishna, IIT Bombay, India, krishnas@cse.iitb.ac.in; Aniket Lal, IIT Bombay, India, aniketlal@cse.iitb.ac.in; Andreas Pavlogiannis, Aarhus University, Aabogade 34, Aarhus, 8200, Denmark, pavlogiannis@cs.au.dk; Omkar Tuppe, IIT Bombay, India, omkarvtuppe@cse.iitb.ac.in.

ACM 2475-1421/2024/1-ART42
https://doi.org/10.1145/3632884

1 INTRODUCTION

Dyck reachability in static analysis. Dyck reachability is an elegant and widespread graph-based formulation of a plethora of static analyses. The reachability problem is phrased on labeled directed graphs $G = (V, E)$, where V is a set of nodes and E is a set of edges, labeled with opening and closing parenthesis symbols. Given two nodes u and v, the task is to determine whether v is reachable from u by means of a path P, such that the sequence of labels of the edges of P produce a parenthesis string that is properly balanced [Yannakakis 1990]. In the static analysis domain, G serves as the program model, where nodes represent basic program constructs such as program variables, pointers, or statements, and edges capture data flow between these constructs. Effectively, program executions carrying data flow between distant program points are captured by paths in G. Naturally, as G is an approximate model of the program, it may have paths that do not correspond to any valid program execution, leading to spurious data flow and thus to false positive warnings. In order to increase the precision of the analysis, parenthesis symbols are used to model certain restrictions in the data flow, such as sensitivity to calling contexts and field accesses [Reps 1997, 2000; Späth et al. 2019]. Focusing on Dyck reachability on G (as opposed to plain reachability) thus filters out paths that are guaranteed to not correspond to valid program executions, thereby increasing the precision of the analysis.

As a modeling formalism, Dyck reachability strikes a remarkable balance between simplicity and expressiveness, and has been used to drive a wide range of static analyses, such as interprocedural data-flow analysis [Reps et al. 1995], slicing [Reps et al. 1994], shape analysis [Reps 1995a], impact analysis [Arnold 1996], type-based flow analysis [Rehof and Fähndrich 2001], taint analysis [Huang et al. 2015], data-dependence analysis [Tang et al. 2017], alias/points-to analysis [Lhoták and Hendren 2006; Xu et al. 2009; Zheng and Rugina 2008], to only name a few. In practice, widely-used analysis tools, such as Wala [Wal 2003] and Soot [Bodden 2012] equip Dyck-reachability techniques to perform the analysis. From a complexity perspective, answering a single Dyck-reachability query "is v reachable from u?" takes $O(n^3)$ time, where n is the number of nodes of G, and although some slight improvements are possible [Chaudhuri 2008], this cubic dependency is often considered prohibitive [Heintze and McAllester 1997].

Bidirectedness. One important and practically motivated variant of Dyck reachability is that of *bidirectedness*. Intuitively, a bidirected graph G has the property that every (directed) edge is present in both directions with complementary labels: an edge $x \xrightarrow{(} y$ is present in G if and only if $y \xrightarrow{)} x$ is also present in G. Bidirectedness makes reachability an *equivalence relation*: if v is reachable from u via a properly balanced path, the same path backwards (i.e., from v to u) is also properly balanced, thereby witnessing the reachability of u from v. Thus, the nodes of G can be partitioned into equivalence classes of inter-reachable nodes, called *Dyck Strongly Connected Components* (or *DSCCs*, for short). For example, in Figure 1 the bidirected graph G (top right) corresponds to the input program, and nodes c and d are inter-reachable via the paths $c \xrightarrow{L} f \xrightarrow{\overline{L}} d$ and $d \xrightarrow{L} f \xrightarrow{\overline{L}} c$, and thus they belong to the same DSCC.

From a *semantics* perspective, bidirectedness has been a standard approach to handle mutable heap data [Lu and Xue 2019; Sridharan and Bodík 2006; Xu et al. 2009; Zhang and Su 2017] – though it can sometimes be relaxed for read-only accesses [Milanova 2020], and the de-facto formulation of demand-driven points-to analyses [Shang et al. 2012; Sridharan et al. 2005; Vedurada and Nandivada 2019; Yan et al. 2011; Zheng and Rugina 2008]. From an *algorithmic* perspective, bidirectedness allows Dyck reachability to be computed more efficiently [Yuan and Eugster 2009; Zhang et al. 2013], with the fastest algorithm running in $O(m + n \cdot \alpha(n))$ time [Chatterjee et al. 2018], where n is the

```
class ATree {
  ATree L;
  ATree R;
}
void main(){
  ATree c,d,e;
  ATree f,g,h;
  g.L=e;
– d=f.L;       ②
  h.L=f;
  f.L=c;
+ h=d.R;       ①
  c.R=g;
  e=f.L
}
```

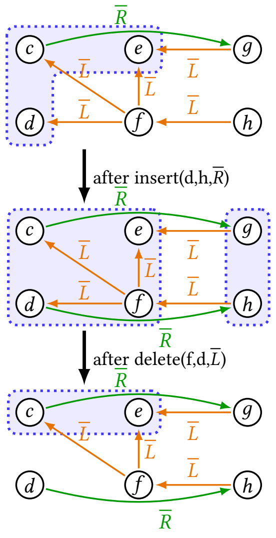

$\overline{R}$
$\overline{L}$
after insert(d,h,$\overline{R}$)
after delete(f,d,$\overline{L}$)
c
d
e
f
g
h

Fig. 1. An input program (left), to which two updates are made, an insertion ① and a deletion ② , and the corresponding flow graphs for field-sensitive alias analysis after each update (right). The edges are labeled $L, \overline{L}, R, \overline{R}$ and correspond to the fields of `ATree`. For notational convenience, we only draw the edges labeled with closing parentheses, the reverse edges with opening parentheses are implied by bidirectedness. On the initial program (before ① and ②), bidirected Dyck-reachability reports four DSCCs: $\{c, d, e\}$, $\{f\}$, $\{g\}$, and $\{h\}$. The addition ① inserts the edge $d \xrightarrow{\overline{R}} h$ in the flow graph, which now has two DSSCs:$\{c, d, e, f\}$ and $\{g, h\}$. The deletion ② deletes the edge $f \xrightarrow{\overline{L}} d$ in the flow graph, leaving it with five DSCCs: $\{c, e\}$, $\{d\}$, $\{f\}$, $\{g\}$, and $\{h\}$.

number of nodes, m is the number of edges, and $\alpha(n)$ is the inverse Ackermann function[1]. Compared to the cubic bound of directed Dyck reachability, bidirectedness thus allows for a large speedup in the order of n^2. Due to this algorithmic benefit, bidirectedness also serves as an overapproximation to directed reachability, and has been suggested as a mechanism to speed up challenging static analysis and verification problems [Ganardi et al. 2022b,a; Li et al. 2020].

On-the-fly static analysis. In the everyday practice of program analysis, the analyzed code base is not static but rather subject to perpetual change; source-code lines are added and removed following patches, new modules and libraries, and bug fixes. In an even more demanding setting, lightweight static analyzers are embedded inside the IDE and run on-the-fly, so as to enable faster, more robust and more productive development. As has been observed in the literature, a major computational challenge in static analyses is in their apparent inability to adapt to these continuous changes efficiently [Arzt and Bodden 2014; Zadeck 1984]. Existing works approach this challenge mostly by tweaking the offline analyses and adapting them to the dynamic setting. Typically, such adaptations only focus on incremental updates (i.e., only the addition of code lines) and are based on some form of caching [Arzt and Bodden 2014; Burke and Ryder 1990; Liu et al. 2019; Pacak et al. 2020; Szabó et al. 2016]. Decremental updates (i.e., the deletion of code lines) is much more difficult, intuitively, due to the fixpoint nature of static analyses. As such, analyses that fully follow all code changes (both incremental and decremental) thus far offer no efficiency guarantees. This paper tackles this challenge of on-the-fly, full dynamic analyses formulated as bidirected Dyck reachability in a rigorous and provably efficient way. Figure 1 illustrates the use of dynamic bidirected Dyck reachability for on-the-fly field-sensitive alias analysis on a small example.

Our contributions. We consider the problem of maintaining the DSCCs of a dynamic graph G, representing the on-the-fly static analysis of a program, where source code changes are both incremental and decremental. In particular, an incremental update inserts an edge in G, while a decremental update deletes edges from G, and the task is to restore the DSCCs of G after each such update. To this end, we develop an algorithm DynamicAlgo, with guarantees as stated in the following theorem.

[1]For all practical purposes, $\alpha(n) \leq 5$, i.e., the function behaves as a constant.

Theorem 1.1. *Given a bidirected graph G of n nodes,* DynamicAlgo *correctly maintains the DSCCs of G across edge insertions and edge deletions, and uses at most $O(n \cdot \alpha(n))$ time for each update.*

Observe that in general this is much faster than the optimal offline algorithm of [Chatterjee et al. 2018], as the latter spends time that is at least proportional to the number of edges m (which can be up to $m = \Theta(n^2)$). On the other hand, it can be easily shown that a single update (either edge insertion or deletion) can affect $\Theta(n)$ DSCCs. Thus DynamicAlgo is effectively optimal, at least in the natural setting where DSCCs have to be explicitly maintained at all times.

We have implemented our dynamic algorithm and evaluated its performance on Dyck graphs that arise on data-dependence analysis and alias analysis, on standard benchmarks. Our experiments show that our dynamic algorithm is consistently, and by far, the top performing algorithm compared to (i) the optimal offline algorithm for the problem, and (ii) a fully dynamic Datalog solver, arguably the two most relevant approaches to our problem setting. In particular, our dynamic algorithm exhibits speedups that are between 100X-1000X for data-dependence analysis and 1000X for alias analysis (which is also the more demanding of the two), compared to the best alternative. Although its theoretical worst-case complexity is linear in n, its average running time is barely (if at all) noticeable for all practical purposes, making it suitable for the on-the-fly analysis setting that runs continuously during development.

A note on earlier approaches. Dynamic bidirected Dyck reachability was studied recently in [Li et al. 2022]. To this end, that work claims a dynamic algorithm with $O(n \cdot \alpha(n))$ running time per update operation, similarly to our Theorem 1.1. Unfortunately, both the complexity and the correctness of [Li et al. 2022] overlook certain cases. In particular, that algorithm exhibits $\Omega(n^2)$ running time, as opposed to the claimed (nearly) linear bound. This quadratic behavior arises even on sparse graphs (i.e., with $m = O(n)$ edges), for which the vanilla offline algorithm runs in $O(n \cdot \alpha(n))$ time. Hence on such graphs, processing a single line deletion in the source code is n times slower than performing the whole analysis from scratch. Moreover, that algorithm also suffers from correctness issues, as it fails to detect that certain nodes become unreachable after updates. In such cases, the on-the-fly static analysis returns wrong results. We provide further details in our technical report [Krishna et al. 2023].

Intuition behind our approach. In high-level, our approach works as follows. First, we observe that the component graph (i.e., the graph with a single node representing each DSCC) is sparse. Each edge insertion can then be handled by executing the offline algorithm of [Chatterjee et al. 2018] not from scratch, but directly on the already constructed component graph, yielding $O(n \cdot \alpha(n))$ running time. The main difficulty arises in edge deletions, as each deletion seemingly requires repeating all the computation from scratch, i.e., operating on the initial graph, which might be dense and thus incur $O(n^2)$ running time. We leverage techniques from dynamic undirected connectivity, together with some sparsification ideas that are specific to our richer bidirected setting, to start the recomputation not from scratch, but from a suitable preliminary component graph, called the *primary component graph*, that is already sparse, thereby recovering the $O(n \cdot \alpha(n))$ running time. Maintaining the primary component graph efficiently is the main technical challenge.

2 PRELIMINARIES

In this section we develop general notation on labeled graphs, Dyck reachability, and fully-dynamic reachability queries. As this is somewhat standard material, our exposition follows closely that of related works (e.g., [Chatterjee et al. 2018; Kjelstrøm and Pavlogiannis 2022; Zhang et al. 2013]).

2.1 Dyck Reachability on Bidirected Graphs

Dyck Languages. Hereinafter, we fix a natural number $k \in \mathbb{N}$. Let $[k]$ denote $\{1, \dots, k\}$. A Dyck alphabet is a set $\Sigma = \{\alpha_i, \overline{\alpha}_i\}_{i \in [k]}$ of k matched symbols, usually referred to as parentheses, where $\text{Open}(\Sigma) = \{\alpha_i\}_{i \in k}$ and $\text{Close}(\Sigma) = \{\overline{\alpha}_i\}_{i \in k}$ are the sets of opening parenthesis symbols and closing parenthesis symbols of Σ, respectively[2]. The Dyck language $\mathcal{D}$ is the set of strings over Σ^* produced by the following grammar with initial non-terminal $\mathcal{I}$:

$$\mathcal{I} \rightarrow \mathcal{I}\ \mathcal{I} \mid \alpha_1 \mathcal{I} \overline{\alpha}_1 \mid \cdots \mid \alpha_k \mathcal{I} \overline{\alpha}_k \mid \epsilon$$

For example, $\alpha_1\alpha_2\overline{\alpha}_2\alpha_3\overline{\alpha}_3\overline{\alpha}_1 \in \mathcal{D}$, but $\alpha_1\alpha_2\alpha_3\overline{\alpha}_3\overline{\alpha}_1\overline{\alpha}_2 \notin \mathcal{D}$. Finally, given a string $\sigma = \gamma_1 \dots \gamma_m \in \Sigma^*$, we let $\overline{\sigma} = \overline{\gamma}_1 \dots \overline{\gamma}_m \in \Sigma^*$, where $\overline{\gamma}_i$ is a closing parenthesis symbol if γ_i is an opening parenthesis symbol, and vice versa. For example, if $\sigma = \alpha_1\alpha_2\overline{\alpha}_2\alpha_3\overline{\alpha}_3\overline{\alpha}_1$ then $\overline{\sigma} = \overline{\alpha}_1\overline{\alpha}_2\alpha_2\overline{\alpha}_3\alpha_3\alpha_1$.

Graphs and Dyck reachability. We consider labeled directed graphs $G = (V, E)$ where V is a set of nodes and E is a set of labeled edges (u, v, l), where $u, v \in V$ and $l \in \Sigma \cup \{\epsilon\}$. For notational convenience, given an edge e, we will refer to its label as $\lambda(e)$. We occasionally are not interested in the label of an edge, in which case we will simply denote it with its endpoints, e.g., $e = (u, v)$. We also adopt the pictorial notation $u \xrightarrow{l} v$ to denote the existence of an edge (u, v, l), and write $u \xrightarrow{l} \cdot$ to denote the existence of an edge outgoing u and labeled with l. Given a set X, we sometimes write $u \xrightarrow{l} X$ to denote that there exists some $v \in X$ such that $u \xrightarrow{l} v$. We also use similar notation for incoming, instead of outgoing edges (e.g., $X \xrightarrow{l} v$). A path is a sequence of edges $P = (u_1, v_1, l_1), \dots, (u_r, v_r, l_r)$ such that, for each $i \in [r-1]$, we have $u_{i+1} = v_i$. The label of P is $\lambda(P) = l_1 \dots l_r$, that is, it is the concatenation of the labels of its edges. A path can also be an empty sequence of edges, in which case its label is ϵ. We often write $P \colon u \rightsquigarrow v$ to denote a path from node u to node v. Naturally, we say that v is reachable from u if such a path exists. Moreover, we say that v is Dyck-reachable from u if there exists a path $P \colon u \rightsquigarrow v$ such that $\lambda(P) \in \mathcal{D}$, in which case we call P a Dyck path.

In alignment to existing literature ([Chatterjee et al. 2018; Zhang et al. 2013]), we consider that the number of parenthesis types is constant compared to the size of G (i.e., $k = O(1)$). From a theoretical standpoint this is not an oversimplification, as any Dyck graph with arbitrary k can be transformed to one with $k = O(1)$, while preserving the reachability relationships (see e.g., [Chistikov et al. 2022]).

Bidirected Graphs and Dyck SCCs. We call a graph $G = (V, E)$ bidirected if every edge in E appears in both ways with complementary labels. Formally, let $\overline{\epsilon} = \epsilon$, and we have:

$$(u, v, s) \in E \qquad \Longleftrightarrow \qquad (v, u, \overline{s}) \in E$$

Bidirectedness turns reachability into an equivalence relation, similarly to plain undirected graphs. To realize this, notice that every path $P \colon u \rightsquigarrow v$ has a reverse version $\overline{P} \colon v \rightsquigarrow u$ with $\lambda(P) = \lambda(\overline{P})$. Hence, if P witnesses the Dyck reachability of v from u, then $\overline{P}$ witnesses the Dyck reachability of u from v. Naturally, the nodes of G are partitioned into strongly-connected components (SCCs), which are maximal sets of pairwise Dyck-inter-reachable nodes. We will call such sets Dyck strongly connected components, or DSCCs, for short. Note that, in contrast to standard SCCs, paths witnessing the Dyck reachability of two nodes in the same DSCC might have to traverse nodes outside the DSCC. For example, in Figure 1 (top right), the paths witnessing the Dyck reachability of nodes c to d and d to c go via node f, which is outside the DSCC $\{c, d\}$. Given a node u, we let $\text{DSCC}(u)$ be the DSCC in which u appears.

[2]We use Greek letters such as α, β to represent opening parenthesis symbols, and $\overline{\alpha}, \overline{\beta}$ for their matching closing parentheses.

For notational convenience, hereinafter we represent bidirected graphs by only explicitly denoting their edges labeled with closing parentheses, with the understanding that the reverse edges (labeled with an opening parentheses) are also present. In our figures, when a graph contains edges of different labels, we color-code the edges for easier visual representation.

The efficiency of DSCC computation. In Dyck-reachability formulations of static analyses, the analysis queries are phrased as Dyck-reachability queries in the underlying graph. As we have seen above, the nodes of bidirected graphs are partitioned into DSCCs. Thus, reachability queries can be handled by first computing these DSCCs. Then, given a reachability query on two nodes u, v, we answer that they are inter-reachable iff they are found in the same DSCC, which costs a simple lookup of constant time. Computing Dyck reachability on general (non-bidirected) graphs takes cubic time $O(n^3)$ even for a single pair. On the other hand, it is known that the DSCCs of bidirected graphs can be computed efficiently.

Theorem 2.1 ([Chatterjee et al. 2018]). *Given a graph G of n nodes and m edges, the DSCCs of G can be computed in $O(m + n \cdot \alpha(n))$ time, where $\alpha(n)$ is the inverse Ackermann function.*

2.2 Dynamic Reachability

Dynamic reachability on bidirected graphs. As the program source is being developed, the underlying graph model changes shape due to the addition and removal of nodes and edges. The goal of the fully dynamic setting is to maintain Dyck-reachability information on the fly under such changes, so that analysis queries can be performed fast. More formally, we are processing a sequence of operations $\mathcal{S} = (o_1, o_2, \dots, o_r)$, where each operation o_i has one of the following types.

(1) insert$(u, v, \overline{\alpha})$, which inserts an edge $u \xrightarrow{\overline{\alpha}} v$ (as well as the implicit inverse edge $v \xrightarrow{\alpha} u$).
(2) delete$(u, v, \overline{\alpha})$, which deletes the edge $u \xrightarrow{\overline{\alpha}} v$ (as well as the implicit inverse edge $v \xrightarrow{\alpha} u$).
(3) DSCCRepr(u), which returns a representative node of the DSCC of u that is the same for all nodes $v \in \mathrm{DSCC}(u)$.

This sequence of operations creates a sequence of graphs $(G_i = (V, E_i))_{i \in [r]}$, where G_1 is the initial graph, and G_{i+1} is obtained from G_i by performing the operation o_i (naturally, if $o_i = \mathrm{DSCCRepr}(u)$ is a query operation, $G_{i+1} = G_i$). Note that we have kept the node set identical across all G_i. Indeed, this is a standard approach for studying such problems: we may simply assume that V contains all nodes that are ever added to the graph, and instead of deleting a node, we can delete all its adjacent edges. Finally, the reachability of v from u can be checked with two successive queries, i.e., testing whether $\mathrm{DSCCRepr}(u) = \mathrm{DSCCRepr}(v)$.

Although in general there might be a spectrum of trade offs in the time taken between operations modifying the graph (insert$(u, v, \overline{\alpha})$, delete$(u, v, \overline{\alpha})$) and reachability queries (DSCCRepr(u)), the linear upper bounds we establish here for updates means that query operations can be done in constant time $O(1)$. This holds because, in theory, after every update operation we can easily, in $O(n)$ time (which is within the upper bound we establish for processing the update operations), retrieve the DSCC of each node to a simple lookup table that will lead to subsequent queries taking $O(1)$ time each. In practice, of course, this approach is to be avoided, as dynamic updates cost much less than $O(n)$ time, hence building the full lookup table causes an unnecessary cost. Moreover, even without a lookup table, queries cost $O(\alpha(n))$ time, which is constant for all practical purposes.

Dynamic reachability on undirected graphs. Problems of dynamic reachability have been studied extensively on plain graphs (i.e., without the restriction on Dyck paths), both for directed

Table 1. Classic results on dynamic undirected reachability.

Reference	Update time	Query time	Guarantee
[Eppstein et al. 1997]	$O(\sqrt{n})$	$O(1)$	Worst-case
[Holm et al. 2001]	$O(\log^2 n)$	$O(\log n)$	Amortized

and undirected variants. However, the richer semantics of Dyck reachability makes the Dyck setting quite more intricate. Our solution to the problem uses as a black box a data structure for dynamic reachability on undirected graphs. Although there exist several results in this direction, Table 1 shows two standard data structures that are relevant to our work. The one developed in [Eppstein et al. 1997] offers standard worst-case guarantees for the time to perform each update (i.e., inserting/deleting an undirected edge) and query (i.e., obtaining the component of a node u). The one developed in [Holm et al. 2001] offers amortized guarantees for the respective tasks. That is, some operations might exceed the stated time bounds, but this excess is balanced in later operations, so that the average time per operation is the one stated. As we will see later, we will be using the former technique to establish our bound in Theorem 1.1, but the latter in our experiments.

3 EFFICIENT DYNAMIC BIDIRECTED DYCK REACHABILITY

In this section we present our dynamic algorithm for bidirected Dyck reachability. As this is the main technical section of our paper, we provide here an outline of its structure to guide the reader.

(1) In Section 3.1 we present the key algorithmic intuition for solving Dyck reachability on bidirected graphs, and revisit the offline algorithm for the problem, as presented in [Chatterjee et al. 2018]. Some aspects of the offline algorithm (in particular, its fixpoint computation) will also be used later in our dynamic algorithm.
(2) In Section 3.2 we present the main technical challenges for handling dynamic updates efficiently, and the key concepts our algorithm uses for handling each challenge. This section is filled with examples that illustrate how each concept is used, and what data structure is used to support it.
(3) In Section 3.3 we present DynamicAlgo in detail, with pseudocode followed by a high-level description of each of its blocks. The pseudocode is also heavy in comments to guide the reader.
(4) In Section 3.4 we give a step-by-step execution of the algorithm on the running example of Figure 1.
(5) In Section 3.5, we establish the correctness and complexity of DynamicAlgo. We state its main invariants, together with some intuition around them and how they are used towards the correctness and complexity lemmas, while we delegate most proofs to [Krishna et al. 2023].

3.1 Offline Reachability

We start with the optimal offline algorithm for computing bidirected Dyck reachability [Chatterjee et al. 2018], i.e., for obtaining Theorem 2.1. This will allow us to build some intuition about the problem. Moreover, our data structure for handling graph updates later will use some components of the offline algorithm.

Intuitive description. The principle of operation of the algorithm behind Theorem 2.1 (initially observed in [Zhang et al. 2013]) is as follows. Assume that we have computed a DSCC S_1 that has two nodes $u, v \in S_1$, and further, G has two edges $u \xrightarrow{\overline{\alpha}} x$ and $v \xrightarrow{\overline{\alpha}} y$. Here u and v are not necessarily distinct, meaning that possibly $u = v$. Then we can conclude that x and y also belong to

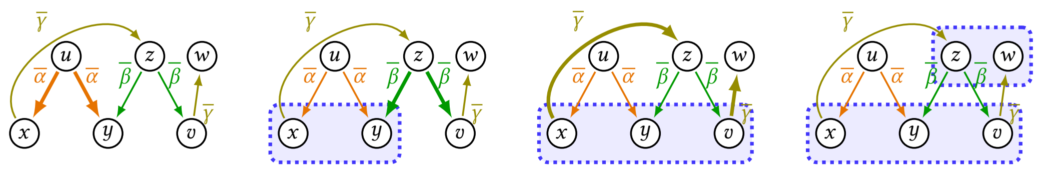


Fig. 2. Illustration of the fixpoint computation of DSCCs. Initially, every node forms its own DSCC. We have $u \xrightarrow{\overline{\alpha}} x$ and $u \xrightarrow{\overline{\alpha}} y$, forming a DSCC $S_1 = \{x, y\}$, witnessed by the paths $x \xrightarrow{\alpha} u \xrightarrow{\overline{\alpha}} y$ and $y \xrightarrow{\alpha} u \xrightarrow{\overline{\alpha}} x$. In turn, the two edges $z \xrightarrow{\overline{\beta}} y$ and $z \xrightarrow{\overline{\beta}} v$ enlarge this DSCC to $S_1 = \{x, y, v\}$. Finally, the two edges $x \xrightarrow{\overline{\gamma}} z$ and $v \xrightarrow{\overline{\gamma}} w$ form another DSCC $S_2 = \{z, w\}$.

the same DSCC, witnessed by the path

$$x \xrightarrow{\alpha} u \rightsquigarrow v \xrightarrow{\overline{\alpha}} y$$

where the intermediate path $u \rightsquigarrow v$ is one witnessing the reachability of v from u – such a path exists since u and v belong to the same DSCC. In fact, any node x' in the DSCC of x can reach any node y' in the DSCC of y, via the path

$$x' \rightsquigarrow x \xrightarrow{\alpha} u \rightsquigarrow v \xrightarrow{\overline{\alpha}} y \rightsquigarrow y'$$

Thus the DSCCs of x and y can be merged into one DSCC S_2. We can now repeat the above process to discover Dyck paths that go through S_2, and so on, up to a fixpoint. Figure 2 illustrates this process on a small example.

Algorithm OfflineAlgo. The above insights are made formal in Algorithm 1, which is an adaptation of the algorithm as appeared initially in [Chatterjee et al. 2018]. The first insight that OfflineAlgo rests on is that, since DSCCs are equivalence classes that only merge together during the fixpoint computation, they can be maintained efficiently using a disjoint-sets data structure that supports Union/Find operations. The second insight is to use linked lists for storing the edges outgoing each DSCC and labeled with a given symbol $\overline{\alpha}$. When two DSCCs are merged into one, the algorithm also merges the corresponding linked lists, which can be done very efficiently, in $O(1)$ time (using simple pointers). In particular, OfflineAlgo performs two big steps. First, it initializes a disjoint-sets data structure DisjointSets and a worklist Q (function Initialization() in Line 1). Then, it uses DisjointSets and Q to compute the fixpoint (function Fixpoint() in Line 2), achieving the running time of $O(m + n \cdot \alpha(n))$. As the correctness and complexity are established in [Chatterjee et al. 2018], we do not re-establish them here.

3.2 Main Concepts and Intuition

Having outlined the basic algorithm for offline reachability, we now turn our attention to the data structure for dynamic reachability. In high level, our data structure will use the same components as OfflineAlgo, in particular, the representation of DSCCs using the DisjointSets data structure, the Edges linked lists for storing the outgoing edges of each component, and the Fixpoint() function for computing the DSCCs as a fixpoint after every graph update. However, instead of starting the computation from scratch, DisjointSets and Edges (as well as the worklist Q of Fixpoint()) will be at a state that represent some DSCCs that are guaranteed to exist in the updated graph, and thus need not be recomputed. The key technical challenge we have to solve is in computing a non-trivial such state, and doing so efficiently.

Algorithm 1: OfflineAlgo

```
Input: A bidirected graph $G = (V, E)$
Output: A DisjointSets map of the DSCCs of $G$
1  Initialization()                                   // Initialize the fixpoint computation
2  Fixpoint()                                         // Compute the fixpoint
3  function Initialization()
4      $Q \leftarrow$ an empty queue over $V^*$             // A worklist over edges for computing the fixpoint
5      Edges $\leftarrow$ a map $V \times \mathrm{Close}(\Sigma) \longrightarrow V^*$ as a linked list   // The outgoing edges of each node
6      DisjointSets $\leftarrow$ a disjoint-sets data structure over $V$
7      foreach $u \in V$ do
8          DisjointSets.MakeSet($u$)                   // constructs the singleton set $\{u\}$
9          foreach label $\overline{\alpha} \in \mathrm{Close}(\Sigma)$ do
10             Edges$[u][\overline{\alpha}] \leftarrow (v : (u, v, \overline{\alpha}) \in E)$
11             if $|$Edges$[u][\overline{\alpha}]| \geq 2$ then Insert $(u, \overline{\alpha})$ in $Q$   // The two edges must merge their endpoints
12 function Fixpoint()
13     while $Q$ is not empty do                       // Repeat until fixpoint
14         Extract $(u, \overline{\alpha})$ from $Q$      // We merge components connected via $u$'s component
15         if $u =$ DisjointSets.Find($u$) then         // $u$ is the root of its component
16             $S \leftarrow \{$DisjointSets.Find($w$) : $w \in$ Edges$[u][\overline{\alpha}]\}$   // The components to be merged
17             if $|S| \geq 2$ then
18                 $x \leftarrow$ some arbitrary element of $S \setminus \{u\}$   // $x$ will be the root of the merged component
19                 Make DisjointSets.Union($S, x$)          // All components merge to $x$'s component
20                 foreach label $\overline{\beta} \in \mathrm{Close}(\Sigma)$ do   // Process each label separately
21                     foreach $v \in S \setminus \{x\}$ do
22                         if $u \neq v$ or $\overline{\alpha} \neq \overline{\beta}$ then   // $(v, \overline{\beta})$ is not the $(u, \overline{\alpha})$ we extracted from $Q$
23                             Move Edges$[v][\overline{\beta}]$ to Edges$[x][\overline{\beta}]$   // In $O(1)$ time, using pointers
24                         else
25                             Append $(x)$ to Edges$[x][\overline{\beta}]$   // All Edges$[u][\overline{\alpha}]$ merged to a self-loop $x \xrightarrow{\overline{\alpha}} x$
26                     if $|$Edges$[x][\overline{\beta}]| \geq 2$ then Insert $(x, \overline{\beta})$ in $Q$   // More components to be merged
27             else
28                 Let $x \leftarrow$ the single node in $S$
29             if $u \notin S$ or $|S| = 1$ then Edges$[u][\overline{\alpha}] \leftarrow (x)$   // $u$'s component now points to $x$'s component
```

In this section we introduce the main concepts of our dynamic algorithm and provide the necessary intuition around them. The precise algorithm is presented in the next section.

High-level description of the dynamic algorithm. The main components that are new to our dynamic algorithm compared to the offline algorithm of Section 3.1 are geared towards handling edge deletions. Assume that we have computed the DSCCs of a graph G, and we now have to process an operation delete$(u, v, \overline{\alpha})$. This will, in general, result in splitting some DSCCs into smaller ones. In high level, we handle such edge deletion in three steps.

(1) We compute a sound overapproximation of the DSCCs that are affected by the edge deletion, i.e., those that might have to be split into smaller components. We compute this overapproximation by effectively performing a forward search starting from DSCC(v) and repeatedly proceeding to neighboring DSCCs of G. In particular given a current DSCC S, we proceed to those DSCCs

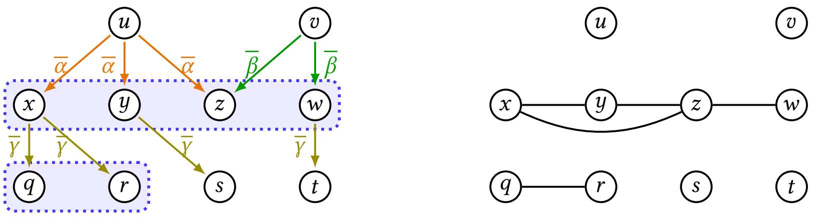


Fig. 3. A bidirected graph G (left) and the corresponding primal graph H (right). There are two non-singleton PDSCCs in G (marked), corresponding to non-singleton connected subgraphs of H.

S' that have at least two incoming edges of the form $x \xrightarrow{\overline{\beta}} y$, with $x \in S$ and $y \in S'$, and some label $\overline{\beta}$. This is because S' might have been formed through S and the corresponding edges $x \xrightarrow{\overline{\beta}} y$. Although this traversal might end up touching $\Theta(n)$ DSCCs, we expect that in practice it will perform much better, as the effect of an average edge deletion is usually very local.

(2) At this point, it would suffice to split all nodes t in the previously discovered DSCCs S' into singleton components, gather in the worklist $\mathcal{Q}$ (see Algorithm 1) all pairs $(s, \overline{\beta})$ corresponding to incoming edges $s \xrightarrow{\overline{\beta}} t$ where t is the unique node of such a singleton component, and restart the fixpoint computation using Fixpoint(). Unfortunately, this would require $\Theta(n^2)$ time, which is beyond our target bound, as there can be $\Theta(n^2)$ such edges $s \xrightarrow{\overline{\beta}} t$, and Line 16 of Algorithm 1 would iterate over all of them (where $u = s$ and $w = t$ in the algorithm). To circumvent this difficulty, we introduce the novel notion of *primary DSCCs* (or *PDSCCs*), which, intuitively, are connected components of small size. We keep track of the formation of PDSCCs dynamically by leveraging techniques from undirected dynamic reachability (see e.g., Table 1). Now, instead of splitting each previously discovered DSCC S' into *singleton components*, we split it into its *PDSCCs*. This allows us to re-initiate the fixpoint computation of the function Fixpoint() after only traversing $O(n)$ edges of the form $s \xrightarrow{\overline{\beta}} t$ (where t now is a node in a PDSCC). In practice and due to the previous step, we typically traverse much fewer edges than n. In turn, this implies that Fixpoint() will converge after $O(n)$ iterations, leading to the desired time bound.

(3) We execute the fixpoint computation, similarly to Algorithm 1.

We now describe the above concepts in detail.

Primal graphs and primary DSCCs. Consider a bidirected graph $G = (V, E)$. The *primal* graph $H = (V, L)$ is an unlabeled, undirected graph with the same node set, and edge set defined as:

$$L = \{(x, y) : \exists u \in V.\ \exists \overline{\alpha} \in \Sigma^C.\ u \xrightarrow{\overline{\alpha}} x, u \xrightarrow{\overline{\alpha}} y \in E\}$$

In words, x and y share an edge in H if they are connected in G via a Dyck path of length 2. A *primary DSCC (PDSCC)* of G is a (maximal) connected component of the primal graph H. It is not hard to see that the PDSCC partitioning of G is a refinement of its DSCC partitioning, i.e., each DSCC contains one or more PDSCCs. See Figure 3 for an illustration. Similarly to the case of DSCCs, given a node u, we let PDSCC(u) be the PDSCC in which u belongs, while PDSCCRepr(u) returns a representative of PDSCC(u) that is common for all nodes $v \in$ PDSCC(u). Hence, two nodes u, w are in the same PDSCC iff PDSCCRepr(u) = PDSCCRepr(v).

Since the PDSCCs of G have a direct representation as connected components in the undirected graph H, we can leverage existing techniques from dynamic undirected connectivity (see, e.g.,

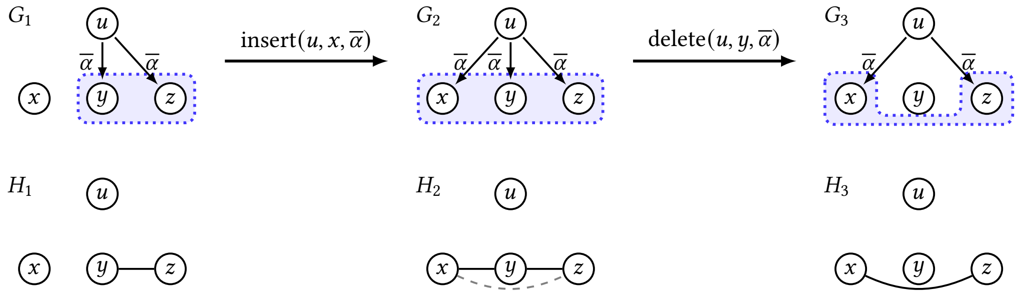

G_1
G_2
G_3
H_1
H_2
H_3
insert$(u, x, \overline{\alpha})$
delete$(u, y, \overline{\alpha})$
u
x
y
z
$\overline{\alpha}$

Fig. 4. Sparsification for the maintenance of the PDSCCs of G_i across edge insertions and deletions (top), by maintaining connected components in the corresponding primal graphs H_i (bottom).

Table 1) to maintain them efficiently. PDSCCs serve the following function. When an edge is deleted, some DSCCs S of G have to be split to smaller DSCCs. This can lead to effectively repeating the fixpoint computation from scratch in G. However, this approach would require $\Theta(n^2)$ running time, which is beyond our complexity bound. Instead, maintaining the PDSCCs allows us to split S to its PDSCCs (as opposed to individual nodes), and thus avoid recomputing the PDSCCs from scratch. In turn, this allows us to achieve the desired $O(n \cdot \alpha(n))$ bound for edge deletions.

A sparsification approach for maintaining the PDSCCs. Since each PDSCC corresponds to a connected component of the undirected primal graph H, we will maintain PDSCCs dynamically, by using any data structure for dynamic reachability on undirected graphs (see Table 1). We call this data structure for undirected reachability PrimCompDS.

An operation $o = \text{insert}(u, v, \overline{\alpha})$ inserts between 0 and $n - 1$ undirected edges in the primal graph H. Indeed, we have an edge (x, v) for every node $x \neq v$ for which already $u \xrightarrow{\overline{\alpha}} x$, and there may be $n - 1$ such nodes x. Note, however, that if there already exist two such nodes, x, y, with $u \xrightarrow{\overline{\alpha}} x$ and $u \xrightarrow{\overline{\alpha}} y$, then x and y already appear in the same PDSCC of G. Hence, we can faithfully maintain the PDSCCs of G by only adding *one* of the two primal edges (v, x) and (v, y) in PrimCompDS. This has the desirable effect of maintaining PDSCCs after edge insertions by inserting $O(1)$ undirected edges in PrimCompDS, as opposed to $O(n)$. This kind of technique is known as *sparsification* [Eppstein et al. 1997], in that we manage to represent the full connectivity of the primal graph H while only storing a subset of its edges. In particular, we achieve this effect by maintaining the outgoing edges of u labeled with $\overline{\alpha}$ in a linked list OutEdges$[u][\overline{\alpha}] = (x_1, \dots, x_r)$, and only inserting in PrimCompDS edges (x_i, x_{i+1}), i.e., between consecutive nodes in OutEdges$[u][\overline{\alpha}]$. Distant nodes become connected transitively in PrimCompDS.

Conversely, an operation $o = \text{delete}(u, v, \overline{\alpha})$ deletes between 0 and $n - 1$ undirected edges in the primal graph H, with reasoning similar to the one above. However, because of the above invariant, we can restore the PDSCCs of G by removing the edges (a, v) and (v, b) in PrimCompDS, where a and b are the neighbors of v in OutEdges$[u][\overline{\alpha}]$. Moreover, as a and b are no longer connected in PrimCompDS, we restore their connectivity by inserting the edge (a, b) in PrimCompDS. Figure 4 illustrates this sparsification technique on a small example. The update insert$(u, x, \overline{\alpha})$ creates two edges in the primal graph H_2, namely (x, y) and (x, z). However, the dashed edge (x, z) is not stored explicitly in PrimCompDS, as x and y are already connected via z. The update delete$(u, y, \overline{\alpha})$ removes the edges (x, y) and (y, z) from the primal graph H_3. To preserve the connectivity of x and z, the dashed edge (x, z) is restored in PrimCompDS.

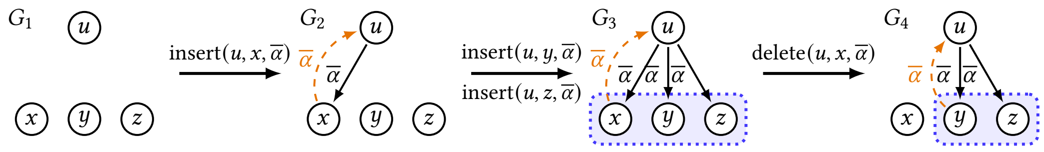

G_1
G_2
G_3
G_4
insert$(u, x, \overline{\alpha})$
insert$(u, y, \overline{\alpha})$
insert$(u, z, \overline{\alpha})$
delete$(u, x, \overline{\alpha})$

Fig. 5. Maintenance of the sets InPrimary (shown in orange dashed) along edge insertions and deletions. The first edge insertion $u \xrightarrow{\overline{\alpha}} x$ leads to $u \in$ InPrimary$[x][\overline{\alpha}]$. The following two edge insertions $u \xrightarrow{\overline{\alpha}} y$ and $u \xrightarrow{\overline{\alpha}} z$ do not modify InPrimary, as x, y and z belong to the same PDSCC, thus u can be retrieved as a $\overline{\alpha}$-neighbor via the set InPrimary$[x][\overline{\alpha}]$. When processing deleting the edge $u \xrightarrow{\overline{\alpha}} x$, we move u to InPrimary$[y][\overline{\alpha}]$, thus u can still be retrieved as a $\overline{\alpha}$-neighbor of the PDSCC $\{y, z\}$.

Maintaining the neighbors of PDSCCs. The maintenance of the PDSCCs allows us to efficiently identify the incoming neighbors of a PDSCC, in $O(n)$ worst-case time, although the graph might have $\Theta(n^2)$ edges. This fact stems from the following observation. For a given node u and label $\overline{\alpha}$, if there are two edges $u \xrightarrow{\overline{\alpha}} x$ and $u \xrightarrow{\overline{\alpha}} y$, then x and y are in the same PDSCC. In other words, for every such u and $\overline{\alpha}$, there is at most one PDSCC that u is a neighbor of via a $\overline{\alpha}$-labeled edge. Thus, given a set of PDSCCs (as described in Item (2) above), we can obtain all edges incoming to them in $O(n)$ time, by iterating over all nodes u and labels $\overline{\alpha}$. Though this is sufficient towards our linear time bound, it can become unnecessarily slow when the number of such PDSCCs is small. Instead, we follow a different approach, which retains the $O(n)$ worst-case behavior but is faster in practice.

For every node x and label $\overline{\alpha}$, we maintain a set InPrimary$[x][\overline{\alpha}]$, which stores nodes u that have an edge $u \xrightarrow{\overline{\alpha}} x$. We maintain the invariant that x will be the *only* node in PDSCC(x) with $u \in$ InPrimary$[x][\overline{\alpha}]$, even though there might be other nodes $y \in$ PDSCC(x) also having an edge $u \xrightarrow{\overline{\alpha}} y$. To identify the neighbors of each PDSCC, it now suffices to iterate over its nodes x, and for each label $\overline{\alpha}$, collect the nodes from InPrimary$[x][\overline{\alpha}]$. The invariant guarantees that every neighbor u will be accessed exactly once per label $\overline{\alpha}$, retaining the $O(n)$ worst-case running time.

The sets InPrimary$[x][\overline{\alpha}]$, together with their invariant, can be maintained as follows. Upon inserting an edge insert$(u, x, \overline{\alpha})$, if u does not have any other outgoing edges labeled with $\overline{\alpha}$, we insert u in InPrimary$[x][\overline{\alpha}]$. Otherwise, there exists another edge $u \xrightarrow{\overline{\alpha}} y$, with already $u \in$ InPrimary$[y][\overline{\alpha}]$. The existence of the two edges $u \xrightarrow{\overline{\alpha}} x$ and $u \xrightarrow{\overline{\alpha}} y$ implies that PDSCC(x) = PDSCC(y), hence it is sound to not insert u in InPrimary$[x][\overline{\alpha}]$, as u is retrievable via y. Similarly, upon deleting an edge delete$(u, x, \overline{\alpha})$, if $u \in$ InPrimary$[x][\overline{\alpha}]$, we move u to another set InPrimary$[y][\overline{\alpha}]$ for which there exists an edge $u \xrightarrow{\overline{\alpha}} y$. Figure 5 illustrates the maintenance of InPrimary on a small example.

Processing a delete operation. We are now in position to outline our approach for processing a delete operation delete$(u, v, \overline{\alpha})$. As the edge $u \xrightarrow{\overline{\alpha}} v$ might be used to connect v to other nodes in DSCC(v), this operation might split DSCC(v) into smaller DSCCs. As nodes become disconnected in DSCC(v), other DSCCs might also be split, if the paths connecting their nodes passed through DSCC(v), resulting in a cascading effect. For instance, consider Figure 1 (middle), having two DSCCs $\{c, d, e, f\}$ and $\{g, h\}$. On deleting the edge $f \xrightarrow{\overline{L}} d$, the node d splits from the DSCC $\{c, d, e, f\}$. In turn, the DSCC $\{g, h\}$ splits into $\{g\}$, $\{h\}$ since they were held together by c, d being in the same DSCC. The split of $\{g, h\}$ results in pulling out f from DSCC(e). Thus, deleting the edge $f \xrightarrow{\overline{L}} d$ results in splitting DSCCs $\{c, d, e, f\}$ and $\{g, h\}$ resulting in DSCCs $\{c, e\}$, $\{d\}$, $\{f\}$, $\{g\}$, $\{h\}$.

Thus, a single edge deletion may have an effect which propagates to the whole graph; however, we can obtain a sound overapproximation of the DSCCs affected by the initial edge deletion using the following observation. If the split of a DSCC S leads to the immediate split of another DSCC S', then there exist two distinct edges $x_i \xrightarrow{\overline{\beta}} y_i$, for $i \in [2]$, such that $x_i \in S$ and $y_i \in S'$. Thus, for delete$(u, v, \overline{\alpha})$, we can obtain our overapproximation by starting a forward search from DSCC(v): given a current DSCC S, we iterate over all labels $\overline{\beta}$ such that S has at least two $\overline{\beta}$-labeled outgoing edges, and proceed to the resulting DSCC S' (note that there can be at most one such S').

After the overapproximation of the set of potentially splitting DSCCs has been performed, each such DSCC S is split to its constituent PDSCCs, using the undirected connectivity data structure PrimCompDS. Note that this splitting is an underapproximation of the set of the final DSCCs, as some PDSCCs have to be merged again. We will discover this by running the fixpoint computation again, up to convergence. For this, we re-insert in the worklist Q (see function Fixpoint() in Algorithm 1) node-label pairs (DSCCRepr$(y), \overline{\beta}$) that represent edges $y \xrightarrow{\overline{\beta}} \cdot$ that might lead to further component merging. Due to the efficient maintenance of neighbors of PDSCCs (see the previous paragraph), this can be achieved by iterating over all nodes x of the PDSCCs, and for each label $\overline{\beta} \in \Sigma^C$, insert in Q, the pair (DSCCRepr$(w), \overline{\beta}$) corresponding to $w \in$ InPrimary$[x][\overline{\beta}]$ if $|$Edges[DSCCRepr$(w)][\overline{\beta}]| \geq 2$. Finally we continue with the fixpoint computation as in OfflineAlgo with the guarantee that upon convergence, DisjointSets will contain the DSCCs of the new graph after deleting the edge $u \xrightarrow{\overline{\alpha}} v$.

Example. Figure 6 illustrates the above process on a small example, processing an update delete$(c, d, \overline{\gamma})$. Before the deletion (top), the graph has four DSCCs, namely S_1, S_2, S_3, S_4 (marked). In the first step, we perform a forward search from S_2, following pairs of same-label edges, which discovers S_2, S_3 and S_4 as the set of DSCCs that are potentially affected by the delete operation. Observe that S_1 does not have to be split, and the algorithm avoids recomputation on S_1. Then, S_2, S_3 and S_4 are split to their constituent PDSCCs (middle). Observe that S_4 is only partly split, as its subset $\{u, v, y\}$ is a PDSCC (with y = PDSCCRepr(u)) in the new graph (after deletion). Hence the algorithm avoids recomputing S_4 from scratch. Afterwards, the algorithm uses the InPrimary data structure to collect the set of pairs $\mathcal{L} = \{(g, \overline{\gamma}), (h, \overline{\gamma}), (c, \overline{\gamma}), (d, \overline{\alpha}), (e, \overline{\alpha}), (d, \overline{\beta}), (y, \overline{\alpha}), (y, \overline{\beta}), (y, \overline{\gamma}), (f, \overline{\beta})\}$, i.e., the pairs (DSCCRepr$(x), \overline{\beta}$) such that DSCC$(x) \xrightarrow{\overline{\beta}}$ PDSCC(r), where r is a node in the newly formed PDSCCs. Observe that the pair $(f, \overline{\gamma})$ is not collected in $\mathcal{L}$ even though $f \xrightarrow{\overline{\gamma}} c$, as this edge is incoming to DSCC S_1 which was deemed as not affected by the edge deletion. Note that some of these PDSCCs can be merged during the Fixpoint computation to obtain the new set of DSCCs post deletion. For this, a pair $(x, \overline{\alpha}) \in \mathcal{L}$ is added to Q, if there are two outgoing edges on $\overline{\alpha}$ from x's component. In this example, only $(y, \overline{\gamma})$ is added in Q since there are two edges on $\overline{\gamma}$ from y's component, namely $u \xrightarrow{\overline{\gamma}} e$ and $v \xrightarrow{\overline{\gamma}} f$. Fixpoint takes this Q and converges to the final DSCCs (bottom), merging the PDSCCs $\{e\}$ and $\{f\}$, thereby partially restoring S_2.

3.3 The Dynamic Algorithm

We now make the concepts of the previous section formal, by developing precise algorithms for handling each insert and delete operation. In a static analysis context, the edges of the underlying graph correspond to statements in the analyzed source code. This means that a certain edge might be inserted several times in the graph, if, for example, it comes from a repeating program statement. Note, however, that only the presence of the edge impacts the reachability analysis and not its

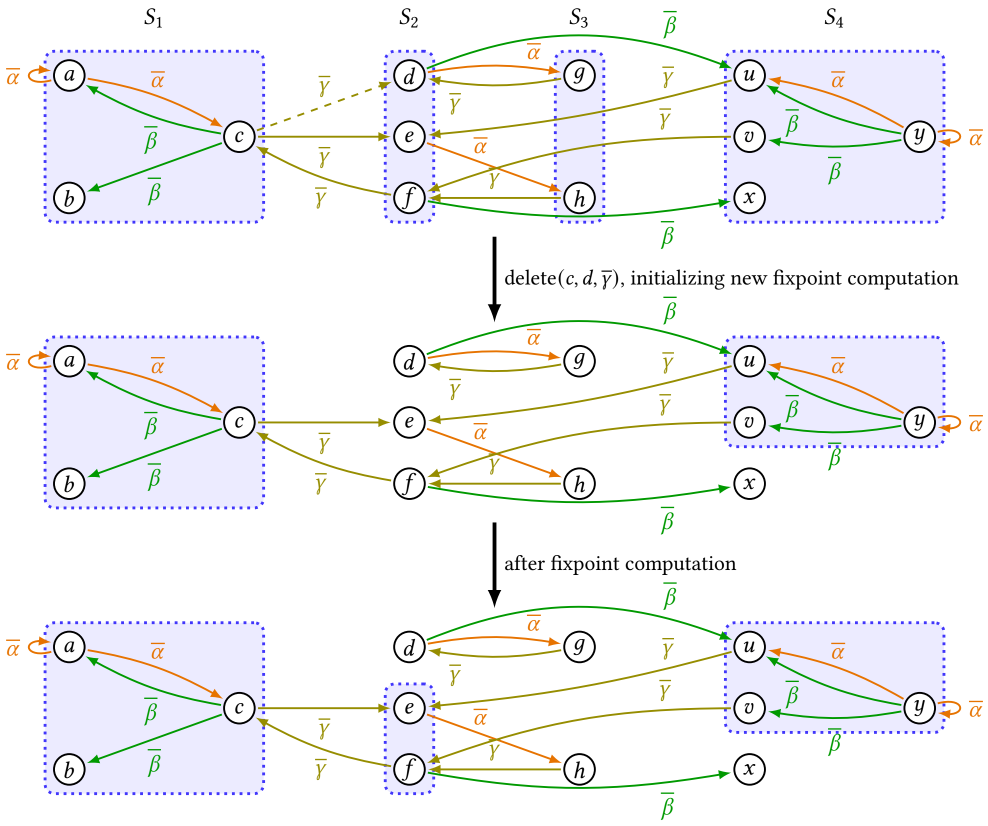

S_1
S_2
S_3
S_4
delete$(c, d, \overline{\gamma})$, initializing new fixpoint computation
after fixpoint computation

Fig. 6. Illustration of handling delete$(c, d, \overline{\gamma})$, showing the state of the algorithm before deletion (top), after preparing the new fixpoint computation (middle) and after completing the fixpoint computation (bottom).

multiplicity. To handle multiple edges, we maintain simple counters in a list data structure Count, so that Count$[u][v][\overline{\alpha}]$ holds the number of $u \xrightarrow{\overline{\alpha}} v$ edges currently in the graph.

Operation insert$(u, v, \overline{\alpha})$. Algorithm 2 handles each insert$(u, v, \overline{\alpha})$ operation. In words, the algorithm first implements the logic for sparsely maintaining the PDSCCs (Line 3 to Line 8). In particular, if v is the first $\overline{\alpha}$-neighbor of u, then u is marked as a $\overline{\alpha}$-neighbor of PDSCC(v) via v, by inserting u in InPrimary$[v][\overline{\alpha}]$ (Line 4). Otherwise, u is already marked via some existing $\overline{\alpha}$-neighbor y of u, which is also in PDSCC(v). Then the algorithm inserts an edge (v, y) in the PrimCompDS data structure (which maintains undirected connectivity) to reflect the change in v's PDSCC.

The second step of Algorithm 2 (Line 9 to Line 13) prepares the fixpoint computation that might be triggered due to the new edge. For this, it marks that v is a new $\overline{\alpha}$-neighbor of DSCC(u) (Line 9, as the DSCC is represented by its root node x). If there already exists another $\overline{\alpha}$-labeled edge out of DSCC(u), then the pair $(x, \overline{\alpha})$ is inserted in Q, and the Fixpoint() is called to continue the fixpoint computation for merging the DSCCs (in DisjointSets) (note that Fixpoint() is also used by OfflineAlgo and defined in Algorithm 1).

Algorithm 2: insert$(u, v, \overline{\alpha})$

```
1  $\text{Count}[u][v][\overline{\alpha}] \leftarrow \text{Count}[u][v][\overline{\alpha}] + 1$    // Update the edge counter
2  if $\text{Count}[u][v][\overline{\alpha}] \geq 2$ then return    // Edge exists already, no change in reachability
3  if $|\text{OutEdges}[u][\overline{\alpha}]| = 0$ then    // $v$ is the first $\overline{\alpha}$-neighbor of $u$
4      Insert $u$ in $\text{InPrimary}[v][\overline{\alpha}]$ // The fact that $u \xrightarrow{\overline{\alpha}} \text{PDSCC}(v)$ is retrievable via $\text{InPrimary}[v][\overline{\alpha}]$
5  else
6      Let $y \leftarrow$ the head of $\text{OutEdges}[u][\overline{\alpha}]$    // The most recent $\overline{\alpha}$-neighbor of $u$
7      PrimCompDS. insert$(v, y)$    // Update the PDSCCs with the edge $(v, y)$
8  Insert $v$ as the head of $\text{OutEdges}[u][\overline{\alpha}]$    // $v$ is a node in the PDSCC formed by $u \xrightarrow{\overline{\alpha}} \cdot$ edges
9  $x \leftarrow$ DisjointSets. Find$(u)$    // $x$ is the root of the component of $u$
10 Insert $v$ in $\text{Edges}[x][\overline{\alpha}]$    // The component of $u$ has an extra outgoing edge
11 if $|\text{Edges}[x][\overline{\alpha}]| \geq 2$ then    // Potentially some component merging occurs, compute new fixpoint
12     Insert $(x, \overline{\alpha})$ in $Q$    // Prepare the new fixpoint computation
13     Fixpoint()    // Compute the new fixpoint due to the new edge
```

Algorithm 3: delete$(u, v, \overline{\alpha})$

```
1  $\text{Count}[u][v]\overline{\alpha}] \leftarrow \text{Count}[u][v][\overline{\alpha}] - 1$    // Update the edge counter
2  if $\text{Count}[u][v][\overline{\alpha}] \geq 1$ then return    // Edge still exists, no change in reachability
3  if $v$ is the tail of $\text{OutEdges}[u][\overline{\alpha}]$ then    // $v$ was the earliest node to have an edge $u \xrightarrow{\overline{\alpha}} \cdot$
4      Remove $u$ from $\text{InPrimary}[v][\overline{\alpha}]$
5      if $|\text{OutEdges}[u][\overline{\alpha}]| \geq 2$ then    // There is another $u \xrightarrow{\overline{\alpha}} \cdot$ edge
6          Let $w \leftarrow$ the penultimate node in $\text{OutEdges}[u][\overline{\alpha}]$
7          Insert $u$ in $\text{InPrimary}[w][\overline{\alpha}]$    // $u \xrightarrow{\overline{\alpha}} \text{PDSCC}(w)$ is now retrievable via $\text{InPrimary}[w][\overline{\alpha}]$
8  Let $a, b \leftarrow \perp, \perp$
9  if $v$ is neither the head nor the tail of $\text{OutEdges}[u][\overline{\alpha}]$ then
10     Let $a, b \leftarrow$ the two neighbors of $v$ in $\text{OutEdges}[u][\overline{\alpha}]$    // We will have to reconnect $a$ and $b$
11 foreach neighbor $x$ of $v$ in $\text{OutEdges}[u][\overline{\alpha}]$ do    // At most two neighbors
12     PrimCompDS. delete$(v, x)$    // The primal graph loses the edge $(v, x)$
13 Remove $v$ from $\text{OutEdges}[u][\overline{\alpha}]$    // $v$ no longer has undirected edges due to $\overline{\alpha}$ via $u$
14 if $a \neq \perp$ and $b \neq \perp$ then    // $v$ had two neighbors in $\text{OutEdges}[u][\overline{\alpha}]$
15     PrimCompDS. insert$(a, b)$    // $a$ and $b$ no longer connected via $v$, connect them directly
16 MakePrimary()    // Prepare the state for the dynamic fixpoint
17 Fixpoint()    // Compute the fixpoint from the current state
```

Operation delete$(u, v, \overline{\alpha})$. Algorithm 3 handles each delete$(u, v, \overline{\alpha})$ operation. Similarly to Algorithm 2, Algorithm 3 first implements the logic for sparsely maintaining the PDSCCs (Line 3 to Line 15). In particular, if v is the tail of $\text{OutEdges}[u][\overline{\alpha}]$ then u is currently retrievable as a $\overline{\alpha}$-neighbor of PDSCC(v) via v. Since u will no longer be a $\overline{\alpha}$-neighbor of v, the algorithm removes u from $\text{InPrimary}[v][\overline{\alpha}]$, and makes u retrievable via w (Line 7), which is the new last $\overline{\alpha}$-neighbor of u (after removing v in Line 6). Then, the algorithm deletes the edge (a, v) and (b, v) in the PrimCompDS data structure, where a and b are the neighbors of v in $\text{OutEdges}[u][\overline{\alpha}]$ to reflect the change in PDSCC(v) (Line 11). After removing v from $\text{OutEdges}[u][\overline{\alpha}]$ (Line 13), a and b become neighboring nodes in $\text{OutEdges}[u][\overline{\alpha}]$, and thus the algorithm inserts a new edge (a, b) in PrimCompDS, to reflect the fact that a and b are now directly connected in the sparse representation of PDSCCs (Line 15). As the edge deletion might lead to the splitting of some DSCCs, the second step of Algorithm 3 splits some of the previously computed DSCCs and re-initiates the fixpoint computation (Line 16 and Line 17).

Function MakePrimary(). Algorithm 4 prepares the new fixpoint computation by partially splitting some DSCCs to their constituent PDSCCs. In words, the algorithm first computes an overapproximation of the splitting DSCCs by running a forward graph search from DSCC(v) (Line 1 to Line 10), every time transitioning to new DSCCs by following pairs of same-label edges outgoing the current DSCC. This transitioning reflects the intuition that such pairs of edges might have used to form the neighboring DSCC – by splitting the current one, we might have to split the neighbor as well. It begins the forward search by populating DSCCRepr(v) in a set $\mathcal{Z}$ and a queue $\mathcal{S}$. Eventually, after Line 1 to Line 10, $\mathcal{Z}$ will contain the roots of all affected DSCCs (including DSCC(v)) while $\mathcal{S}$ helps with the forward search of affected DSCCs.

The second step (Line 11 to Line 22) of Algorithm 4 splits these potentially affected DSCCs to their constituent PDSCCs. In Line 19, the algorithm performs this split in DisjointSets by iterating over the nodes of the DSCCs stored in $\mathcal{Z}$. $\mathcal{R}$ holds the roots of all the newly formed PDSCCs after splitting (Line 20). Then Edges$[z][\cdot]$ is reset, where z is either the root node of an affected DSCC (Line 22) or is the root node of an unaffected DSCC which has an outgoing edge entering one of the affected DSCCs (Line 18). The re-initialization is done keeping in mind the split of the affected DSCCs. These will be re-populated in the third step with edges outgoing or entering the nodes of the newly formed PDSCCs. Although we are guaranteed that these DSCCs do not have to be split below their PDSCCs, it may be the case that certain PDSCCs have to merge again.

In its third step, Algorithm 4 identifies the edges incoming to these PDSCCs, and inserts them in the set $\mathcal{L}$ (Line 23 to Line 30). $\mathcal{L}$ is later used to decide whether two PDSCCs must be merged or not. It also populates Edges with (i) edges incoming to the PDSCCs (lines Line 27 to Line 29) as well as with (ii) edges (Line 31 to Line 34) outgoing from the PDSCCs, and entering an unaffected DSCC (those whose roots are not in $\mathcal{R}$). Note that edges coming out of a PDSCC and entering an affected DSCC (those whose roots are in $\mathcal{R}$) are covered by Line 27 to Line 29.

To merge PDSCCs, the worklist Q is populated using relevant entries of $\mathcal{L}$ to drive the new fixpoint computation (Line 23 to Line 37). In more detail, the algorithm iterates over all PDSCCs (Line 24), and for each node t in each PDSCC (Line 25) and label $\overline{\beta}$ (Line 26), it identifies the edges $s \xrightarrow{\overline{\beta}} t$ by iterating over the nodes $s \in$ InPrimary$[t][\overline{\beta}]$ (Line 27). It then identifies the root node x of s's component (Line 28), and inserts t in Edges$[x][\overline{\beta}]$ as well as the pair $(x, \overline{\beta})$ in $\mathcal{L}$, to record that we have an incoming edge $\cdot \xrightarrow{\overline{\beta}} t$ to a PDSCC that the fixpoint computation needs to process. Then, Line 34 inserts in Edges$[r][\overline{\beta}]$ the edges $t \xrightarrow{\overline{\beta}} y$ that outgo t's component (rooted at r) and for which y is not in a breaking component. Finally, the algorithm identifies the pairs $(x, \overline{\beta})$ in $\mathcal{L}$ for which it has recorded at least two $\overline{\beta}$ neighbors out of x's component, and inserts them in Q to be processed by the fixpoint algorithm later (Line 35).

By structuring MakePrimary this way we have the following benefits: (i) we spend no time in the non-affected DSCCs, and (ii) because we maintain the PDSCCs, the time cost is only linear in the number of *nodes* of these affected DSCCs (and hence $O(n)$ in the worst case), as opposed to linear in the number of the *edges* incoming to these affected DSCCs (which would be $O(n^2)$). This ensures that $\ell = \sum_{x,\overline{\alpha}} |\text{Edges}[x][\overline{\alpha}]| = O(n)$ at the end of MakePrimary before calling Fixpoint; as shown in [Chatterjee et al. 2018, Lemma 3.4, Lemma 3.5], a call to Fixpoint() takes time $O(\ell \cdot \alpha(n))$.

Algorithm 4: MakePrimary()

```
   // 1. A forward search from DSCC($u$) to gather the DSCCs potentially affected by delete($u, v, \overline{\alpha}$)
1  $\mathcal{S} \leftarrow$ an empty queue over $V$    // A simple queue to perform the forward search
2  $\mathcal{Z} \leftarrow$ an empty set over $V$    // Stores the roots of DSCCs potentially affected by delete($u, v, \overline{\alpha}$)
3  Insert DisjointSets.Find($v$) in $\mathcal{Z}$ and $\mathcal{S}$    // Mark DSCC($v$) as potentially affected by the delete
4  while $\mathcal{S}$ is not empty do    // Start the forward search from DSCC($v$)
5      Extract $x$ from $\mathcal{S}$    // Proceed the search to DSCC($x$)
6      foreach label $\overline{\beta} \in \Sigma^C$ do
7          if exist nodes $s \neq t$ with DSCC($x$) $\xrightarrow{\overline{\beta}}$ $s$ and DSCC($x$) $\xrightarrow{\overline{\beta}}$ $t$ then    // Note that DSCC($s$) = DSCC($t$)
8              $z \leftarrow$ DisjointSets.Find($t$)    // The root representative of $t$'s component
9              if $z \notin \mathcal{Z}$ then    // This is the first time we encounter DSCC($z$)
10                 Insert $z$ in $\mathcal{Z}$ and $\mathcal{S}$    // Mark DSCC($z$) as potentially affected by the delete
   // 2. Every DSCC potentially affected is split to its PDSCCs
11 $\mathcal{R} \leftarrow$ an empty set over $V$    // Gathers the roots of the PDSCCs that DSCCs are split to
12 foreach $z \in \mathcal{Z}$ do
13     foreach node $t$ in the component of $z$ do    // Iterate over the nodes of the affected DSCCs
14         foreach label $\overline{\beta} \in \Sigma^C$ do
15             foreach $s \in$ InPrimary[$t$][$\overline{\beta}$] do    // Incoming neighbor to the affected DSCC($z$)
16                 $x \leftarrow$ DisjointSets.Find($s$)    // The root representative of DSCC($s$)
17                 if $x \notin \mathcal{Z}$ then    // DSCC($x$) is unaffected by the delete
18                     Reinitialize Edges[$x$][$\overline{\beta}$] to an empty list    // Will be re-populated later in Line 29
19     Split the nodes of $z$'s component in DisjointSets to its PDSCCs, using PrimCompDS
20     Insert the roots of the new components in $\mathcal{R}$    // Some of the PDSCCs might merge again
21     foreach label $\overline{\beta} \in \Sigma^C$ do
22         Reinitialize Edges[$z$][$\overline{\beta}$] to an empty list // $z$ no longer represents a component in DisjointSets
   // 3. Gather edges incoming to the newly initialized components for new fixpoint computation
23 $\mathcal{L} \leftarrow$ an empty set over $V \times \Sigma^C$ // Gathers pairs ($x, \overline{\beta}$) for which DSCC($x$) $\xrightarrow{\overline{\beta}}$ PDSCC($r$), for $r \in \mathcal{R}$
24 foreach $r \in \mathcal{R}$ do    // Iterate over the roots of the newly split components
25     foreach node $t$ in the component of $r$ do // Iterate over the nodes of each newly split component
26         foreach label $\overline{\beta} \in \Sigma^C$ do
27             foreach $s \in$ InPrimary[$t$][$\overline{\beta}$] do    // Retrieve that $s \xrightarrow{\overline{\beta}}$ PDSCC($t$)
28                 $x \leftarrow$ DisjointSets.Find($s$)    // The root representative of $s$'s component
29                 Insert $t$ in Edges[$x$][$\overline{\beta}$]    // The component of $s$ has an outgoing edge to $t$
30                 Insert ($x, \overline{\beta}$) in $\mathcal{L}$    // Mark that we have inserted edges in Edges[$x$][$\overline{\beta}$]
31             if |OutEdges[$t$][$\overline{\beta}$]| $\geq 1$ then    // $t$ has a $\overline{\beta}$ out-neighbor
32                 $y \leftarrow$ the first node in OutEdges[$t$][$\overline{\beta}$]    // An arbitrary $\overline{\beta}$ out-neighbor of $t$
33                 if DisjointSets.Find($y$) $\notin \mathcal{R}$ then    // The component of $y$ is not splitting
34                     Insert $y$ in Edges[$r$][$\overline{\beta}$]    // The component of $t$ has a $\overline{\beta}$ edge to $y$
35 foreach ($x, \overline{\beta}$) $\in \mathcal{L}$ do    // Re-initialize the queue $Q$ for the new fixpoint
36     if |Edges[$x$][$\overline{\beta}$]| $\geq 2$ then    // There are two outgoing $\overline{\beta}$-labeled edges from $x$'s component
37         Insert ($x, \overline{\beta}$) in $Q$
```

3.4 Example

Here we give a step-by-step illustration of DynamicAlgo on the example of Figure 1. We start with a description of the contents of the various data structures before handling the edge insertion.

Let us fix some notations. Let $(v_1, v_2, \overline{R})\colon k$ denote that there are k edges from v_1 to v_2 labeled $\overline{R}$. The elements of the set Count will contain entries of this kind. Likewise, let $(v, \overline{L})\colon [v_1, v_2, \dots, v_k]$ represent that there are out edges labeled $\overline{L}$ from v to $v_1, \dots, v_k$. OutEdges contains entries of this kind. We let InPrimary to contain entries of the kind $(v, \overline{L}) : [v_1, \dots, v_k]$ to denote that there are edges labeled $\overline{L}$ from each of $v_1, \dots, v_k$ to v. The components in the DisjointSets and PrimCompDS data structures are listed as sequences of nodes $\{(a), b, c, \dots\}$, where the node in the parenthesis denotes the component representative.

(1) Count : $\{(c, g, \overline{R})\colon 1,\ (f, c, \overline{L})\colon 1,\ (f, e, \overline{L})\colon 1,\ (f, d, \overline{L})\colon 1,\ (g, e, \overline{L})\colon 1,\ (h, f, \overline{L})\colon 1\}$
(2) OutEdges : $\{(f, \overline{L})\colon [e, c, d],\ (g, \overline{L})\colon [e],\ (h, \overline{L})\colon [f],\ (c, \overline{R})\colon [g]\}$
(3) InPrimary : $\{(d, \overline{L})\colon [f],\ (e, \overline{L})\colon [g],\ (f, \overline{L})\colon [h],\ (g, \overline{R})\colon [c]\}$
(4) Edges : $\{(f, \overline{L}) : [c],\ (c, \overline{R}) : [g],\ (g, \overline{L}) : [e],\ (h, \overline{L}) : [f]\}$
(5) DisjointSets : $\{(c), d, e\},\ \{(g)\},\ \{(f)\},\ \{(h)\}$
(6) PrimCompDS – PDSCC : $\{(c), d, e\}, \{(g)\}, \{(f)\},\ \{(h)\}$
(7) $Q : \emptyset$

We now describe the development of the data structures during the edge insertion $(d, h, \overline{R})$.

(1) Count$[d][h][\overline{R}] = 1$.
(2) Since $|$OutEdges$[d][\overline{R}]| == 0$, the algorithm adds d to the InPrimary$[h][\overline{R}]$. Then, the algorithm inserts h as the head of OutEdges$[d][\overline{R}]$.
(3) Get the root of DSCC containing d , i.e. $c =$ DisjointSets.Find(d). Then add h to Edges$[c][\overline{R}]$. So, updated Edges$[c][\overline{R}] = (g, h)$.
(4) Since, $|$Edges$[c][\overline{R}]| \geq 2$. The algorithm inserts $(c, \overline{R})$ in Q and invokes Fixpoint().
(5) Inside Fixpoint(): First, we extract $(c, \overline{R})$ from Q. Then we get the set $S = \{g, h\}$. Since $|S| \geq 2$, the algorithm merges DSCCs $[(g)], [(h)]$ into single DSCC$[(h), g]$. Then, moves Edges$[g][\overline{L}]$ to Edges$[h][\overline{L}]$. Now, since $|$Edges$[h][\overline{L}]| \geq 2$, insert $(h, \overline{L})$ to Q. This will lead to merging of DSCCs $[(c), d, e]$ and $[(f)]$ into DSCC $[(f), c, d, e]$. This finishes the fixpoint.

After the insertion, the data structures are in the following state.

(1) Count: new entry $(d, h, \overline{R})\colon 1$; OutEdges: new entry $(d, \overline{R})\colon [h]$; InPrimary: new entry $(h, \overline{R})\colon [d]$; $Q : \emptyset$
(2) Edges : $\{(f, \overline{L}) : [c],\ (f, \overline{R}) : [h],\ (h, \overline{L}) : [f]$
(3) DisjointSets : $\{(f), c, d, e\},\ \{(h), g\}$
(4) PrimCompDS – PDSCC : $\{(c), d, e\},\ \{(g)\},\ \{(f)\},\ \{(h)\}$

We now describe the development of the data structures during the edge deletion $(f, d, \overline{L})$.

(1) Count$[f][d][\overline{L}] = 0$. Since d is at tail of OutEdges$[f][\overline{L}]$, remove f from InPrimary$[d][\overline{L}]$.
(2) In OutEdges$[f][\overline{L}]$, c is penultimate element. Hence, the algorithm inserts f in InPrimary$[c][\overline{L}]$.
(3) Remove edge (c, d) from PrimCompDS.
(4) Remove d from OutEdges$[f][\overline{L}]$.
(5) Invoke MakePrimary():
    (1) Insert $f =$ DSCCRepr(d) to $\mathcal{S}$ and $\mathcal{Z}$.
    (2) In the loop from Line 4 to Line 10 in the algorithm inserts $h =$ DSCCRepr(g) to $\mathcal{S}$ and $\mathcal{Z}$.

(3) $\mathcal{Z}$ contains f, h and by following InPrimary, algorithm re-initializes Edges$[f][\overline{L}]$, Edges$[f][\overline{R}]$ and Edges$[h][\overline{L}]$ to an empty list.
(4) Since $\mathcal{Z}$ contains f and h, we split DSCC(f) and DSCC(h) to their PDSCCs. Then the algorithm inserts roots of new components in $\mathcal{R}$. So, we get $\mathcal{R} = \{c, d, f, g, h\}$.
(5) In the next step the algorithm gathers edges for fixpoint computations. For each root r in $\mathcal{R}$, the algorithm iterate over all nodes in DSCC(r). For each node t in DSCC(r) and each edge type $\overline{\beta} \in \{\overline{L}, \overline{R}\}$ the algorithm checks if there exists node $s \in$ InPrimary$[t][\overline{\beta}]$. For all such nodes s, the algorithm inserts t in Edges[DSCCRepr(s)]$[\overline{\beta}]$ and (DSCCRepr$(s), \overline{\beta})$ in $\mathcal{L}$ (lines Line 27 to Line 30) . In lines Line 31 - Line 34 the algorithm checks if OutEdges$[t][\overline{\beta}] \geq 1$. If yes, then inserts $x =$ first node of OutEdges$[t][\overline{\beta}]$ in Edges$[r][\overline{\beta}]$.
So after Line 34 the updated Edges and $\mathcal{L}$ will be:
(1) Edges : $\{(c, \overline{R})\colon [g],\ (f, \overline{L})\colon [c],\ (h, \overline{L})\colon [f],\ (g, \overline{L})\colon [e], (d, \overline{R})\colon [h]$.
(2) $\mathcal{L}$: $(d, \overline{R}), (f, \overline{L}), (c, \overline{R}), (g, \overline{L}), (h, \overline{L})$.
Then the algorithm iterates over all $(x, \overline{\beta}) \in \mathcal{L}$, and checks if Edges$[x][\overline{\beta}] \geq 2$. If yes, then inserts $(x, \overline{\beta})$ in Q. So, for the $\mathcal{L}$, computed in above step, no element will be added to the Q.
(6) Since Q is empty, the call to Fixpoint() will not merge any components.

After the deletion, the data structures are in the following state.

(1) Count : $\{(c, g, \overline{R})\colon 1,\ (f, c, \overline{L})\colon 1,\ (f, e, \overline{L})\colon 1,\ (f, d, \overline{L})\colon 0,\ (g, e, \overline{L})\colon 1,\ (h, f, \overline{L})\colon 1,\ (d, h, \overline{R})\colon 1\}$,
(2) OutEdges : $\{(f, \overline{L})\colon [e, c],\ (g, \overline{L})\colon [e],\ (h, \overline{L})\colon [f],\ (c, \overline{R})\colon [g],\ (d, \overline{R})\colon [h]\}$,
(3) InPrimary : $\{(c, \overline{L})\colon [f],\ (e, \overline{L})\colon [g],\ (f, \overline{L})\colon [h],\ (g, \overline{R})\colon [c],\ (h, \overline{R})\colon [d]\}$
(4) Edges : $\{(c, \overline{R}) : [g],\ (f, \overline{L}) : [c],\ (h, \overline{L}) : [f],\ (g, \overline{L}) : [e],\ (d, \overline{R}) : [h]\}$
(5) DisjointSets : $\{(c), e\}, \{(d)\}, \{(g)\}, \{(f)\}, \{(h)\}$
(6) PrimCompDS – PDSCC : $\{(c), e\}, \{(d)\}, \{(g)\}, \{(f)\}, \{(h)\}$
(7) $Q : \emptyset$

3.5 Correctness and Complexity

Finally, we state the correctness and complexity of DynamicAlgo.

Theorem 1.1. *Given a bidirected graph G of n nodes,* DynamicAlgo *correctly maintains the DSCCs of G across edge insertions and edge deletions, and uses at most $O(n \cdot \alpha(n))$ time for each update.*

Next, we state the main invariants that DynamicAlgo maintains which support its correctness and complexity, as well as some intuition behind them. For proofs, we refer to [Krishna et al. 2023].

Correctness. The basis of the correctness of DynamicAlgo is a number of invariants that are maintained along edge insertions and deletions. Observe that OutEdges$[u][\overline{\alpha}]$ is, at all times, a linked list representation of the edge set $u \xrightarrow{\overline{\alpha}} \cdot$.

Our first invariant concerns the correct maintenance of the PDSCCs of G in the PrimCompDS data structure maintaining undirected connectivity. To prove the invariant, we argue that DynamicAlgo inserts and removes sufficient and necessary undirected edges in PrimCompDS to (sparsely) represent the connected components of the corresponding primal graph. This follows directly from the sparsification approach outlined in Section 3.2 and Figure 4. Indeed, the algorithm maintains in PrimCompDS a set of edges that connect neighboring nodes in each linked list OutEdges$[u][\overline{\alpha}]$. Thus, the omitted edges (i.e., between non-neighboring nodes in OutEdges$[u][\overline{\alpha}]$) are anyway transitively connected in PrimCompDS. Formally, we have the following lemma.

Lemma 3.1. *After every insert and delete operation, the connected components in* PrimCompDS *are precisely the PDSCCs of G.*

The next invariant concerns the correct maintenance of the InPrimary data structure, and is established in Lemma 3.2 and Lemma 3.3. In words, the invariant states that $x \in \text{InPrimary}[z][\overline{\beta}]$ iff we have an edge $x \xrightarrow{\overline{\beta}} z$ and z is the last node in $\text{OutEdges}[x][\overline{\beta}]$. Thus, when MakePrimary() constructs the PDSCCs and discovers their incoming edges, the edge $x \xrightarrow{\overline{\beta}} \text{PDSCC}(z)$ is correctly discovered by finding that $x \in \text{InPrimary}[z][\overline{\beta}]$ (Line 27).

Lemma 3.2 is concerned with the soundness, and is straightforward. Any time the algorithm inserts $x \in \text{InPrimary}[z][\overline{\beta}]$, this is followed by inserting z in $\text{OutEdges}[x][\overline{\beta}]$ (in the case of edge insertions, Line 4 and Line 8 in Algorithm 2). In the case of edge deletions, the insertion $x \in \text{InPrimary}[z][\overline{\beta}]$ is preceded by inserting z to $\text{OutEdges}[x][\overline{\beta}]$ in a previous update (Line 6 and Line 7 in Algorithm 3). In this case, $x \in \text{InPrimary}[z][\overline{\beta}]$ takes place when z becomes the last node in $\text{OutEdges}[x][\overline{\beta}]$ after deletion of edges $x \xrightarrow{\overline{\beta}} v$, where v appeared later than z in $\text{OutEdges}[x][\overline{\beta}]$.

Lemma 3.2. *After every insert and delete operation, for every two nodes $x, z \in V$ and label $\overline{\beta} \in \Sigma^C$, if $x \in \text{InPrimary}[z][\overline{\beta}]$ then $x \xrightarrow{\overline{\beta}} z$ and z is the last node in $\text{OutEdges}[x][\overline{\beta}]$.*

Similarly, for Lemma 3.3, if $x \xrightarrow{\overline{\beta}} \text{PDSCC}(y)$, then the last node z in $\text{OutEdges}[x][\overline{\beta}]$ is also in $\text{PDSCC}(y)$, while the algorithm maintains that $x \in \text{InPrimary}[z][\overline{\beta}]$. Hence, the fact that $x \xrightarrow{\overline{\beta}} \text{PDSCC}(y)$ is recoverable via discovering that $x \in \text{InPrimary}[z][\overline{\beta}]$.

Lemma 3.3. *After every insert and delete operation, for every pair of nodes $x, y \in V$ and label $\overline{\beta} \in \Sigma^C$, if $x \xrightarrow{\overline{\beta}} \text{PDSCC}(y)$ then there exists a node $z \in \text{PDSCC}(y)$ such that $x \in \text{InPrimary}[z][\overline{\beta}]$.*

The next lemma captures the correctness of MakePrimary() and the invariants concerning the state that it passes on to function Fixpoint() for the final fixpoint computation after processing a delete$(u, v, \overline{\alpha})$ update. Recall that MakePrimary() identifies an overapproximation of the DSCCs that have to be split and rebuilt as a result of this edge deletion. The lemma has two parts. Item (1) states that the components that MakePrimary() passes on to Fixpoint() (i.e., those found in DisjointSets) are a refinement of the DSCC decomposition of G after the deletion, i.e., it suffices to merge some of them in order to arrive at the correct DSCC-decomposition of the graph after the edge deletion. The Fixpoint() function will perform this merging by processing the edges found in the Edges data structure. Item (2) states that MakePrimary() populates the Edges data structure with sufficiently many edges for Fixpoint() to process and arrive at the correct DSCC decomposition of G.

Lemma 3.4. *At the end of* MakePrimary(), *the following assertions hold.*

(1) Every component in DisjointSets *is a (not necessarily maximal) DSCC of G.*
(2) For every component in DisjointSets *rooted in some node x, the following hold.*
   (a) For every node y and label $\overline{\beta}$, if $y \in \text{Edges}[x][\overline{\beta}]$ then there is an edge $z \xrightarrow{\overline{\beta}} w$, where z is a node in the component of x, and w is a node in the component of y in DisjointSets.

(b) For every node z and label $\overline{\beta}$ such that (i) z is in the component rooted at node x in DisjointSets*, and (ii) there is an edge $z \xrightarrow{\overline{\beta}} w$, there exists a node y in the component of w in* DisjointSets *such that $y \in$* Edges$[x][\overline{\beta}]$.

Given the above invariants, MakePrimary creates a correct state of the worklist $\mathcal{Q}$ and the Edges linked lists for the Fixpoint() call of Algorithm 3 (Line 17) to compute the correct DSCC decomposition. This leads ot the correctness of DynamicAlgo.

Lemma 3.5. *After every* insert$(u, v, \overline{\alpha})$ *and* delete$(u, v, \overline{\alpha})$*,* DisjointSets *contains the DSCCs of G.*

Complexity. We now turn our attention to the complexity bound of Theorem 1.1. We always start with a graph of n nodes but without any edges, for which the initialization of all data structures takes $O(n)$ time. In practice, the initial graph might already have some edges, which can be thought of being inserted one-by-one.

In high level, the time DynamicAlgo spends in each edge insertion and deletion is the sum of two parts: (i) the time taken for maintaining the PDSCCs in the PrimCompDS data structure, and (ii) the time taken for all other computations. Regarding (i), so far we have not specified the precise data structure for implementing PrimCompDS, as DynamicAlgo treats PrimCompDS as a black box. The theoretical guarantees of Theorem 1.1 can be obtained by using the data structure of [Eppstein et al. 1997] for undirected connectivity (see Table 1). Although this guarantees $O(\sqrt{n})$ and $O(1)$ time for edge insertions/deletions and queries, respectively, here we only use the fact that both bounds are less than n. Regarding (ii), the algorithm spends $O(n \cdot \alpha(n))$ for each operation, which stems from the fact that the algorithm encounters each node of G across all data structures a constant number of times (recall that we have $k = O(1)$ labels in G). In particular, we have the following lemma.

Lemma 3.6. *Every* insert$(u, v, \overline{\alpha})$ *and* delete$(u, v, \overline{\alpha})$ *operation is processed in $O(n \cdot \alpha(n))$ time.*

Proof. After every insert and delete operation, Fixpoint() is invoked for merging the DSCCs (in DisjointSets). As shown in [Chatterjee et al. 2018, Lemma 3.4, Lemma 3.5], a call to Fixpoint() takes time $O(\ell \cdot \alpha(n))$, where $\ell = \sum_{u,\overline{\alpha}} |$Edges$[x][\overline{\alpha}]|$ is the total number of nodes stored in the Edges linked lists, and $\alpha(n)$ is the inverse Ackermann function. After Fixpoint() has completed, we have $|$Edges$[x][\overline{\alpha}]| \leq 1$ for each x and $\overline{\alpha}$. Thus to prove Lemma 3.6, it suffices to argue that every operation takes $O(n)$ time before calling Fixpoint(). This implies that $\ell = O(n)$ before the call, and thus Fixpoint() takes $O(n \cdot \alpha(n))$ time, yielding $O(n) + O(n \cdot \alpha(n)) = O(n \cdot \alpha(n))$ total time.

Edge insertions. Consider the processing of an operation insert$(u, v, \overline{\alpha})$ by Algorithm 2. Observe that the algorithm has no loops, thus the running time is dominated by the time taken to access the various data structures that the algorithm maintains. In particular, OutEdges is a simple linked list, and checking its length (Line 3), as well as accessing and inserting in the head (Line 6 and Line 8) takes constant time. Similarly, the data structures InPrimary and Edges are simple sets and linked lists with $O(1)$ accesses. As we have already argued in Section 2.2, performing a DisjointSets. Find(v) operation (Line 9) takes $O(1)$ time. Finally, inserting the edge (v, y) in PrimCompDS takes $O(\sqrt{n})$ time [Eppstein et al. 1997].

Edge deletions. Consider the processing of an operation delete$(u, v, \overline{\alpha})$ by Algorithm 3. The time spent in the body of Algorithm 3 is $O(n)$, by an analysis very similar to Algorithm 2 for edge deletions, and we will not repeat it here. Instead, we focus on the time spent in the call to MakePrimary(), which is the more complex part of processing the edge deletion (Algorithm 4). In the first step

(Line 1 to Line 10), for each iteration of the while loop of Line 4, the condition in Line 7 can be checked in $O(|\mathrm{DSCC}(x)|)$ time, by iterating over all nodes of $\mathrm{DSCC}(x)$. Observe that $\mathrm{DSCC}(x)$ is examined only once throughout this loop of Line 4, and since the DSCCs partition the node set, total time for running this loop is $O(n)$.

The time spent in the second step (Line 11 to Line 22) is the sum of two parts. The first part corresponds to the time spent in the nested loops, which is proportional to the number of times the inner-most loop in Line 15 is taken. That is, the number of nodes x that exist in the lists $\mathrm{InPrimary}[t][\overline{\beta}]$ of nodes t in the components represented by their roots in $\mathcal{Z}$. It suffices to argue that there do not exist nodes x, s_1, s_2 and label $\overline{\beta}$ such that $x \in \mathrm{InPrimary}[s_1][\overline{\beta}]$ and $x \in \mathrm{InPrimary}[s_2][\overline{\beta}]$. Indeed, by Lemma 3.2, if $x \in \mathrm{InPrimary}[s_1][\overline{\beta}]$ then s_1 is the last node in $\mathrm{OutEdges}[x][\overline{\beta}]$. Clearly, as a linked list, $\mathrm{OutEdges}[x][\overline{\beta}]$ can have at most one last node, hence $x \notin \mathrm{InPrimary}[s_2][\overline{\beta}]$. Thus, for every label $\overline{\beta}$, the loop in Line 15 is executed at most once per node x leading to $O(n)$ total iterations. The second part corresponds to the total time spent in Line 19. In Line 19, the algorithm splits the potentially affected DSCCs (in DisjointSets) into its PDSCCs. This takes $O(n)$ time as the algorithm simply iterates over the nodes of the DSCCs stored in $\mathcal{Z}$. For each such node, the algorithm performs a single membership query to PrimCompDS, which takes $O(1)$ time [Eppstein et al. 1997].

Finally, we consider the time spent by MakePrimary() in the third step (Line 23 to Line 37). Note that the time spent in these nested loops is proportional to the number of times the inner-most loop (Line 27) is executed. Again, it suffices to argue that there do not exist nodes x, s_1, s_2 and label $\overline{\beta}$ such that $x \in \mathrm{InPrimary}[s_1][\overline{\beta}]$ and $x \in \mathrm{InPrimary}[s_2][\overline{\beta}]$. Indeed, by Lemma 3.2, if $x \in \mathrm{InPrimary}[s_1][\overline{\beta}]$ then s_1 is the last node in $\mathrm{OutEdges}[x][\overline{\beta}]$, hence $x \notin \mathrm{InPrimary}[s_2][\overline{\beta}]$. Thus, for every label $\overline{\beta}$, the loop in Line 27 is executed at most once per node x, leading to $O(n)$ total iterations. In the end, the loop in Line 35 runs for $|\mathcal{L}|$ iterations, which is bounded by the iterations of the previous loop.

Since for every label $\overline{\beta}$ and node x the loop in Line 27 is executed at most once, it follows that Line 29 will be executed $O(n)$ times. This leads to total $O(n)$ new entries to Edges at Line 29. Similarly, Line 34 is executed $O(n)$ times leading to $O(n)$ new entries to Edges. Therefore, at the end of MakePrimary() we have $\sum_{x,\overline{\alpha}} |\mathrm{Edges}[x][\overline{\alpha}]| = O(n)$. The loop in Line 35 runs for $|\mathcal{L}| = O(n)$ iterations. This leads to $O(n)$ entries in $\mathcal{Q}$. Thus, at the end of MakePrimary(), $|\mathcal{Q}| = O(n)$. Thus before calling Fixpoint() we have $\sum_{u,\overline{\alpha}} |\mathrm{Edges}[u][\overline{\alpha}]| = O(n)$ and $|\mathcal{Q}| = O(n)$. □

4 EXPERIMENTS

In this section we report on an implementation of DynamicAlgo algorithm behind Theorem 1.1, and an evaluation of its performance on various datasets on real-world static analyses. To some extent, our experimental setting follows [Li et al. 2022].

Compared algorithms. We compare three standard approaches to bidirected Dyck reachability[3].

(1) OfflineAlgo, as developed and implemented in [Chatterjee et al. 2018]. For each graph update (edge insertion/deletion), the algorithm is invoked from scratch to handle the updated graph.
(2) Our DynamicAlgo, which is implemented in C/C++, and closely follows the pseudocode presented in Section 3. DynamicAlgo uses as a black box a data structure PrimCompDS for dynamic undirected connectivity (see Table 1). Although the one developed in [Eppstein et al. 1997] works best towards the complexity guarantees of Theorem 1.1, in our implementation we use

[3]Since the algorithm of [Li et al. 2022] has complexity and correctness issues, we have not included it in our evaluation.

the one developed in [Holm et al. 2001] (and implemented in [Tseng 2020]), as it is conceptually simpler and well-performing in practice.

(3) A declarative approach in which the production rules of Dyck reachability are encoded as Datalog constraints and dispatched to a Datalog solver [Reps 1995b]. Datalog-based static analyses have been popularized in the Flix programming language [Madsen and Lhoták 2020] and the Doop framework [Bravenboer and Smaragdakis 2009]. Our bidirected setting allows us to optimize the Datalog program by explicitly focusing only on closing-parentheses edges, in similar style to OfflineAlgo and DynamicAlgo. To be fair in our comparison, we follow this approach here. In particular, we use the following Datalog program.

```
Reaches(u,u)
Close(x,u,ᾱ):-  Edge(x,u,ᾱ)
Close(x,u,ᾱ):-  Edge(y,u,ᾱ),  Reaches(x,y)
Reaches(u,v):-  Close(x,u,ᾱ),  Close(x,v,ᾱ)
Reaches(u,v):-  Reaches(u,x),  Reaches(x,v)
```

To handle our dynamic setting, we rely on an efficient, fully dynamic Datalog solver [Ryzhyk and Budiu 2019], as opposed to solving the Datalog program from scratch every time.

The two algorithms (OfflineAlgo and dynamic Datalog) that support our experimental comparison serve as very fitting baselines. OfflineAlgo is an algorithm dedicated to the *problem* we are solving (i.e., bidrected Dyck reachability) but not dedicated to the *setting* we are solving it in (i.e., under dynamic updates). On the other hand, the dynamic-Datalog approach is dedicated to the dynamic setting, but not dedicated to the bidirected Dyck-reachability problem. As such, both algorithms are theoretically of worse complexity that our DynamicAlgo, yet still the closest that exist in the literature for this problem and setting. Our experiments aim to highlight to what extent the theoretical superiority of DynamicAlgo is realized in practice.

Benchmarks. We evaluate the above algorithms on two popular static analyses.

(1) Context-sensitive data dependence analysis as formulated in [Tang et al. 2015], and evaluated on benchmark programs from [SPE 2008]. In this case the parenthesis labels represent calling contexts, and a properly-balanced-parenthesis path represents an interprocedurally-valid dataflow via parameter-passing in function invocation and return.
(2) Field-sensitive alias analysis for Java as formulated in [Yan et al. 2011; Zhang et al. 2013], and evaluated on DaCapo benchmarks [Blackburn et al. 2006]. In this case the parenthesis labels represent field accesses on composite objects, as illustrated in Figure 1 of Section 1.

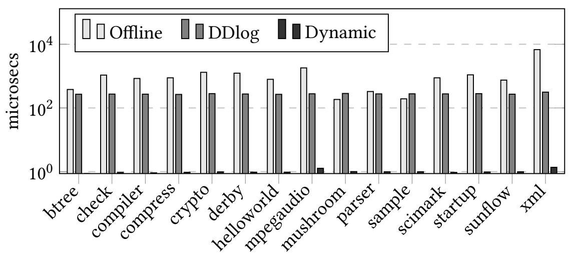
Our graph models for the above analyses and benchmarks are obtained from [Li et al. 2022].

On-the-fly formulation and update sequences. To simulate the on-the-fly setting where the source code undergoes a sequence of changes, for each benchmark graph G, we generate three sequences $\mathcal{S}_G$ of updates (edge insertions/deletions), as follows.

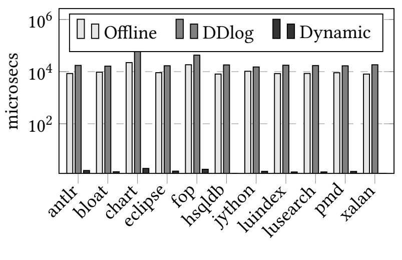
(1) *Incremental setting:* We randomly select a set E^+ of 90% of the edges of G and remove them from the graph. We create a sequence of edge insertions $\mathcal{S}_G^{\text{inc}}$ as a random permutation of E^+. This is a fully incremental setting, where code lines are only added but never removed in the program.
(2) *Decremental setting:* We randomly select a set E^- of 90% of the edges of G. We create a sequence of edge deletions $\mathcal{S}_G^{\text{dec}}$ as a random permutation of E^-. This is a fully decremental setting, where code lines are only removed but never added in the program.
(3) *Mixed setting:* We randomly split the edges of G into two sets E^+ and E^-, with proportion 10% to 90%, and start with an initial graph containing the edges of E^-. We create a sequence of

Data Dependence Analysis

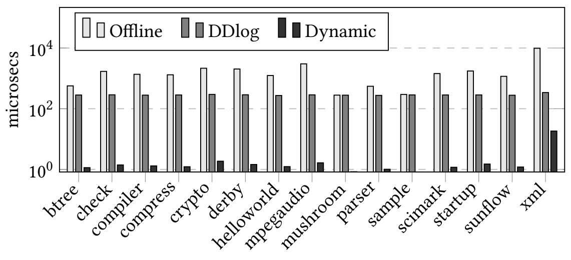
Alias Analysis



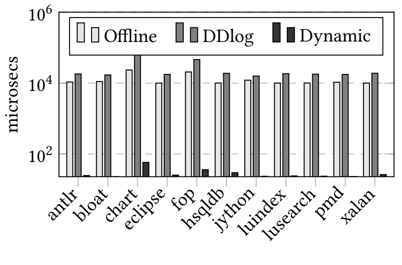
(a) Incremental updates.



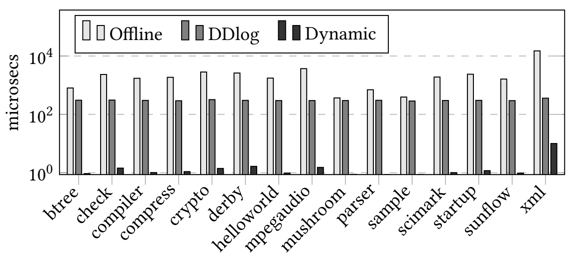
(b) Incremental updates.



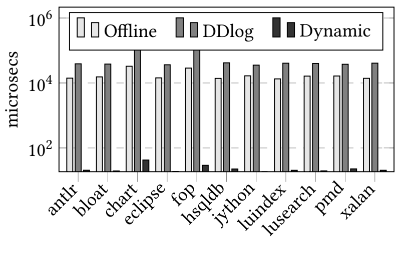
(c) Decremental updates.



(d) Decremental updates.



(e) Mixed updates.



(f) Mixed updates.

Fig. 7. The average time to handle a single update on on-the-fly data-dependence analysis (a, c, e) and on-the-fly alias analysis (b, d, f) for sequence files generated with 90-10 split of original graph. Note that all results are in log-scale.

mixed operations (both insertions and deletions) $\mathcal{S}_G^{\text{dec}}$ by repeated stochastic sampling: in each step, we randomly choose the next operation as an edge insertion/edge deletion. In the former case, we randomly select an edge from E^+, move it to E^-, and insert it in G. In the latter case, we randomly select an edge from E^-, move it to E^+, and delete it from G. The length of the sequence is equal to 90% of the number of edges in G.

In each case, for each benchmark we report the amortized time that each algorithm took to handle the whole sequence: that is, the total running time over the whole sequence divided by the length of the sequence. To gain more confidence in our results, we repeat the above process three times and report the average numbers. We run our experiments on a conventional laptop with a 2.6GHz CPU and 16GBs of memory, which was always sufficient for the analysis. As a sanity check, we have verified that all three algorithms give the same results on each benchmark and update sequence.

Results on data-dependence analysis. Our experimental results on on-the-fly the data-dependence analysis are shown in Figure 7 (left column). We see that in all three settings (incremental/decremental/mixed), the dynamic-Datalog approach is measurably faster than OfflineAlgo. Although, in the worst case, the Datalog solver has worse complexity than OfflineAlgo, the nature of the updates on the analyses graphs allows an efficient Datalog solver dedicated to dynamic updates to perform faster, as it never exhibits its worst-case performance. Still, the time spent by the Datalog solver is typically quite large, given that we are looking at updates by a *single edge* (i.e., a single source-code line).

On the other hand, our DynamicAlgo spends much less time than both OfflineAlgo and the dynamic Datalog solver, leading to typical speedups between two and three orders of magnitude. The only difference is in `xml`, where DynamicAlgo appears to spend more time than in the rest of the benchmarks. We have identified that this benchmark has a disproportionately large number of parenthesis symbols, which is the likely cause of this behavior. Still, DynamicAlgo is by far the fastest approach also on this benchmark. Naturally, decremental updates yield the least speedup on average. This is expected given how much more complex our procedure for handling deletions is compared to that of handling insertions. Indeed, when processing a insert$(u, v, \overline{\alpha})$ update, our algorithm only merges DSCCs, while processing a delete$(u, v, \overline{\alpha})$ update both splits and merges DSCCs. Nevertheless, the speedups are still in the range of two orders of magnitude, enough to render DynamicAlgo the clear best approach overall.

Results on alias analysis. Our experimental results on on-the-fly the alias analysis are shown in Figure 7 (right column). In contrast to the data-dependence analysis, here the dynamic-Datalog approach is always a bit worse that OfflineAlgo, indicating that reachability patterns in these graphs are more challenging; enough so to push the dynamic-Datalog solver to worse performance than the from-scratch OfflineAlgo. Note, also, that the analysis times here are considerably larger than in the case of data-dependence analysis.

DynamicAlgo is again the best-performing approach by far, consistently by three orders of magnitude. In several benchmarks, its running time is not visible despite the log-scale of the plots. Finally, we again observe that decremental updates are overall more challenging that incremental updates.

In summary. Our experiments clearly show that DynamicAlgo is the right approach to on-the-fly static analyses formulated as bidirected Dyck reachability, giving several orders of magnitude speedups over both (i) the offline algorithm that is dedicated (and optimal) for bidirected Dyck reachability, and (ii) the dynamic Datalog approach that is dedicated to dynamic updates (but agnostic to the setting of bidirected Dyck reachability). Although the worst-case running time of DynamicAlgo is linear, we rarely observed this in our experiments. Instead, the time cost of each update is barely (if at all) noticeable to the human eye, and thus the algorithm is suitable for continuous analysis, e.g., integrated inside an IDE.

5 RELATED WORK

The importance of Dyck reachability in static analyses has lead to a systematic study of its complexity in various settings. The problem has a simple $O(n^3)$ upper bound [Yannakakis 1990], which has resisted improvements beyond logarithmic [Chaudhuri 2008]. Due to this reason, the complexity of Dyck reachability has also been studied in terms of lower bounds. Even for a single-pair query, the problem has been known to be 2NPDA-hard [Heintze and McAllester 1997], while its combinatorial cubic hardness persists even on constant-treewidth graphs [Chatterjee et al. 2018]. All-pairs Dyck reachability was recently shown to have a conditional $n^{2.5}$ lower bound based on popular complexity-theoretic hypotheses [Koutris and Deep 2023]. Despite the cubic hardness of the general problem, it is known to have sub-cubic certificates for both positive and negative instances [Chistikov et al. 2022]. All-pairs reachability with $k = 1$ parenthesis (aka one-counter systems) was recently shown to admit an $O(n^{\omega} \cdot \log^2 n)$ bound [Mathiasen and Pavlogiannis 2021], where ω is the matrix multiplication exponent, which is also tight even for single-pair queries [Cetti Hansen et al. 2021]. We refer to [Pavlogiannis 2023] for a recent survey on this rich problem.

The technique developed in this work for on-the-fly bidirected Dyck reachability is motivated by the same setting on undirected connectivity, which has been studied extensively. We leverage the data structure developed in [Eppstein et al. 1997] for maintaining the PDSCCs of a graph as connected components of the underlying primary graph (which is indeed undirected), as well as a sparsification technique that is specific to our setting (even though the concept of sparsification was developed in [Eppstein et al. 1997] to handle undirected connectivity). It would be interesting to investigate whether techniques from undirected connectivity can be used further in our setting so as to reduce the complexity to sublinear (as is the status quo in undirected connectivity). However, as a single update can create or merge $\Theta(n)$ DSCCs, sublinear complexity can only arise in the amortized sense, or by not requiring the explicit maintenance of DSCCs throughout updates. Though we can easily obtain an $O(\alpha(n))$ amortized insertion cost for our technique (by amortizing over the linear cost of deletes), tighter bounds appear non-trivial and are open to interesting future work.

Bidirected Dyck reachability is very similar to unification closure, though the later problem is typically phrased with labels on the nodes as opposed to the edges of the input graph [Kanellakis and Revesz 1989]. Unification closure has found widespread applications in programming languages, such as in efficient binding-time analysis [Henglein 1991], simple first-order type-inference [Møller and Schwartzbach 2018], efficient dynamic type inference for LISP [Henglein 1992], as well as Steensgaard's famous pointer analysis [Steensgaard 1996]. The insights developed here for the on-the-fly setting are likely extendable to these applications, though this merits further investigation.

In this work we have focused on online analyses where the analyzed source code changes frequently. Another "on-the-fly" style of analysis is that of *on-demand* analysis. Here the analyzed program remains unchanged, but the task is to answer a sequence of analysis queries that is not known in advance. This setting has been studied a lot in the context of pointer analysis [Heintze and Tardieu 2001; Sridharan et al. 2005; Yan et al. 2011; Zheng and Rugina 2008] and data flow analysis [Horwitz et al. 1995; Lerch et al. 2015; Naeem et al. 2010]; the latter have also been efficiently parameterized by treewidth [Chatterjee et al. 2016, 2020, 2015] and treedepth [Goharshady and Zaher 2023].

6 CONCLUSION

On-the-fly analysis is a very appealing feature to static analyzers, in order to run in real-time during code development and incorporate the constant changes in the source code. However, on-the-fly analysis algorithms with provable complexity benefits have been missing, due to the intricacies of typical static analyses. In this work we have considered a wide class of static analyses phrased as bidirected Dyck reachability. We have developed a dynamic algorithm for handling the addition and removal of source code lines, with a provable guarantee that each such modification takes only (nearly) linear time in the worst case. Our experiments show that our dynamic algorithm is extremely performant in practice, with a clear advantage over the (optimal) offline static analysis algorithm, as well as dynamic Datalog approaches, with typical speedups of three orders of magnitude. From a practical standpoint, our results indicate that our algorithm can directly support lightweight analyses inside IDEs at a time cost that is barely (if at all) noticeable to the developers.

ACKNOWLEDGMENTS

Andreas Pavlogiannis was partially supported by a research grant (VIL42117) from VILLUM FONDEN. S. Krishna was partially supported by the SERB MATRICS grant MTR/2019/000095.

REFERENCES

2003. T. J. Watson Libraries for Analysis (WALA). https://github.com.

2008. SPECjvm2008 Benchmark Suit. http://www.spec.org/jvm2008/.

Robert S. Arnold. 1996. *Software Change Impact Analysis.* IEEE Computer Society Press, Los Alamitos, CA, USA.

Steven Arzt and Eric Bodden. 2014. Reviser: Efficiently Updating IDE-/IFDS-Based Data-Flow Analyses in Response to Incremental Program Changes. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014).* Association for Computing Machinery, New York, NY, USA, 288–298. https://doi.org/10.1145/2568225.2568243

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

Eric Bodden. 2012. Inter-procedural Data-flow Analysis with IFDS/IDE and Soot. In *SOAP.* ACM, New York, NY, USA.

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. *SIGPLAN Not.* 44, 10 (oct 2009), 243–262. https://doi.org/10.1145/1639949.1640108

M.G. Burke and B.G. Ryder. 1990. A critical analysis of incremental iterative data flow analysis algorithms. *IEEE Transactions on Software Engineering* 16, 7 (1990), 723–728. https://doi.org/10.1109/32.56098

Jakob Cetti Hansen, Adam Husted Kjelstrøm, and Andreas Pavlogiannis. 2021. Tight bounds for reachability problems on one-counter and pushdown systems. *Inform. Process. Lett.* 171 (2021), 106135. https://doi.org/10.1016/j.ipl.2021.106135

Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. 2018. Optimal Dyck Reachability for Data-Dependence and Alias Analysis. *Proc. ACM Program. Lang.* 2, POPL, Article 30 (Dec. 2018), 30 pages.

Krishnendu Chatterjee, Amir Kafshdar Goharshady, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. 2016. Algorithms for Algebraic Path Properties in Concurrent Systems of Constant Treewidth Components. *SIGPLAN Not.* 51, 1 (jan 2016), 733–747. https://doi.org/10.1145/2914770.2837624

Krishnendu Chatterjee, Amir Kafshdar Goharshady, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. 2020. Optimal and Perfectly Parallel Algorithms for On-demand Data-Flow Analysis. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 112–140.

Krishnendu Chatterjee, Rasmus Ibsen-Jensen, Andreas Pavlogiannis, and Prateesh Goyal. 2015. Faster Algorithms for Algebraic Path Properties in Recursive State Machines with Constant Treewidth. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15).* Association for Computing Machinery, New York, NY, USA, 97–109. https://doi.org/10.1145/2676726.2676979

Swarat Chaudhuri. 2008. Subcubic Algorithms for Recursive State Machines. *SIGPLAN Not.* 43, 1 (Jan. 2008), 159–169. https://doi.org/10.1145/1328897.1328460

Dmitry Chistikov, Rupak Majumdar, and Philipp Schepper. 2022. Subcubic Certificates for CFL Reachability. *Proc. ACM Program. Lang.* 6, POPL, Article 41 (jan 2022), 29 pages. https://doi.org/10.1145/3498702

David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. 1997. Sparsification—a Technique for Speeding up Dynamic Graph Algorithms. *J. ACM* 44, 5 (sep 1997), 669–696. https://doi.org/10.1145/265910.265914

Moses Ganardi, Rupak Majumdar, Andreas Pavlogiannis, Lia Schütze, and Georg Zetzsche. 2022b. Reachability in Bidirected Pushdown VASS. In *49th International Colloquium on Automata, Languages, and Programming (ICALP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 229)*, Mikołaj Bojańczyk, Emanuela Merelli, and David P. Woodruff (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 124:1–124:20. https://doi.org/10.4230/LIPIcs.ICALP.2022.124

Moses Ganardi, Rupak Majumdar, and Georg Zetzsche. 2022a. The Complexity of Bidirected Reachability in Valence Systems. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '22).* Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3531130.3533345

Amir Kafshdar Goharshady and Ahmed Khaled Zaher. 2023. Efficient Interprocedural Data-Flow Analysis Using Treedepth and Treewidth. In *Verification, Model Checking, and Abstract Interpretation*, Cezara Dragoi, Michael Emmi, and Jingbo Wang (Eds.). Springer Nature Switzerland, Cham, 177–202.

Nevin Heintze and David McAllester. 1997. On the Cubic Bottleneck in Subtyping and Flow Analysis. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science (LICS '97).* IEEE Computer Society, Washington, DC, USA, 342–. http://dl.acm.org/citation.cfm?id=788019.788876

Nevin Heintze and Olivier Tardieu. 2001. Demand-Driven Pointer Analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, USA) *(PLDI '01)*. Association for Computing Machinery, New York, NY, USA, 24–34. https://doi.org/10.1145/378795.378802

Fritz Henglein. 1991. Efficient Type Inference for Higher-Order Binding-Time Analysis. In *Functional Programming Languages and Computer Architecture (Lecture Notes in Computer Science)*, John Hughes (Ed.). Springer, Berlin, Heidelberg, 448–472. https://doi.org/10.1007/3540543961_22

Fritz Henglein. 1992. Global Tagging Optimization by Type Inference. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming (LFP '92)*. Association for Computing Machinery, New York, NY, USA, 205–215. https://doi.org/10.1145/141471.141542

Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. 2001. Poly-Logarithmic Deterministic Fully-Dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-Edge, and Biconnectivity. *J. ACM* 48, 4 (jul 2001), 723–760. https://doi.org/10.1145/502090.502095

Susan Horwitz, Thomas Reps, and Mooly Sagiv. 1995. Demand Interprocedural Dataflow Analysis. *SIGSOFT Softw. Eng. Notes* (1995).

Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. 2015. Scalable and Precise Taint Analysis for Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 106–117. https://doi.org/10.1145/2771783.2771803

Paris C. Kanellakis and Peter Z. Revesz. 1989. On the Relationship of Congruence Closureand Unification. *Journal of Symbolic Computation* 7, 3-4 (March 1989), 427–444. https://doi.org/10.1016/S0747-7171(89)80018-5

Adam Husted Kjelstrøm and Andreas Pavlogiannis. 2022. The Decidability and Complexity of Interleaved Bidirected Dyck Reachability. *Proc. ACM Program. Lang.* 6, POPL, Article 12 (jan 2022), 26 pages. https://doi.org/10.1145/3498673

Paraschos Koutris and Shaleen Deep. 2023. The Fine-Grained Complexity of CFL Reachability. *Proc. ACM Program. Lang.* 7, POPL, Article 59 (jan 2023), 27 pages. https://doi.org/10.1145/3571252

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]

Johannes Lerch, Johannes Späth, Eric Bodden, and Mira Mezini. 2015. Access-Path Abstraction: Scaling Field-Sensitive Data-Flow Analysis with Unbounded Access Paths. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska) *(ASE '15)*. IEEE Press, 619–629. https://doi.org/10.1109/ASE.2015.9

Ondřej Lhoták and Laurie Hendren. 2006. Context-Sensitive Points-to Analysis: Is It Worth It?. In *Proceedings of the 15th International Conference on Compiler Construction (CC)*. 47–64.

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

Yuanbo Li, Qirun Zhang, and Thomas Reps. 2020. Fast Graph Simplification for Interleaved Dyck-Reachability. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 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, Article 6 (mar 2019), 31 pages. https://doi.org/10.1145/3293606

Jingbo Lu and Jingling Xue. 2019. Precision-Preserving yet Fast Object-Sensitive Pointer Analysis with Partial Context Sensitivity. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 148 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360574

Magnus Madsen and Ondřej Lhoták. 2020. Fixpoints for the Masses: Programming with First-Class Datalog Constraints. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 125 (nov 2020), 28 pages. https://doi.org/10.1145/3428193

Anders Alnor Mathiasen and Andreas Pavlogiannis. 2021. The Fine-Grained and Parallel Complexity of Andersen's Pointer Analysis. *Proc. ACM Program. Lang.* 5, POPL, Article 34 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434315

Ana Milanova. 2020. FlowCFL: Generalized Type-Based Reachability Analysis: Graph Reduction and Equivalence of CFL-Based and Type-Based Reachability. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 178 (Nov. 2020), 29 pages. https://doi.org/10.1145/3428246

Anders Møller and Michael I. Schwartzbach. 2018. *Static Program Analysis*. Technical Report. Department of Computer Science, Aarhus University. http://cs.au.dk/~amoeller/spa/

Nomair A. Naeem, Ondřej Lhoták, and Jonathan Rodriguez. 2010. Practical Extensions to the IFDS Algorithm. In *Compiler Construction*, Rajiv Gupta (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 124–144.

André Pacak, Sebastian Erdweg, and Tamás Szabó. 2020. A Systematic Approach to Deriving Incremental Type Checkers. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 127 (nov 2020), 28 pages. https://doi.org/10.1145/3428195

Andreas Pavlogiannis. 2023. CFL/Dyck Reachability: An Algorithmic Perspective. *ACM SIGLOG News* 9, 4 (feb 2023), 5–25. https://doi.org/10.1145/3583660.3583664

Jakob Rehof and Manuel Fähndrich. 2001. Type-base Flow Analysis: From Polymorphic Subtyping to CFL-reachability. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 54–66.

Thomas Reps. 1995a. Shape Analysis As a Generalized Path Problem. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '95)*. ACM, 1–11.

Thomas Reps. 1997. Program Analysis via Graph Reachability. In *Proceedings of the 1997 International Symposium on Logic Programming (ILPS)*. 5–19.

Thomas Reps. 2000. Undecidability of Context-sensitive Data-dependence Analysis. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 162–186.

Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*. ACM, New York, NY, USA.

Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. 1994. Speeding Up Slicing. *SIGSOFT Softw. Eng. Notes* 19, 5 (1994), 11–20.

Thomas W. Reps. 1995b. *Demand Interprocedural Program Analysis Using Logic Databases.* Springer US, Boston, MA, 163–196. https://doi.org/10.1007/978-1-4615-2207-2_8

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

Lei Shang, Xinwei Xie, and Jingling Xue. 2012. On-demand Dynamic Summary-based Points-to Analysis. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. ACM, 264–274.

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, Article 48 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290361

Manu Sridharan and Rastislav Bodík. 2006. Refinement-based Context-sensitive Points-to Analysis for Java. *SIGPLAN Not.* 41, 6 (2006), 387–400.

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven Points-to Analysis for Java. In *OOPSLA*.

Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 32–41. https://doi.org/10.1145/237721.237727

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* (Singapore, Singapore) *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 320–331. https://doi.org/10.1145/2970276.2970298

Hao Tang, Di Wang, Yingfei Xiong, Lingming Zhang, Xiaoyin Wang, and Lu Zhang. 2017. Conditional Dyck-CFL Reachability Analysis for Complete and Efficient Library Summarization. In *Programming Languages and Systems*, Hongseok Yang (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 880–908.

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)*. 83–95.

Tom Tseng. 2020. Dynamic connectivity data structure by Holm, de Lichtenberg, and Thorup. https://github.com/tomtseng/dynamic-connectivity-hdt.

Jyothi Vedurada and V. Krishna Nandivada. 2019. Batch Alias Analysis. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 936–948. https://doi.org/10.1109/ASE.2019.00091

Guoqing Xu, Atanas Rountev, and Manu Sridharan. 2009. Scaling CFL-Reachability-Based Points-To Analysis Using Context-Sensitive Must-Not-Alias Analysis. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming (Genoa)*. 98–122.

Dacong Yan, Guoqing Xu, and Atanas Rountev. 2011. Demand-driven Context-sensitive Alias Analysis for Java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA)*. 155–165.

Mihalis Yannakakis. 1990. Graph-theoretic Methods in Database Theory. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*. 230–242.

Hao Yuan and Patrick Eugster. 2009. An Efficient Algorithm for Solving the Dyck-CFL Reachability Problem on Trees. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009 (ESOP)*. 175–189.

Frank Kenneth Zadeck. 1984. Incremental Data Flow Analysis in a Structured Program Editor. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction* (Montreal, Canada) *(SIGPLAN '84)*. Association for Computing Machinery, New York, NY, USA, 132–143. https://doi.org/10.1145/502874.502888

Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. 2013. Fast Algorithms for Dyck-CFL-reachability with Applications to Alias Analysis *(PLDI)*. ACM.

Qirun Zhang and Zhendong Su. 2017. Context-Sensitive Data-Dependence Analysis via Linear Conjunctive Language Reachability. *SIGPLAN Not.* 52, 1 (Jan. 2017), 344–358. https://doi.org/10.1145/3093333.3009848

Xin Zheng and Radu Rugina. 2008. Demand-driven Alias Analysis for C. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, 197–208.

Received 2023-07-11; accepted 2023-11-07