Indexing the Extended Dyck-CFL Reachability for Context-Sensitive Program Analysis

QINGKAI SHI, Ant Group, China
YONGCHAO WANG, The Hong Kong University of Science and Technology, China
PEISEN YAO, The Hong Kong University of Science and Technology, China
CHARLES ZHANG, The Hong Kong University of Science and Technology, China

Many context-sensitive dataflow analyses can be formulated as an extended Dyck-CFL reachability problem, where function calls and returns are modeled as partially matched parentheses. Unfortunately, despite many works on the standard Dyck-CFL reachability problem, solving the extended version is still of quadratic space complexity and nearly cubic time complexity, significantly limiting the scalability of program analyses. This paper, for the first time to the best of our knowledge, presents a cheap approach to transforming the extended Dyck-CFL reachability problem to conventional graph reachability, a much easier and well-studied problem. This transformation allows us to benefit from recent advances in reachability indexing schemes, making it possible to answer any reachability query in a context-sensitive dataflow analysis within almost constant time plus only a few extra spaces. We have implemented our approach in two common context-sensitive dataflow analyses, one determines pointer alias relations and the other tracks information flows. Experimental results demonstrate that, compared to their original analyses, we can achieve orders of magnitude ($10^2\times$ to $10^5\times$) speedup at the cost of only a moderate space overhead. Our implementation is publicly available.

CCS Concepts: • **Mathematics of computing → Graph algorithms**; • **Theory of computation → Program analysis**; • **Software and its engineering → Automated static analysis**.

Additional Key Words and Phrases: Dyck-CFL reachability, context sensitivity, reachability indexing scheme.

ACM Reference Format:
Qingkai Shi, Yongchao Wang, Peisen Yao, and Charles Zhang. 2022. Indexing the Extended Dyck-CFL Reachability for Context-Sensitive Program Analysis. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 176 (October 2022), 31 pages. https://doi.org/10.1145/3563339

1 INTRODUCTION

The problem of context-free-language (CFL) reachability is a generalization of the problem of conventional graph reachability [Yannakakis 1990]. A vertex v is CFL-reachable from a vertex u if and only if there is a path from the vertex u to the vertex v, and the string of the edge labels follows a given context-free grammar. CFL reachability has been broadly used in program analysis for a wide range of applications, including context-sensitive dataflow analysis [Reps et al. 1995], program slicing [Reps et al. 1994], shape analysis [Reps 1995], type-based flow analysis [Kodumal and Aiken 2004; Milanova 2020; Pratikakis et al. 2006; Rehof and Fähndrich 2001], pointer analysis [Pratikakis et al. 2006; Shang et al. 2012; Sridharan et al. 2005; Xu et al. 2009; Yan et al. 2011; Zhang et al. 2013, 2014; Zheng and Rugina 2008], and debugging [Cai et al. 2018], to name just a few.

Authors' addresses: Qingkai Shi, Ant Group, China, qingkai.sqk@antgroup.com; Yongchao Wang, The Hong Kong University of Science and Technology, China, ywanghz@cse.ust.hk; Peisen Yao, The Hong Kong University of Science and Technology, China, pyao@cse.ust.hk; Charles Zhang, The Hong Kong University of Science and Technology, China, charlesz@cse.ust.hk.

2475-1421/2022/10-ART176
https://doi.org/10.1145/3563339

This paper focuses on the problem of context-sensitive dataflow analysis, where an extended Dyck-CFL is used to capture the paired function calls and returns using matched parentheses. The Dyck-CFL is "extended" because the standard Dyck-CFL fails to capture many valid dataflows that contain partially matched parentheses [Kodumal and Aiken 2004]. Intuitively, the extended Dyck-CFL includes all sub-strings of a standard Dyck word. As an example, the graph in Figure 1 demonstrates propagation of values. In the graph, edges labeled by $[\![_\kappa$ and $]\!]_\kappa$ respectively represent inter-procedural assignments via function calls and function returns at Line κ. An edge labeled by ϵ represents an intra-procedural assignment. The vertex j is context-sensitively reachable from the vertex b because the label string of the path, $]\!]_8]\!]_{19}$, does not contain any mismatched parentheses and, thus, is a sub-string of a standard Dyck word, $[\![_{19}[\![_8]\!]_8]\!]_{19}$. In contrast, the vertex j is not context-sensitively reachable from the vertex g because the label string of the path, $[\![_{17}[\![_7]\!]_7[\![_8]\!]_8]\!]_{19}$, contains mismatched parentheses. To distinguish from the standard Dyck-CFL reachability problem, we refer to the extended version as the extended Dyck-CFL reachability problem, or the context-sensitive reachability (CS-reachability) problem as it is specially used in context-sensitive dataflow analysis.

Recently, the problem of Dyck-CFL reachability has been extensively studied in two directions. First, some fast algorithms have been proposed to address the "standard" Dyck-CFL reachability problem on a few special graphs, such as trees [Yuan and Eugster 2009; Zhang et al. 2013], bidirected graphs [Chatterjee et al. 2017; Zhang et al. 2013], and graphs of constant tree-width [Chatterjee et al. 2017]. Second, there also have been studies on the problem of interleaved Dyck-CFL reachability, which use two "standard" Dyck-CFLs to formulate context- and field-sensitivity at the same time [Li et al. 2020, 2021; Zhang and Su 2017]. On one hand, works in both directions focus on "standard" Dyck-CFLs and do not address the CS-reachability problem. On the other hand, while the formulation of interleaved Dyck-CFL reachability provides a way to simultaneously model context and field sensitivity, it is essentially not a CFL-reachability problem, known to be undecidable, and does not have a precise solution [Kjelstrøm and Pavlogiannis 2022; Reps 2000].[1] In practice, to precisely answer CS-reachability queries, most context-sensitive dataflow analyses, including alias analysis [Li et al. 2011, 2013], information-flow analysis [Arzt et al. 2014; Lerch et al. 2014], and all other IFDS-based dataflow analyses, still rely on the typical tabulation algorithm [Reps et al. 1995, 1994], either in an exhaustive or a demand-driven manner. The exhaustive manner computes a transitive closure and allows for answering any query in constant time. However, it is of at least quadratic complexity in general to compute a transitive closure, which is not affordable when analyzing large-scale software. The demand-driven manner traverses the graph for every reachability query and, thus, is not efficient at responding to every single query.

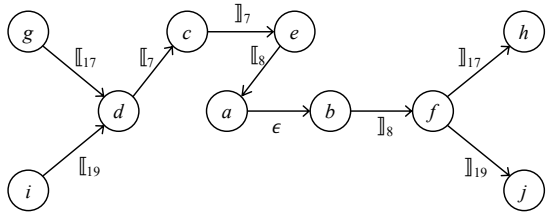
In this work, we make no claims of any breakthroughs to the innate $O(n^3/\log n)$ time complexity and $O(n^2)$ space complexity of general Dyck-CFL reachability problems. However, we note that the graphic code representations used in context-sensitive dataflow analyses often exhibit a modular structure, which makes it easy to achieve significant speedup by transforming CS-reachability to conventional graph reachability, a much easier and well-studied problem. Such a transformation is made by building a common directed graph, namely the indexing graph, without any overhead except for a standard and cheap preprocessing procedure that builds the so-called summary edges. Then, we can answer any CS-reachability query by checking a conventional reachability relation on the indexing graph. Thanks to the recent advances in the field of graph databases, a huge number of indexing schemes [Cheng et al. 2013; Jin et al. 2011, 2009; Wang et al. 2006; Yildirim et al. 2010] can be directly employed to answer the conventional reachability queries, as well as

[1]While we focus on the non-interleaved Dyck-CFL to address the context-sensitivity problem alone, our approach can be used in many context- and field-sensitive program analyses. This is because the interleaved Dyck-CFL is just one of the many ways to formulate both context and field sensitivity. We will discuss more in Section 4.

```
1.  int* foo(int* a) {
2.      int* b = a;
3.      return b;
4.  }
5.
6.  int* qux(int* d) {
7.      int* e = bar(d);
8.      int* f = foo(e);
9.      return f;
10. }
```

(a)

```
11. int* bar(int* c) {
12.     return c;
13. }
14.
15. void main() {
16.     int* g = malloc(…);
17.     int* h = qux(g);
18.     int* i = malloc(…)
19.     int* j = qux(i)
20. }
```



(b)

Fig. 1. The value-flow graph of the code snippet. The edges, (u, v), labeled by $[\![_\kappa$ and $]\!]_\kappa$ represent assignments from the variable u to the variable v via function call and function return at Line κ, respectively. An edge labeled by ϵ represents an intra-procedural assignment.

the CS-reachability queries due to the equivalent transformation, in almost constant time without computing an expensive transitive closure or performing full graph traversals. Basically, these indexing schemes pre-compute and store some cheap intermediate reachability information, which is known as the reachability index, to enable "almost" constant querying time. We will discuss the details of reachability indexing schemes in the next section.

In addition to speeding up CS-reachability queries, it has many other practical merits that motivate us to use indexing and other graph database techniques. First, when analyzing modern large-scale software, it becomes common to use a graph database to store intermediate program analysis results for reuse [Urma and Mycroft 2015; Yamaguchi et al. 2014]. When using a graph database to manage program analysis results, it is natural to use graph database techniques for optimization. Second, indexing-like techniques are widely available as graph database techniques have been well studied, mature, and widely deployed. Modern graph databases often integrate many indexing techniques that we can directly use for optimization once we transform CS-reachability to conventional reachability. Third, as detailed in Section 6.2, after transforming CS-reachability to conventional reachability with our approach, many more database techniques can be directly used for further optimization and other program analysis applications.

We have implemented our indexing approach in two existing context-sensitive dataflow analyses, one for IFDS-based information-flow analysis and the other for value-flow-based alias analysis. After building the indexing graphs for the two analyses, we then apply state-of-the-art indexing schemes for conventional reachability to speed up the CS-reachability queries. In the evaluation, we conducted experiments on twelve standard benchmark programs and four open-source systems to measure the time cost of building the indexes, the space cost of storing the indexes, and the query time using the indexes. Compared to the original analyses, the analyses armed with the indexing approach achieved orders of magnitude ($10^2\times$ to $10^5\times$) speedup for answering an alias or information-flow query with only a moderate overhead to build and store the indexes. In summary, the principal contributions of this paper are three-fold and listed as follows:

- We propose an approach of almost linear time and space complexity to transforming the CS-reachability problem into the conventional graph reachability problem. To the best of our knowledge, this is the first work that shows the possibility of such a transformation.
- We implement our indexing approach in two existing typical applications of context-sensitive program analyses. Through the two applications, we also summarize the criteria for selecting a proper indexing scheme in practice.

- We evaluate the time and the space overhead of building the indexes and compare our method to conventional techniques. The results showed orders of magnitude speedup for answering CS-reachability queries with just a moderate space overhead.

The remainder of the paper is organized as follows. Section 2 reviews the background of the CS-reachability problem as well as existing indexing schemes for conventional graph reachability. In Section 3, with an example that briefly illustrates how we transform CS-reachability to conventional graph reachability, we formally establish our theory and prove its correctness. We discuss the applications of our results in Section 4 and present the evaluation results in Section 5. Section 6 surveys the related work and Section 7 concludes the paper.

2 BACKGROUND

This section introduces the background knowledge of context-sensitive reachability (Section 2.1), reviews closely related problems (Section 2.2), discusses existing indexing schemes for conventional graph reachability (Section 2.3), and states the core problem we try to address (Section 2.4).

2.1 Context-Sensitive Reachability

In this paper, we study the all-pairs version of an extended Dyck-CFL reachability problem. It is often used in context-sensitive dataflow analysis and, thus, is referred to as the context-sensitive reachability problem. By "all-pairs", we mean that we do not focus on the reachability relations between some specific vertices but aim to efficiently reply to reachability queries between any vertex pair. The flow graphs, including the program dependence graph [Ferrante et al. 1987], the value-flow graph [Cherem et al. 2007; Sui et al. 2014], the exploded super graph [Reps et al. 1995], and many others, can be uniformly defined as a program-valid graph, which captures the modular program structure [Chatterjee et al. 2017].

Definition 2.1 (Program-Valid Graph). Given an alphabet $\Sigma_k = \{\epsilon\} \cup \{[\![_i,]\!]_i\}_{i=1}^{k}$, a program-valid graph G is a Σ_k-labeled directed graph that can be partitioned to sub-graphs such that every sub-graph has only ϵ-labeled edges, and there exists a constant $\alpha \geq 0$ such that every sub-graph has α or fewer vertices with $[\![_i$-labeled incoming edges or $]\!]_i$-labeled outgoing edges.

Intuitively, a sub-graph of the program-valid graph represents a function of a program. The constant α indicates that every function has only a few function parameters and return values. For example, the graph in Figure 1 is program-valid because it can be partitioned into four parts, $\{a, b\}$, $\{c\}$, $\{d, e, f\}$, and $\{g, h, i, j\}$. The four parts correspond to the four functions, *foo*, *bar*, *qux*, and *main*, respectively. Each part has at most two vertices with $[\![_i$-labeled incoming edges or $]\!]_i$-labeled outgoing edges. From now on, given a program-valid graph, we use V to represent the vertex set, $E \subseteq V \times V$ the edge set, and $L(u, v) \in \Sigma_k$ the label of an edge $(u, v) \in E$.

Definition 2.2 (Context-Sensitive Reachability). Given two vertices v_0 and v_m on a program-valid graph, we say the vertex v_m is context-sensitively reachable (or CS-reachable) from the vertex v_0 if and only if there is a path $(v_0, v_1, v_2, \ldots, v_m)$ on the graph such that the concatenation of the edge labels, $L(v_0, v_1)L(v_1, v_2) \ldots L(v_{m-1}, v_m)$, can be derived from the extended Dyck-CFL grammar shown in Figure 2.

By definition, the context-free grammar allows the following three kinds of CS-reachable paths on the program-valid graph. In other words, to answer a CS-reachability query, we need to check if there is a P-path, N-path, or PN-path between two vertices:

(1) P-Path. The label string of a P-path can be derived from the symbol P of the grammar. A parenthesis on a P-path is either a right-parenthesis or correctly matched. In a program

$$\begin{array}{lcl} S & \rightarrow & P\ N \\ P & \rightarrow & M\ P \mid]_i\ P \mid \epsilon \\ N & \rightarrow & M\ N \mid [_i\ N \mid \epsilon \\ M & \rightarrow & [_i\ M\ ]_i \mid M\ M \mid \epsilon \end{array}$$

Fig. 2. The context-free grammar of an extended Dyck-CFL, which is defined on the alphabet $\Sigma_k = \{\epsilon\} \cup \{[_i,]_i\}_{i=1}^{k}$ for achieving context-sensitivity [Kodumal and Aiken 2004]. The grammar of the standard Dyck-CFL only has the last production, M, that produces perfectly matched parentheses.

analysis, a P-path often represents the propagation of a dataflow fact from a callee function to a caller function. For instance, the path (b, f, h), labeled by $]_8]_{17}$, in Figure 1 is a P-path.

(2) N-Path. The label string of an N-path can be derived from the symbol N of the grammar. A parenthesis on an N-path is either a left-parenthesis or correctly matched. In a program analysis, an N-path often represents the propagation of a dataflow fact from a caller function to a callee function. For instance, the path (g, d, c), labeled by $[_{17}[_7$, in Figure 1 is an N-path.

(3) PN-Path. A PN-path is the concatenation of a P-paths and an N-path, which implies that a dataflow fact returned from a callee function is passed again to a callee function. For instance, the path (c, e, a), labeled by $]_7[_8$, in Figure 1 is a PN-path, where the prefix (c, e) is a P-path and the suffix (e, a) is an N-path.

2.2 State of The Arts

In this subsection, we discuss four groups of recent works closely related to Dyck-CFL reachability and explain their difficulties in addressing the all-pairs CS-reachability problem.

2.2.1 Standard Dyck-CFL Reachability on Bidirected Graphs. A common use of Dyck-CFL reachability is to resolve field-sensitive pointer relations on a bidirected graph. Different from the program-valid graph, in a bidirected graph as illustrated in Figure 3(a), each edge (u, v) labeled by a left-parenthesis corresponds to an inverse edge (v, u) labeled by a right-parenthesis. Zhang et al. [2013] and Chatterjee et al. [2017] show that a transitive closure can be computed in almost linear time for standard Dyck-CFL reachability over a bidirected graph. These recent works focus on bidirected graphs, standard Dyck-CFL reachability, and field sensitivity, whereas we focus on program-valid graphs, CS-reachability, and context sensitivity. Thus, we study a different problem and they do not address the CS-reachability problem studied in the paper.

2.2.2 Interleaved Standard Dyck-CFL Reachability. When modeling both field sensitivity and context sensitivity, one may use a pair of standard Dyck-CFL grammars together, one for field sensitivity and the other for context sensitivity. For instance, when $[\![$ and $(\!|$ in Figure 3(b) belong to two Dyck-CFL grammars, the last vertex is reachable from the first in terms of interleaved Dyck-CFL reachability but not reachable from the first vertex in terms of Dyck-CFL reachability. However, on one hand, recent approaches to interleaved Dyck-CFL reachability only work for "standard" Dyck-CFLs and do not address the "extended" Dyck-CFL reachability problem studied in this paper; on the other hand, the interleaved Dyck-CFL reachability problem essentially is not a CFL-reachability problem and is known to be undecidable [Kjelstrøm and Pavlogiannis 2022; Reps 2000]. Thus, only approximation algorithms can be provided [Späth et al. 2019; Zhang and Su 2017]. In this paper, we will not discuss these approaches as we focus on the exact algorithms for CFL-reachability, and the problem of interleaved Dyck-CFL reachability is out of this scope. Nevertheless, in Section 4, we will detail how our approach is used in a dataflow analysis that requires both context sensitivity and field sensitivity.

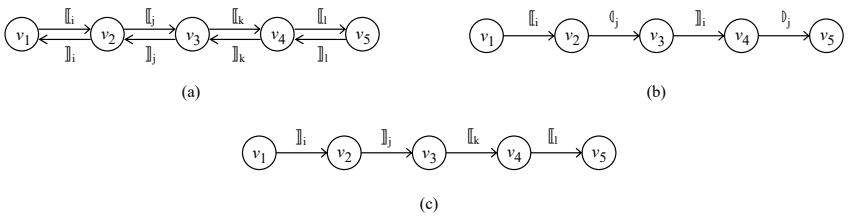


Fig. 3. Examples to show the differences between the problem studied in the paper and the state of the arts. (a) Bidirected graph ≠ Program-valid graph. (b) The vertex v_5 is not reachable from the vertex v_1 in terms of Dyck-CFL reachability but reachable in terms of the interleaved Dyck-CFL reachability. (c) The vertex v_i is not reachable from the vertex $v_j (j < i)$ in terms of the standard Dyck-CFL reachability but reachable in terms of the extended Dyck-CFL reachability, or CS-reachability.

2.2.3 Standard Dyck-CFL Reachability on Program-Valid Graphs. The most recent and closely related work is the one proposed by Chatterjee et al. [2017]. By utilizing the "constant tree-width" feature of the program-valid graphs, they propose an algorithm of linear complexity to compute the transitive closure for the standard Dyck-CFL reachability problem. This means that their approach only supports the production M in Figure 2 and does not support the others. Our work supports all productions in Figure 2 and, thus, are expected to be useful in more application scenarios. As illustrated in Figure 3(c), each vertex v_i is CS-reachable from each vertex $v_j (j < i)$ but Chatterjee et al. [2017]'s approach will miss all these CS-reachability relations.

2.2.4 Extended Dyck-CFL Reachability (CS-Reachability) on Program-Valid Graphs. As discussed above, while there have been many works related to Dyck-CFL reachability, few of them address the CS-reachability problem in an efficient manner. In practice, to fully address the CS-reachability problem, we still need to follow two classic approaches. On one hand, we can build a transitive closure, which can provide constant query time for CS-reachability but is notoriously expensive due to the sub-cubic, i.e., $O(|V|^3/\log|V|)$, complexity [Chaudhuri 2008]. To avoid confusion, we would like to stress again that, for computing the transitive closure of a Dyck-CFL reachability problem, existing linear algorithms are proposed either for bidirected graphs (Section 2.2.1) or for the standard Dyck-CFL reachability problem (Section 2.2.3), not for the CS-reachability problem over a program-valid graph. Thus, for CS-reachability, we still have to use the traditional $O(|V|^3/\log|V|)$ algorithm to compute the transitive closure.

On the other hand, we can choose not to build an expensive transitive closure but, instead, for every single CS-reachability query, we traverse the input program-valid graph using the traditional tabulation algorithm [Reps et al. 1995, 1994]. Basically, the tabulation algorithm performs a depth-first traversal over the input program-valid graph to answer a reachability query. What makes it different from a depth-first traversal is that it uses summary edges to avoid repetitively visiting a function body. As defined in Definition 2.3, a summary edge tabulates an input and an output of a function. For instance, in Figure 1, we can add a summary edge from the vertex e to the vertex f, which are the input and output of the function *foo* at Line 8. This summary edge allows us to skip the function foo whenever a graph traversal reaches the vertex e. Since we never visit a function more than once, answering a CS-reachability query using the tabulation algorithm is of linear complexity with respect to the graph size and the number of summary edges. Reps et al. [1994] showed that the number of summary edges is bounded by $O(\alpha^2|V|)$ and the time of building all summary edges is bounded by $O(\alpha|E| + \alpha^3|V|)$. As mentioned before, α is a constant. Thus, the complexity is linear in practice. More recently, Chatterjee et al. [2017] also proposed an algorithm that can build all summary edges in almost linear time.

Definition 2.3 (Summary Edge and Summary Path). A summary edge (v_0, v_m) is an extra ϵ-labeled edge added to the program-valid graph $G = (V, E)$ such that $v_0, v_m \in V$ and there is a path $(v_0, v_1, v_2, \ldots, v_m)$, which we refer to as a summary path, such that the label string is in the form of $[\![_i M]\!]_i$, where M is the symbol in the context-free grammar.

To summarize, building a transitive closure is of high complexity and the tabulation algorithm is also not efficient as it needs to traverse the input program-valid graph for every single query. To resolve the dilemma between the two extreme approaches, this paper proposes a novel use of the summary edges, which are cheap to build and allow us to answer any CS-reachability query within "almost" constant time via existing indexing schemes for conventional graph reachability.

2.3 Indexing Schemes for Conventional Graph Reachability

Quickly answering a query of conventional graph reachability has been the focus of research for over thirty years due to its wide spectrum of applications. Like the problem of CS-reachability, in general, answering conventional reachability queries also relies on two "extreme approaches". The first can answer any query in $O(1)$ time but needs to pre-compute an expensive transitive closure. The other extreme approach replies to a query by traversing the graph by depth-first or breadth-first search. This approach takes linear time for each query and, thus, is apparently very slow for frequent queries on a large graph. Fortunately, recent studies on indexing schemes have found promising trade-offs lying in-between the two extremes — pre-computing and storing some cheap intermediate reachability information, which is known as the reachability index, to enable "almost" constant querying time. The state-of-the-art indexing schemes can be put into two groups:

(1) **Compression of transitive closure**, e.g., [Chen and Chen 2008; Cohen et al. 2003; Jin et al. 2011, 2009; Wang et al. 2006]. Approaches in this group aim to reduce the cost of computing and storing the transitive closure. For instance, assuming k is a variable far less than $|V|$, the dual-labeling method takes $O(|V| + |E| + k^3)$ time to compress the size of transitive closure from $O(|V|^2)$ to $O(|V| + k^2)$ but preserves the capability of answering each reachability query in constant time [Wang et al. 2006].

(2) **Pruned search**, e.g., [Chen et al. 2005; Seufert et al. 2013; Wei et al. 2014; Yildirim et al. 2010]. Approaches in this group pre-compute information to speed up the depth-first or breadth-first graph traversal by pruning unnecessary searches. Grail is such an indexing scheme that labels each vertex with a constant number of intervals [Yildirim et al. 2010]. We then can tell if a vertex is NOT reachable from another by testing the interval containment in constant time. In other cases, it falls back to a graph traversal but can use the intervals to prune unreachable paths.

To show more insights into these indexing schemes, we provide two detailed examples, which are put at the end of this paper (see Appendix A) to avoid distraction from our main focus. Nevertheless, in the next section, we will provide a detailed example to illustrate how an indexing scheme for conventional graph reachability speeds up CS-reachability queries.

We would like to clarify that, in addition to the indexing schemes that aim to speed up reachability queries, there are also many other indexing schemes not for reachability queries. For instance, the traditional B-tree indexing schemes break a database into many small blocks so that the scope of a query can be limited in small blocks, thus reducing the time of querying some specific data (rather than reachability relations between data) [Comer 1979]. Weiss et al. [2008] proposed an indexing scheme to manage data from the web and applied the indexing scheme to program analysis [Weiss et al. 2015]. Their indexing scheme is not for accelerating reachability queries and CFL-reachability queries. Thus, their approach is not related to ours.

2.4 Problem Statement

The reachability indexing schemes discussed above can easily accelerate conventional reachability queries on a common directed graph. However, owing to the constraint of the context-free grammar, they cannot speed up CS-reachability queries on a program-valid graph, unless we can address the following problem, i.e., reduce the CS-reachability problem to a conventional reachability problem:

> Given a program-valid graph $G = (V, E)$, find a common directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and two functions $src : V \mapsto \mathcal{V}$ and $dst : V \mapsto \mathcal{V}$, such that, for any pair of vertices, $u, v \in V$, the vertex v is CS-reachable from the vertex u if and only if the vertex $dst(v)$ is reachable from the vertex $src(u)$ on the common directed graph $\mathcal{G}$.

In what follows, we introduce an efficient transformation of almost linear complexity from CS-reachability to conventional graph reachability, which allows us to directly profit from the reachability indexing schemes discussed in the last subsection.

3 TRANSFORMING CS-REACHABILITY TO CONVENTIONAL REACHABILITY

With an example that briefly illustrates our approach (Section 3.1), we then discuss the intuition of the correctness (Section 3.2) and provide formal proofs (Section 3.3). In the end, we discuss an optimization, which allows us to transform CS-reachability to conventional reachability without any structural change of the input program-valid graph (Section 3.4).

3.1 Flare in a Nutshell

Figure 4(a) shows a program-valid graph that describes the value propagation in the code snippet — each directed edge (u, v) represents an assignment from the variable u to the variable v. An edge is labeled by $[\![_i$ if the assignment happens at the time of the function call, i.e., passing an actual parameter to a formal parameter at the ith call site. An edge is labeled by $]\!]_i$ if the assignment happens when the function call returns, i.e., passing a return value to its receiver at the ith call site. A summary edge is added over the call site $d = baz(b)$, connecting the input b and the output d. Next, we use the example to show how our approach, namely Flare, transforms a CS-reachability query on the program-valid graph into an equivalent conventional reachability query on a common directed graph that we referred to as the indexing graph. After the transformation, we can directly use existing indexing schemes to quickly answer the conventional reachability queries and, thus, the equivalent CS-reachability queries. Next, we illustrate (1) how we build the indexing graph, (2) how we transform a CS-reachability query to an equivalent query of conventional reachability, and (3) how a typical indexing scheme for conventional reachability speeds up a CS-reachability query.

3.1.1 Building the Indexing Graph. Figure 4(b) shows the indexing graph, which consists of two copies of the original program-valid graph. The copies of each vertex v are distinguished by the subscripts v_1 and v_2. In the first copy, we remove all edges labeled by the left-parentheses, i.e., the call edges. In the second copy, we remove all edges labeled by the right-parentheses, i.e., the return edges. For each vertex v, we add an edge from its first copy v_1 to its second copy v_2. All edge labels are removed from the indexing graph.

Although, in this example, we show that we need to copy the input program-valid graph to build the indexing graph, we show in Section 3.4 that such copies are not necessary. Thus, the only overhead of building an indexing graph is to compute the summary edges, which, as discussed before, can be built in almost linear time and, thus, is very cheap.

3.1.2 Querying CS-Reachability. To tell if a vertex v is CS-reachable from a vertex u in Figure 4(a), we only need to tell if the vertex v_2 is reachable from the vertex u_1 in the indexing graph, i.e.,

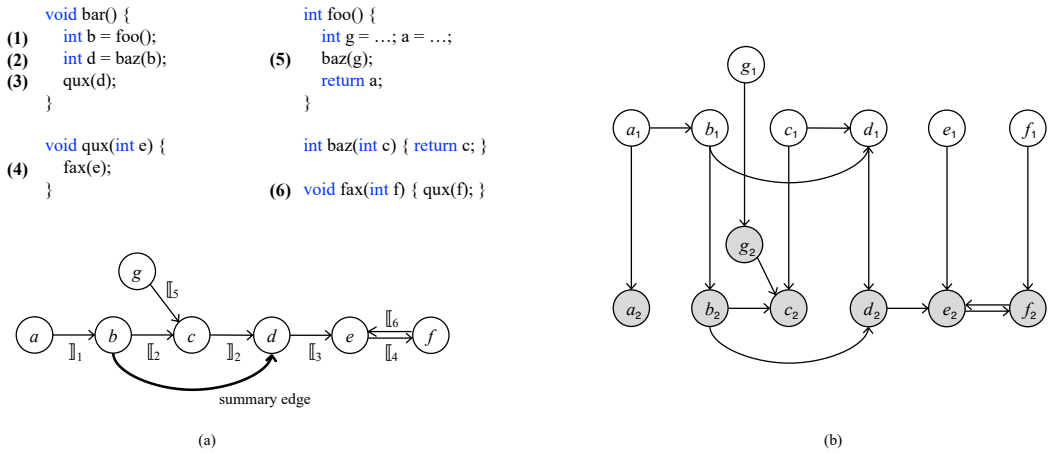


Fig. 4. (a) A code snippet and its program-valid graph with a summary edge. (b) The indexing graph we build to transform CS-reachability to the conventional graph reachability.

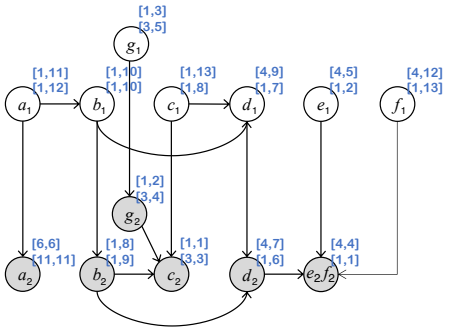
Figure 4(b). Note that all edge labels have been intentionally removed in the indexing graph as we only check conventional reachability in the indexing graph. Two examples are illustrated below.

(1) To tell if the vertex f is CS-reachable from the vertex a in Figure 4(a), we only need to check if the vertex f_2 is reachable from the vertex a_1 in Figure 4(b). Since there is a path from the vertex a_1 to the vertex f_2 in Figure 4(b), the vertex f is CS-reachable from the vertex a in Figure 4(a). Let us check the CS-reachability relation on the original program-valid graph. We can find a PN-path (a, b, c, d, e, f) labeled by the string, $]_1[_2]_2[_3[_4$, which satisfies the context-free grammar of CS-reachability.

(2) To tell if the vertex f is CS-reachable from the vertex g in Figure 4(a), we only need to check if the vertex f_2 is reachable from the vertex g_1 in Figure 4(b). Since there is not any path connecting the vertex f_2 and the vertex g_1, the vertex f is not CS-reachable from the vertex g. Let us check the CS-reachability relation on the original program-valid graph. We can find that the label strings of the paths between the vertex g and the vertex f always contain a pair of mismatched parentheses, i.e. $[_5]_2$, thus not satisfying the context-free grammar.

3.1.3 Speeding up Reachability Queries via Indexing. We have shown that a CS-reachability query over the program-valid graph can be transformed to a conventional reachability query over the indexing graph. Thus, we can employ existing indexing schemes for conventional reachability to speed up CS-reachability queries. Let us use the Grail indexing scheme [Yildirim et al. 2010] as an example. Basically, Grail labels each vertex in a common directed graph to speed up reachability queries. More detailed examples of Grail and other indexing schemes can be found in the appendix.

Given the indexing graph shown in Figure 4(b), which is a common directed graph, the first step of Grail is to find all strongly connected components (SCC) in the graph and merge each SCC into a single vertex. As illustrated in Figure 5(a), we merge the vertex e_2 and the vertex f_2 into a single vertex e_2f_2, thus forming a directed acyclic graph (DAG). This step is cheap as finding all SCCs is of linear complexity [Tarjan 1972]. This step does not affect reachability queries because vertices in an SCC are reachable from each other and any reachability query related to a vertex in an SCC will be transformed into a reachability query related to the merged vertex. For instance, to test if the vertex f_2 is reachable from the vertex g_1 in Figure 4(b), it is equivalent to test if the vertex e_2f_2 is reachable from the vertex g_1 in Figure 5(a).



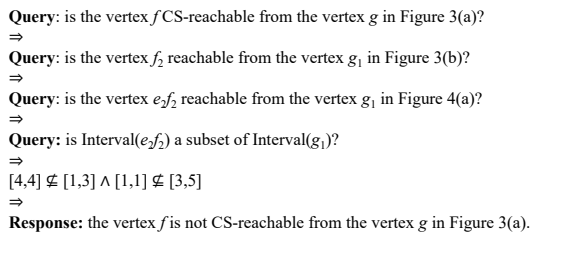


Fig. 5. (a) The indexing graph labeled by a pair of intervals generated by Grail, an indexing scheme for conventional reachability [Yildirim et al. 2010]. (b) The steps of answering a CS-reachability query.

The second step of Grail is to label each vertex in the DAG, i.e., Figure 5(a), with multiple intervals. Each interval $[L_v, H_v]$ of a vertex v is labeled by a post-order traversal on the DAG. For each interval, H_v is the rank of the vertex v in a post-order traversal, where the ranks are assumed to begin at 1; L_v denotes the lowest rank for any vertex in the sub-graph rooted at the vertex v. To produce multiple intervals, Grail performs the post-order traversal multiple times and, at each time, randomizes the children's order of each vertex. This step is also of linear complexity.

The intervals produced by Grail allow us to answer many CS-reachability queries in constant time. Figure 5(b) shows the steps of checking if the vertex f is CS-reachable from the vertex g in the original program-valid graph. This CS-reachability query is transformed to a conventional reachability query between f_2 and g_1 in Figure 4(b), then a conventional reachability query between e_2f_2 and g_1 in Figure 5(a), and eventually an interval containment query. Since the interval of e_2f_2 is not subsumed by that of g_1, we can immediately conclude that e_2f_2 is not reachable from g_1 and, hence, f is not CS-reachable from g in the original program-valid graph.

To sum up, the example shows that an indexing scheme for conventional graph reachability can speed up CS-reachability queries in the sense that, it allows us to answer CS-reachability queries in constant time without computing an expensive transitive closure, which is of quadratic space complexity and nearly cubic time complexity. Instead, we may only need linear time and space to compute and store the intervals.

3.2 Intuition of the Correctness

Using the previous example, we now discuss the intuition of why the transformation from CS-reachability to conventional reachability is correct. The discussion serves as a warm-up construction for our formalization in the next subsection.

One key preprocessing procedure of our approach is that we utilize the modular structure of a program-valid graph to build the summary edges, within $O(\alpha|E| + \alpha^3|V|)$ [Reps et al. 1994]. This complexity is cubic in α and is consistent with the innate cubic complexity of general CFL-reachability problems. However, as noted before, α denotes the number of function parameters and return values in program analysis and, thus, is a constant. This means that the summary edges can be built very efficiently in almost linear time. There have been other works, e.g., [Chatterjee et al. 2017], that demonstrate that summary edges can be built in almost linear time and, thus, is very cheap. Thus, building summary edges is not a limitation of our approach.

The rationale behind the reduction is that, each CS-reachable path, i.e., P-path, N-path, or PN-path, on the program-valid graph corresponds to a path on the indexing graph. First, we explain

that each P-path corresponds to a path in the first copy of the program-valid graph. Recall that (1) each summary edge represents a path labeled by $[\![_i M]\!]_i$, and (2) the production of P-paths can be rewritten as $P \rightarrow ([\![_i M]\!]_i \mid \epsilon)^* P \mid]\!]_i P \mid \epsilon$. This production implies that a P-path is composed of sub-paths labeled by $[\![_i M]\!]_i$ and edges labeled by $]\!]_i$ or ϵ. After adding summary edges, since each sub-path labeled by $[\![_i M]\!]_i$ can be replaced by a summary edge, a P-path then can be represented as a path only labeled by $]\!]_i$ and ϵ. Based on this observation, in the first copy of the program-valid graph, we remove all edges labeled by $[\![_i$ and only preserve edges labeled by $]\!]_i$ and ϵ, so that each path in the copy represents a P-path in the original program-valid graph. For instance, in Figure 4(b), after replacing the sub-path (b, c, d) in the P-path (a, b, c, d) with a summary edge (b, d), the path is represented by the path (a_1, b_1, d_1) in the first copy.

Similarly, in the second copy of the program-valid graph, we only preserve edges labeled by $[\![_i$ and ϵ, so that each path in the copy represents an N-path in the original program-valid graph. Since a PN-path is a path with a P-path as its prefix and an N-path as its suffix, in Figure 4(b), we connect the two copies so that a path from the first copy to the second copy represents a PN-path. For instance, the PN-path (a, b, c, d, e, f) on the program-valid graph can be split into two sub-paths, the P-path (a, b, c, d) and the N-path (d, e, f), which respectively correspond to the path (a_1, b_1, d_1) and the path (d_2, e_2, f_2), connected by (d_1, d_2), on the indexing graph.

To sum up, each path on the indexing graph corresponds to a P-path, N-path, or PN-path in the program-valid graph. Thus, we can safely transform CS-reachability queries on the program-valid graph into conventional reachability queries on the indexing graph. We note that using summary edges to optimize CS-reachability algorithms can be dated back to 1990s [Reps 1998; Reps et al. 1995, 1994]. Nevertheless, in this work, for the first time to the best of our knowledge, we show the possibility of transforming the CS-reachability problem to a conventional reachability problem, which allows up to $10^5\times$ speedup as shown in our evaluation.

3.3 Formalization

We now formalize the solutions to the two key problems: how we build the indexing graph, and how we transform a CS-reachability query to an equivalent query of conventional graph reachability.

3.3.1 Building the Indexing Graph. A preprocessing step of our approach is to compute all summary edges for a given program-valid graph. Reps et al. [1994] have presented an algorithm to compute the summary edges and proved the following lemma about the cost of the algorithm.

Lemma 3.1 (Complexity of the Summary Edges [Reps et al. 1994]). *The number of summary edges is bounded by $O(\alpha^2|V|)$ and the time to build all summary edges is bounded by $O(\alpha|E| + \alpha^3|V|)$.*

Since α, which denotes the number of function parameters and return values, is a constant in program analysis, we can build summary edges with almost linear complexity in the graph size. In our evaluation, we will also show that it is cheap to compute the summary edges. With the summary edges, we can define the indexing graph as follows.

Definition 3.2 (Indexing Graph). Given a program-valid graph $G = (V, E)$ where $E = E^\epsilon \cup E^l \cup E^r$ is the union set of edges labeled by ϵ, left-parentheses, and right-parentheses, and the set E^s of summary edges, an indexing graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ can be built in two steps:

(1) Build two copies of the program-valid graph as well as the summary edges:
- $G_1 = (V_1, E_1 = E_1^\epsilon \cup E_1^r \cup E_1^s)$,
- $G_2 = (V_2, E_2 = E_2^\epsilon \cup E_2^l \cup E_2^s)$,

where the vertex set V_i is a copy of the vertex set V, and E_i^ϵ, E_i^r, E_i^l, and E_i^s are copies of E^ϵ, E^r, E^l, and E^s, respectively.

(2) Build the indexing graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$:
- $\mathcal{V} = V_1 \cup V_2$,
- $\mathcal{E} = E_1 \cup E_2 \cup \{(v_1, v_2) : v_1 \in V_1, v_2 \in V_2\}$,

where we use v_i to represent the copy of the vertex $v \in V$ in V_i.

Since we only copy the input program-valid graph and the summary edges twice, we have the following lemma showing that the size of the indexing graph is linear to the size of the original program-valid graph.

Lemma 3.3 (Complexity of the Indexing Graph). *Assuming α is a constant, the time and space complexity of building the indexing graph is $O(|V| + |E|)$.*

3.3.2 Querying CS-Reachability. The indexing graph allows us to answer CS-reachability queries according to the conventional reachability relations on the indexing graph. That is, given a program-valid graph $G = (V, E)$ and its indexing graph $\mathcal{G} = (\mathcal{V} = V_1 \cup V_2, \mathcal{E})$, to determine if a vertex $v \in V$ is CS-reachable from a vertex $u \in V$ on the program-valid graph, we only need to check if the vertex $v_2 \in V_2$ is reachable from the vertex $u_1 \in V_1$ on the indexing graph. To prove the correctness of this transformation, we need to understand the following rewriting of the extended Dyck-CFL grammar in Figure 2. First, by simply unfolding $M \rightarrow MM$, the production M can be rewritten as follows, where $*$ is the Kleene star operator:

$$M \quad \rightarrow \quad [\![_i M]\!]_i \mid MM \mid \epsilon \quad \rightarrow \quad ([\![_i M]\!]_i \mid \epsilon)*$$

Hence, the production P and the production N can be rewritten as follows by replacing the non-terminal symbol M with the right-hand-side sequence, $([\![_i M]\!]_i \mid \epsilon)*$:

$$P \quad \rightarrow \quad MP \mid]\!]_i P \mid \epsilon \quad \rightarrow \quad ([\![_i M]\!]_i \mid \epsilon) * P \mid]\!]_i P \mid \epsilon \quad \rightarrow \quad ([\![_i M]\!]_i \mid]\!]_i \mid \epsilon)*$$

$$N \quad \rightarrow \quad NM \mid N[\![_i \mid \epsilon \quad \rightarrow \quad N([\![_i M]\!]_i \mid \epsilon) * \mid N[\![_i \mid \epsilon \quad \rightarrow \quad ([\![_i M]\!]_i \mid [\![_i \mid \epsilon)*$$

By Definition 2.3, a path labeled by a string $[\![_i M]\!]_i$ is a summary path. Hence, the rewriting above allows us to redefine the P-path and N-path as follows.

Definition 3.4 (P-path). A path on the program-valid graph is a P-path if and only if it can be partitioned into multiple segments, each of which is a summary path, a $]\!]_i$-labeled edge, or an ϵ-labeled edge.

Definition 3.5 (N-path). A path on the program-valid graph is an N-path if and only if it can be partitioned into multiple segments, each of which is a summary path, a $[\![_i$-labeled edge, or an ϵ-labeled edge.

Next, we prove the correctness of our approach in two steps, i.e., the necessity and the sufficiency.

Lemma 3.6 (Necessity). *Given a program-valid graph $G = (V, E)$ and its indexing graph $\mathcal{G} = (\mathcal{V} = V_1 \cup V_2, \mathcal{E})$, if there is a CS-reachable path from $u \in V$ to $v \in V$ on the program-valid graph, then there must exist a path from $u_1 \in V_1$ to $v_2 \in V_2$ on the indexing graph.*

Proof. According to our discussion before, a CS-reachable path on the program-valid graph may be a P-path, N-path, or PN-path, which are discussed below, respectively.

(1) According to Definition 3.4, a P-path can be partitioned into multiple segments, each of which is a summary path, a $]\!]_i$-labeled edge, or an ϵ-labeled edge. By definition of the indexing graph, each of the segment $(x, \ldots, y)$ has a corresponding edge (x_1, y_1) in the first copy G_1 of the indexing graph. Hence, any P-path, $(u, \ldots, v)$, has a corresponding path, $(u_1, \ldots, v_1)$, in the indexing graph. Since $(v_1, v_2) \in \mathcal{E}$, we have the path $(u_1, \ldots, v_2)$ in the indexing graph.

(2) According to Definition 3.5, an N-path can be partitioned into multiple segments, each of which is a summary path, a $[\![_i$-labeled edge, or an ϵ-labeled edge. By definition, each of the segment $(x, \ldots, y)$ has a corresponding edge (x_2, y_2) in the second copy G_2 of the indexing graph. Hence, any N-path, $(u, \ldots, v)$, has a corresponding path, $(u_2, \ldots, v_2)$, in the indexing graph. Since $(u_1, u_2) \in \mathcal{E}$, we have the path $(u_1, \ldots, v_2)$ in the indexing graph.

(3) By definition, a PN-path, $(u, \ldots, v)$ can be split into two segments, say $(u, \ldots, x)(x, \ldots, v)$, where the prefix segment is a P-path and the suffix segment is an N-path. Based on the above discussions, we have $(u_1, \ldots, x_1)$ in G_1 and $(x_2, \ldots, v_2)$ in G_2. Since $(x_1, x_2) \in \mathcal{E}$, we have the path $(u_1, \ldots, x_1, x_2, \ldots, v_2)$ in the indexing graph. □

Lemma 3.7 (Sufficiency). *Given a program-valid graph $G = (V, E)$ and its indexing graph $\mathcal{G} = (\mathcal{V} = V_1 \cup V_2, \mathcal{E})$, if there is a path from $u_1 \in V_1$ to $v_2 \in V_2$ on the indexing graph, there must exist a CS-reachable path from $u \in V$ to $v \in V$ on the program-valid graph.*

Proof. Given a path $(u_1, \ldots, v_2)$ in the indexing graph, it must be in one of the following three forms: $(u_1, \ldots, v_1, v_2)$, $(u_1, u_2 \ldots, v_2)$, or $(u_1, \ldots, x_1)(x_1, x_2)(x_2 \ldots, v_2)$.

(1) The path is in the form of $(u_1, \ldots, v_1, v_2)$. By definition, all vertices from u_1 to v_1 are in V_1, and each edge (x_1, y_1) on the path is in $E_1^\epsilon \cup E_1^r \cup E_1^s$. Each edge $(x_1, y_1) \in E_1^\epsilon$ corresponds to an ϵ-labeled edge (x, y) in the program-valid graph. Each edge $(x_1, y_1) \in E_1^r$ corresponds to a $[\![_i$-labeled edge (x, y) in the program-valid graph. Each edge $(x_1, y_1) \in E_1^s$ corresponds to a summary path, i.e., $[\![_i M]\!]_i$-labeled path, in the program-valid graph. Hence, each path $(u_1, \ldots, v_1)$ corresponds to a path $(u, \ldots, v)$ in the program-valid graph, which is composed by ϵ-labeled edges, $[\![_i$-labeled edges, or summary paths. By Definition 3.4, the path $(u, \ldots, v)$ is a P-path.

(2) The path is in the form of $(u_1, u_2 \ldots, v_2)$. By definition, all vertices from u_2 to v_2 are in V_2, and each edge (x_2, y_2) on the path is in $E_2^\epsilon \cup E_2^l \cup E_2^s$. Each edge $(x_2, y_2) \in E_2^\epsilon$ corresponds to an ϵ-labeled edge (x, y) in the program-valid graph. Each edge $(x_2, y_2) \in E_2^l$ corresponds to a $[\![_i$-labeled edge (x, y) in the program-valid graph. Each edge $(x_2, y_2) \in E_2^s$ corresponds to a summary path, i.e., $[\![_i M]\!]_i$-labeled path, in the program-valid graph. Hence, each path $(u_2, \ldots, v_2)$ corresponds to a path $(u, \ldots, v)$ in the program-valid graph, which is composed by ϵ-labeled edges, $[\![_i$-labeled edges, or summary paths. By Definition 3.5, the path $(u, \ldots, v)$ is an N-path.

(3) The path is $(u_1, \ldots, x_1)(x_1, x_2)(x_2 \ldots, v_2)$. Based on the discussions in Case 1 and Case 2, the prefix $(u_1, \ldots, x_1)$ and the suffix $(x_2, \ldots, v_2)$ correspond to a P-path $(u, \ldots, x)$ and an N-path $(x, \ldots, v)$ in the program-valid graph, respectively. Thus, the concatenation of the two paths, i.e., $(u, \ldots, x, \ldots, v)$, is a PN-path. □

Putting Lemma 3.3, Lemma 3.6, and Lemma 3.7 together, we have the following theorem that summarizes our result.

Theorem 3.8. *The CS-reachability problem on a program-valid graph can be reduced to a conventional graph reachability problem on the indexing graph in linear time and space with respect to the size of the input program-valid graph.*

3.4 Building the Indexing Graph without Any Copy

Instead of a sophisticated algorithm, our approach simply copies the input program-valid graph twice to build the indexing graph for addressing the all-pairs CS-reachability problem. In practice, we can take a further step to make the approach simpler — we even do not need to physically copy the program-valid graph for building the indexing graph.

Our key insight is that the indexing graph shares most vertices and edges with the original program-valid graph $G = (V, E^\epsilon \cup E^r \cup E^l \cup E^s)$. Thus, we do not need to physically generate

Algorithm 1: Iterating vertices and successors of a given vertex in an indexing graph.

```
1  Procedure vertices(G = (V, E))
2      /* E = E^ϵ ∪ E^r ∪ E^l ∪ E^s */
3      foreach v ∈ V and i ∈ {1, 2} do
4          do some operation on (v, i); /* iterate all v_i ∈ 𝒱 */
5  Procedure successors(G = (V, E), (v, i))
6      /* E = E^ϵ ∪ E^r ∪ E^l ∪ E^s */ /* v ∈ V, i ∈ {1, 2} */
7      if i = 1 then
8          do some operation on (v, 2); /* (v_1, v_2) ∈ ℰ */
9          foreach (v, u) ∈ E^ϵ ∪ E^r ∪ E^s do
10             do some operation on (u, 1) /* (v_1, u_1) ∈ ℰ */
11     else
12         foreach (v, u) ∈ E^ϵ ∪ E^l ∪ E^s do
13             do some operation on (u, 2) /* (v_2, u_2) ∈ ℰ */
```

the copies, G_1 and G_2, but reuse the data structure of G and logically distinguish the copies using an extra integer in $\{1, 2\}$. That is, we do not generate the physically copied vertices $v_i \in \mathcal{V}$ but represent the vertex v_i as a pair $(v \in V, i \in \{1, 2\})$, which reuses the vertex v in the original program-valid graph. With this representation, Algorithm 1 demonstrates the basic operations over the indexing graph, i.e., iterating all vertices and iterating all successors of a given vertex, without any physically copied vertices. Since the algorithm has essentially represented the indexing graph as an adjacent list, one primary data structure for graphs, it is sufficient for implementing any graph algorithm over the indexing graph including querying reachability via the indexing schemes.

To conclude, in practice, the only overhead of building an indexing graph is to compute the summary edges. In other words, building summary edges is sufficient to transform CS-reachability to conventional reachability. While saving a copy of the program-valid graph does not affect the algorithm complexity and does not significantly increase the algorithm performance, we argue that the approach is valuable in the following two aspects:

(1) The approach has its practical value as it follows the design principle of Occam's razor, which encourages us to keep a method simple [Gauch 2002]. In practice, the approach brings two advantages. First, it makes it easy to implement the proposed approach as we no longer need to copy the graph. Second, it makes it easy for data maintenance. It is infamous that data clones increase the cost of maintenance. For example, assuming that we store two copies of a program-valid graph in a database, we then need extra effort to keep the two copies consistent when one of them is revised. Our approach avoids such data clones and, thus, eases the efforts in maintenance.

(2) This approach has its conceptual value. In the past decades, it has been widely-known that "building summary edges can transform CS-reachability to regular-language reachability" [Reps 1998; Reps et al. 1995, 1994]. Based on this observation, the classic tabulation algorithm, one of the baseline approaches in our evaluation, was proposed and is still widely used in state-of-the-art techniques. With the way of building the indexing graph without any copy, for the first time to the best of our knowledge, we now can update this statement to "building summary edges can transform CS-reachability to conventional reachability". We believe this update could push forward the development of techniques in this field because conventional reachability is much easier and well studied.

Table 1. Two applications and the indexing schemes we apply to them.

Application	IFDS-Based Information-Flow Analysis[1]	Value-Flow-Based Alias Analysis[2]
Program-Valid Graph	Exploded Super-Graph	Value-Flow Graph
State-of-the-Art Algorithm to Answer CS-Reachability Queries	Reps et al. [1995, 1994]'s Tabulation Algorithm	
Indexing Scheme	Grail [Yildirim et al. 2010]	PathTree [Jin et al. 2011]
Index Size	$O(k\|V\|)$	$O(k\|V\|)$
Indexing Time	$O(k(\|V\|+\|E\|))$	$O(k\|E\|)$
Time per Query	$O(k)$ or $O(k(\|V\|+\|E\|))$	$O(1)$ or $O(\log^2 k)$

[1] $k \leq 5$ is the times we randomly traverse the graph to build the index.
[2] k is the number of paths that can cover the input graph.

4 APPLICATIONS

Our results can speed up a wide range of context-sensitive dataflow analyses working on different program-valid graphs. We have applied our results to two common context-sensitive dataflow analyses, one is IFDS-based information-flow analysis, e.g., [Lerch et al. 2014; Schubert et al. 2019], and the other is value-flow-based alias analysis, e.g., [Li et al. 2013; Shi et al. 2018; Sui and Xue 2020], as shown in Table 1. The two kinds of analyses work on two different program-valid graphs, which are known as the *exploded super-graph* and the *value-flow graph*, respectively. In practice, there are many different techniques for building these program-valid graphs. However, for the problem studied in this paper, it is not important how to build the graphs as we focus on speeding up CS-reachability queries after a graph is given. For evaluation, we use recent techniques to build the graphs, which will be detailed in the next section.

Despite many differences, it is common for both applications to formulate their problems as CS-reachability queries and follow the same spirit of Reps et al. [1995, 1994]'s tabulation algorithm to answer CS-reachability queries. As discussed in Section 2.1, the tabulation algorithm needs to traverse the program-valid graph for answering every CS-reachability query and, thus, is inefficient. To improve the performance, as the workflow shown in Figure 6, we first build the indexing graph by adding the summary edges and compute the reachability indexes over the indexing graph. The indexing graph allows us to transform any CS-reachability query into a conventional reachability query, which then can be significantly accelerated by the reachability indexes. Given that, as discussed in Section 2.3, there are many indexing schemes we can choose, we advocate two rules that help select indexing schemes for a given program analysis.

Rule 1. *If a program analysis needs to return paths between two vertices when responding positively to a CS-reachability query, we prefer to use the pruned-search-based indexing schemes. These indexing schemes are cheaper and keep the capability of replying to queries with negative answers in almost constant time and replying queries with positive answers by searching paths within almost linear complexity in the path length.*

Rule 2. *If a program analysis does not need to return any paths when responding to a CS-reachability query, we prefer to use the indexing schemes that compress the transitive closure because they exhibit almost constant complexity, no matter whether the queries have positive or negative answers.*

4.1 Indexing the IFDS-Based Information-Flow Analysis

Information flow is the transfer of information between two variables in a given program. A valid information flow exists between two variables if and only if there is a CS-reachable path

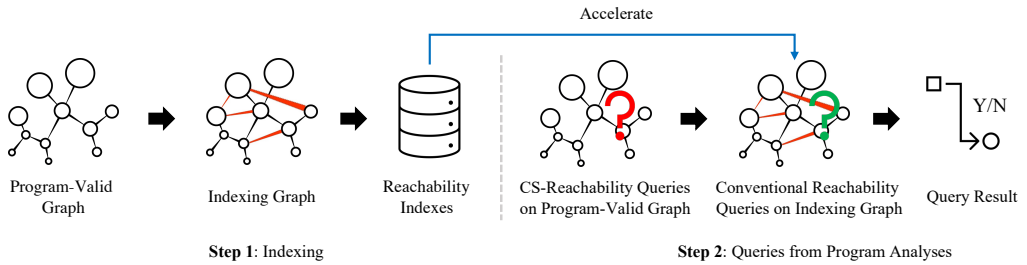


Fig. 6. Workflow of an indexed context-sensitive program analysis.

between them. When answering a CS-reachability query, this application not only needs to check the reachability relation but also needs to find one or multiple CS-reachable paths that lead to the information flow. This is critical when the information-flow analysis is used to detect security violations as we need to check the violation-triggering paths so as to fix the violations.

Hence, we follow Rule 1 to use the pruned-search-based indexing schemes as they can provide paths as the evidence of reachability. In the implementation, we use the Grail indexing scheme [Yildirim et al. 2010] (see Appendix A for an example of the indexing scheme). Grail builds the reachability index by randomly traversing an input graph k times ($k \leq 5$ in practice and we use $k = 5$ in the implementation). Grail allows us to answer most unreachable queries in $O(k)$ time and, for reachable queries, it returns a path within linear time and space with respect to the path length. Apparently, the reachability indexing scheme significantly improves the query efficiency over the original algorithm that needs to perform a full graph traversal for every query.

4.2 Indexing the Value-Flow-Based Alias Analysis

Alias analysis statically determines if two pointer variables can point to the same memory location during program execution. Two pointer variables, such as the pointer a and the pointer f in Figure 1, are aliases if and only if there exists a CS-reachable path between them in the graph. Since we often only need to answer "yes" or "no" when querying if a pointer is the alias of the other, it is not necessary to find any CS-reachable path between two pointer variables.

Hence, we follow Rule 2 to use the indexing scheme, known as PathTree [Jin et al. 2011], that compresses the transitive closure (see Appendix A for an example of the indexing scheme). Basically, PathTree partitions the input graph into k paths [Jin et al. 2011] and takes $O(k|E|)$ time to compress the size of transitive closure from $O(|V|^2)$ to $O(k|V|)$. The compressed transitive closure, a.k.a., the index, allows us to answer reachability queries in $O(1)$ time for most cases and in $O(\log^2 k)$ time for the others. This is far more efficient than a traditional approach that always performs a full graph traversal for every query.

4.3 Dealing with Field-Sensitivity

As discussed in Section 2.2, there have been a few studies on the interleaved Dyck-CFL reachability problem, where one Dyck-CFL is used to formulate context sensitivity and the other for field sensitivity. Compared to them, our approach focuses on the problem of context sensitivity alone. However, this does not mean our approach cannot be used in a context- and field-sensitive analysis as the interleaved Dyck-CFL formulation is just one of many ways to formulate context- and field-sensitivity. In practice, our approach can be used in many program analyses that require both context and field sensitivity. For instance, the aforementioned two applications, i.e., the IFDS-based information-flow analysis and the value-flow-based alias analysis, are both context- and

```
1.  struct List { int val; List *next }
2.  void create() {
3.      List lst1, lst2;
4.      lst1.val = 1; lst1.next = null;
5.      lst2.val = move_value(lst1.val)
6.      lst2.next = move_next(lst1.next);
7.  }
8.  int move_value(int val) {
9.      int ret_val = val; return ret_val;
10. }
11.
12. List *move_next(List *lst) {
13.     List *ret_lst = lst; return ret_lst;
14. }
```

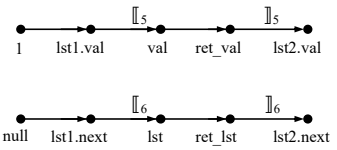


Fig. 7. Example to illustrate how our approach is used in a context- and field-sensitive analysis.

field-sensitive. To achieve field sensitivity, they build field-sensitive program-valid graphs where each vertex represents a single structure field and only one Dyck-CFL is required to formulate context sensitivity. As an example, Figure 7 shows the field-sensitive value-flow graph built by the state-of-the-art value-flow analyses [Li et al. 2011, 2013; Shi et al. 2018; Sui and Xue 2016, 2020]. In the graph, each vertex represents a constant, variable, or single structure field. The edges represent value propagation between them. An edge label, $[\![_i$ or $]\!]_i$, stands for a function call or return at Line i in the code. Clearly, tracking value flows along these edges is both context- and field-sensitive. Since the value-flow graph follows the definition of the program-valid graph, our approach can be directly used to speed up the context- and filed-sensitive value flow analysis.

5 EVALUATION

In this section, we discuss the evaluation results of our approach. All experiments were run on a server with eighty "Intel Xeon CPU E5-2698 v4 @ 2.20GHz" processors and 256GB of memory running Ubuntu-16.04. The source code of our tool is publicly available: https://github.com/qingkaishi/context-sensitive-reachability.

5.1 Experiment Setup

This subsection discusses the implementation of the baseline approaches and their indexed counterparts, followed by some clarifications on the baselines and discussions on the benchmark programs.

5.1.1 Baseline Approaches. We implemented the IFDS-based information-flow analysis based on the Phasar framework [Schubert et al. 2019]. The Phasar framework allows us to build the program-valid graph, i.e., the exploded super-graph. Information-flow queries then can be formulated as CS-reachability queries on the graph. As an IFDS-based approach, Phasar employs the standard tabulation algorithm to answer each CS-reachability query. We then implement our indexing approach in the analysis and evaluate how it can speed up the queries. As discussed in the last section, we use the Grail indexing scheme [Yildirim et al. 2010] in this application.

We also follow a recent work [Shi et al. 2018] to build a value-flow graph, which allows us to formulate the pointer alias problem as CS-reachability queries and answer each CS-reachability query via a standard tabulation algorithm. We then implement our indexing approach in the value-flow-based analysis and evaluate how our indexing schemes can speed up the queries. As discussed in the last section, we use PathTree [Jin et al. 2011] as the indexing scheme in this application.

For both of our applications, we organize the experiments in three parts. In the first two parts, we show that it is not possible to compute a full transitive closure for answering the CS-reachability queries while we can compute the reachability indexes within a reasonable time and space overhead. The experiments of computing the transitive closure are run with a limit of six hours. In the third part, we show that the indexed analyses are much faster than their original counterparts, which follow the same spirit of Reps et al. [1995, 1994]'s tabulation algorithm and traverse the input program-valid graph to answer each CS-reachability query.

Table 2. Benchmark programs from SPEC2000 and the real world.

ID	Program	LoC	Information Flow		Alias Analysis	
			# V	# E	# V	# E
1	mcf	2K	33.3K	35.0K	22.2K	29.4K
2	bzip2	3K	257.3K	277.8K	59.5K	76.2K
3	gzip	6K	314.5K	332.2K	135.6K	182.8K
4	parser	8K	540.5K	566.7K	574.2K	749.1K
5	vpr	11K	3.0M	3.4M	347.7K	421.7K
6	crafty	13K	651.0K	678.5K	280.2K	362.1K
7	twolf	18K	635.4K	696.6K	468.0K	624.0K
8	eon	22K	798.8K	969.9K	766.0K	852.0K
9	gap	36K	689.8K	788.3K	3.2M	3.8M
10	vortex	49K	659.1K	714.2K	4.4M	5.6M
11	perlbmk	73K	2.4M	2.6M	9.2M	11.7M
12	gcc	135K	9.7M	10.7M	17.0M	22.2M
13	git-2.32.0	248K	5.4M	5.7M	20.6M	25.4M
14	vim-8.2.3047	386K	14.7M	19.0M	40.0M	50.0M
15	icu-69.1	594K	5.9M	6.5M	14.4M	18.2M
16	ffmpeg-3.0	940K	5.3M	6.2M	33.8M	44.6M

5.1.2 Clarifications on Baseline Approaches. Since there have been many Dyck-CFL algorithms, information-flow analyses, and alias analyses, we clarify the following points on the baselines.

(1) There have been a large number of Dyck-CFL algorithms proposed in the past decades. These algorithms may also have been applied to alias analysis or information-flow analysis. However, as discussed in Section 2.2, they require different modeling of the program analysis problems and cannot address the CS-reachability problem studied in this paper. Hence, it does not make sense to compare our approach to them. In practice, the most common way to address the CS-reachability problem is to use the tabulation algorithm or compute a transitive closure, which are the correct baselines we need to compare.

(2) Our core contribution is an indexing approach to accelerating program analyses based on CS-reachability queries. To show the contribution, we design a controlled experiment where an indexed analysis is compared with its non-indexed counterpart. We want to clarify that it is not meaningful to compare an indexed analysis, e.g., the indexed value-flow-based alias analysis, with an analysis that follows an orthogonal methodology, e.g., an alias analysis not based on value flows but the LCL formulation [Zhang and Su 2017]. This is because, even if we can show that the indexed value-flow-based alias analysis is faster than the LCL-based one, we cannot make any conclusion about our indexing technique as the speed improvement may come from two sources, i.e., the indexing method or the value-flow-based formulation.

(3) There are many IFDS-based information-flow analyses and value-flow-based alias analyses proposed in the past decades. We want to clarify that they differ from each other mainly because of their approaches to building the exploded-super graphs, e.g., [Arzt and Bodden 2014; Schubert et al. 2019], and the value-flow graphs, e.g., [Cherem et al. 2007; Shi et al. 2018; Sui and Xue 2016], which, however, is actually not important for our evaluation. This is because the core contribution of the paper is to speed up queries after a graph is given, rather than an approach to building the graphs. In our implementation, as discussed before, we follow two recent works to build the graphs.

(4) We notice that many query caching mechanisms [Zhou et al. 2018], parallel querying techniques [Han et al. 2013], and graph simplification algorithms [Li et al. 2020] can also improve query performance. These techniques are orthogonal to our approach. Thus, it does not make sense to compare with them. In practice, our approach can be used together with them for better performance. For instance, our approach can be performed on a simplified program-valid graph, which will lead to a smaller indexing graph and faster query speed.

ID	**Time** (seconds)			**Memory** (MB)		
	I.G.	Grail	**Total**	I.G.	Grail	**Total**
1	0.36	0.03	0.39	0.25	1.52	1.78
2	2.34	0.33	2.67	1.96	11.78	13.74
3	3.36	0.24	3.60	2.40	14.39	16.79
4	4.83	0.42	5.25	4.12	24.74	28.87
5	30.39	5.25	35.64	23.20	139.18	162.38
6	6.06	1.26	7.32	4.97	29.80	34.77
7	6.01	2.49	8.50	4.85	29.08	33.93
8	8.84	1.56	10.40	6.09	36.57	42.66
9	6.63	0.78	7.41	5.26	31.57	36.84
10	5.58	2.16	7.74	5.03	30.17	35.20
11	21.32	6.3	27.62	18.38	110.29	128.67
12	95.92	22.71	118.63	74.36	446.14	520.50
13	43.95	3.9	47.85	41.46	248.76	290.22
14	117.18	80.7	197.88	129.51	657.08	786.60
15	45.81	2.76	48.57	45.35	272.12	317.47
16	41.45	3.99	45.44	40.63	243.80	284.44

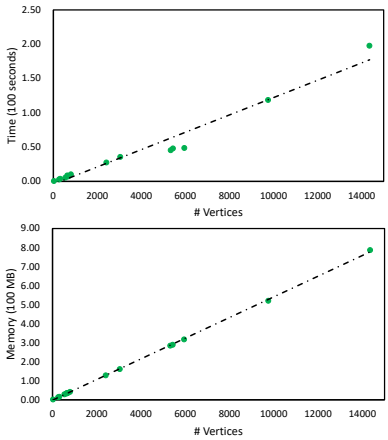


Fig. 8. The cost of building the indexing graph (I.G.) and the Grail index for the information-flow analysis.

5.1.3 Benchmark Programs. Our experiments are conducted over twelve programs from the standard benchmark suite, SPEC2000 [Henning 2000], and four much larger real-world programs: *git*, *vim*, *icu*, and *ffmpeg*. The basic information of the sixteen benchmark programs is listed in Table 2, including the lines of code (LoC) of each program as well as the number of vertices (#V) and edges (#E) in their program-valid graphs. As shown in the table, the sizes of the programs range from a few thousand lines of code to nearly one million, and a program-valid graph may contain tens of millions of vertices and edges, which makes it challenging to answer conventional reachability queries quickly, let alone CS-reachability queries.

5.2 IFDS-Based Information-Flow Analysis

The evaluation consists of three parts, aiming to show that (1) computing the transitive closure is not practical, (2) compared to computing a transitive closure, the overhead of computing the reachability index is reasonable, and (3) the reachability index significantly speeds up CS-reachability queries.

5.2.1 Transitive Closure is not Practical. Computing a transitive closure is always the most ideal approach to answering conventional reachability queries and CS-reachability queries. This is because, a transitive closure allows us to answer an unreachable CS-reachability query (or, in this application, an information-flow query) in constant time and, for reachable cases, the transitive closure allows us to find paths between two vertices within linear time and space with respect to the path length. However, due to the high complexity of computing the transitive closure [Chaudhuri 2008], we failed to compute the transitive closure for ten of our sixteen benchmark programs (ID from 7 to 16) in six hours. Thus, while computing a transitive closure is an ideal approach to the information-flow analysis, it is not practical for analyzing large-scale software.

5.2.2 Overhead of Indexing is Reasonable. Compared to the transitive closure, the reachability index is much cheaper to compute. In the information-flow analysis, the cost of computing the reachability index comes from two parts — computing the indexing graph and building the Grail index [Yildirim et al. 2010] over the indexing graph. As discussed in Section 3.4, the main cost of building an indexing graph is to compute the summary edges. Figure 8 lists the cost of building the indexing graph as well as the cost of computing the Grail index. We can observe that building the summary edges and the indexing graph is very cheap. For the largest program-valid graph that contains tens of millions of vertices, we only need less than 2 minutes and about 80 MB of space.

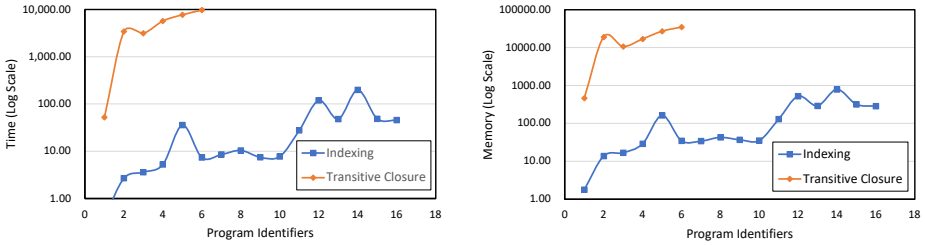


Fig. 9. Reachability indexes vs. Transitive closure for the information-flow analysis.

Table 3. Time cost (milliseconds) of answering 10,000 reachable queries (R), 10,000 unreachable queries ($\neg R$) in the information-flow analysis.

ID	IFDS-IFA + FLARE			IFDS-IFA		
	R	$\neg R$	**Total**	R	$\neg R$	**Total**
1	22	2	24	35.5K	2.1K	37.6K
2	150	62	212	34.0K	35.2K	69.2K
3	214	240	454	329.3K	139.0K	468.3K
4	352	132	484	714.6K	150.2K	864.8K
5	2,317	320	2,637	2.8M	983.5K	3.7M
6	838	23	861	786.2K	41.6K	827.8K
7	290	9	299	31.9K	30.0K	61.9K
8	2,123	181	2,304	558.3K	255.4K	813.7K
9	1,321	114	1,435	1.5M	269.2K	1.8M
10	1,194	101	1,295	1.7M	351.1K	2.0M
11	3,654	205	3,859	5.7M	193.1K	5.9M
12	6,416	315	6,731	21.6M	11.5M	33.1M
13	1,012	166	1,178	1.5M	1.4M	2.9M
14	2,787	50	2,837	1.8M	1.7M	3.5M
15	1,898	242	2,140	14.5M	555.6K	15.0M
16	1,532	213	1,745	1.1M	1.2M	2.3M
min	110×	565×	206×			
med	1.3K×	2.3K×	1.4K×		N/A	
max	7.6K×	36.5K×	7.0K×			

Figure 8 also plots the relationship between the input graph size and the total cost of building both the indexing graph and the Grail index, which shows a linear complexity in practice. Therefore, the indexing scheme for the information-flow analysis scales quite gracefully in practice. Figure 9 shows that, compared to the cost of computing a transitive closure, the overhead of computing the reachability index is negligible in practice. Even for the six small programs for which we succeed in computing the transitive closure, computing the index is 540× faster and saves 99.7% of the space compared to computing the transitive closure.

5.2.3 Indexing Enables Much Faster Queries. As listed in Table 3, the reachability index allows the indexed information flow analysis (IFDS-IFA+Flare) to answer 10,000 reachable queries in 6,500 milliseconds (R) and 10,000 unreachable queries in 350 milliseconds ($\neg R$), 110× to 7,642× and 565× to 36,541× faster than the baseline tabulation algorithm (IFDS-IFA). Readers may notice that answering a reachable query takes much longer than answering an unreachable query. This is because, for a reachable query in the information-flow analysis, we need to additionally compute a path from the source vertex to the target vertex. Readers may also argue that it takes some time to build the reachability index while it does not need any preprocessing to perform the tabulation

ID	Time (seconds)			Memory (MB)		
	I.G.	PathTree	**Total**	I.G.	PathTree	**Total**
1	0.09	1.80	1.89	0.21	1.08	1.29
2	0.33	8.79	9.12	0.56	2.91	3.47
3	1.53	34.60	36.13	1.28	7.95	9.23
4	8.00	105.67	113.67	5.13	28.76	33.89
5	3.25	41.19	44.44	3.16	16.80	19.96
6	0.84	33.55	34.39	2.36	13.26	15.62
7	3.26	69.85	73.11	4.07	23.97	28.04
8	1.97	83.15	85.12	8.33	38.46	46.79
9	11.63	404.20	415.83	25.50	142.98	168.48
10	33.85	645.69	679.54	36.00	204.34	240.34
11	37.39	1894.22	1931.61	72.07	427.44	499.51
12	110.80	6507.63	6618.43	141.52	925.09	1066.61
13	130.52	3144.47	3274.99	173.92	890.78	1064.70
14	196.54	10038.21	10234.75	320.35	1763.15	2083.50
15	124.52	3580.17	3704.69	127.60	684.07	811.67
16	113.56	10737.98	10851.54	297.56	1558.47	1856.03

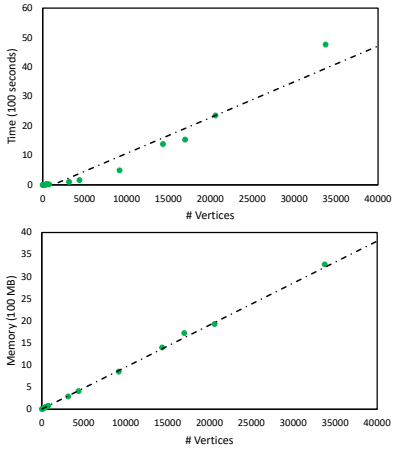


Fig. 10. The cost of building the indexing graph (I.G.) and the PathTree index for the alias analysis.

algorithm. For this issue, we would like to clarify that, computing the reachability index is a one-time effort, which means that it is built offline and only once to benefit all future queries. Hence, it is not necessary to consider the cost of indexing here as we have shown it can be built efficiently.

5.3 Value-Flow-Based Alias Analysis

The evaluation of the alias analysis also consists of three parts, aiming to show that (1) computing a transitive closure is not practical, (2) the overhead of computing the reachability index is reasonable, and (3) the reachability index significantly speeds up the CS-reachability queries.

5.3.1 Transitive Closure is not Practical. Computing a transitive closure allows us to answer both reachable and unreachable CS-reachability queries (or, in this application, aliasing queries) in constant time. However, it is of sub-cubic time complexity and quadratic space complexity to compute a transitive closure [Chaudhuri 2008], which is not affordable in practice. In the experiment, we finished the computation only for the programs with less than 40 KLoC and failed for all other larger programs. This result shows that computing a transitive closure is not practical.

5.3.2 Overhead of Indexing is Reasonable. Same as the information flow analysis, the cost of the indexing procedure also includes two parts. The first part is to compute the indexing graph and the second part is to apply the PathTree indexing scheme [Jin et al. 2011] to the indexing graph. As discussed in Section 3.4, the main cost of building the indexing graph is to compute the summary edges. Figure 10 lists the time and memory cost of computing the indexing graph as well as the cost of computing the PathTree index. We can observe that, for the value-flow-based alias analysis, the cost of building the summary edges and the indexing graph is also of linear complexity in the input graph size. For the largest graph that contains about forty million vertices, we only need three minutes and 320MB of space to build the indexing graph. Compared to the Grail indexing scheme used in the information-flow analysis, both the time cost and the memory cost of PathTree are higher but still tend to be linear as shown in Figure 10. For the largest program, it takes less than 3 hours to build the index. It is noteworthy that the scalability of PathTree has been shown in previous works [Jin et al. 2011]. In practice, if an application cannot afford its overhead, we can choose other indexing schemes discussed in Section 2.3.

We show in Figure 11 that, compared to a transitive closure, the reachability index is much cheaper. Even in a log-scale coordinate system, the curves of computing the transitive closure are

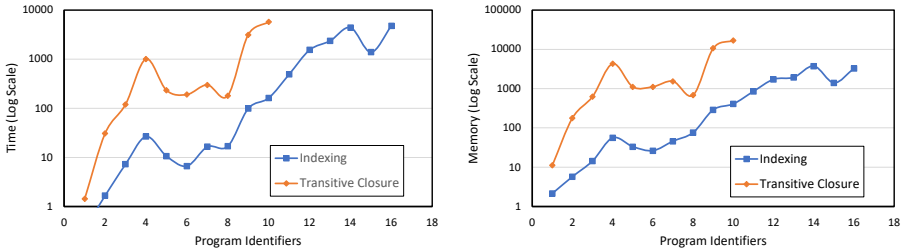


Fig. 11. Reachability indexes vs. Transitive closure for the alias analysis.

Table 4. Time cost (milliseconds) of answering 10,000 reachable queries (R) and 10,000 unreachable queries ($\neg R$) in the alias analysis.

ID	VF-AA + FLARE			VF-AA		
	R	$\neg R$	**Total**	R	$\neg R$	**Total**
1	80	18	98	6.5K	2.5K	9.0K
2	102	6	108	83.0K	21.8K	104.8K
3	114	12	126	319.8K	57.5K	377.3K
4	124	8	132	690.4K	52.3K	742.7K
5	108	8	116	243.9K	38.7K	282.6K
6	123	8	131	160.3K	27.3K	187.6K
7	157	7	164	244.0K	20.1K	264.1K
8	149	8	157	289.5K	34.8K	324.3K
9	172	11	183	835.0K	50.2K	885.2K
10	155	10	165	3.6M	55.4K	3.6M
11	191	8	199	4.8M	52.3K	4.9M
12	222	12	234	14.4M	122.9K	14.5M
13	196	11	207	7.8M	225.2K	8.0M
14	200	10	210	3.2M	402.7K	3.6M
15	195	15	210	16.6M	402.1K	17.0M
16	216	24	240	58.5M	1.5M	60.0M
min	81×	134×	91×			
med	5.2K×	5.3K×	5.2K×		N/A	
max	270.9K×	47.2K×	248.8K×			

much higher than those of computing the reachability index. Particularly, we failed to compute the transitive closure for programs with more than four million vertices while we can finish computing the index for all benchmark programs. Meanwhile, compared to the transitive closures we succeed in computing, the reachability index is much smaller in size, saving 99.1% of the space.

5.3.3 Indexing Enables Much Faster Queries. As shown in Table 4, the reachability index allows the indexed value-flow-based alias analysis (VF-AA + Flare) to answer 10,000 reachable queries in 200 milliseconds and 10,000 unreachable queries in 30 milliseconds, 81× to 270,885× and 134× to 47,199× faster than the baseline tabulation algorithm approach. In total, compared to the baseline approach, the indexing scheme for the alias analysis can speed up the context-sensitive aliasing queries with a speedup from 91× to 248, 812×, with 5, 227× as the median.

In addition to the promising speedup over the original approach, we can observe that the query performance is very stable and works like constant time complexity — the time cost of answering 10,000 reachable and unreachable queries is always around or less than 200 milliseconds and 30 milliseconds, respectively. The differences in the query performance between the reachable and the unreachable cases are caused by the internal design of PathTree, which is out of the scope of the paper and, thus, is omitted. The stable query time confirms that, with a moderate space overhead, our approach enables us to answer CS-reachability queries in almost constant time.

5.4 Summary

In the end, we would like to summarize our experiment results as follows, which include the main takeaway messages we want to convey via the experiments.

(1) Building summary edges is a preprocessing procedure of our indexing approach, which has been shown in previous works that it is in almost linear complexity [Chatterjee et al. 2017; Reps et al. 1994] and has been demonstrated in this paper that it is cheap to build in practice. Hence, it is practical to build summary edges and this is not a limitation of our approach.
(2) Building the reachability indexes is of linear complexity in practice and is much cheaper than computing a transitive closure. In addition, note that building indexes is a one-time effort, which means that we can build the indexes offline only once to benefit all future queries. Hence, it improves the query speed without extra overhead added to an analysis.
(3) Except for querying paths for a positive reachability query, after indexing, the time cost of answering a CS-reachability query exhibits a constant complexity in practice, i.e., the cost does not grow with the increase of graph size. Hence, it significantly improves the scalability of program analysis.

6 RELATED WORK

In addition to the related works discussed in Section 2.2, we discuss two strands of related works — one on the language-reachability problems in program analysis and the other on some advanced reachability indexing schemes with the potential of optimizing program analyses.

6.1 Language Reachability for Program Analysis

Many program analysis problems can be formulated as CFL-reachability problems. Particularly, Dyck-CFL lays the basis for "almost all of the applications of CFL reachability in program analysis" [Kodumal and Aiken 2004]. Studies on solving the problem of all-pairs CFL or Dyck-CFL reachability have been conducted in various contexts such as recursive state machines [Alur et al. 2005] and visibly push-down languages [Alur and Madhusudan 2004]. They often work with a dynamic programming algorithm, which is a generalization of the CYK algorithm for CFL-recognition [Younger 1967] and is of cubic complexity [Kodumal and Aiken 2004; Reps et al. 1995; Yannakakis 1990]. Melski and Reps [2000] and Kodumal and Aiken [2004] studied the relationship between CFL reachability and set constraints, but they did not break through the cubic bottleneck. Chaudhuri [2008] and Zhang et al. [2014] showed that the Four Russians' Trick could be employed to achieve sub-cubic algorithms. These works are different from our approach as they do not utilize indexing schemes and do not try to transform CFL-reachability queries to conventional reachability queries. Next, we discuss two main applications of language reachability.

6.1.1 Context-Sensitive Analysis. Tang et al. [2015] formulated the context-sensitive data-dependence analysis in the presence of callbacks as a tree-adjoining-language reachability problem. Their algorithms are of $O(|V|^6)$ and, thus, are not scalable in practice. Chatterjee et al. [2017] solved the problem by utilizing the "constant tree-width" feature of a local data-dependence graph. However, their algorithm only checks the CS-reachability relation between vertices in the same calling context while we can check CS-reachability between vertices from different calling contexts. DOOP-based analyses also utilize CFL-reachability to achieve context-sensitivity, but they often achieve only a small $k < 3$ w.r.t. kCFA [Kastrinis and Smaragdakis 2013; Smaragdakis et al. 2014] while we aim for context-sensitive reachability implied by paths of any length, which means $k = \infty$.

Zhang and Su [2017] formulated the context-sensitive and field-sensitive data-dependence analysis as a linear-conjunctive-language reachability problem, or interleaved Dyck-CFL reachability problem. Compared to CFL-reachability, the interleaved CFL-reachability formulation provides a

more precise model due to field sensitivity. However, the problem of interleaved CFL-reachability is known to be undecidable [Kjelstrøm and Pavlogiannis 2022; Reps 2000]. Thus, only approximation algorithms can be provided [Späth et al. 2019; Zhang and Su 2017]. To improve the algorithm efficiency in practice, Li et al. [2020] proposed an approach to reducing the graph size before conducting the interleaved Dyck-CFL reachability analysis.

6.1.2 Pointer Analysis. The other common use of CFL reachability is to resolve pointer relations on a bidirected graph. When the underlying language is the standard Dyck language, Zhang et al. [2013] and Chatterjee et al. [2017] have shown that a transitive closure can be computed in almost linear time. However, it is much harder when the graph is not bidirected and the CFL is not a standard Dyck-CFL. Many techniques have been proposed to optimize the CFL-based pointer analysis. Zheng and Rugina [2008] proposed a demand-driven method to resolve pointer relations without computing a transitive closure. Xu et al. [2009] compute must-not-alias information for all pairs of variables, which then can be used to quickly filter out infeasible paths during the more precise pointer analysis. Dietrich et al. [2015] proposed a novel transitive-closure data structure with a pre-computed set of potentially matching load/store pairs to accelerate the fix-point calculation. Wang et al. [2017] proposed a graph system that allows CFL-based pointer analysis to work in a single machine by utilizing the disk space. Our work is different from theirs because we do not focus on bidirected graphs and the underlying CFL is different.

6.2 Potential Use of Graph Database Techniques in Program Analysis

Since our approach reduces the CS-reachability problem to the conventional reachability problem, it enables us to benefit from many other advanced graph database techniques, thereby enabling more program analysis applications. We briefly discuss some of them below.

6.2.1 Indexing for Dynamic Graph Reachability. Beyond the indexing schemes for conventional graph reachability discussed in Section 2, recent studies also consider additional constraints when evaluating reachability queries. Some of these indexing schemes can be used directly to optimize program analyses. Roditty and Zwick [2008], Bouros et al. [2009], and Zhu et al. [2014] proposed improved reachability algorithms to handle dynamic graphs, where edges and vertices may be dynamically created, updated, or deleted.

These indexing schemes, which we refer to as incremental indexing schemes, can be applied to incremental program analysis where program variables or dependence relations are changed during software evolution. When software evolves, we continuously revise the indexing graph proposed in the paper. An incremental indexing scheme can quickly capture the graph changes and rebuild the indexes for answering CS-reachability queries.

6.2.2 Indexing for Label-Constraint Reachability. Jin et al. [2010] proposed the problem of label-constraint reachability (LCR), in which a vertex v is reachable from a vertex u if and only if there exists a path from the vertex u to the vertex v and the set of edge labels on the path is a subset of a given label set. This problem has been extensively studied [Valstar et al. 2017; Zou et al. 2014]. Recently, Peng et al. [2020] proposed an indexing scheme to answer billion-scale LCR queries. Hassan et al. [2016] and Rice and Tsotras [2010] proposed approaches to finding the shortest path for LCR problems.

Many program analyses can be modeled as an LCR problem. For instance, in a taint analysis, by modeling the sanitizing operations as edge labels, we can employ the LCR indexing schemes to check if the tainted data are propagated to a destination with proper sanitizations. Using the indexing graph proposed in the paper, we can label its edges with sanitization labels and use an LCR indexing scheme to enable an efficient context-sensitive taint analysis.

6.2.3 Benefiting from Other Graph Database Techniques. Since we have transformed CS-reachability queries into conventional reachability queries in a common directed graph, we can profit from a lot of existing graph database techniques developed for common directed graphs, such as query caching techniques and graph simplification algorithms. The query caching techniques, such as C-Graph [Zhou et al. 2018], explore the data locality to speed up graph processing tasks. All modern graph databases such as Neo4j [Neo4j 2022] implement such caching mechanisms for acceleration. These approaches are orthogonal to our idea and can be used together with our approach for better performance. The graph simplification techniques reduce the graph size so as to accelerate reachability queries. First, we can discover and merge vertices with equivalent reachability relations, e.g., vertices in a strongly connected component or vertices with the same successors and predecessors, thereby reducing the graph size [Fan et al. 2012; Zhou et al. 2017]. Second, we can perform transitive reduction, which removes unnecessary edges with respect to reachability queries [Aho et al. 1972; Habib et al. 1993; Simon 1988; Valdes et al. 1982; Williams 2012; Zhou et al. 2017].

7 CONCLUSION

Different from many existing works that only focus on the standard Dyck-CFL reachability, this paper addresses the problem of the extended version by transforming it into the problem of conventional graph reachability. This transformation allows us to answer any context-sensitive reachability query in almost constant time via the indexing schemes of conventional graph reachability. We apply our approach to speeding up two context-sensitive dataflow analyses and compare them to their original approaches. The evaluation results demonstrate we can achieve orders of magnitude speedup with only a linear space overhead to build and store the indexes. Since reducing the complexity of CFL-reachability is theoretically very hard in a general setting, providing proper indexing schemes could be a promising solution for speeding up CFL-reachability queries.

ACKNOWLEDGEMENTS

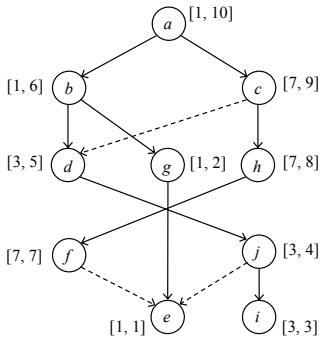
We would like to thank the anonymous reviewers and Dr. Qirun Zhang for their valuable feedback on the earlier draft of this paper. This work was supported in part by the Ant Research Program and the PRP/004/21FX and ITS/440/18FP grants. This work was done when Qingkai Shi was with Ant Group. He is currently with Purdue University and available via email at shi553@purdue.edu.

A INDEXING SCHEMES FOR CONVENTIONAL REACHABILITY

Existing indexing schemes for conventional graph reachability can be put into two groups: (1) compression of transitive closure (e.g., [Chen and Chen 2008; Cohen et al. 2003; Jin et al. 2011, 2009; Wang et al. 2006]) and (2) pruned search (e.g., [Chen et al. 2005; Seufert et al. 2013; Wei et al. 2014; Yildirim et al. 2010]). This section provides detailed examples. Note that all indexing schemes are assumed to work on a directed acyclic graph (DAG) because, given a directed graph, we can always merge vertices in a strongly connected component (SCC) into a single vertex v. Since vertices in an SCC are reachable from each other, any reachability query between a vertex outside an SCC, e.g., u, and a vertex inside the SCC can be equivalently transformed to a query between the vertex u and the vertex v in the DAG.

A.1 Compression of Transitive Closure

Figure 12 illustrates the indexing scheme called "dual labeling" [Wang et al. 2006], which can be regarded as a simplified version of the PathTree indexing scheme [Jin et al. 2011] used in our application. The indexing scheme firstly finds a spanning tree of a given DAG and labels each vertex v with an interval $[L_v, H_v]$ according to a post-order traversal of the tree. For each interval,



(a)

(1) $[7, 9] \rightarrow [3, 5]$
(2) $[3, 4] \rightarrow [1, 1]$
(3) $[7, 7] \rightarrow [1, 1]$
(4) $[7, 9] \rightarrow [1, 1]$

The Transitive Link Table
((4) is the transitive closure of (1) and (2))

(b)

Fig. 12. (a) A graph where one of its spanning trees is represented by the solid edges and non-tree edges are the dashed edges. Vertices are labeled by intervals for reachability queries. (b) The transitive link table for encoding reachability relations implied by the non-tree edges.

H_v is the rank of the vertex v in a post-order traversal of the tree, where the ranks are assumed to begin at 1; L_v denotes the lowest rank for any vertex in the sub-tree rooted at the vertex v. This approach guarantees that the vertex v is reachable from the vertex u on the tree if and only if $[L_v, H_v] \subseteq [L_u, H_u]$, because the post-order traversal enters a vertex before all its descendants and leaves after visiting all its descendants.

For each non-tree edge, we record it in a transitive link table (TLT) as illustrated in Figure 12(b), and compute a transitive closure. For instance, since the edge from the vertex c to the vertex d is a non-tree edge, we include the entry $[7, 9] \rightarrow [3, 5]$ in the TLT, where $[7, 9]$ and $[3, 5]$ are the labels of the two vertices, respectively. Similarly, we have $[7, 7] \rightarrow [1, 1]$ and $[7, 9] \rightarrow [1, 1]$ in the TLT. We then compute a transitive closure of the TLT. For instance, since $[7, 9] \rightarrow [3, 5]$ and $[3, 4] \rightarrow [1, 1]$ are in the TLT and $[3, 4] \subseteq [3, 5]$ implies $[3, 5] \rightarrow [3, 4]$, we also include $[7, 9] \rightarrow [1, 1]$ in the TLT.

We then can determine the reachability relation on the graph as follows: the vertex v is reachable from the vertex u if and only if $[L_v, H_v] \subseteq [L_u, H_u]$ or there exists an entry $[L_x, H_x] \rightarrow [L_y, H_y]$ in the link table such that $[L_x, H_x] \subseteq [L_u, H_u] \wedge [L_v, H_v] \subseteq [L_y, H_y]$. Assuming the graph has $|V|$ vertices, $|E|$ edges, and k (k is far less than $|E|$) non-tree edges, we need $O(|V| + |E|)$ time to label the vertices and $O(k^3)$ time to compute the transitive link table, which consume $O(|V|)$ and $O(k^2)$ space, respectively. Wang et al. [2006] showed that, when answering a reachability query, it is not necessary to take $O(k^2)$ time to find the entry $[L_x, H_x] \rightarrow [L_y, H_y]$ in the link table. It is actually a special range-temporal aggregation problem and can be solved in $O(1)$ time. Thus, we can answer each reachability query in constant time.

We say this indexing scheme speeds up reachability queries by compressing the transitive closure because it keeps the capability of answering a query in constant time, just like that we have a transitive closure, but significantly reduces the complexity of computing a transitive closure from $O(|V||E|)$ to $O(|V| + |E| + k^3)$ in terms of time and from $O(|V|^2)$ to $O(|V| + k^2)$ in terms of space.

A.2 Pruned Search

Figure 13 illustrates the Grail indexing scheme [Yildirim et al. 2010]. Given a DAG, Grail labels each vertex v with an interval $[L_v, H_v]$ as in the last example but based on a post-order traversal on the DAG instead of the spanning tree. That is, for each interval, H_v is the rank of the vertex v

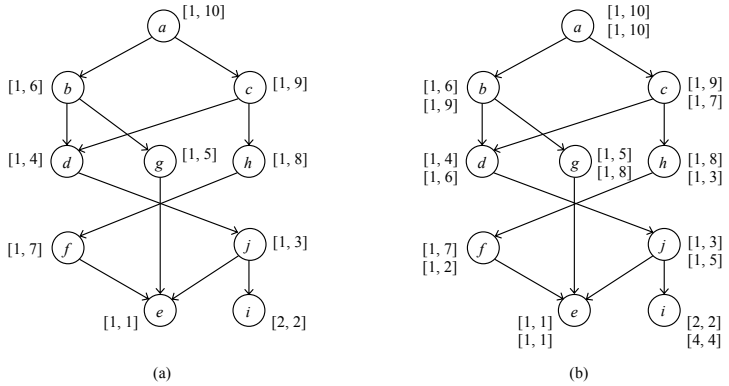


Fig. 13. (a) A graph where each vertex is labeled by an interval for reachability queries. (b) Vertices are labeled by multiple intervals to reduce false positives.

in a post-order traversal of the DAG, where the ranks are assumed to begin at 1; L_v denotes the lowest rank for any vertex in the sub-graph rooted at the vertex v. The basic idea is that, on the DAG, although $[L_v, H_v] \subseteq [L_u, H_u]$ cannot imply that the vertex v is reachable from the vertex u, $[L_v, H_v] \not\subseteq [L_u, H_u]$ is sufficient to decide that the vertex v is NOT reachable from the vertex u. For instance, in Figure 13(a), $[L_h, H_h] = [1, 8] \not\subseteq [1, 3] = [L_j, H_j]$ implies that the vertex h is NOT reachable from the vertex j. However, $[L_j, H_j] = [1, 3] \subseteq [1, 8] = [L_h, H_h]$ does not imply that the vertex j is reachable from the vertex h.

To prune such false positives induced by checking the "$\subseteq$" relation, Grail performs a randomized post-order traversal on the graph multiple times, leading to multiple interval labels as shown in Figure 13(b). The randomized traversal is performed by randomly ordering the children of each vertex at the time of graph traversal. With multiple interval labels, we now can easily determine that the vertex j is not reachable from the vertex h, because the second interval of the vertex j, $[1, 5]$, is not a subset of the second interval of the vertex h, $[1, 3]$.

In most cases, we can deny the reachability query between two vertices by checking the "$\not\subseteq$" relation in constant time. In other cases, Grail falls back to a graph traversal but is capable of using the intervals to prune unreachable paths. For instance, assume that we want to check if the vertex j is reachable from the vertex c using the labels in Figure 13(b). Since $[L_j, H_j] = [1, 3][1, 5] \subseteq [1, 9][1, 7] = [L_c, H_c]$, we cannot decide the reachability relations between the two vertices. Hence, we need to traverse the graph from the vertex c, trying to find a path between the two vertices. Assume that the second vertex we visit during the graph traversal is the vertex h, one child of the vertex c. Since $[L_j, H_j] = [1, 3][1, 5] \not\subseteq [1, 8][1, 3] = [L_h, H_h]$, we can decide that the vertex h cannot reach the vertex j. Hence, we can stop the traversal along this path. This strategy prunes unnecessary paths to avoid a full graph traversal, thus improving the query performance. This is also why we call Grail a pruned-search-based indexing scheme.

REFERENCES

Alfred Aho, Michael Garey, and Jeffrey Ullman. 1972. The transitive reduction of a directed graph. *SIAM J. Comput.* 1, 2 (1972), 131–137. https://doi.org/10.1137/0201008

Rajeev Alur, Michael Benedikt, Kousha Etessami, Patrice Godefroid, Thomas Reps, and Mihalis Yannakakis. 2005. Analysis of recursive state machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 27, 4 (2005), 786–818. https://doi.org/10.1145/1075382.1075387

Rajeev Alur and Parthasarathy Madhusudan. 2004. Visibly pushdown languages. In *Proceedings of the 36th ACM Symposium on Theory of Computing (STOC '04)*. ACM, 202–211. https://doi.org/10.1145/1007352.1007390

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 (ICSE '14)*. ACM, 288–298. https://doi.org/10.1145/2568225.2568243

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, 259–269. https://doi.org/10.1145/2594291.2594299

Panagiotis Bouros, Spiros Skiadopoulos, Theodore Dalamagas, Dimitris Sacharidis, and Timos Sellis. 2009. Evaluating reachability queries over path collections. In *Proceedings of the 21st International Conference on Scientific and Statistical Database Management (SSDBM '09)*. Springer, 398–416. https://doi.org/10.1007/978-3-642-02279-1_29

Cheng Cai, Qirun Zhang, Zhiqiang Zuo, Khanh Nguyen, Guoqing Xu, and Zhendong Su. 2018. Calling-to-reference context translation via constraint-guided CFL-reachability. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*. ACM, 196–210. https://doi.org/10.1145/3192366.3192378

Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. 2017. Optimal Dyck reachability for data-dependence and alias analysis. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 30:1–30:30. https://doi.org/10.1145/3158118

Swarat Chaudhuri. 2008. Subcubic algorithms for recursive state machines. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, 159–169. https://doi.org/10.1145/1328438.1328460

Li Chen, Amarnath Gupta, and M. Erdem Kurul. 2005. Stack-based algorithms for pattern matching on dags. In *Proceedings of the 31st International Conference on Very Large Data Bases*. VLDB Endowment, 493–504. https://doi.org/10.14778/2180912.2180919

Yangjun Chen and Yibin Chen. 2008. An efficient algorithm for answering graph reachability queries. In *Proceedings of the 24nd International Conference on Data Engineering (ICDE '08)*. IEEE, 893–902. https://doi.org/10.1109/ICDE.2008.4497498

James Cheng, Silu Huang, Huanhuan Wu, and Ada Wai-Chee Fu. 2013. TF-Label: A topological-folding labeling scheme for reachability querying in a large graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, 193–204. https://doi.org/10.1145/2463676.2465286

Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. 2007. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, 480–491. https://doi.org/10.1145/1250734.1250789

Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355. https://doi.org/10.1137/S0097539702403098

Douglas Comer. 1979. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)* 11, 2 (1979), 121–137. https://doi.org/10.1145/356770.356776

Jens Dietrich, Nicholas Hollingum, and Bernhard Scholz. 2015. Giga-scale exhaustive points-to analysis for java in under a minute. In *Proceedings of the 2015 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '15)*. ACM, 535–551. https://doi.org/10.1145/2814270.2814307

Wenfei Fan, Jianzhong Li, Xin Wang, and Yinghui Wu. 2012. Query preserving graph compression. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, 157–168. https://doi.org/10.1145/2213836.2213855

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349. https://doi.org/10.1145/24039.24041

Hugh Gauch. 2002. *Scientific Method in Practice*. Cambridge University Press. https://doi.org/10.1017/CBO9780511815034

Michel Habib, Michel Morvan, and J-X Rampon. 1993. On the calculation of transitive reduction-closure of orders. *Discrete Mathematics* 111, 1-3 (1993), 289–303. https://doi.org/10.1016/0012-365X(93)90164-O

Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. 2013. TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '13)*. ACM, 77–85. https://doi.org/10.1145/2487575.2487581

Mohamed Hassan, Walid Aref, and Ahmed Aly. 2016. Graph indexing for shortest-path finding over dynamic sub-graphs. In *Proceedings of the 2016 ACM International Conference on Management of Data (SIGMOD '16)*. ACM, 1183–1197. https://doi.org/10.1145/2882903.2882933

John L. Henning. 2000. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer* 33, 7 (2000), 28–35. https://doi.org/10.1109/2.869367

Ruoming Jin, Hui Hong, Haixun Wang, Ning Ruan, and Yang Xiang. 2010. Computing label-constraint reachability in graph databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. ACM,

123–134. https://doi.org/10.1145/1807167.1807183

Ruoming Jin, Ning Ruan, Yang Xiang, and Haixun Wang. 2011. Path-tree: An efficient reachability indexing scheme for large directed graphs. *ACM Transactions on Database Systems (TODS)* 36, 1 (2011), 7:1–7:44. https://doi.org/10.1145/1929934.1929941

Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 2009. 3-hop: A high-compression indexing scheme for reachability query. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD '09)*. ACM, 813–826. https://doi.org/10.1145/1559845.1559930

George Kastrinis and Yannis Smaragdakis. 2013. Hybrid Context-Sensitivity for Points-to Analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, 423–434. https://doi.org/10.1145/2491956.2462191

Adam Husted Kjelstrøm and Andreas Pavlogiannis. 2022. The Decidability and Complexity of Interleaved Bidirected Dyck Reachability. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 12:1–12:26. https://doi.org/10.1145/3498673

John Kodumal and Alex Aiken. 2004. The set constraint/CFL reachability connection in practice. In *Proceedings of the 25th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '04)*. ACM, 207–218. https://doi.org/10.1145/996841.996867

Johannes Lerch, Ben Hermann, Eric Bodden, and Mira Mezini. 2014. FlowTwist: Efficient context-sensitive inside-out taint analysis for large codebases. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE '14)*. ACM, 98–108. https://doi.org/10.1145/2635868.2635878

Lian Li, Cristina Cifuentes, and Nathan Keynes. 2011. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Proceedings of the 13th European Software Engineering Conference Held Jointly with the 19th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE '11)*. ACM, 343–353. https://doi.org/10.1145/2025113.2025160

Lian Li, Cristina Cifuentes, and Nathan Keynes. 2013. Precise and scalable context-sensitive pointer analysis via value flow graph. In *Proceedings of the 2013 International Symposium on Memory Management (ISMM '13)*. ACM, 85–96. https://doi.org/10.1145/2491894.2466483

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 (PLDI '20)*. ACM, 780–793. https://doi.org/10.1145/3385412.3386021

Yuanbo Li, Qirun Zhang, and Thomas Reps. 2021. On the complexity of bidirected interleaved Dyck-reachability. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 59:1–59:28. https://doi.org/10.1145/3434340

David Melski and Thomas Reps. 2000. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science* 248, 1-2 (2000), 29–98. https://doi.org/10.1016/S0304-3975(00)00049-9

Ana Milanova. 2020. FlowCFL: generalized type-based reachability analysis: graph reduction and equivalence of CFL-based and type-based reachability. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 178:1–178:29. https://doi.org/10.1145/3428246

Neo4j. 2022. Graph data platform. https://neo4j.com/.

You Peng, Ying Zhang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2020. Answering billion-scale label-constrained reachability queries within microsecond. *Proceedings of the VLDB Endowment* 13, 6 (2020), 812–825. https://doi.org/10.14778/3380750.3380753

Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2006. Existential label flow inference via CFL reachability. In *Proceedings of the 13th International Static Analysis Symposium (SAS '06)*. Springer, 88–106. https://doi.org/10.1007/11823230_7

Jakob Rehof and Manuel Fähndrich. 2001. Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '01)*. ACM, 54–66. https://doi.org/10.1145/360204.360208

Thomas Reps. 1995. 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. https://doi.org/10.1145/215465.215466

Thomas Reps. 1998. Program analysis via graph reachability. *Information and Software Technology* 40, 11-12 (1998), 701–726. https://doi.org/10.1016/S0950-5849(98)00093-7

Thomas Reps. 2000. Undecidability of context-sensitive data-dependence analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22, 1 (2000), 162–186. https://doi.org/10.1145/345099.345137

Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. ACM, 49–61. https://doi.org/10.1145/199448.199462

Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. 1994. Speeding up slicing. In *Proceedings of the 2nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE '94)*. ACM, 11–20. https://doi.org/10.1145/193173.195287

Michael Rice and Vassilis Tsotras. 2010. Graph indexing of road networks for shortest path queries with label restrictions. *Proceedings of the VLDB Endowment* 4, 2 (2010), 69–80. https://doi.org/10.14778/1921071.1921074

Liam Roditty and Uri Zwick. 2008. Improved dynamic reachability algorithms for directed graphs. *SIAM J. Comput.* 37, 5 (2008), 1455–1471. https://doi.org/10.1137/060650271

Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2019. Phasar: An inter-procedural static analysis framework for c/c++. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 393–410. https://doi.org/10.1007/978-3-030-17465-1_22

Stephan Seufert, Avishek Anand, Srikanta Bedathur, and Gerhard Weikum. 2013. Ferrari: Flexible and efficient reachability range assignment for graph indexing. In *Proceedings of the 29nd International Conference on Data Engineering (ICDE '13)*. IEEE, 1009–1020. https://doi.org/10.1109/ICDE.2013.6544893

Lei Shang, Xinwei Xie, and Jingling Xue. 2012. On-demand dynamic summary-based points-to analysis. In *Proceedings of the 10th International Symposium on Code Generation and Optimization (CGO '12)*. ACM, 264–274. https://doi.org/10.1145/2259016.2259050

Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*. ACM, 693–706. https://doi.org/10.1145/3192366.3192418

Klaus Simon. 1988. An improved algorithm for transitive closure on acyclic digraphs. *Theoretical Computer Science* 58, 1-3 (1988), 325–346. https://doi.org/10.1007/3-540-16761-7_87

Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective Analysis: Context-Sensitivity, across the Board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, 485–495. https://doi.org/10.1145/2594291.2594320

Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, Flow-, and Field-Sensitive Data-Flow Analysis Using Synchronized Pushdown Systems. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 48:1–48:29. https://doi.org/10.1145/3290361

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, 59–76. https://doi.org/10.1145/1094811.1094817

Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction (CC '16)*. ACM, 265–266. https://doi.org/10.1145/2892208.2892235

Yulei Sui and Jingling Xue. 2020. Value-flow-based demand-driven pointer analysis for C and C++. *IEEE Transactions on Software Engineering* 46, 8 (2020), 812–835. https://doi.org/10.1109/TSE.2018.2869336

Yulei Sui, Ding Ye, and Jingling Xue. 2014. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering* 40, 2 (2014), 107–122. https://doi.org/10.1109/TSE.2014.2302311

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 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, 83–95. https://doi.org/10.1145/2676726.2676997

Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160. https://doi.org/10.1137/0201010

Raoul-Gabriel Urma and Alan Mycroft. 2015. Source-Code Queries with Graph Databases-with Application to Programming Language Usage and Evolution. *Science of Computer Programming* 97, P1 (2015), 127–134. https://doi.org/10.1016/j.scico.2013.11.010

Jacobo Valdes, Robert Tarjan, and Eugene Lawler. 1982. The recognition of series parallel digraphs. *SIAM J. Comput.* 11, 2 (1982), 298–313. https://doi.org/10.1137/0211023

Lucien DJ Valstar, George HL Fletcher, and Yuichi Yoshida. 2017. Landmark indexing for evaluation of label-constrained reachability queries. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '11)*. ACM, 345–358. https://doi.org/10.1145/3035918.3035955

Haixun Wang, Hao He, Jun Yang, Philip S. Yu, and Jeffrey Xu Yu. 2006. Dual labeling: Answering graph reachability queries in constant time. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE '06)*. IEEE, 75–75. https://doi.org/10.1109/ICDE.2006.53

Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, 389–404. https://doi.org/10.1145/3037697.3037744

Hao Wei, Jeffrey Xu Yu, Can Lu, and Ruoming Jin. 2014. Reachability querying: An independent permutation labeling approach. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1191–1202. https://doi.org/10.14778/2732977.2732992

Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. 2008. Hexastore: Sextuple Indexing for Semantic Web Data Management. *Proceedings of the VLDB Endowment* 1, 1 (2008), 1008–1019. https://doi.org/10.14778/1453856.1453965

Cathrin Weiss, Cindy Rubio-González, and Ben Liblit. 2015. Database-backed program analysis for scalable error propagation. In *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*. IEEE, 586–597. https://doi.org/10.1109/ICSE.2015.75

Virginia Vassilevska Williams. 2012. Multiplying matrices faster than Coppersmith-Winograd. In *Proceedings of the 44th ACM Symposium on Theory of Computing (STOC '12)*. ACM, 887–898. https://doi.org/10.1145/2213977.2214056

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 Object-Oriented Programming (ECOOP '09)*. Springer, 98–122. https://doi.org/10.1007/978-3-642-03013-0_6

Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P '14)*. IEEE, 590–604. https://doi.org/10.1109/SP.2014.44

Dacong Yan, Guoqing Xu, and Atanas Rountev. 2011. Demand-driven context-sensitive alias analysis for Java. In *Proceedings of the 20th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '11)*. ACM, 155–165. https://doi.org/10.1145/2001420.2001440

Mihalis Yannakakis. 1990. Graph-theoretic methods in database theory. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '90)*. ACM, 230–242. https://doi.org/10.1145/298514.298576

Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. 2010. Grail: Scalable reachability index for large graphs. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 276–284. https://doi.org/10.14778/1920841.1920879

Daniel H. Younger. 1967. Recognition and parsing of context-free languages in time n3. *Information and Control* 10, 2 (1967), 189–208. https://doi.org/10.1016/S0019-9958(67)80007-X

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 (ESOP '09)*. Springer, 175–189. https://doi.org/10.1007/978-3-642-00590-9_13

Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. 2013. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, 435–446. https://doi.org/10.1145/2499370.2462159

Qirun Zhang and Zhendong Su. 2017. Context-sensitive data-dependence analysis via linear conjunctive language reachability. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*. ACM, 344–358. https://doi.org/10.1145/3009837.3009848

Qirun Zhang, Xiao Xiao, Charles Zhang, Hao Yuan, and Zhendong Su. 2014. Efficient subcubic alias analysis for C. In *Proceedings of the 2014 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '14)*. ACM, 829–845. https://doi.org/10.1145/2660193.2660213

Xin Zheng and Radu Rugina. 2008. Demand-driven alias analysis for C. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, 197–208. https://doi.org/10.1145/1328438.1328464

Junfeng Zhou, Shijie Zhou, Jeffrey Xu Yu, Hao Wei, Ziyang Chen, and Xian Tang. 2017. DAG reduction: Fast answering reachability queries. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, 375–390. https://doi.org/10.1145/3035918.3035927

Li Zhou, Ren Chen, Yinglong Xia, and Radu Teodorescu. 2018. C-Graph: A highly efficient concurrent graph reachability query framework. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP '18)*. ACM, 79:1–79:10. https://doi.org/10.1145/3225058.3225136

Andy Diwen Zhu, Wenqing Lin, Sibo Wang, and Xiaokui Xiao. 2014. Reachability queries on large dynamic graphs: A total order approach. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, 1323–1334. https://doi.org/10.1145/2588555.2612181

Lei Zou, Kun Xu, Jeffrey Xu Yu, Lei Chen, Yanghua Xiao, and Dongyan Zhao. 2014. Efficient processing of label-constraint reachability queries in large graphs. *Information Systems* 40, MAR (2014), 47–66. https://doi.org/10.1016/j.is.2013.10.003