Fast Client-Driven CFL-Reachability via Regularization-Based Graph Simplification

CHENGHANG SHI, Institute of Computing Technology, CAS, China and University of Chinese Academy of Sciences, China
DONGJIE HE, Chongqing University, China
HAOFENG LI[*], Institute of Computing Technology, CAS, China
JIE LU, Institute of Computing Technology, CAS, China
LIAN LI[*], Institute of Computing Technology, CAS, China, University of Chinese Academy of Sciences, China, and Zhongguancun Laboratory, China
JINGLING XUE, University of New South Wales, Australia

Context-free language (CFL) reachability is a critical framework for various program analyses, widely adopted despite its computational challenges due to cubic or near-cubic time complexity. This often leads to significant performance degradation in client applications. Notably, in real-world scenarios, clients typically require reachability information only for specific source-to-sink pairs, offering opportunities for targeted optimization.

We introduce MoYe, an effective regularization-based graph simplification technique designed to enhance the performance of client-driven CFL-reachability analyses by pruning non-contributing edges—those that do not participate in any specified CFL-reachable paths. MoYe employs a regular approximation to ensure exact reachability results for all designated node pairs and operates linearly with respect to the number of edges in the graph. This lightweight efficiency makes MoYe a valuable pre-processing step that substantially reduces both computational time and memory requirements for CFL-reachability analysis, outperforming a recent leading graph simplification approach. Our evaluations with two prominent CFL-reachability client applications demonstrate that MoYe can substantially improve performance and reduce resource consumption.

CCS Concepts: • **Theory of computation → Grammars and context-free languages**.

Additional Key Words and Phrases: CFL-Reachability, Graph Simplification

ACM Reference Format:
Chenghang Shi, Dongjie He, Haofeng Li, Jie Lu, Lian Li, and Jingling Xue. 2025. Fast Client-Driven CFL-Reachability via Regularization-Based Graph Simplification. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 287 (October 2025), 28 pages. https://doi.org/10.1145/3763065

1 Introduction

CFL-reachability [Reps 1998; Yannakakis 1990] is a foundational framework used to formalize a wide array of program analysis tasks, including pointer analysis [He et al. 2023; Lu and Xue 2019; Sridharan et al. 2005], inter-procedural data flow analysis [Arzt et al. 2014; Reps et al. 1995], shape

[*]Corresponding authors

Authors' Contact Information: Chenghang Shi, Institute of Computing Technology, CAS, Beijing, China and University of Chinese Academy of Sciences, Beijing, China, shichenghang21s@ict.ac.cn; Dongjie He, Chongqing University, Chongqing, China, dongjiehe@cqu.edu.cn; Haofeng Li, Institute of Computing Technology, CAS, Beijing, China, lihaofeng@ict.ac.cn; Jie Lu, Institute of Computing Technology, CAS, Beijing, China, lujie@ict.ac.cn; Lian Li, Institute of Computing Technology, CAS, Beijing, China and University of Chinese Academy of Sciences, Beijing, China and Zhongguancun Laboratory, Beijing, China, lianli@ict.ac.cn; Jingling Xue, University of New South Wales, Sydney, Australia, j.xue@unsw.edu.au.

ACM 2475-1421/2025/10-ART287
https://doi.org/10.1145/3763065

analysis [Reps 1998], and program slicing [Li et al. 2016; Reps et al. 1994; Sridharan et al. 2007]. This approach models program analysis by utilizing a context-free language L and represents the analyzed program as an edge-labeled graph G. It determines if, for every potential source-to-sink pair, there exists a path p in G where the sequence of labels along p is a string in L.

CFL-reachability is computationally demanding, exhibiting classic cubic time complexity of $O(|V|^3)$, where V denotes the node set of G. Although a slightly subcubic algorithm with a complexity of $O(|V|^3/\log(|V|))$ exists [Chaudhuri 2008], achieving truly subcubic bounds of $O(|V|^{3-\varepsilon})$, where $\varepsilon > 0$, for general CFL-reachability is notably challenging [Melski and Reps 2000] and continues to be an unresolved problem [Chatterjee et al. 2017]. Despite the introduction of several recent acceleration techniques [Lei et al. 2022a; Shi et al. 2023, 2022; Wang et al. 2017], the performance gains provided still fall short of the needs of client applications that rely on CFL-reachability.

In this paper, we introduce MoYe, a novel approach to optimizing CFL-reachability analysis for client-specific applications. Our primary insight is that clients often need specific source-to-sink reachability information, rather than all possible pairs, which traditional CFL-reachability algorithms [Lei et al. 2022a; Shi et al. 2023, 2022; Zheng and Rugina 2008] process, causing considerable inefficiencies by including numerous extraneous paths. Conventional graph simplification methods [Hardekopf and Lin 2007; Rountev and Chandra 2000], although effective, still retain unnecessary reachability information. A recent state-of-the-art technique, Gf [Lei et al. 2023b], improves efficiency by contracting trivial edges—those irrelevant to any reachable paths—and merging their incident nodes. Diverging from these methods, MoYe exclusively focuses on pruning irrelevant paths, substantially reducing unnecessary computations and enhancing performance.

To achieve this, MoYe uses a unique graph simplification strategy to enhance efficiency. During pre-processing for client-driven CFL-reachability, it removes *non-contributing* (or *useless*) edges—those not part of any source-to-sink paths specified by clients. CFL-reachability algorithms then analyze this simplified graph to determine reachability. By eliminating only these edges, MoYe improves analysis efficiency while maintaining result accuracy for clients.

Addressing the challenges of crafting an effective yet lightweight graph simplification technique, we employ a regular approximation, L', of the context-free language L. This approximation covers all path strings of L and strategically targets the removal of non-contributing edges. By leveraging the relationship between L'-reachability and the transitions of its associated finite automaton, we have developed a novel regularization-based graph simplification algorithm. Operating in linear time relative to the number of edges in the input graph G, this algorithm adeptly eliminates non-contributing edges, ensuring both efficiency and effectiveness in addressing pre-analysis tasks.

We have developed MoYe, a standalone tool that employs our regularization-based graph simplification technique for two significant client analyses: points-to analysis in Java and alias analysis in C/C++. Our extensive testing shows that MoYe significantly speeds up CFL-reachability resolution by removing many non-essential edges, outperforming Gf [Lei et al. 2023b], the current leading method. In points-to analysis, MoYe achieves node and edge reductions of 63.82% and 70.81%, respectively, resulting in a speedup of 6.99× and a memory reduction of 71.65%. Similarly, in alias analysis, it reduces 59.14% of nodes and 65.36% of edges, leading to a speedup of 15.72× and a memory reduction of 80.95%. In comparison, Gf achieves node and edge reductions of 40.22% (39.84%) and 24.44% (36.89%) respectively in these analyses, with speedups of 4.36× (4.21×) and memory reductions of 56.7% (44.28%). Additionally, integrating MoYe with Gf enhances reductions and performance gains in both analyses. Our experiments in a batch setting also confirm that MoYe further enhances the performance of CFL-reachability analysis.

MoYe represents the first regularization-based graph simplification approach specifically designed to enhance CFL-reachability analysis by removing non-contributing edges related to client-specified source-to-sink pairs. In summary, this paper presents the following principal contributions:

- We introduce MoYe, a novel regularization-based graph simplification approach that enhances CFL-reachability analysis by selectively removing non-contributing edges from the input graph of a program, while delivering exact reachability results.
- We develop MoYe as a lightweight regularization-based pre-analysis technique that operates efficiently in linear time relative to the input graph size.
- We implement MoYe as a standalone tool and validate its utility through its application to two common CFL-reachability client analyses, covering both Java and C/C++ programs.
- We demonstrate that MoYe significantly reduces both computational time and memory overhead in CFL-reachability analysis by minimizing graph size through extensive testing, outperforming the current leading graph simplification technique, Gf [Lei et al. 2023b]. Additionally, we demonstrate that MoYe complements existing simplification methods, especially when combined with Gf, achieving even greater performance enhancements.

The rest of this paper is structured as follows. Section 2 provides background on CFL-reachability and introduces a motivating example. Section 3 outlines the specific problem this paper addresses. Section 4 elaborates on our regularization-based approach for identifying contributing edges in client-specified source-to-sink CFL-reachability. Section 5 details our regularization-based graph simplification algorithm. Section 6 presents our evaluation results. Section 7 reviews related work in the field. Section 8 concludes the paper and explores potential directions for future research.

2 Background and Motivation

We start with a review of CFL-reachability (Sections 2.1 and 2.2) and present an illustrative example (Section 2.3) that highlights our research motivation. In the final section on graph simplification (Section 2.4), we discuss prior work and introduce our regularization-based approach, emphasizing the challenges addressed and the innovative solutions proposed.

2.1 Context-Free Language

A context-free language (CFL) L is defined by the strings generated from a context-free grammar (CFG). A CFG is characterized by a 4-tuple (N, Σ, P, S), where N and Σ are disjoint finite sets of nonterminals and terminals, respectively, with Σ also referred to as the *alphabet* of the language. The set P contains a finite number of production rules, each formatted as $N \longrightarrow (N \cup \Sigma)^*$. Here, $S \in N$ serves as the start (nonterminal) symbol of the grammar.

Following [Mohri and Nederhof 2001], we define the *productions of* A as those productions whose left-hand side is $A \in N$. For any subset $N' \subseteq N$, the *productions of* N' are defined as the union of all rules whose left-hand sides belong to any $A \in N'$.

A grammar is *left-linear* (*right-linear*) if all its productions in P are of the form $A \longrightarrow B\ \omega$ ($A \longrightarrow \omega\ B$) or $A \longrightarrow \omega$, where $A, B \in N$ and $\omega \in \Sigma^*$. A context-free language L is considered a *regular language* if its corresponding CFG is either left-linear or right-linear.

2.2 CFL-Reachability

CFL-reachability is a widely used framework for various program analyses [Reps 1998]. In this framework, a CFG is used to formalize an analysis problem, and the program being analyzed is represented as an edge-labeled graph $G = (V, E)$, with V and E denoting the sets of nodes and edges, respectively. Each edge in G is labeled with a terminal $t \in \Sigma$.

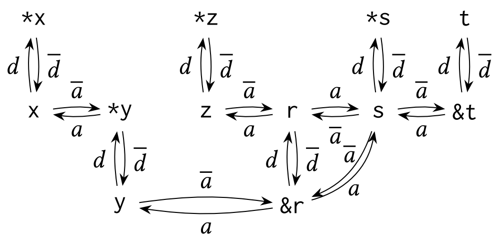
Path Strings. Consider a path p in G, represented as $v_0 \xrightarrow{t_1} v_1 \xrightarrow{t_2} \dots \xrightarrow{t_k} v_k$, where each t_i for $i \in [1, k]$ is a terminal in Σ. The *realized path string*, $R(p)$, is the sequence obtained by concatenating the labels of the edges along the path, i.e., $R(p) = t_1 \dots t_k$.

```
1 s = &t;
2 s = &r;
3 z = r;
4 y = &r;
5 s = r;
6 x = *y;
7 ... = *x+2;
8 *s = ...;
9 ... = *x+2;
```

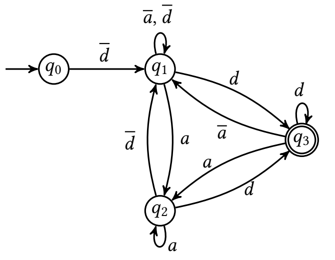
(a) Code snippet

$$
\begin{array}{lcl}
M & \longrightarrow & \overline{d}\, V\, d \\
V & \longrightarrow & \overline{A}\, V \mid V\, A \\
 & & \mid M \mid \epsilon \\
A & \longrightarrow & a\, M \mid a \\
\overline{A} & \longrightarrow & M\, \overline{a} \mid \overline{a}
\end{array}
$$

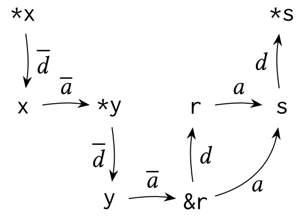
(b) Context-free grammar



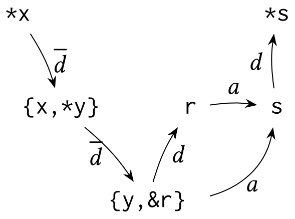
(c) Input graph G



(d) Deterministic finite automaton



(e) Simplified input graph G' by MoYe



(f) Simplified input graph by combing MoYe and Gf

Fig. 1. A motivating example.

Reachable Paths/Pairs. A node v is said to be X*-reachable* from a node u if there exists a path p from u to v in G such that the path string $R(p)$ belongs to the language generated by the nonterminal X in the grammar. Formally, this is expressed as $X \longrightarrow^{*} R(p)$, where $\longrightarrow^{*}$ indicates zero or more applications of productions from P. Such a path, referred to as an X*-path*, is depicted as a *summary edge* in G, labeled by X to explicitly encode the reachability information. In the special case when X is the start symbol (i.e., $X = S$), node v is described as L*-reachable* from node u, with the path p termed an L*-path*. Consequently, the pair (u, v) is defined as an L*-reachable pair*. A summary edge can be derived multiple times through distinct reachable paths, with each path corresponding to a distinct *derivation check* [Lei et al. 2022a; Shi et al. 2023].

Essentially, CFL-reachability algorithms [Lei et al. 2022a; Shi et al. 2023, 2022] iteratively discover new reachable paths and generate summary edges in G. This process continues until no further paths can be discovered, indicating that a fixed point has been reached. The overall time complexity of this process is (sub)cubic relative to $|V|$ [Chaudhuri 2008].

2.3 A Motivating Example

Consider the code snippet in Figure 1a as part of an available expression analysis. This analysis determines whether the expression *x+2 at line 9 can reuse the value computed at line 7 or if it requires recomputation. The reusability of *x+2 hinges on whether *x remains unchanged from line 7 to line 9. A crucial ***alias query*** then emerges: Are the memory locations *x and *s aliased? If they are, the assignment to *s at line 8 might modify *x, thereby rendering the expression *x+2, previously computed at line 7, unavailable at line 9.

This alias query can be addressed using CFL-reachability-based alias analysis [Zheng and Rugina 2008]. Figure 1b illustrates the grammar defining alias relations between program expressions. In this grammar, the terminals d, a, and ϵ symbolize pointer dereferencing, assignment, and the empty string, respectively. The nonterminals include M (*memory aliasing*), indicating expressions that may refer to the same memory location; V (*value aliasing*), signifying expressions that may

Table 1. Step-by-step derivation of $M \longrightarrow^* \overline{d}\ \overline{a}\ \overline{d}\ \overline{a}\ d\ a\ d$ for the motivating example in Figure 1.

Step	Current String	Production
0	M	$M \longrightarrow \overline{d}\ V\ d$
1	$\overline{d}\ V\ d$	$V \longrightarrow \overline{A}\ V$
2	$\overline{d}\ \overline{A}\ V\ d$	$\overline{A} \longrightarrow \overline{a}$
3	$\overline{d}\ \overline{a}\ V\ d$	$V \longrightarrow V\ A$
4	$\overline{d}\ \overline{a}\ V\ A\ d$	$V \longrightarrow M$
5	$\overline{d}\ \overline{a}\ M\ A\ d$	$M \longrightarrow \overline{d}\ V\ d$
6	$\overline{d}\ \overline{a}\ \overline{d}\ V\ d\ A\ d$	$V \longrightarrow \overline{A}\ V$
7	$\overline{d}\ \overline{a}\ \overline{d}\ \overline{A}\ V\ d\ A\ d$	$\overline{A} \longrightarrow \overline{a}$
8	$\overline{d}\ \overline{a}\ \overline{d}\ \overline{a}\ V\ d\ A\ d$	$V \longrightarrow \epsilon$
9	$\overline{d}\ \overline{a}\ \overline{d}\ \overline{a}\ d\ A\ d$	$A \longrightarrow a$
10	$\overline{d}\ \overline{a}\ \overline{d}\ \overline{a}\ d\ a\ d$	

evaluate to a common pointer; and A, which captures both direct ($A \longrightarrow a$) and indirect ($A \longrightarrow a\ M$) assignments. Additionally, for any symbol X, $\overline{X}$ denotes the *inverse relation* of X. For additional details about this grammar, please refer to [Zheng and Rugina 2008].

Figure 1c displays the input graph G extracted from the code snippet in Figure 1a, where each node corresponds to an expression in the code. This graph consists of four types of edges: a-edges, $\overline{a}$-edges, d-edges and $\overline{d}$-edges. The alias query, as discussed earlier, is resolved by determining the reachability from the source node `*x` to the sink node `*s` in G.

In this example, `*x` and `*s` are memory aliases because the following path exists in G:

$$\texttt{*x} \xrightarrow{\overline{d}} \texttt{x} \xrightarrow{\overline{a}} \texttt{*y} \xrightarrow{\overline{d}} \texttt{y} \xrightarrow{\overline{a}} \texttt{\&r} \xrightarrow{d} \texttt{r} \xrightarrow{a} \texttt{s} \xrightarrow{d} \texttt{*s} \tag{1}$$

The corresponding path string $\overline{d}\ \overline{a}\ \overline{d}\ \overline{a}\ d\ a\ d$ can be derived from the nonterminal M, as shown in Table 1, indicating that `*s` is M-reachable from `*x`. Consequently, in Figure 1a, the assignment at line 8 invalidates the availability of the expression `*x+2` at line 9.

2.4 Graph Simplification

We aim to develop a graph simplification technique that accelerates CFL-reachability analysis by reducing the input graph size. As shown in Figure 1, our alias analysis query requires establishing reachability only between `*x` and `*s`. Therefore, any simplification that maintains this specific reachability can effectively optimize the analysis for this client query. Below, we will review prior work and elaborate on our approach, focusing on the challenges and our innovative solution.

Prior Work. Existing graph simplification techniques often retain much irrelevant reachability information, leading to only modest reductions in graph size. For instance, the classic cycle elimination technique [Hardekopf and Lin 2007] does not apply in our motivating example because it may lead to missing A-edges produced by the production $A \longrightarrow a\ M$, which relies on transitive a-edges, potentially compromising analysis soundness [Xu et al. 2024]. Moreover, a-edges do not form cycles in G. A recent method known as Graph Folding (Gf) [Lei et al. 2023b], guided by recursive state machines, identifies foldable adjacent nodes by analyzing their incoming and outgoing edges. Adjacent node pairs are then merged, and all connecting edges considered trivial—primarily transitive edges in practical applications—are removed, since their contraction preserves CFL-reachability. In Figure 1, Gf contracts only three a-edges (and their reverse edges): $\texttt{*y} \xrightarrow{a} \texttt{x}$, $\texttt{\&r} \xrightarrow{a} \texttt{y}$, and $\texttt{r} \xrightarrow{a} \texttt{z}$, achieving a 25% reduction in nodes and 25% reduction in edges. However, Gf cannot contract $\texttt{r} \xrightarrow{a} \texttt{s}$, $\texttt{\&r} \xrightarrow{a} \texttt{s}$, and $\texttt{\&t} \xrightarrow{a} \texttt{s}$ due to the multiple incoming a-edges at `s`.

Our Work. Unlike existing graph simplification techniques [Hardekopf and Lin 2007; Lei et al. 2023b], MoYe focuses on simplifying the input graph G by preserving primarily the reachability information relevant to client needs. This strategy allows for the removal of many edges that do not contribute to the specific CFL-reachability paths of interest, though some redundancy may still remain. Figure 1e illustrates the resulting simplified graph G', which preserves those edges that form the L-reachable path from `*x` to `*s` (Equation (1)), removing 16 out 17 non-contributing edges for this particular example. However, the edge $\texttt{\&r} \xrightarrow{a} \texttt{s}$ is conservatively retained by MoYe due to its approximation approach. Utilizing the simplified G' instead of the original G substantially enhances CFL-reachability analysis efficiency, applicable to both whole-program and demand-driven scenarios. For instance, an alias query starting at `*x` in G would unnecessarily traverse to `*z`, whereas in G', with `*z`, `z`, and associated edges removed, such irrelevant paths are eliminated.

Challenges. Designing MoYe as a pre-analysis presents two key challenges: maximizing the removal of non-contributing edges to minimize the computational and memory overhead of CFL-reachability analysis while ensuring the pre-analysis remains lightweight. Ideally, we would precisely identify contributing edges and remove all others. A naive approach would compute all CFL-reachable paths and mark non-contributing edges, but this would defeat the purpose, as it requires solving CFL-reachability upfront, incurring significant computational costs.

Key Idea. MoYe over-approximates the context-free language in Figure 1b with a regular language, which is then converted into a deterministic finite automaton (DFA) $\mathcal{D}$, as shown in Figure 1d. By analyzing the DFA's transition rules, it over-approximately identifies contributing edges and removes non-contributing edges from G under-approximately. The DFA has a small number of states, comparable to the nonterminals in the original CFG, ensuring efficient computation.

Let us summarize four key advantages offered by MoYe in the context of our motivating example:

(1) **Effectiveness**. MoYe efficiently removes all non-contributing edges, achieving a reduction of 33.33% (4 out of 12) of nodes and 66.67% (16 out of 24) of edges, as shown in Figure 1e. Nodes such as `z` and `&t` are automatically removed as they no longer have incident edges after simplification. In contrast, Gf [Lei et al. 2023b] only eliminates only 25% of nodes and 25% of edges through edge contraction. Notably, in the evaluation presented in Section 6, MoYe consistently outperforms Gf in speeding up CFL-reachability analysis, often significantly.

(2) **Preservation**. MoYe simplifies the input graph G into G' while preserving the reachability needed for the specific alias query considered. The M-path from `*x` to `*s` given in Equation (1) is retained in G' (Figure 1e), ensuring the reachability results remain unchanged.

(3) **Efficiency**. MoYe operates with a time complexity linear to the number of edges in the input graph G, compared to the cubic complexity of CFL-reachability algorithms, making it highly efficient as a pre-analysis step to accelerate these algorithms.

(4) **Compatibility**. MoYe is compatible with existing methods like Gf [Lei et al. 2023b], enhancing graph simplification. Gf can contract contributing edges (e.g., $\texttt{x} \xrightarrow{\overline{a}} \texttt{*y}$, $\texttt{y} \xrightarrow{\overline{a}} \texttt{\&r}$) to shorten paths, while MoYe can remove non-contributing edges that Gf cannot contract to prune irrelevant paths. Combining both methods, as shown in Figure 1f, results in a reduction of nodes by 50% and edges by 75%, surpassing the performance of using either technique alone. This synergy will be further analyzed in Section 6.

3 Problem Formulation

In this work, we focus on addressing client-driven CFL-reachability problems, in which the source and sink sets are defined by client applications, as formally stated below.

Definition 3.1 (Client-Driven CFL-Reachability Problem). An instance of a *client-driven CFL-reachability problem* is represented as a quadruple $(L, G, V_{\mathrm{src}}, V_{\mathrm{snk}})$, where L is a context-free language, G is an edge-labeled graph, and $V_{\mathrm{src}} \subseteq V$ and $V_{\mathrm{snk}} \subseteq V$ are the sets of source and sink nodes specified by a client. The goal is to compute all L-reachable node pairs $(u, v) \in V_{\mathrm{src}} \times V_{\mathrm{snk}}$ in G.

This scenario is typical in real-world program analysis tools, where queries are often processed in batches [Vedurada and Nandivada 2020]. For instance, in pointer analysis performed for constructing a call graph in a Java program, only the points-to sets of pointers that act as base variables for virtual invocation sites in the program are required.

Given a specific client analysis, we can formally define the "usefulness" of an edge in G by determining whether it contributes to particular source-to-sink reachable paths.

Definition 3.2 (L-Contributing Edges). Let $I = (L, G, V_{\mathrm{src}}, V_{\mathrm{snk}})$ be an instance of a client-driven CFL-reachability problem. An edge $u \xrightarrow{t} v \in G$ is defined as an L-*contributing edge* if and only if there exists an L-reachable path $v_{\mathrm{src}} \to \cdots \to u \xrightarrow{t} v \to \cdots \to v_{\mathrm{snk}}$ in G, where $v_{\mathrm{src}} \in V_{\mathrm{src}}$ and $v_{\mathrm{snk}} \in V_{\mathrm{snk}}$. If no such path exists, the edge is classified as a *non-contributing edge* of I.

Therefore, not all edges in G are necessary for calculating the desired L-reachable pairs. By removing non-contributing edges, we can improve the performance of CFL-reachability analysis on a simplified graph G', while maintaining exactly the same reachability results for clients.

Example 3.3. Revisiting our motivating example in Figure 1, the edges in the path p in Equation (1) are contributing edges, as they form the M-path from the source node `*x` to the sink node `*s`. However, the remaining edges are non-contributing. Although there is an M-path from `*x` to `*z`, it is irrelevant to the specific alias analysis considered.

We formulate our graph simplification problem as follows:

Given an instance $(L, G, V_{\mathrm{src}}, V_{\mathrm{snk}})$ of a client-driven CFL-reachability problem, the goal is to produce a simplified graph G' by removing non-contributing edges from G.

The technique proposed in this paper can be used as a pre-processing step for any client-driven CFL-reachability problem, whether applied in a demand-driven manner [Shi et al. 2022; Sridharan et al. 2005; Zheng and Rugina 2008] or for all-pairs reachability [Lei et al. 2022a; Shi et al. 2023].

4 Identifying L-Contributing Edges

Given $(L, G, V_{\mathrm{src}}, V_{\mathrm{snk}})$, we over-approximately identify L-contributing edges by first regularizing L into L' (Section 4.1), and then determining L'-contributing edges for V_{src} and V_{snk} in G (Section 4.2).

4.1 Regularizing L to L'

We first introduce a standard regular approximation technique for a CFL (Section 4.1.1) and then explain how to use it to soundly over-approximate the set of L-contributing edges (Section 4.1.2).

4.1.1 MN-Transformation. This represents a simple yet effective algorithm for converting a CFL L into a regular language L' [Mohri and Nederhof 2001].

Given a CFG (N, Σ, P, S), let $\mathcal{R}$ be the relation defined on its nonterminals $A, B \in N$:

$$A \,\mathcal{R}\, B \Leftrightarrow (\exists\, \alpha, \beta \in (\Sigma \cup N)^*, A \to^* \alpha B \beta) \wedge (\exists\, \alpha, \beta \in (\Sigma \cup N)^*, B \to^* \alpha A \beta)$$

Intuitively, $\mathcal{R}$ defines an equivalence relation that partitions the set N of nonterminals into subsets of *mutually recursive nonterminals*. For each partition $\mathscr{P}$, two steps are applied to convert each production into right-linear form if its productions are not all left-linear or right-linear:

$$\begin{array}{lcl@{\qquad\qquad}lcl}
M & \longrightarrow & \overline{d}\,V & M' & \longrightarrow & V' \mid A' \mid \overline{a}\,\overline{A'} \mid \epsilon \\
V & \longrightarrow & \overline{A} \mid M \mid V' & V' & \longrightarrow & d\,M' \mid A \\
A & \longrightarrow & a\,M \mid a\,A' & A' & \longrightarrow & V' \\
\overline{A} & \longrightarrow & M \mid \overline{a}\,\overline{A'} & \overline{A'} & \longrightarrow & V
\end{array}$$

Fig. 2. The regular grammar derived from the context-free grammar given in Figure 1b.

(1) *Introduce new nonterminal symbols.* For each nonterminal $A \in \mathscr{P}$, introduce a new nonterminal $A' \in N$, and add the ϵ-production $A' \longrightarrow \epsilon$ to the grammar.
(2) *Replace productions.* For each production of the form $A \longrightarrow \alpha_0 B_1 \alpha_1 B_2 \alpha_2 \ldots B_m \alpha_m$, where $B_1, \ldots, B_m \in \mathscr{P}$, $\alpha_0, \ldots, \alpha_m \in (\Sigma \cup (N - \mathscr{P}))^*$, and $m \geq 0$, replace it with:

$$\begin{aligned}
A &\longrightarrow \alpha_0 B_1 \\
B'_1 &\longrightarrow \alpha_1 B_2 \\
&\cdots \\
B'_{m-1} &\longrightarrow \alpha_{m-1} B_m \\
B'_m &\longrightarrow \alpha_m A'
\end{aligned}$$

In the case of $m = 0$ (where the production is $A \longrightarrow \alpha_0$), replace it with $A \longrightarrow \alpha_0 A'$.

Strongly regular grammars generate regular languages and can be converted into equivalent finite automata using the standard algorithm outlined in [Mohri and Nederhof 2001] and further detailed in [Nederhof 2000]. *Strongly regular grammars* are characterized by productions within each partition $\mathscr{P}$ being exclusively left-linear or right-linear. When determining the linearity of a production in $\mathscr{P}$, nonterminals not within $\mathscr{P}$ are treated as terminals.

By construction, L' is a superset of L, meaning any string derivable in L is also derivable in L'. For each nonterminal X in L, at most one new nonterminal X' is added to L', where X and X' represent the start and end of string recognition generated by X in L. This ensures that the number of nonterminals in L' is at most double that of L, which is beneficial in practice.

As an optimization, Step (1) can be refined by adding the ϵ-production $A' \longrightarrow \epsilon$ only when $A \in \mathscr{P}$ is directly reachable from another set of mutually recursive nonterminals $\mathscr{P}'$ [Mohri and Nederhof 2001]. In this case, $\mathscr{P}'$ uses $\mathscr{P}$ to generate strings in the language of A. This refinement ensures that $\mathscr{P}$ begins with A and concludes with the production $A' \longrightarrow \epsilon$, effectively preventing undesirable strings from being introduced into the transformed grammar.

Example 4.1. In Figure 1b, the CFG contains a single partition of mutually recursive nonterminals, $\{M, V, A, \overline{A}\}$. The corresponding regular grammar obtained via the MN-transformation is shown in Figure 2. Since we focus on the memory alias relation in this example, M is the start symbol. With the refinement for Step (1) mentioned above, only $M' \longrightarrow \epsilon$ is introduced in Step (1) during the MN-transformation. This ensures that string generation terminates exclusively with $M' \longrightarrow \epsilon$.

For the M-path given in Equation (1), denoted here as p, the path string $R(p)$ can obviously be derived from this regular grammar as follows: $M \longrightarrow \overline{d}\;V \longrightarrow \overline{d}\;\overline{A} \longrightarrow \overline{d}\;\overline{a}\;\overline{A'} \longrightarrow \overline{d}\;\overline{a}\;V \longrightarrow \overline{d}\;\overline{a}\;M \longrightarrow \overline{d}\;\overline{a}\;\overline{d}\;V \longrightarrow \overline{d}\;\overline{a}\;\overline{d}\;\overline{A} \longrightarrow \overline{d}\;\overline{a}\;\overline{d}\;\overline{a}\;\overline{A'} \longrightarrow \overline{d}\;\overline{a}\;\overline{d}\;\overline{a}\;V \longrightarrow \overline{d}\;\overline{a}\;\overline{d}\;\overline{a}\;V' \longrightarrow \overline{d}\;\overline{a}\;\overline{d}\;\overline{a}\;d\;M' \longrightarrow \overline{d}\;\overline{a}\;\overline{d}\;\overline{a}\;d\;V' \longrightarrow \overline{d}\;\overline{a}\;\overline{d}\;\overline{a}\;d\;A \longrightarrow \overline{d}\;\overline{a}\;\overline{d}\;\overline{a}\;d\;a\;A' \longrightarrow \overline{d}\;\overline{a}\;\overline{d}\;\overline{a}\;d\;a\;V' \longrightarrow \overline{d}\;\overline{a}\;\overline{d}\;\overline{a}\;d\;a\;d\;M' \longrightarrow \overline{d}\;\overline{a}\;\overline{d}\;\overline{a}\;d\;a\;d$. However, the underlying regular language also includes unwanted strings. For instance, consider path p':

$$\texttt{*x} \xrightarrow{\overline{d}} \texttt{x} \xrightarrow{\overline{a}} \texttt{*y} \xrightarrow{\overline{d}} \texttt{y} \xrightarrow{\overline{a}} \texttt{\&r} \xrightarrow{a} \texttt{s} \xrightarrow{d} \texttt{*s} \tag{2}$$

$R(p')$ represents an unwanted string derived as follows: $M \longrightarrow^* \overline{d}\;\overline{a}\;\overline{d}\;\overline{a}\;V' \longrightarrow \overline{d}\;\overline{a}\;\overline{d}\;\overline{a}\;A \longrightarrow \overline{d}\;\overline{a}\;\overline{d}\;\overline{a}\;a\;A' \longrightarrow \overline{d}\;\overline{a}\;\overline{d}\;\overline{a}\;a\;V' \longrightarrow \overline{d}\;\overline{a}\;\overline{d}\;\overline{a}\;a\;d\;M' \longrightarrow \overline{d}\;\overline{a}\;\overline{d}\;\overline{a}\;a\;d$. Here, p' qualifies as an L'-path but not an L-path, since L' fails to account for the matching behavior between the terminals $\overline{d}$ and

d. Without the refinement in Step (1), the production $\overline{A}' \to \epsilon$ would be included, allowing $R(p)$'s derivation to prematurely terminate at $M \to^* \overline{d}\ \overline{a}\ \overline{A'}$, resulting in the incomplete string $\overline{d}\ \overline{a}$, which is also not part of the original language.

4.1.2 Correctness for Approximating L-Contributing Edges. We now examine the correctness of our graph simplification technique. Given a CFL L and its regular approximation L', L' "relaxes" language reachability (Lemma 4.2), allowing us to over-approximate the set of L-contributing edges in G (Lemma 4.3) and ensure the correctness of the graph simplification (Theorem 4.6).

Lemma 4.2 (Relaxed Reachability). Consider any two nodes u and v in G. If v is L-reachable from u via a path p in G, then v must also be L'-reachable from u via p.

Proof Sketch. Let the set of strings generated by L be $\mathcal{L}(L)$. Without loss of generality, let us assume v is L-reachable from u via path p such that $R(p) \in \mathcal{L}(L)$, and since L' over-approximates L ($\mathcal{L}(L) \subseteq \mathcal{L}(L')$), it follows that $R(p) \in \mathcal{L}(L')$. Therefore, v is L'-reachable from u via p. □

Lemma 4.3. In G, if an edge $u \xrightarrow{t} v$ is L-contributing, then it is also L'-contributing.

Proof Sketch. Assume that $u \xrightarrow{t} v$ contributes to an L-reachable path p. According to Lemma 4.2, p must also be an L'-reachable path, which $u \xrightarrow{t} v$ contributes to. □

Corollary 4.4. In G, if an edge $u \xrightarrow{t} v$ is non-contributing to L'-reachability, then it is also non-contributing to L-reachability.

Example 4.5. Continuing from Example 4.1, where L is provided in Figure 1b and L' in Figure 2, the M-reachable path from `*s` to `*x` under L is also M-reachable under L'. Evidently, any path that qualifies as an M-path in L-reachability also qualifies in L'-reachability. However, if we instead select `*x` and `r` as the source and sink, and let p' be the sub-path from `*x` to `r` in Equation (1), we find that p' qualifies as an M-path in L'-reachability but not in L-reachability.

By removing edges in G that do not contribute to L', we create a simplified graph G'. Since these edges do not contribute to any L-reachable pair in $V_{\text{src}} \times V_{\text{snk}}$, verifying L-reachability on both G and G' should yield the same set of L-reachable pairs. This confirms the correctness of this regular approximation as a graph simplification technique, as formally stated below.

Theorem 4.6 (Correctness of Regular Approximation). Given an instance $(L, G, V_{\text{src}}, V_{\text{snk}})$ of a client-driven CFL-reachability problem, solving L-reachability from any source in V_{src} to any sink in V_{snk} on G and its simplified version G' yields the same set of L-reachable source-to-sink pairs.

Proof Sketch. Let E_C and E'_C represent the sets of L-contributing and L'-contributing edges, respectively. According to Lemma 4.3, $E_C \subseteq E'_C$, making E'_C an over-approximation of E_C. Therefore, the edges that are not in E'_C are non-contributing to L-reachability (Corollary 4.4) and can be safely removed from G, resulting in the simplified graph G'. This removal does not affect the L-reachability of any source-to-sink paths from V_{src} to V_{snk}, ensuring the simplification maintains the accuracy of CFL-reachability analysis on G'. □

It is important to note that the graph simplification process reduces not only the edge set but also the node set of G. Intuitively, a node u with only non-contributing incoming and outgoing edges can be safely removed. This effect is demonstrated in Figure 1e from our motivating example, where four extraneous nodes—`*z`, `z`, `t`, and `&t`—are removed from the input graph G to obtain G'.

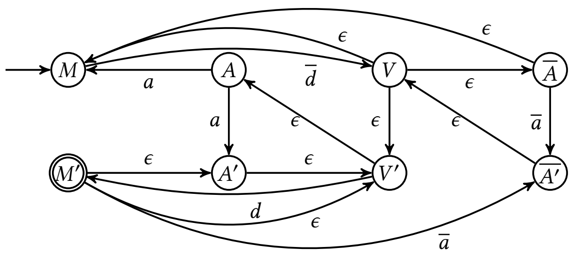


Fig. 3. NFA derived from the regular grammar in Figure 2, with the initial state M and as the final state M'.

4.2 Identifying L'-Contributing Edges

After regularizing L to L' as described in Section 4.1, we proceed to identify L'-contributing edges as an over-approximation of L-contributing edges. To achieve this, we examine the transition rules of a finite automaton, which is equivalent to the regular language L'. Specifically, we track the state transitions of nodes in G, using them to guide the identification of L'-contributing edges.

4.2.1 From L' to Finite Automaton. L', obtained by regularizing L in Section 4.1.1, is strongly regular, enabling its straightforward conversion into a finite automaton [Nederhof 2000].

Finite Automata. A deterministic finite automaton (DFA) is a 5-tuple $\mathcal{A} = (Q, \Sigma, \delta, q_{\text{init}}, F)$, where Q is a finite set of states, Σ a finite alphabet, $\delta : Q \times \Sigma \to Q$ the transition function, $q_{\text{init}} \in Q$ the initial state, and $F \subseteq Q$ the set of final states. A nondeterministic finite automaton (NFA) extends a DFA by allowing $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$, where ϵ permits transitions without consuming an input symbol and $\mathcal{P}(Q)$ allows multiple possible next states.

Definition 4.7 (Transition Sequence). Consider a string $\omega = t_1 t_2 \dots t_k$, where $t_i \in \Sigma$. If there exists a chain of state transitions: $q_0 \xrightarrow{t_1} q_1 \xrightarrow{t_2} \dots \xrightarrow{t_k} q_k$ in a finite automaton $\mathcal{A}$, such that each transition $q_{i-1} \xrightarrow{t_i} q_i$ is defined in the transition function δ for all i from 1 to k, then this sequence of transitions represents the *transition sequence* for string ω in $\mathcal{A}$.

Let us examine this definition in more detail. If $\mathcal{A}$ is deterministic, the transition sequence for a string is unique, provided it exists. Note that q_0 in this definition may not necessarily be the initial state q_{init}. The transition sequence of a string serves as a witness to the computation of $\mathcal{A}$ as it processes the string. A string is *accepted* by the automaton if, and only if, its transition sequence begins at the initial state q_{init} and ends at a final state $q_{\text{f}} \in F$.

Translation to NFA. Given a strongly regular grammar L' (Section 4.1.1), we convert it into an equivalent NFA $\mathcal{N}$ following [Mohri and Nederhof 2001]. We set S as the initial state and S' as the final state of $\mathcal{N}$, where S is the start nonterminal of the original grammar L, and S' is a new nonterminal introduced in the MN-transformation (Section 4.1). The production $S' \to \epsilon$ allows S' to generate the empty string, completing string recognition. Hence, a string in L' is accepted by $\mathcal{N}$ if it follows a transition sequence from S to S'.

Conversion to DFA. The NFA $\mathcal{N}$ is converted into an equivalent DFA $\mathcal{D}$ using the standard subset construction algorithm [Aho et al. 2006]. We then employ Hopcroft's algorithm [Hopcroft 1971] for state minimization to eliminate redundant states. The resulting DFA accurately simulates the behavior of the original NFA, facilitating predictable and efficient computations.

Example 4.8. Figure 3 shows the NFA derived from the regular grammar L' in Figure 2, with the corresponding DFA in Figure 1d. Revisiting the M-path from `*x` to `*s` (Equation (1)) in our motivating example, Figure 4 illustrates the relationship of this path (reproduced in Figure 4a) with

$$\begin{tikzcd}
\texttt{*x} \arrow[r, "\overline{d}"] & \texttt{x} \arrow[r, "\overline{a}"] & \texttt{*y} \arrow[r, "\overline{d}"] & \texttt{y} \arrow[r, "\overline{a}"] & \texttt{\&r} \arrow[r, "d"] & \texttt{r} \arrow[r, "a"] & \texttt{s} \arrow[r, "d"] & \texttt{*s}
\end{tikzcd}$$

(a) The M-path from `*x` to `*t` in Equation (1)

$$\begin{tikzcd}
M \arrow[r, "\overline{d}"] & V \arrow[r, "\epsilon"] & \overline{A} \arrow[r, "\overline{a}"] & \overline{A'} \arrow[r, "\epsilon"] & V \arrow[r, "\epsilon"] & M \arrow[r, "\overline{d}"] & V \arrow[r, "\epsilon"] & \overline{A} \arrow[r, "\overline{a}"] & \overline{A'} \arrow[d, "\epsilon"] \\
& M' & V' \arrow[l, "d"'] & A' \arrow[l, "\epsilon"'] & A \arrow[l, "a"'] & V' \arrow[l, "\epsilon"'] & M' \arrow[l, "\epsilon"'] & V' \arrow[l, "d"'] & V \arrow[l, "\epsilon"']
\end{tikzcd}$$

(b) Transition sequence on the NFA in Figure 3

$$\begin{tikzcd}
q_0 \arrow[r, "\overline{d}"] & q_1 \arrow[r, "\overline{a}"] & q_1 \arrow[r, "\overline{d}"] & q_1 \arrow[r, "\overline{a}"] & q_1 \arrow[r, "d"] & q_3 \arrow[r, "a"] & q_2 \arrow[r, "d"] & q_3
\end{tikzcd}$$

(c) Transition sequence on the DFA in Figure 1d

Fig. 4. The M-path from Figure 1 (as given in Equation (1)) and the transition sequences of its path string.

the transition sequences in both the NFA and DFA. The NFA moves from state M to M' through 16 transitions (Figure 4b) , while the DFA progresses from state q_0 to q_3 in just 7 transitions (Figure 4c), illustrating the DFA's efficiency. Both sequences correspond to the M-path defined by L'.

4.2.2 Determining L'-Contributing Edges. Since the regular language L' and the DFA $\mathcal{D}$ are equivalent, we can utilize the DFA's state transitions to identify L'-contributing edges.

The DFA $\mathcal{D}$ provides an alternative representation of the regular language L'. Consider a path p in G: $v_0 \xrightarrow{t_1} v_1 \xrightarrow{t_2} \dots \xrightarrow{t_k} v_k$, where the path string $R(p) = t_1t_2 \dots t_k$ forms a string $\omega \in \Sigma^*$. Since $\mathcal{D}$ is deterministic, the transition sequence for ω is unique, if it exists. Although determinism is not crucial for identifying L'-contributing edges, converting the original NFA $\mathcal{N}$ to a state-minimal DFA $\mathcal{D}$ often reduces the number of states practically, despite no theoretical guarantee—a well-established fact. This reduction, demonstrated in Figures 7 and 8 for our two key client analyses, accelerates the identification process, as explained in Example 4.8.

Correspondence between L'-Reachable Paths and Transition Sequences. Since L' and $\mathcal{D}$ are equivalent, it is well-known that a string ω belongs to L' if and only if it is accepted by $\mathcal{D}$.

Lemma 4.9. Let L' be a regular language and $\mathcal{A} = (Q, \Sigma, \delta, q_{\text{init}}, F)$ its equivalent finite automaton. A path $p = v \longrightarrow \cdots \longrightarrow v'$ in G is L'-reachable if and only if the path string $R(p)$ has a transition sequence starting at q_{init} and ending at a final state $q_{\text{f}} \in F$.

Proof Sketch. Given the equivalence between L' and $\mathcal{A}$, this proof is derived directly from the definitions of L'-path (Section 2.2) and transition sequence (Definition 4.7). □

Configurations. This lemma illustrates that comparing a path p with its transition sequence $R(p)$ aligns each automaton state with a specific node in G, forming a sequence of configurations: $(v_0, q_0) \xrightarrow{t_1} (v_1, q_1) \xrightarrow{t_2} \dots \xrightarrow{t_k} (v_k, q_k)$. Each pair (v_i, q_i) from $V \times Q$ constitutes a *configuration* for node v_i. A node in G can have multiple configurations based on its paths. We focus on identifying *realizable configurations*, which are crucial for establishing L'-reachable pairs $(u, v) \in V_{\text{src}} \times V_{\text{snk}}$.

Definition 4.10 (Realizable Configurations)**.** A configuration $(v, q) \in V \times Q$ is called *realizable* if the following two conditions are both satisfied:

(1) There exists a path in G from some node $v_{\text{src}} \in V_{\text{src}}$ to v that corresponds to a transition sequence from the initial state q_{init} to q, and
(2) There exists a path in G from v to some node $v_{\text{snk}} \in V_{\text{snk}}$ that corresponds to a transition sequence from q to a final state $q_{\text{f}} \in F$.

Note that v_{src} and v_{snk} may not be necessarily distinct from v.

Intuitively, this definition implies the existence of an L'-path from v_{src} to v_{snk} via node v. According to the two conditions, there exists a path p in G of the form $v_{\text{src}} \to \cdots \to v \to \cdots \to v_{\text{snk}}$. The corresponding path string $R(p)$ has an associated transition sequence from q_{init} to q_{f}, passing through state q at node v. By Lemma 4.9, this means that v_{snk} is L'-reachable from v_{src}.

Finally, we present a necessary and sufficient condition based on realizable configurations that enables us to identify L'-contributing edges in G efficiently.

Theorem 4.11 (Determining L'-Contributing Edges)**.** In G, an edge $u \xrightarrow{t} v$ is L'-contributing if and only if there exist realizable configurations (u, q_1) and (v, q_2), such that $q_1 \times t \to q_2 \in \delta$.

Proof Sketch. To prove the implication "$\Longrightarrow$", assume that $u \xrightarrow{t} v$ is part of an L'-path $v_{\text{src}} \to \cdots \to u \xrightarrow{t} v \to \cdots \to v_{\text{snk}}$. According to Lemma 4.9, there exists a configuration chain $(v_{\text{src}}, q_{\text{init}}) \to \cdots \to (u, q_1) \xrightarrow{t} (v, q_2) \to \cdots \to (v_{\text{snk}}, q_{\text{f}})$. By Definition 4.10, the configurations (u, q_1) and (v, q_2) are realizable. Furthermore, the transition from (u, q_1) to (v, q_2) is effected by the edge $u \xrightarrow{t} v$, verifying that this edge contributes to the reachability from v_{src} to v_{snk}.

To prove the implication "$\Longleftarrow$", assume (u, q_1) and (v, q_2) are realizable configurations. This implies there exist (1) a path from v_{src} to u corresponding to a transition sequence $q_{\text{init}} \to \cdots \to q_1$, and (2) a path from v to v_{snk} corresponding to a transition sequence $q_2 \to \cdots \to q_{\text{f}}$. The transition $q_1 \times t \to q_2$ effectively connects these two paths, establishing an L'-path as outlined in Lemma 4.9. This linkage confirms that $u \xrightarrow{t} v$ is an L'-contributing edge, completing our proof. □

Example 4.12. Returning to our motivating example in Figure 4, based on the relationship between the M-path in Figure 4a and the transition sequence in Figure 4c, we deduce a configuration chain:

$$(*\mathsf{x}, q_0) \to (\mathsf{x}, q_1) \to (*\mathsf{y}, q_1) \to (\mathsf{y}, q_1) \to (\&\mathsf{r}, q_1) \to (\mathsf{r}, q_3) \to (\mathsf{s}, q_2) \to (*\mathsf{s}, q_3)$$

All configurations in this chain are realizable, and according to Theorem 4.11, the edges in this M-path are L'-contributing (thus L-contributing since L' over-approximates L). Let us now consider the path p' in Equation (2). While all its edges are identified as L'-contributing due to $(*\mathsf{x}, q_0) \to \cdots \to (\&\mathsf{r}, q_1) \to (\mathsf{s}, q_2) \to (*\mathsf{s}, q_3)$, the edge $\&r \xrightarrow{a} s$ is deemed L-contributing spuriously.

5 Graph Simplification Algorithm

We introduce our regularization-based graph simplification algorithm designed to identify L'-contributing edges, following the guidelines set forth in Theorem 4.11. The algorithm utilizes the DFA $\mathcal{D}$, derived from L', to iteratively compute realizable configurations during the traversal of G.

5.1 Computing Realizable Configurations

Algorithm 1 details the algorithm for computing the set C of realizable configurations. This algorithm operates by intersecting two sets of reachable configurations: one set generated in the forward phase (lines 5-13), originating from sources in V_{src} and the initial state, and the other set in the backward phase (lines 14-23), starting from sinks in V_{snk} and the final states. The intersection ensures that each realizable configuration (v, q) forms part of a configuration chain from $(v_{\text{src}}, q_{\text{init}})$ through (v, q) to $(v_{\text{snk}}, q_{\text{f}})$, where $v_{\text{src}} \in V_{\text{src}}$, $v_{\text{snk}} \in V_{\text{snk}}$, and $q_{\text{f}} \in F$.

Initially, three sets are initialized as empty: the worklist W, FC for storing visited configurations during the forward phase, and C for storing realizable configurations (lines 2-4).

In the forward phase (lines 5-13), pairs from $V_{\text{src}} \times \{q_{\text{init}}\}$ are added to W and FC as initial configurations (lines 5-7). The loop from lines 8-13 continuously processes configurations that are reachable during forward traversal until no new configurations can be reached. A configuration (v', q') is added to FC and W if it has not been visited before ($(v', q') \notin FC$).

Algorithm 1: Computing realizable configurations.

Input: $\mathcal{A} = (Q, \Sigma, q_{\text{init}}, \delta, F)$, $G = (V, E)$, V_{src}, and V_{snk}
Output: the set C of all realizable configurations

```
1  Proc ComputeRLZConf($\mathcal{A}, G, V_{\text{src}}, V_{\text{snk}}$):
2  $W = \emptyset$;                      /* WorkList */
3  $FC = \emptyset$;  /* Configurations in forward phase */
4  $C = \emptyset$;         /* Realizable configurations */
5  for $v \in V_{\text{src}}$ do
6      if $(v, q_{\text{init}}) \notin FC$ then
7          add $(v, q_{\text{init}})$ to $FC$ and $W$;
8  while $W \neq \emptyset$ do
9      pop $(v, q)$ from $W$;
10     for $v \xrightarrow{t} v' \in E$ do
11         for $q \times t \rightarrow q' \in \delta$ do
12             if $(v', q') \notin FC$ then
13                 add $(v', q')$ to $FC$ and $W$;
14 for $v \in V_{\text{snk}}$ do
15     for $q_f \in F$ do
16         if $(v, q_f) \in FC \wedge (v, q_f) \notin C$ then
17             add $(v, q_f)$ to $C$ and $W$;
18 while $W \neq \emptyset$ do
19     pop $(v, q)$ from $W$;
20     for $v' \xrightarrow{t} v \in E$ do
21         for $q' \times t \rightarrow q \in \delta$ do
22             if $(v', q') \in FC \wedge (v', q') \notin C$ then
23                 add $(v', q')$ to $C$ and $W$;
24 return $C$;
```

The backward phase (lines 14-23) operates similarly to the forward phase but starts with configurations in $V_{\text{snk}} \times F$ (lines 14-17). During this traversal (lines 18-23), a configuration (v', q') is considered only if previously visited in the forward phase ($(v', q') \in FC$) and not yet visited in the backward phase ($(v', q') \notin C$). This selective process efficiently performs the intersection, minimizing unnecessary traversal and focusing on potentially realizable configurations.

Time Complexity. Algorithm 1 operates with a time complexity of $O(|Q|^2 \times |E|)$, where Q represents the states of $\mathcal{A}$. The sets W, FC, and C are managed using hash tables, enabling operations like insertions and lookups in lines 6-7, 12-13, 16-17, and 22-23 to occur in $O(1)$ amortized time.

The forward phase, delineated in lines 5-13, initializes initial configurations based on V_{src}, resulting in $O(|V_{\text{src}}|)$ complexity. The main loop from lines 8-13 processes the worklist until no new configurations are found, iterating over potential configurations bounded by $|Q| \times |V|$. Each node v can have up to $|V| \times |\Sigma|$ outgoing edges. As $\mathcal{A}$ is deterministic, there is at most one output state from state q via any terminal t. Given these dynamics, the while loop in lines 8-13 might seem to have a complexity of $O(|Q| \times |V|^2 \times |\Sigma|)$ in a naive analysis. However, since each node v in G can host at most $|Q|$ configurations and each configuration (v, q) can only initiate transitions corresponding to v's outgoing edges — each potentially yielding up to one resultant state — an amortized analysis leads us to a complexity of $O(|Q| \times |E|)$ for the forward phase.

The backward phase, processed in lines 14-23, contributes $O(|V_{\text{snk}}| \times |F| + |Q|^2 \times |E|)$ to the overall time complexity, as an output state can have up to $|Q|$ possible input states transited via a terminal t (line 21). Combining these, Algorithm 1 exhibits a complexity of $O(|V_{\text{src}}| + |V_{\text{snk}}| \times |F| + |Q| \times |E| + |Q|^2 \times |E|)$. Assuming $|V| \leq |E|$, and since V_{src} and V_{snk} are subsets of V, and F is a subset of Q, the complexity simplifies to $O(|Q|^2 \times |E|)$, covering all phases of the algorithm.

Space Complexity. The space complexity of Algorithm 1 is $O(|Q| \times |V|)$, reflecting the maximum potential number of configurations that can be stored or processed within the algorithm.

Algorithm 2: Graph simplification.

```
Input: $\mathcal{A} = (Q, \Sigma, q_{\text{init}}, \delta, F)$, $G = (V, E)$, $V_{\text{src}}$, and $V_{\text{snk}}$
Output: a simplified graph $G'$
1 Proc GraphSim($\mathcal{A}, G, V_{\text{src}}, V_{\text{snk}}$):
2     $C$ = ComputeRLZConf($\mathcal{A}, G, V_{\text{src}}, V_{\text{snk}}$);
3     $E'_C = \emptyset$                    /* Approximate contributing edges */
4     for $u \xrightarrow{t} v \in E$ do
5         if $\exists q \times t \to q' \in \delta \wedge (u, q) \in C \wedge (v, q') \in C$ then
6             $E'_C = E'_C \cup \{u \xrightarrow{t} v\}$;
7     $V' = \{u \mid u \xrightarrow{t} v \in E'_C\} \cup \{v \mid u \xrightarrow{t} v \in E'_C\}$;
8     return $G' = (V', E'_C)$;
```

5.2 Overall Algorithm

Algorithm 2 describes the overall regularization-based graph simplification process for the input graph G. It starts by identifying realizable configurations using the ComputeRLZConf procedure from Algorithm 1. The loop in lines 4-6 processes the edge set E to determine if each edge contributes to L'-reachability, as specified by Theorem 4.11 and implemented in line 5. The algorithm completes by creating a simplified graph G', where non-contributing edges are removed.

For an edge $u \xrightarrow{t} v$, label t can result in up to $|Q|$ transitions, since the output state is uniquely determined by the input state provided it exists, giving the loop at lines 4-6 in Algorithm 2 a time complexity of $O(|Q| \times |E|)$. Including the complexity of the ComputeRLZConf procedure, the total time complexity of Algorithm 2 is $O(|Q|^2 \times |E|)$. The space complexity of Algorithm 2, primarily determined by ComputeRLZConf, is $O(|Q| \times |V|)$.

In practice, the DFA $\mathcal{D}$ derived from L' typically features only a few states; for instance, the DFAs in our study in Section 6 contain just 3 and 4 states each. Consequently, the time complexity of Algorithm 2 is linear to the number of edges in the input graph G. Since G is generally sparse in real-world program analysis, where $|E| = O(|V|)$, this translates to approximately linear complexity with respect to $|V|$. Therefore, MoYe is highly efficient, making it an effective pre-processing technique to streamline the input graph for CFL-reachability, as evaluated below.

6 Evaluation

We evaluate MoYe with two major clients: field-sensitive points-to analysis for Java [Sridharan et al. 2005] and alias analysis for C/C++ [Zheng and Rugina 2008], both extensively studied [Lei et al. 2022a; Shi et al. 2023; Wang et al. 2017; Xu et al. 2024]. Our evaluation confirms MoYe's efficiency and effectiveness in enhancing CFL-reachability tasks. Notably, MoYe accelerates a recent CFL-reachability algorithm [Lei et al. 2022a] and outperforms the leading graph simplification method Gf [Lei et al. 2023b]. Combined with Gf, MoYe delivers even greater performance gains, proving its value in optimizing CFL-reachability. In a batch setting, MoYe further boosts performance.

Our experiments are designed to address the following four research questions (RQs):

- RQ1: How efficient is MoYe when used as a pre-analysis tool in the all-queries setting?
- RQ2: To what extent does MoYe reduce the size of input graphs in the all-queries setting?
- RQ3: How much does MoYe accelerate CFL-reachability analysis in terms of reduced runtime overhead and memory footprint in the all-queries setting?
- RQ4: How significantly does MoYe accelerate CFL-reachability analysis in a batch setting?

$$\begin{array}{lcl}
\mathit{FlowsTo} & \longrightarrow & \mathit{new} \mid \mathit{FlowsTo}\ \mathit{Assign} \\
\overline{\mathit{FlowsTo}} & \longrightarrow & \overline{\mathit{new}} \mid \overline{\mathit{Assign}}\ \overline{\mathit{FlowsTo}} \\
\mathit{Assign} & \longrightarrow & \mathit{assign} \mid \mathit{put}_f\ \mathit{Alias}\ \mathit{get}_f \\
\overline{\mathit{Assign}} & \longrightarrow & \overline{\mathit{assign}} \mid \overline{\mathit{get}_f}\ \mathit{Alias}\ \overline{\mathit{put}_f} \\
\mathit{Alias} & \longrightarrow & \overline{\mathit{FlowsTo}}\ \mathit{FlowsTo}
\end{array}$$

(a) Original grammar

$$\begin{array}{lcl}
\mathit{FlowsTo} & \longrightarrow & \mathit{new} \mid \mathit{FlowsTo}\ \mathit{Assign} \\
\mathit{Assign} & \longrightarrow & \mathit{assign} \mid \mathit{PutAlias}_f\ \mathit{get}_f \\
\mathit{PutAlias}_f & \longrightarrow & \mathit{put}_f\ \mathit{Alias} \\
\overline{\mathit{FlowsTo}} & \longrightarrow & \overline{\mathit{new}} \mid \overline{\mathit{Assign}}\ \overline{\mathit{FlowsTo}} \\
\overline{\mathit{Assign}} & \longrightarrow & \overline{\mathit{assign}} \mid \overline{\mathit{get}_f}\ \overline{\mathit{PutAlias}_f} \\
\overline{\mathit{PutAlias}_f} & \longrightarrow & \mathit{Alias}\ \overline{\mathit{put}_f} \\
\mathit{Alias} & \longrightarrow & \overline{\mathit{FlowsTo}}\ \mathit{FlowsTo}
\end{array}$$

(b) Normalized grammar

Fig. 5. Original and normalized CFGs specifying points-to analysis for Java, with *FlowsTo* as the start symbol.

$$\begin{array}{lcl}
V & \longrightarrow & \overline{A}\ V \mid V\ A \mid \overline{f_i}\ V\ f_i \mid M \mid \epsilon \\
M & \longrightarrow & \overline{d}\ V\ d \\
A & \longrightarrow & a \mid a\ M \\
\overline{A} & \longrightarrow & \overline{a} \mid M\ \overline{a}
\end{array}$$

(a) Original grammar

$$\begin{array}{lcl}
V & \longrightarrow & \overline{A}\ V \mid V\ A \mid FV_i\ f_i \mid M \mid \epsilon \\
FV_i & \longrightarrow & \overline{f_i}\ V \\
M & \longrightarrow & DV\ d \\
DV & \longrightarrow & \overline{d}\ V \\
A & \longrightarrow & a \mid a\ M \\
\overline{A} & \longrightarrow & \overline{a} \mid M\ \overline{a}
\end{array}$$

(b) Normalized grammar

Fig. 6. Original and normalized CFGs specifying alias analysis for C/C++, with V as the start symbol.

We assess MoYe's performance in two distinct settings. First, for RQ1, RQ2, and RQ3, we adopt the all-queries approach—resolving every source–sink pair in $V_{\text{src}} \times V_{\text{snk}}$ via a single pass, following Cfl's and Gf's evaluation methodology [Lei et al. 2022a, 2023b]. This method efficiently handles large sets of queries by establishing all-pairs reachability, allowing constant-time solutions for each query once computed, and thus provides an approximate lower bound on MoYe's performance gains in accelerating CFL-reachability analysis. Additionally, it offers an upper bound on MoYe's resource usage and a lower bound on its graph reduction potential. In this setting, a demand-driven approach, which handles each query individually, has no theoretical advantage: answering a single query can take $O(|V|^3)$ [Yannakakis 1990], and many queries share overlapping paths—resulting in redundant work [Vedurada and Nandivada 2020; Zheng and Rugina 2008].

For RQ4, we use a batch-based approach, where queries are grouped into smaller batches of source–sink pairs from $V'_{\text{src}} \times V'_{\text{snk}}$ ($V'_{\text{src}} \subseteq V_{\text{src}}$ and $V'_{\text{snk}} \subseteq V_{\text{snk}}$). This method is particularly effective for handling a moderate number of queries in a demand-driven manner.

6.1 Implementation

Field-Sensitive Points-to Analysis for Java. Figure 5a displays the standard CFG used for specifying points-to analysis in Java. This CFL-reachability analysis designates allocation and virtual-invocation statements as sources and sinks, respectively, focusing only on those virtual-invocation statements within application code and new statements where allocated objects are utilized within the application [Ali and Lhoták 2012]. This setup facilitates the construction of a call graph by establishing CFL-reachability for all source-to-sink pairs. The standard CFL-reachability algorithm [Melski and Reps 2000] mandates that the CFG be normalized, ensuring that each production's right-hand side contains at most two symbols. The normalized version is depicted in Figure 5b. The alphabet Σ includes four types of terminals: *new*, *assign*, put_f, and get_f, representing allocation, assignment, field write, and field read, respectively. In points-to analysis graphs G [He et al. 2023, 2022; Sridharan et al. 2005; Xu et al. 2009], as described by [Reps 1998] and constructed using Tai-e [Tan and Li 2023], each edge is bidirectional; for any edge $v \xrightarrow{t} u$, a corresponding inverse edge $u \xrightarrow{\overline{t}} v$ exists. This setup utilizes terminals like $\overline{\mathit{new}}$, $\overline{\mathit{assign}}$, $\overline{\mathit{put}_f}$, and $\overline{\mathit{get}_f}$ to denote

$$
\begin{array}{lcl}
\mathit{FlowsTo} & \longrightarrow & \mathit{new}\ \mathit{FlowsTo}' \\
\mathit{FlowsTo}' & \longrightarrow & \mathit{Assign} \mid \mathit{Alias}' \mid \epsilon \\
\overline{\mathit{FlowsTo}} & \longrightarrow & \overline{\mathit{new}}\ \overline{\mathit{FlowsTo}'} \mid \overline{\mathit{Assign}} \\
\mathit{Assign} & \longrightarrow & \mathit{assign}\ \mathit{Assign}' \mid \mathit{put}_f\ \mathit{Alias} \\
\mathit{Assign}' & \longrightarrow & \mathit{FlowsTo}' \\
\overline{\mathit{Assign}} & \longrightarrow & \overline{\mathit{assign}}\ \overline{\mathit{Assign}'} \mid \overline{\mathit{get}_f}\ \mathit{Alias} \\
\overline{\mathit{Assign}'} & \longrightarrow & \overline{\mathit{FlowsTo}} \\
\mathit{Alias} & \longrightarrow & \overline{\mathit{FlowsTo}} \\
\mathit{Alias}' & \longrightarrow & \mathit{get}_f\ \mathit{Assign}' \mid \overline{\mathit{put}_f}\ \overline{\mathit{Assign}'} \\
\overline{\mathit{FlowsTo}'} & \longrightarrow & \mathit{FlowsTo}
\end{array}
$$

(a) Approximated regular grammar

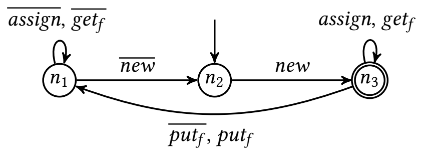
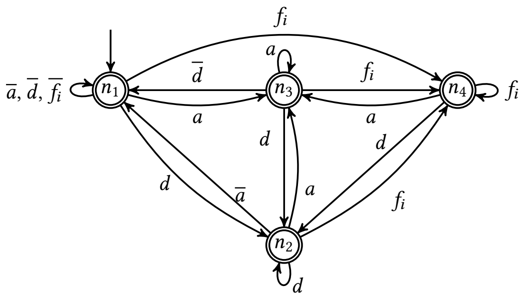

$\overline{assign}$, $\overline{get_f}$
$\overline{new}$
new
$assign$, get_f
n_1
n_2
n_3
$\overline{put_f}$, put_f
(b) State-minimal DFA

Fig. 7. Field-sensitive points-to analysis for Java.

$$
\begin{array}{lcl}
V & \longrightarrow & \overline{f_i}\ V \mid \overline{A} \mid M \mid V' \\
V' & \longrightarrow & d\ M' \mid f_i\ V' \mid A \mid \epsilon \\
M & \longrightarrow & \overline{d}\ V \\
M' & \longrightarrow & V' \mid A' \mid \overline{a}\ \overline{A'} \\
A & \longrightarrow & \mathit{assign}\ M \mid \mathit{assign}\ A' \\
A' & \longrightarrow & V' \\
\overline{A'} & \longrightarrow & V \\
\overline{A} & \longrightarrow & M \mid \overline{a}\ \overline{A'}
\end{array}
$$

(a) Approximated regular grammar



(b) State-minimal DFA

Fig. 8. Field-sensitive alias analysis for C/C++.

these inverse edge labels. The start symbol *FlowsTo* indicates the flow of an object to a pointer, while its inverse, $\overline{\mathit{FlowsTo}}$, represents standard points-to relations. Additionally, the normalization step often introduces new nonterminals, such as $\mathit{PutAlias}_f$ and $\overline{\mathit{PutAlias}_f}$, as seen in Figure 5b.

Field-Sensitive Alias Analysis for C/C++. Figure 6 presents both the original and normalized CFGs utilized for alias analysis in C/C++. This CFL-reachability analysis designates store and load statements as sources and sinks, respectively, and determines the value alias relation (denoted by the start symbol V) between sources and sinks, facilitating the tracking of indirect value flows through memory accesses [Li et al. 2011; Shi et al. 2018; Sui and Xue 2018; Yao et al. 2024]. The original CFG includes a field-sensitive production $V \longrightarrow f_i\ V\ \overline{f_i}$, where f_i represents the i-th object field. Alias analysis graphs G are constructed by the open-source SVF tool [Sui and Xue 2016].

CFL-Reachability Solver. To address CFL-reachability problems, we use Pocr, a recent CFL-reachability algorithm [Lei et al. 2022a], which reduces transitive redundancy via ordered derivations, along with bit-vector set operations for subcubic performance [Chaudhuri 2008] implemented by us. The tool is sourced from the Pocr artifact [Lei et al. 2022b]. We refer to this solver as Cfl, which requires a normalized CFG as input, like those shown in Figures 5b and 6b.

MoYe. We developed MoYe in two components: The first component, a Python3 module, transforms a CFG into a DFA via an intermediary regular grammar, producing identical DFAs from original or normalized grammars in under a second. This one-time process per CFG is illustrated in Figures 7 and 8, with resulting DFAs for points-to and alias analysis having 3 and 4 states, respectively (Figures 7b and 8b). The second component, which handles graph simplification, uses these DFAs and is implemented in LLVM-14.0.0 (Algorithms 1 and 2).

Validation of Correctness. We empirically validated the correctness of our graph simplification technique MoYe on all Java and C/C++ programs where Cfl is scalable, listed in Tables 2 and 3, by confirming that Cfl produces the same set of L-reachable pairs in $V_{\text{src}} \times V_{\text{snk}}$ on both the original graph G and the simplified graph G' generated by MoYe.

6.2 Experimental Setup

Baselines. We evaluate MoYe's effectiveness in improving CFL-reachability analysis via graph size reduction and compare it against Gf, the current leading approach [Lei et al. 2023b]. Because their methods are orthogonal, we also test Combined, applying Gf first to contract edges, then MoYe to remove non-contributing edges. Running MoYe before Gf is infeasible because Gf assumes bidirectional graphs for points-to/alias analysis, a key to optimizing edge scanning [Lei et al. 2023a]; doing so would disrupt Gf's foundational assumptions (Figure 1e). Furthermore, Gf is faster, making it more practical to apply before MoYe, as demonstrated in our results (Tables 2 and 3). We measure performance gains by executing the Cfl solver on both the original and simplified graphs.

Experimental Settings. All experiments are conducted on a server with dual 12-core Intel(R) Xeon(R) Gold 5317 CPUs at 3.00GHz and 2 TB of RAM, with each run capped at 6 hours and 512 GB of memory. We report the mean (μ) time and memory usage over 6 runs for each experiment. Following standard evaluation guidelines [Georges et al. 2007], we calculated the *standard variation* σ, observing a maximum *coefficient of variation* ($CV = \frac{\sigma}{\mu}$) of 4.98%, indicating minimal variation [Westgard nd]. We then calculate statistics on average metrics across benchmarks—such as speedups and reduction rates—using *geometric means*.

Benchmarks and Graph Construction. Following the latest related work [He et al. 2024a], we have selected 13 Java programs from the well-known DaCapo benchmark [Blackburn et al. 2006] (version `6cf0380`) for points-to analysis, coupled with a large Java library (`JRE1.8.0_31`) and `TamiFlex` [Bodden et al. 2011] for reflection handling. The number of classes and reachable methods are displayed in Columns 2-3 in Table 2. Consistent with prior studies [He et al. 2024b; Thiessen and Lhoták 2017], we have excluded `jython` due to its overly conservative reflection log, which rendered it unscalable within our time budget. We employ Tai-e [Tan and Li 2023], an open-source static analysis framework, to translate these Java programs into bytecode for graph extraction. For alias analysis, we evaluate 10 C/C++ programs from the SPEC 2017 suite. They are compiled with Clang and linked into LLVM bitcode using wllvm[1], then analyzed with the SVF framework [Sui and Xue 2016] to generate input graphs. Sources and sinks are identified during graph generation. Detailed statistics on the number of nodes, edges, sources, and sinks in each program—tailored to points-to and alias analyses—are listed in Table 2 and Table 3, respectively.

6.3 RQ1: Evaluating MoYe's Efficiency as Pre-Analysis

In this first RQ, we measure the computational and memory overhead of MoYe as a pre-analysis technique in the all-queries setting by solving all source–sink pairs in $V_{\text{src}} \times V_{\text{snk}}$ in a single pass, which approximates an upper bound on MoYe's resource usage. Tables 2 and 3 show the analysis time and memory usage for the graph simplification techniques—MoYe, Gf, and Combined —applied to each program under points-to and alias analyses, respectively. For comparison, we also run Cfl on the original (unsimplified) graphs, providing a reference point.

Points-to Analysis for Java. Table 2 shows that MoYe is lightweight, taking about one minute for `fop` at its slowest, while Gf typically completes in under ten seconds for all 13 Java programs. The analysis times for both tools are significantly shorter compared to Cfl's. Combined, leveraging Gf's efficiency and the pre-simplified graph it provides, operates faster than MoYe. In terms of

[1]https://github.com/travitch/whole-program-llvm

Table 2. Performance results for points-to analysis of Java programs using Cfl for CFL-reachability analysis and three graph simplification methods (Gf, MoYe, Combined) in the all-queries setting. Columns show analysis time (seconds), peak memory (MBs), number of classes, reachable methods (cols 2–3), and the graph's nodes, edges, sources, and sinks (cols 4–7). A "-" indicates the analysis did not finish within 6 hours.

Bench.	#Cls	#RM	#Nodes	#Edges	#Srcs	#Snks	Cfl		Gf		MoYe		Combined	
							Time	Mem	Time	Mem	Time	Mem	Time	Mem
avrora	8,737	12,643	86,457	314,544	8,282	4,529	4,712.79	4,267.74	0.27	54.50	1.89	81.88	1.08	54.47
batik	11,858	29,378	199,790	902,310	16,863	17,366	-	-	1.62	128.13	14.91	210.16	8.74	127.57
eclipse	10,134	15,992	119,123	488,122	10,020	12,027	-	-	0.53	72.65	4.46	118.61	2.57	72.53
fop	12,871	49,629	370,007	1,856,600	32,099	62,416	-	-	9.08	250.27	68.15	402.78	38.62	249.95
h2	10,712	21,343	151,070	668,916	10,924	22,685	-	-	0.76	98.46	8.18	153.59	4.66	98.14
luindex	8,433	16,994	119,191	430,822	8,855	12,564	15,485.27	8,830.43	0.43	71.85	3.93	110.75	2.28	71.78
lusearch	8,433	8,719	62,110	229,392	5,391	897	1,756.85	2,343.93	0.15	36.71	0.93	57.82	0.57	36.75
pmd	9,954	24,918	185,812	973,112	13,368	31,232	-	-	1.69	126.74	16.58	209.78	9.25	126.42
sunflow	7,399	16,626	115,608	430,460	10,664	2,090	17,582.17	8,883.26	0.41	69.02	3.58	108.21	2.17	68.92
tomcat	8,433	9,486	66,045	244,086	5,773	747	2,146.93	2,649.05	0.16	38.85	1.07	61.46	0.66	38.94
tradebeans	9,954	10,330	74,438	273,734	6,713	747	3,238.06	3,443.98	0.19	43.12	1.36	69.46	0.86	43.17
tradesoap	9,954	10,330	74,438	273,734	6,713	747	3,195.30	3,446.23	0.19	43.23	1.40	69.48	0.87	43.15
xalan	10,841	15,062	115,361	455,242	9,189	13,317	11,493.15	6,788.44	0.45	69.84	3.86	114.31	2.41	69.90
Geo. Mean	9,715	16,333	117,027	472,990	9,828	5,611	5,248.16	4,497.75	0.52	72.24	4.18	115.10	2.48	72.17

Table 3. Performance results for alias analysis of C/C++ programs using Cfl for CFL-reachability and Gf, MoYe, and Combined for graph simplification in the all-queries setting, with column definitions from Table 2.

Bench.	#Nodes	#Edges	#Srcs	#Snks	Cfl		Gf		MoYe		Combined	
					Time	Mem	Time	Mem	Time	Mem	Time	Mem
nab	18,151	40,910	867	1,501	68.38	343.80	0.04	21.19	0.07	15.18	0.05	21.38
xz	12,795	27,848	377	568	30.38	195.86	0.03	14.94	0.04	10.81	0.03	15.00
cactus	157,319	335,256	8,211	11,337	-	-	0.45	175.96	5.80	129.58	3.42	175.85
leela	22,861	51,958	1,884	2,282	140.97	423.57	0.05	27.44	0.12	20.24	0.09	27.31
x264	68,316	157,582	4,748	6,768	792.87	2,655.51	0.18	78.24	0.63	56.93	0.51	78.33
povray	78,052	185,288	3,228	5,138	-	-	0.23	89.97	1.60	63.11	1.17	90.38
parest	120,601	259,762	8,290	13,610	9,536.01	6,985.11	0.37	136.27	4.46	101.29	2.31	136.37
imagick	120,056	322,144	3,120	4,700	-	-	0.40	144.19	3.00	100.18	2.96	144.12
omnetpp	244,498	521,674	12,061	29,887	-	-	0.85	272.49	14.39	201.42	8.59	272.54
perlbench	160,837	424,272	5,906	17,600	-	-	0.61	194.42	9.47	140.38	6.60	194.48
Geo. Mean	69,249	159,613	3,277	5,510	294.46	880.44	0.20	80.34	1.14	58.17	0.79	80.45

memory usage, both MoYe and Gf consume only a fraction of what Cfl requires, with MoYe using 1.38× more than Gf. For Combined, the total pre-analysis time is $t_1 + t_2$: Gf operates on the original graph (t_1 corresponds to the data in Table 2, 10th column), and MoYe operates on the simplified graph (t_2 is usually less than the data in the 12th column). Thus, Combined's pre-analysis time (14th column) does not equal the sum of the 10th and 12th columns. Combined's peak memory usage is the higher of Gf's or MoYe's. Since the peak occurs during Gf's stage, the values in the 15th column are very close to those of Gf (11th column). Finally, it is important to note that Cfl failed to complete analyses for five benchmarks—batik, eclipse, fop, h2, and pmd, highlighting the critical role of graph simplification techniques in enhancing performance.

Alias Analysis for C/C++. Table 3 reveals that Cfl failed to complete the analysis for half of the C/C++ programs tested, including cactus, povray, imagick, omnetpp, and perlbench, within the 6-hour time limit. These programs also required more time and resources for MoYe, Gf, and Combined. Despite these demands, all three pre-analysis techniques finished much faster than Cfl, emphasizing their efficiency and lower asymptotic time complexity in comparison.

Compared to Cfl, both MoYe and Gf consume significantly less analysis time and memory, making them well-suited as pre-analysis techniques to accelerate CFL-reachability analysis. Specifically, for benchmarks where Cfl completes within 6 hours, MoYe and Gf require only 0.037% (0.085%)

and 0.005% (0.028%) of the analysis time, while using 1.809% (3.249%) and 1.147% (4.453%) of the memory consumed by Cfl in points-to (alias) analysis, respectively. Combined also shows similar benefits, underscoring the value of integrating these two graph simplification methods.

6.4 RQ2: Evaluating MoYe's Effectiveness in Graph Size Reduction as Pre-Analysis

In this second RQ, we assess MoYe's effectiveness in reducing input graph sizes relative to Gf in the all-queries setting, thus establishing an approximate lower bound on MoYe's graph reduction potential. Section 6.6 demonstrates that MoYe achieves even greater edge reduction in the batch setting than shown here. We find MoYe to be both lightweight and highly effective, which enhances the advantages of Combined —integrating MoYe and Gf together.

Figures 9 and 10 show the reduction rates for nodes and edges in the input graphs for points-to and alias analyses, respectively. In points-to analysis, Gf reduces nodes by 40.22% and edges by 24.44% on average, while MoYe achieves higher reductions of 63.82% (nodes) and 70.81% (edges). The combined approach, Combined, yields the best performance, cutting nodes by 77.79% and edges by 78.32%. In alias analysis, Gf, MoYe, and Combined reduce nodes by 39.84%, 59.14%, and 76.71%, and edges by 36.89%, 65.36%, and 79.0%, respectively.

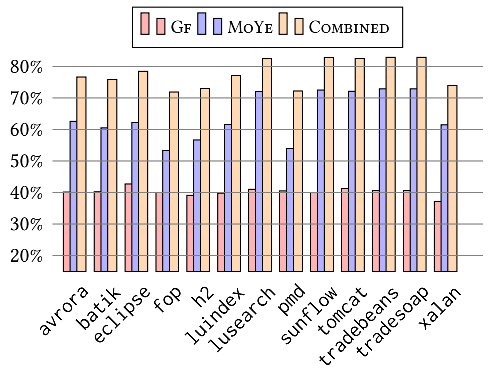
Combined attains higher reduction rates than MoYe or Gf alone because they reduce the input graph based on distinct principles: MoYe prunes non-contributing edges to eliminate irrelevant paths, while Gf contracts trivial edges to shorten paths (see [Lei et al. 2023b] for details on Gf). In our analyses, Gf only contracts transitive edges—such as *assign*-edges in points-to analysis and *a*-edges in alias analysis—which generally represent assignments in CFL-based program analyses. In contrast, MoYe removes non-contributing edges of any type from the input graph. Let E_{Gf} be the set of edges contracted by Gf, and E_{MoYe} the set removed by MoYe. The two methods are compatible for graph simplification. Gf may contract edges that MoYe identifies as necessary for preserving CFL-reachability; contracting these edges retains reachability, whereas removing them may cause unsoundness. Such edges lie in $E_{\text{Gf}} - E_{\text{MoYe}}$. Conversely, edges deemed non-contributing by MoYe may not appear trivial to Gf due to its approximations, so these edges are in $E_{\text{MoYe}} - E_{\text{Gf}}$. With respect to node reduction, Gf consolidates two nodes by contracting the edges between them, while MoYe removes a node only if all its incident edges are non-contributing.

To further demonstrate the effectiveness of MoYe, we conducted an additional cast-may-fail analysis for Java, utilizing source-to-sink pairs distinct from those in the call graph construction client. Specifically, we selected new statements as sources and cast statements as sinks, reusing the grammars and DFA presented in Figures 5 and 7. This analysis focuses on determining the potential failure of cast statements, like "`a = (A) b;`", based on the points-to set of the pointer `b`. Figure 11 shows MoYe's reduction rates of nodes and edges in the input graphs, averaging 69.91% and 72.62%, respectively. These rates are comparable to those observed in the call graph construction client (Figure 11), highlighting MoYe's robustness across different analysis scenarios.

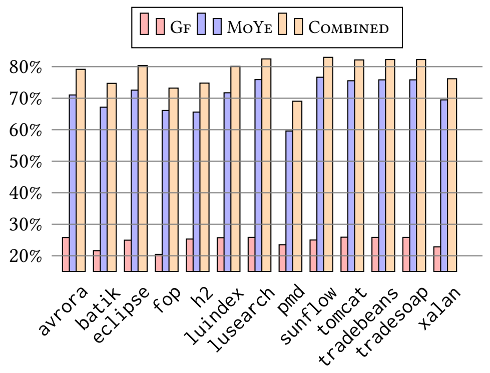
6.5 RQ3: Evaluating MoYe's Impact on CFL-Reachability in the All-Queries Setting

In this third RQ, we assess MoYe's ability to enhance Cfl (CFL-reachability analysis) by reducing analysis time and memory usage relative to Gf in the all-queries setting. We also demonstrate how Combined, which integrates MoYe and Gf, achieves even greater performance, underscoring MoYe's pivotal role in significantly boosting analysis efficiency.

As standard practice dictates [Reps et al. 1995], summary edges derived from grammar productions are added during CFL-reachability solving (Section 2.2). By reducing the input graph size, the number of summary edges decreases, enabling Cfl to run faster and use less memory. Hereafter, edges in the input graph are referred to as *graph edges* to distinguish them from summary edges.

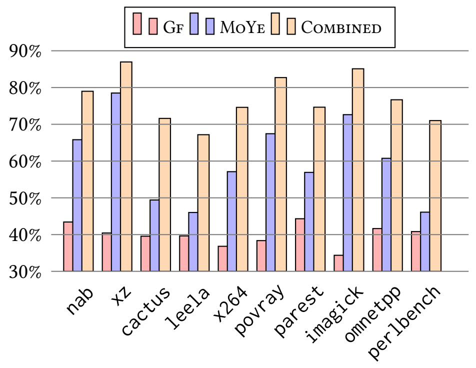


(a) Node reduction

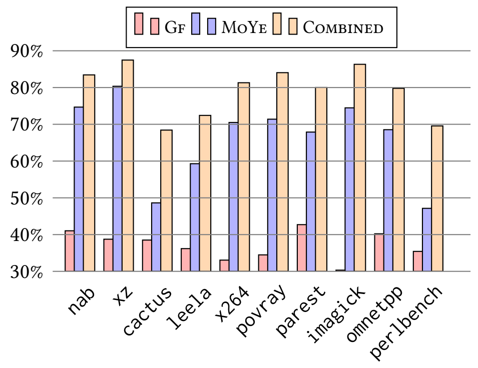


(b) Edge reduction

Fig. 9. Reduction rates for nodes and edges in points-to analysis input graphs in the all-queries setting.



(a) Node reduction



(b) Edge reduction

Fig. 10. Reduction rates of nodes and edges in alias analysis input graphs in the all-queries setting.

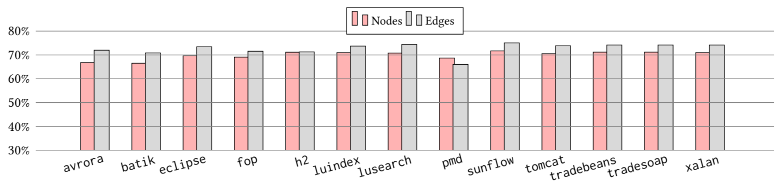
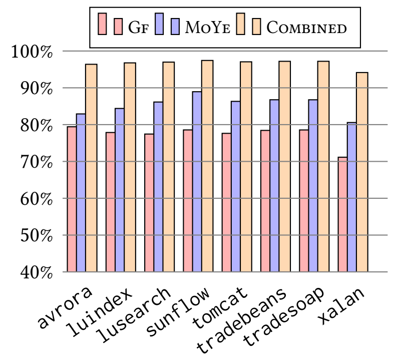


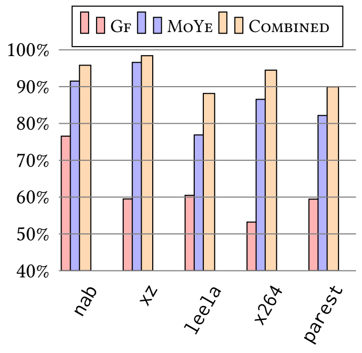
Fig. 11. Reduction rates for nodes and edges in cast-may-fail points-to analysis, a significant client analysis for Java, achieved by MoYE in the all-queries setting.

Reduction of Summary Edges. Figure 12 shows the summary edge reduction rates achieved by MoYE, GF, and COMBINED, focusing on programs CFL can analyze within 6 hours on the original graphs. While higher input graph reductions often correlate with greater summary edge reductions, this is not always the case, as summary edge count also depends on factors like edge types.

In points-to analysis, the average summary edge reductions for MoYE and GF are 85.31% and 77.34%, respectively. Although MoYE removes significantly more graph edges (70.81%) than GF (24.44%), their summary edge reduction rates are comparable because GF specifically targets transitive edges, which propagate reachability and lead to new summary edges. By focusing on transitive edges, GF effectively eliminates summary edges despite a relatively small reduction in graph edges, leaving room for orthogonal optimizations such as those proposed in this work. In

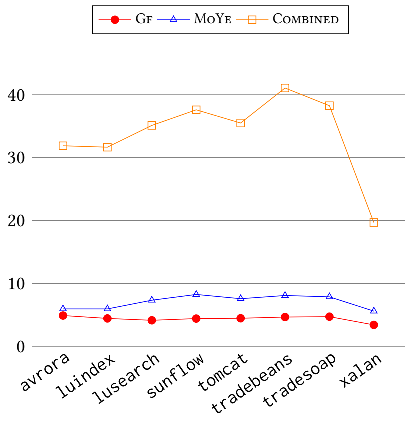


(a) Points-to analysis

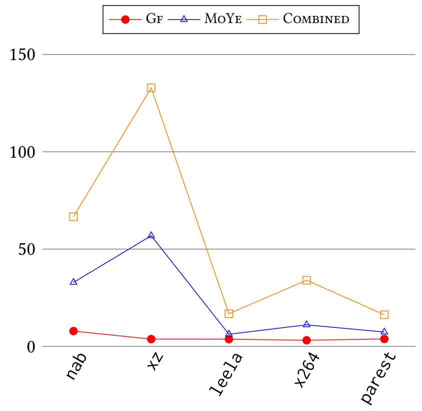


(b) Alias analysis

Fig. 12. Reduction rates of summary edges achieved by Gf, MoYe and Combined in the all-queries setting.



(a) Points-to analysis



(b) Alias analysis

Fig. 13. Speedups in CFL-reachability analysis achieved by Gf, MoYe and Combined in the all-queries setting.

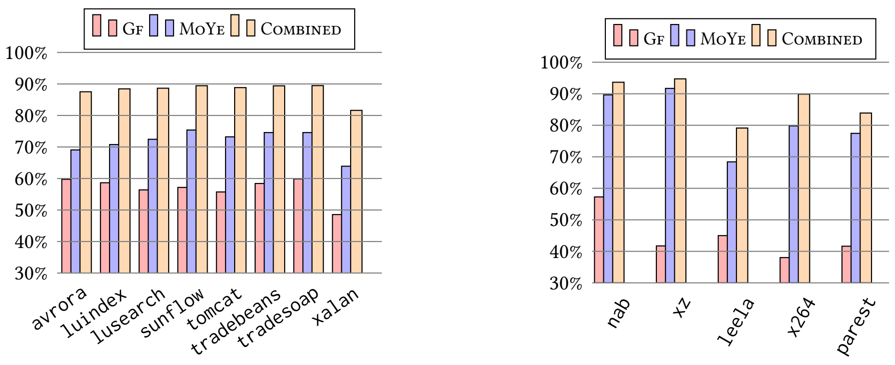
alias analysis, MoYe substantially outperforms Gf, achieving 86.47% in summary edge reduction versus 61.38% for Gf.

Comparing Gf and MoYe individually with their combination, Combined, we observe that Combined achieves significantly greater summary edge reductions across all programs, averaging 96.64% for points-to analysis and 93.28% for alias analysis. Additionally, Combined's performance trends closely mirror that of the better-performing of the two graph simplification techniques.

Performance Improvement. Figures 13 and 14 present the performance speedups and memory reduction rates in CFL-reachability analysis achieved by Gf, MoYe, and Combined. Across different programs, the trends in these two figures align with the summary edge reduction rates, as both Gf and MoYe enhance CFL-reachability performance by limiting the number of inserted summary edges, leading to shorter analysis time and lower memory usage.

In points-to analysis for Java, MoYe outperforms Gf with an average speedup of 6.99× (vs. 4.36×) and a memory reduction of 71.65% (vs. 56.7%). For C/C++ alias analysis, Gf achieves an average speedup of 4.21× and a memory reduction of 44.28%, while MoYe reaches 15.72× and 80.95%, respectively. Meanwhile, Combined attains an average speedup of 33.17× (38.28×) and a memory reduction of 87.89% (88.05%) for points-to (alias) analysis.

Improved Scalability. Table 4 demonstrates how graph simplification greatly improves the scalability of CFL-reachability analysis for difficult-to-analyze programs. Among the ten Java and



(a) Points-to analysis (b) Alias analysis

Fig. 14. Reduction rates of memory usage achieved by Gf, MoYe and Combined in the all-queries setting.

Table 4. Scalability improvements of Cfl for ten Java and C/C++ benchmarks that could not be analyzed on original graphs (Tables 2 and 3), using the simplified graphs G_{Gf}, G_{MoYe}, and G_{Combined} generated by Gf, MoYe, and Combined, respectively, in the all-queries setting.

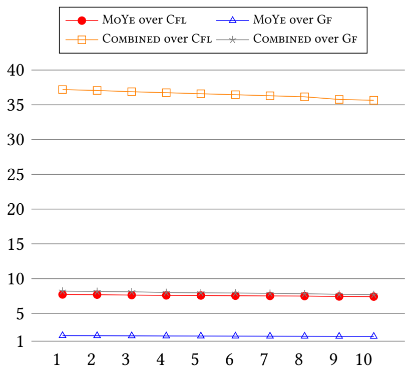
Benchmark	Language	G_{Gf}		G_{MoYe}		G_{Combined}	
		Time	Mem	Time	Mem	Time	Mem
batik	Java	-	-	-	-	6,618.04	4,449.13
eclipse	Java	5,602.95	4,645.72	4,435.76	3,345.78	1,128.10	1,552.13
fop	Java	-	-	-	-	-	-
h2	Java	18,605.60	9,589.94	11,160.27	6,285.78	3,579.64	3,208.65
pmd	Java	-	-	-	-	8,127.69	5,307.03
cactus	C/C++	-	-	-	-	14,307.28	6,274.87
povray	C/C++	10,681.60	8,920.30	8,216.03	3,589.66	1,901.83	2,039.74
imagick	C/C++	-	-	11,337.23	4,641.95	2,250.87	2,242.45
omnetpp	C/C++	-	-	17,980.23	7,983.35	4,908.00	4,979.92
perlbench	C/C++	-	-	-	-	-	-

C/C++ programs Cfl could not complete within six hours (Tables 2 and 3), using the simplified graphs, G_{Gf}, G_{MoYe}, or G_{Combined}, enabled Cfl to finish analyzing 3, 5, and 8 of these programs, respectively—underscoring MoYe's significant impact.

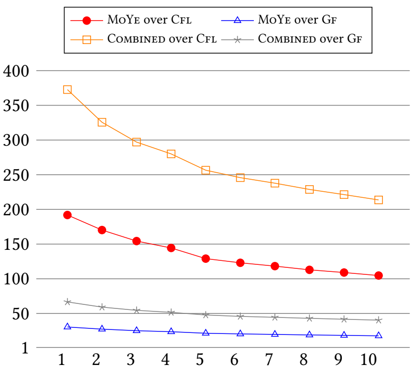
6.6 RQ4: Evaluating MoYe's Impact on CFL-Reachability in a Batch Setting

In this final RQ, we evaluate MoYe's performance gains over Cfl and Gf in a demand-driven batch setting. For each Java or C++ program, we randomly sample 10 query groups Q_i ($i \in [1, 10]$), each containing 1% sources (V_{src}) and 1% sinks (V_{snk}). For each Q_i, MoYe generates a set of contributing edges E_i. We then invoke Cfl to resolve Q_i, starting with a simplified graph $G^{i,\text{start}}_{\text{MoYe}}$ and ending with $G^{i,\text{end}}_{\text{MoYe}}$, which incorporates the summary edges created by Cfl. Here, $G^{i,\text{start}}_{\text{MoYe}}$ is built incrementally from $G^{i-1,\text{end}}_{\text{MoYe}}$ by adding $E_i \setminus \bigcup_{j\in[1,i-1]} E_j$ and their incident nodes (with $G^{0,\text{end}}_{\text{MoYe}}$ being empty). This incremental construction reuses reachability results (i.e., summary edges) from $G^{i-1,\text{end}}_{\text{MoYe}}$, obtained while answering Q_{i-1}, to assist with Q_i through a well-known caching mechanism [Reps et al. 1995; Zheng and Rugina 2008]. We define the *contributing edge rate* for the first i batches as $\frac{|\bigcup_{j\in[1,i]} E_j|}{|E|}$, where E is the set of edges in the original graph G of the program being analyzed.

In the batch setting, Cfl, Gf, and MoYe handle queries differently, allowing us to evaluate MoYe's performance gains below. Both Cfl and Gf are designed for all-pairs reachability with sources and sinks defined independently of queries (Section 6.1). Consequently, Cfl operates directly on



(a) Points-to analysis



(b) Alias analysis

Fig. 15. Speedups achieved by MoYe and Combined over Cfl (w/o simplification) and Gf in the batch setting .

the original graph G, answering all queries $\bigcup_{i\in[1,10]} Q_i$ in one run, taking time t_{Cfl}. Meanwhile, Gf contracts the same *assign*-edges for points-to analysis (or *a*-edges for alias analysis) for every batch Q_i, so it runs once per program, incurring a pre-analysis time p_{Gf} to produce the simplified graph G_{Gf}. We then invoke Cfl on G_{Gf} to answer all queries $\bigcup_{i\in[1,10]} Q_i$, incurring time t_{Gf}. In contrast, MoYe allows queries to be handled on demand. For each query Q_i, p^i_{MoYe} is the pre-analysis time, and t^i_{MoYe} is the main Cfl analysis time (including incrementally adding new contributing edges and their incident nodes to $G^{i,\text{start}}_{\text{MoYe}}$). Therefore, MoYe's speedups over Cfl and Gf for Q_i are conservatively calculated, including time for previous queries, as follows:

$$\text{Speedup over Cfl} = \frac{t_{\text{Cfl}}}{\sum_{j\in[1,i]}(p^j_{\text{MoYe}} + t^j_{\text{MoYe}})}, \quad \text{Speedup over Gf} = \frac{p_{\text{Gf}} + t_{\text{Gf}}}{\sum_{j\in[1,i]}(p^j_{\text{MoYe}} + t^j_{\text{MoYe}})}.$$

Figure 15 gives MoYe's speedups over Cfl and Gf. For points-to analysis (Figure 15a), we compare MoYe with Cfl on eight benchmarks and with Gf on ten benchmarks, where Cfl completes within six hours on G and G_{Gf}, respectively. For alias analysis (Figure 15b), we similarly compare MoYe with Cfl on five benchmarks and with Gf on six benchmarks, under the same time constraints.

MoYe achieves significant speedups over both Cfl and Gf, though these gains gradually decrease as the number of batches increases, as expected (due to the way these speedups are calculated). The rate of decline slows over time. Additionally, MoYe's pre-analysis overhead remains minimal compared to Cfl's main analysis (Section 6.3). In Figure 15, MoYe substantially accelerates Cfl in both points-to and alias analyses, with performance declining from 7.72× to 7.41× and from 191.93× to 104.66×, respectively (red line with circle markers). Similarly, MoYe consistently outperforms Gf, with speedups decreasing from 1.8× to 1.69× and from 30.5× to 17.77× in points-to and alias analyses, respectively (blue line with triangle markers). These results underscore MoYe's strong effectiveness in enhancing CFL-reachability in the batch setting, especially for alias analysis.

In Figure 15, Combined substantially outperforms MoYe, which had been the best performer among MoYe, Gf, and Cfl, in both points-to and alias analyses (orange lines with square markers; gray lines with star markers). The speedups of Combined over MoYe range from 4.55× to 4.57× in points-to analysis and from 2.19× to 2.27× in alias analysis, over 10 batches. These results highlight the compatibility of MoYe with Gf in further boosting the performance of CFL-reachability analysis.

Table 5 presents the batch-wise contributing edge rates for MoYe and Combined. Table 5a shows these rates for points-to analysis, while Table 5b covers alias analysis. In each analysis, for $i \in [1, 10]$, M_i represents the contributing edge rate of the first i batches under MoYe, compared to the all-queries setting ($M_{\max}$). Similarly, C_i and $C_{\max}$ are the corresponding metrics for Combined.

Table 5. Contributing edge rates by MoYe and Combined. M_i (C_i) represents the contributing edge rate for the first i batches by MoYe (Combined), while $M_{\max}$ ($C_{\max}$) denotes the rates when considering all queries ($V_{\text{src}} \times V_{\text{snk}}$). To save space, $M_{i/\ldots/j}$ ($C_{i/\ldots/j}$) denotes the sequence $M_i/\cdots/M_j$ ($C_i/\cdots/C_j$). Only programs on which Cfl terminates on G or G_{GF} within 6 hours are included.

Batch	avrora	eclipse	h2	luindex	lusearch	sunflow	tomcat	tradebeans	tradesoap	xalan
$M_{1/2}$	26.27/26.30	23.00/23.09	26.12/26.27	24.16/24.21	23.52/23.53	22.40/22.41	23.99/23.99	23.75/23.75	23.75/23.75	24.07/24.15
$M_{3/4}$	26.33/26.35	23.13/23.42	26.48/26.64	24.25/24.30	23.54/23.54	22.53/22.54	24.00/24.00	23.76/23.77	23.76/23.76	24.22/24.29
$M_{5/6}$	26.38/26.40	23.46/23.49	26.77/26.85	24.35/24.40	23.54/23.55	22.55/22.56	24.00/24.01	23.77/23.77	23.76/23.76	24.42/24.50
$M_{7/8}$	26.42/26.44	23.53/23.57	27.01/27.17	24.45/24.50	23.55/23.56	22.57/22.59	24.01/24.02	23.77/23.78	23.77/23.77	24.56/24.60
$M_{9/10}$	26.52/26.55	23.62/23.66	27.30/27.39	24.54/24.57	23.56/23.57	22.60/22.60	24.02/24.03	23.78/23.78	23.78/23.78	24.68/24.76
$M_{\max}$	28.98	27.44	34.42	28.27	24.08	23.35	24.46	24.18	24.18	30.53
$C_{1/2}$	19.08/19.10	16.98/17.01	19.60/19.69	17.38/17.40	17.28/17.28	16.59/16.59	17.62/17.62	17.52/17.52	17.52/17.52	18.81/18.86
$C_{3/4}$	19.12/19.13	17.02/17.25	19.82/19.93	17.43/17.45	17.28/17.28	16.60/16.60	17.62/17.62	17.52/17.53	17.52/17.52	18.91/18.96
$C_{5/6}$	19.14/19.15	17.26/17.27	19.99/20.05	17.47/17.50	17.28/17.28	16.60/16.61	17.62/17.62	17.53/17.53	17.52/17.52	19.07/19.12
$C_{7/8}$	19.16/19.17	17.29/17.32	20.16/20.28	17.53/17.56	17.28/17.29	16.61/16.62	17.63/17.63	17.53/17.53	17.53/17.53	19.16/19.19
$C_{9/10}$	19.24/19.25	17.34/17.35	20.37/20.43	17.58/17.60	17.29/17.29	16.62/16.63	17.63/17.63	17.53/17.53	17.53/17.53	19.25/19.30
$C_{\max}$	20.84	19.64	25.21	19.89	17.54	17.04	17.85	17.73	17.73	23.82

(a) Points-to analysis

Batch	nab	xz	leela	x264	povray	parest
$M_{1/2/3}$	8.30/8.44/8.57	5.44/5.64/5.72	7.39/8.15/9.11	3.17/3.95/4.16	15.75/15.98/16.10	5.81/6.09/6.52
$M_{4/5/6}$	8.82/8.97/9.07	5.93/6.23/6.24	9.63/10.25/10.52	4.44/4.68/4.89	16.27/16.37/16.63	6.68/6.88/7.12
$M_{7/8/9}$	9.21/9.35/9.49	6.27/6.31/6.36	10.74/10.87/11.18	5.37/5.58/5.78	16.69/16.76/16.84	7.30/7.51/7.68
$M_{10/\max}$	9.55/25.32	6.41/19.66	11.55/40.72	5.96/29.50	16.93/28.61	7.89/32.11
$C_{1/2/3}$	5.12/5.18/5.28	3.43/3.56/3.61	3.46/5.23/5.91	1.85/2.05/2.19	8.74/8.89/8.96	3.53/3.71/3.97
$C_{4/5/6}$	5.45/5.53/5.60	3.75/3.93/3.94	6.28/6.74/6.92	2.40/2.56/2.69	9.08/9.14/9.31	4.07/4.21/4.36
$C_{7/8/9}$	5.69/5.77/5.88	3.95/3.98/4.02	7.07/7.16/7.40	2.84/2.98/3.12	9.35/9.40/9.44	4.48/4.61/4.71
$C_{10/\max}$	5.92/16.56	4.05/12.53	7.66/27.58	3.24/18.69	9.50/15.95	4.85/19.97

(b) Alias analysis

Query batches often share many L'-contributing edges due to the regular approximation of L into L' (Section 4.1), enhancing the caching mechanism for summary edges [Reps et al. 1995; Zheng and Rugina 2008]. For points-to analysis (Table 5a), M_i (C_i) are slightly lower than $M_{\max}$ ($C_{\max}$). Consequently, Figure 15a shows marginally higher speedups in the batch setting compared to the all-queries setting. Since most L'-contributing edges appear in the first batch (M_1, C_1), speedups decline only slightly over 10 batches. For alias analysis (Table 5b), the batch-based contributing edge rates for MoYe and Combined are noticeably lower than $M_{\max}$ ($C_{\max}$). As a result, both achieve more significant speedups in the batch setting compared to the all-queries setting (Figure 15b), with speedups declining more visibly across the 10 batches.

7 Related Work

In this section, we review work closely related to our graph simplication approach, focusing on CFL-reachability, graph simplification, and grammar regularization.

7.1 CFL-Reachability

Originally developed for Datalog chain queries in the database community [Bravenboer and Smaragdakis 2009; Jordan et al. 2016; Yannakakis 1990], CFL-reachability has become a key framework in program analysis [Reps 1998]. Recent advancements include subcubic algorithms [Chaudhuri 2008; Zhang et al. 2014], disk-based parallel computation [Wang et al. 2017], transitive redundancy elimination [Lei et al. 2022a], multi-derivation [Shi et al. 2023, 2024b], skewed tabulation [Lei et al. 2024], online cycle elimination [Xu et al. 2024], and staged solving [Shi et al. 2024a]. Efficient algorithms have also been proposed for specialized cases like bidirected Dyck-CFL reachability [Chatterjee et al. 2017; Xu et al. 2009; Zhang et al. 2013]. These efforts focus on boosting efficiency and scalability. MoYe is orthogonal to these online optimizations and can act as a pre-processing step to further improve their performance.

7.2 Graph Simplification

Graph simplification techniques, aiming to optimize language reachability by reducing graph size, must approximate to some degree to minimize pre-analysis runtime overhead, thus preventing performance declines. Guided by recursive state machines, GF approximates the states of nodes to contract trivial edges, extending offline variable substitution methods for general CFL-reachability problems [Alur et al. 2005; Lei et al. 2023b; Rountev and Chandra 2000]. For interleaved-Dyck language reachability, a non-context-free scenario, Li et al. [2020] suggest eliminating specific parenthesis-labeled edges that contribute nothing to interleaved-Dyck paths. This method uses bidirectionality and independent resolutions of multiple CFLs for efficiency and is not directly comparable with MoYe. The mutual refinement approach [Ding and Zhang 2023] generalizes this by intersecting graphs for multiple CFL reachabilities. MoYe, employing regular approximation, uniquely accelerates client-driven CFL-reachability analysis.

7.3 Regular Approximation and Its Applications

Regular approximation of context-free languages is critical for applications like language recognition and efficient parsing [Mohri and Pereira 1998; Nederhof 1998]. MN-transformation [Mohri and Nederhof 2001] is a straightforward and practical technique that maintains grammar readability and modifiability. While more complex methods for tighter approximations exist [Eğecioğlu 2009], our experience shows that they often lead to combinatorial explosions with minimal precision gains. In context-sensitive pointer analysis, CFL-reachability helps identify precision-critical variables or objects. These are treated context-sensitively to retain analysis precision, while others are handled context-insensitively to reduce performance costs [He et al. 2021; Lu and Xue 2019]. This involves verifying conditions against multiple interleaved CFLs, often using regular approximations for efficiency. In this work, we utilize regular approximation to prune non-contributing edges from the input graph, significantly enhancing CFL-reachability performance.

8 Conclusion

In this paper, we introduce MoYe, a conceptually simple yet highly effective regularization-based graph simplification technique for enhancing CFL-reachability analysis. MoYe uses a classic regular approximation to convert a context-free language into a regular one. By analyzing the transition rules of a deterministic finite automaton—an equivalent form of a regular language—we identify key contributing edges for CFL-reachability. Extensive experiments in field-sensitive points-to analysis for Java and alias analysis for C/C++ show that MoYe not only accelerates CFL-reachability analysis but also outperforms a leading graph simplification method. When combined with this method, MoYe delivers further performance gains, underscoring its utility in optimizing CFL-reachability analysis. In batch settings, MoYe continues to boost CFL-reachability analysis performance.

Future work could explore developing a tighter approximation that maximizes the removal of non-contributing edges while maintaining the efficiency of pre-analysis. Additionally, tailoring the approximation specifically for program analysis workloads could be beneficial, as the context-free grammars used in CFL-based analyses often exhibit simpler structures.

Acknowledgments

We thank all reviewers for their valuable feedback. This work is supported by the National Key R&D Program of China (2022YFB3103900), the Jiangsu Provincial Key R&D Program (BG2024028), the National Natural Science Foundation of China (62402474, 62132020, 62202452, and HW2024006), and the China Postdoctoral Science Foundation (2024M753295).

Data Availability Statement

Our artifact is publicly available at [Shi et al. 2025].

References

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA. doi:10.5555/1177220

Karim Ali and Ondřej Lhoták. 2012. Application-Only Call Graph Construction. In *ECOOP 2012 – Object-Oriented Programming*, James Noble (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 688–712. doi:10.1007/978-3-642-31057-7_30

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.

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. *Acm Sigplan Notices* 49, 6 (2014), 259–269. doi:10.1145/2666356.2594299

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: java benchmarking development and analysis. *SIGPLAN Not.* 41, 10 (oct 2006), 169–190. doi:10.1145/1167515.1167488

Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*. Association for Computing Machinery, New York, NY, USA, 241–250. doi:10.1145/1985793.1985827

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. Association for Computing Machinery, New York, NY, USA, 243–262. doi:10.1145/1640089.1640108

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), 1–30. doi:10.1145/3158118

Swarat Chaudhuri. 2008. Subcubic algorithms for recursive state machines. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 159–169. doi:10.1145/1328438.1328460

Shuo Ding and Qirun Zhang. 2023. Mutual Refinements of Context-Free Language Reachability. In *Static Analysis*, Manuel V. Hermenegildo and José F. Morales (Eds.). Springer Nature Switzerland, Cham, 231–258. doi:10.1007/978-3-031-44245-2_12

Ömer Eğecioğlu. 2009. Strongly Regular Grammars and Regular Approximation of Context-Free Languages. In *Proceedings of the 13th International Conference on Developments in Language Theory* (Stuttgart, Germany) *(DLT '09)*. Springer-Verlag, Berlin, Heidelberg, 207–220. doi:10.1007/978-3-642-02737-6_16

Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices* 42, 10 (2007), 57–76. doi:10.1145/1297027.1297033

Ben Hardekopf and Calvin Lin. 2007. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 290–299. doi:10.1145/1273442.1250767

Dongjie He, Yujiang Gui, Wei Li, Yonggang Tao, Changwei Zou, Yulei Sui, and Jingling Xue. 2023. A Container-Usage-Pattern-Based Context Debloating Approach for Object-Sensitive Pointer Analysis. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 971–1000. doi:10.1145/3622832

Dongjie He, Jingbo Lu, and Jingling Xue. 2021. Context Debloating for Object-Sensitive Pointer Analysis. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 79–91. doi:10.1109/ASE51524.2021.9678880

Dongjie He, Jingbo Lu, and Jingling Xue. 2022. Qilin: A New Framework For Supporting Fine-Grained Context-Sensitivity in Java Pointer Analysis. In *36th European Conference on Object-Oriented Programming (ECOOP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 30:1–30:29. doi:10.4230/LIPIcs.ECOOP.2022.30

Dongjie He, Jingbo Lu, and Jingling Xue. 2024a. Artifact of "A CFL-Reachability Formulation of Callsite-Sensitive Pointer Analysis with Built-In On-The-Fly Call Graph Construction". doi:10.5281/zenodo.11061891

Dongjie He, Jingbo Lu, and Jingling Xue. 2024b. A CFL-Reachability Formulation of Callsite-Sensitive Pointer Analysis with Built-In On-The-Fly Call Graph Construction. In *38th European Conference on Object-Oriented Programming (ECOOP 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2024.18

John Hopcroft. 1971. An n log n Algorithm for Minimizing States in a Finite Automaton. In *Theory of Machines and Computations*, Zvi Kohavi and Azaria Paz (Eds.). Academic Press, 189–196. doi:10.1016/B978-0-12-417750-5.50022-1

Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On synthesis of program analyzers. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28*. Springer, 422–430. doi:10.1007/978-3-319-41540-6_23

Yuxiang Lei, Camille Bossut, Yulei Sui, and Qirun Zhang. 2024. Context-Free Language Reachability via Skewed Tabulation. *Proc. ACM Program. Lang.* 8, PLDI, Article 221 (jun 2024), 24 pages. doi:10.1145/3656451

Yuxiang Lei, Yulei Sui, Shuo Ding, and Qirun Zhang. 2022a. Taming transitive redundancy for context-free language reachability. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 1556–1582. doi:10.1145/3563343

Yuxiang Lei, Yulei Sui, Ding Shuo, and Qirun Zhang. 2022b. Artifact of "Taming transitive redundancy for context-free language reachability". (2022). doi:10.5281/zenodo.7066401

Yuxiang Lei, Yulei Sui, Shin Hwei Tan, and Qirun Zhang. 2023a. Artifact of "Recursive State Machine Guided Graph Folding for Context-Free Language Reachability". (2023). doi:10.5281/zenodo.7787371

Yuxiang Lei, Yulei Sui, Shin Hwei Tan, and Qirun Zhang. 2023b. Recursive State Machine Guided Graph Folding for Context-Free Language Reachability. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 318–342. doi:10.1145/3591233

Lian Li, Cristina Cifuentes, and Nathan Keynes. 2011. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. Association for Computing Machinery, New York, NY, USA, 343–353. doi:10.1145/2025113.2025160

Yue Li, Tian Tan, Yifei Zhang, and Jingling Xue. 2016. Program tailoring: Slicing by sequential criteria. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECOOP.2016.15

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*. 780–793. doi:10.1145/3385412.3386021

Jingbo Lu and Jingling Xue. 2019. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 148:1–148:29. doi:10.1145/3360574

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. doi:10.1016/S0304-3975(00)00049-9

Mehryar Mohri and Mark-Jan Nederhof. 2001. *Regular Approximation of Context-Free Grammars through Transformation*. Springer Netherlands, Dordrecht, 153–163. doi:10.1007/978-94-015-9719-7_6

Mehryar Mohri and Fernando C. N. Pereira. 1998. Dynamic compilation of weighted context-free grammars *(ACL '98/COLING '98)*. Association for Computational Linguistics, USA, 891–897. doi:10.3115/980691.980716

Mark-Jan Nederhof. 1998. Context-free parsing through regular approximation *(FSMNLP '09)*. Association for Computational Linguistics, USA, 13–24. doi:doi/10.5555/1611533.1611535

Mark-Jan Nederhof. 2000. Regular approximation of CFLs: a grammatical view. In *Advances in Probabilistic and other Parsing Technologies*. Springer, 221–241. doi:10.1007/978-94-015-9470-7_12

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

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*. 49–61. doi:10.1145/199448.199462

Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. 1994. Speeding up slicing. *ACM SIGSOFT Software Engineering Notes* 19, 5 (1994), 11–20. doi:10.1145/193173.195287

Atanas Rountev and Satish Chandra. 2000. Off-line variable substitution for scaling points-to analysis. *Acm Sigplan Notices* 35, 5 (2000), 47–56. doi:10.1145/349299.349310

Chenghang Shi, Dongjie He, Haofeng Li, Jie Lu, Lian Li, and Jingling Xue. 2025. *Artifact of "Fast Client-Driven CFL-Reachability via Regularization-Based Graph Simplification"*. doi:10.5281/zenodo.16911404

Chenghang Shi, Haofeng Li, Jie Lu, and Lian Li. 2024a. Better Not Together: Staged Solving for Context-Free Language Reachability. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. Association for Computing Machinery, New York, NY, USA, 1112–1123. doi:10.1145/3650212.3680346

Chenghang Shi, Haofeng Li, Yulei Sui, Jie Lu, Lian Li, and Jingling Xue. 2023. Two Birds with One Stone: Multi-Derivation for Fast Context-Free Language Reachability Analysis. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 624–636. doi:10.1109/ASE56229.2023.00118

Chenghang Shi, Haofeng Li, Yulei Sui, Jie Lu, Lian Li, and Jingling Xue. 2024b. PEARL: A Multi-Derivation Approach to Efficient CFL-Reachability Solving. *IEEE Transactions on Software Engineering* (2024). doi:10.1109/TSE.2024.3437684

Qingkai Shi, Yongchao Wang, Peisen Yao, and Charles Zhang. 2022. Indexing the extended Dyck-CFL reachability for context-sensitive program analysis. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 1438–1468. doi:10.1145/3563339

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. Association for Computing Machinery, New York, NY, USA, 693–706. doi:10.1145/3192366.3192418

Manu Sridharan, Stephen J Fink, and Rastislav Bodik. 2007. Thin slicing. In *Proceedings of the 28th ACM SIGPLAN conference on programming language design and implementation*. 112–122. doi:10.1145/1250734.1250748

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. *ACM SIGPLAN Notices* 40, 10 (2005), 59–76. doi: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*. ACM, 265–266. doi:10.1145/2892208.2892235

Yulei Sui and Jingling Xue. 2018. Value-flow-based demand-driven pointer analysis for C and C++. *IEEE Transactions on Software Engineering* 46, 8 (2018), 812–835. doi:10.1109/TSE.2018.2869336

Tian Tan and Yue Li. 2023. Tai-e: A Developer-Friendly Static Analysis Framework for Java by Harnessing the Good Designs of Classics *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1093–1105. doi:10.1145/3597926.3598120

Rei Thiessen and Ondřej Lhoták. 2017. Context transformations for pointer analysis. *ACM SIGPLAN Notices* 52, 6 (2017), 263–277. doi:10.1145/3062341.3062359

Jyothi Vedurada and V. Krishna Nandivada. 2020. Batch alias analysis. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 936–948. doi:10.1109/ASE.2019.00091

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. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 389–404. doi:10.1145/3037697.3037744

James O. Westgard. n.d.. Lesson 34: What is an acceptable CV? https://westgard.com/lessons/z-stats-basic-statistics/lesson34.html Accessed: 2025-01-12.

Guoqing Xu, Atanas Rountev, and Manu Sridharan. 2009. Scaling CFL-Reachability-Based Points-To Analysis Using Context-Sensitive Must-Not-Alias Analysis. In *ECOOP*, Vol. 9. Springer, 98–122. doi:10.1007/978-3-642-03013-0_6

Pei Xu, Yuxiang Lei, Yulei Sui, and Jingling Xue. 2024. Iterative-Epoch Online Cycle Elimination for Context-Free Language Reachability. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 145 (April 2024), 26 pages. doi:10.1145/3649862

Mihalis Yannakakis. 1990. Graph-theoretic methods in database theory. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 230–242. doi:10.1145/298514.298576

Peisen Yao, Jinguo Zhou, Xiao Xiao, Qingkai Shi, Rongxin Wu, and Charles Zhang. 2024. Falcon: A Fused Approach to Path-Sensitive Sparse Data Dependence Analysis. *Proc. ACM Program. Lang.* 8, PLDI, Article 170 (June 2024), 26 pages. doi:10.1145/3656400

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*. 435–446. doi:10.1145/2491956.2462159

Qirun Zhang, Xiao Xiao, Charles Zhang, Hao Yuan, and Zhendong Su. 2014. Efficient subcubic alias analysis for C. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. 829–845. doi:10.1145/2660193.2660213

Xin Zheng and Radu Rugina. 2008. Demand-driven alias analysis for C. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 197–208. doi:10.1145/1328438.1328464

Received 2024-10-15; accepted 2025-08-12