Iterative-Epoch Online Cycle Elimination for Context-Free Language Reachability

PEI XU[*], University of Technology Sydney, Australia and University of New South Wales, Australia
YUXIANG LEI[*], University of New South Wales, Australia
YULEI SUI, University of New South Wales, Australia
JINGLING XUE, University of New South Wales, Australia

Context-free language reachability (CFL-reachability) is a fundamental framework for implementing various static analyses. CFL-reachability utilizes context-free grammar (CFG) to extend the expressiveness of ordinary graph reachability from an unlabeled graph to an edge-labeled graph. Solving CFL-reachability requires a (sub)cubic time complexity with respect to the graph size, which limits its scalability in practice. Thus, an approach that can effectively reduce the graph size while maintaining the reachability result is highly desirable. Most of the existing graph simplification techniques for CFL-reachability work during the preprocessing stage, i.e., before the dynamic CFL-reachability solving process. However, in real-world CFL-reachability analyses, there is a large number of reducible nodes and edges that can only be discovered during dynamic solving, leaving significant room for on-the-fly improvements.

This paper aims to reduce the graph size of CFL-reachability dynamically via online cycle elimination. We propose a simple yet effective approach to detect collapsible cycles in the graph based on the input context-free grammar. Our key insight is that symbols with particular forms of production rules in the grammar are the essence of transitivity of reachability relations in the graph. Specifically, in the graph, a reachability relation to a node v_i can be "transited" to another node v_j if there is a transitive relation from v_i to v_j, and cycles formed by transitive relations are collapsible. In this paper, we present an approach to identify the *transitive symbols* in a context-free grammar and propose an *iterative-epoch* framework for online cycle elimination. From the perspective of non-parallelized CFL-reachability solving, our iterative-epoch framework is well compatible with both the standard (unordered) solver and the recent ordered solver, and can significantly improve their performance. Our experiment on context-sensitive value-flow analysis for C/C++ and field-sensitive alias analysis for Java demonstrates promising performance improvement by our iterative-epoch cycle elimination technique. By collapsing cycles online, our technique accelerates CFL-reachability solving by 17.17× and 13.94× for value-flow analysis and alias analysis, respectively, with memory reductions of 48.8% and 45.0%.

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

Additional Key Words and Phrases: CFL-reachability, online graph simplification, performance

ACM Reference Format:
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. https://doi.org/10.1145/3649862

[*]These two authors contributed equally to this research.

Authors' addresses: Pei Xu, University of Technology Sydney, Sydney, Australia and University of New South Wales, Sydney, Australia, pei.xu@student.uts.edu.au; Yuxiang Lei, University of New South Wales, Sydney, Australia, yuxiang.lei@unsw.edu.au; Yulei Sui, University of New South Wales, Sydney, Australia, y.sui@unsw.edu.au; Jingling Xue, University of New South Wales, Sydney, Australia, j.xue@unsw.edu.au.

ACM 2475-1421/2024/4-ART145
https://doi.org/10.1145/3649862

1 INTRODUCTION

A context-free language (CFL) is a set of strings that are accepted by a context-free grammar (CFG). In program analysis, a large variety of problems, including points-to analysis [Zheng and Rugina 2008], dataflow analysis [Reps et al. 1995], and shape analysis [Reps 1995], polymorphic flow analysis [Rehof and Fähndrich 2001], taint analysis [Huang et al. 2015], can be formulated into CFL-reachability, which determines whether specified sources and sinks in a directed graph are connected by a path where the sequence of edge labels forms a string belonging to a given CFL.

Considering the (sub)cubic complexity bottleneck of CFL-reachability solving regarding the graph size, methods for simplifying the graph is always desired. Among the existing graph simplification techniques, cycle elimination [Fähndrich et al. 1998; Hardekopf and Lin 2007a; Nuutila and Soisalon-Soininen 1994; Tarjan 1972] is the most common approach to simplify graphs for transitive closure. Edge contraction [Hardekopf and Lin 2007b; Rountev and Chandra 2000] is another approach to reduce graph sizes by contracting edges whose nodes are equivalent regarding a particular analysis. The above two techniques are widely applied to Andersen's pointer analysis [Andersen 1994] where the type of collapsible cycles is obvious and detecting collapsing cycles is simple. Recently, algorithms [Lei et al. 2023; Li et al. 2020] are proposed to remove redundant edges for CFL-reachability. The aforementioned techniques have time complexities no more than quadratic, making them asymptotically more efficient. However, the common drawback of those approaches is that they can only be performed in the preprocessing stage. Since the edges are gradually increased in the graph of CFL-reachability during the dynamic solving, there is a large number of redundant nodes and edges that can only be detected in the online solving process, making online graph simplification techniques for CFL-reachability desired.

In the literature, online graph simplification for CFL-reachability has seldom been investigated. There are two main challenges. The first one is identifying the reducable edges, which is similar to offline graph simplification. The difference is that offline graph simplification identifies edges labeled by terminals, which are given in the initial input graph, whereas online graph simplification identifies edges labeled by non-terminals, which are generated and added to the graph during the dynamic solving. The second challenge, also the most important one, is balancing the dynamic processes of edge addition and edge reduction. This is because solving CFL-reachability is a process of dynamically generating and adding edges to the graph [Melski and Reps 2000; Yannakakis 1990], whereas graph simplification is to remove edges (and nodes) from the graph. The process of reduction can interfere with the solving process and, hence, lower the efficiency of solving and destroy the soundness of the solution. In particular, the soundness of the reduction depends on ensuring that the simplified graph produces equivalent results to the original graph. In other words, an online graph simplification technique needs to address *what* to reduce, and *how* to perform the online reduction.

In this paper, we present an *online cycle elimination* technique for CFL-reachability. With respect to "what to reduce", we exploit transitivity in CFL-reachability. Our key insight is that, in some context-free grammars, there are non-terminals that can extend other non-terminals. Such a non-terminal can be regarded as the "transitive symbol". An intuitive example is the symbol A for $B ::= B\ A$ and $C ::= A\ C$. We develop a method to identify the transitive symbols in a context-free grammar, and show that a cycle comprised of edges labeled by the transitive symbol has all of its nodes equivalent regarding the CFL-reachability problem, meaning that the cycle can be collapsed.

With respect to "how to perform the reduction", we try to find the sweet spot to address the tension between detecting and collapsing cycles too early and too late. This is because detecting cycles too early will result in the overhead due to repeatedly sweeping the graph, while detecting cycles too late will reduce the benefits of cycle elimination in the dynamic solving process. We propose an *iterative-epoch framework* for solving CFL-reachability solving, with online cycle elimination embedded. In each epoch, we separate the processes of deriving edges into two groups – transitive edges and other edges – and perform cycle elimination for transitive edges before solving other edges. In this way, the wasted edge generations caused by transitive cycles are expected to be reduced to the largest extent. In particular, the cycle elimination algorithm [Nuutila and Soisalon-Soininen 1994] we adopt in our iterative-epoch framework has a linear time complexity regarding the graph size, which is asymptotically faster than the cubic CFL-reachability solving.

By dynamically reducing the graph size during reachability solving, our technique can effectively reduce wasted computation, hence accelerating CFL-reachability analysis and reducing its memory overhead. From the perspective of non-parallelized CFL-reachability solving, our iterative-epoch framework shows good compatibility with existing solvers. We implemented the framework into both the standard solver [Melski and Reps 2000], and a recent partially-ordered solver Pocr [Lei et al. 2022], yielding two algorithms Iea and Iea-Ocr. Experimental evaluations on context-sensitive value-flow analysis [Sui and Xue 2016] and field-sensitive alias analysis [Sridharan et al. 2005] yield promising results. On the one hand, Iea accelerates the standard solver by 17.17× and 13.94× for value-flow analysis and alias analysis, respectively, with memory reductions of 48.8% and 45.0%. On the other hand, Iea-Ocr accelerates the ordered solver by 14.32× and 8.36× for value-flow analysis and alias analysis, respectively, with memory reductions of 55.2% and 57.8%.

The main contributions of this paper are listed as follows:

- We propose an approach to identify transitive symbols in context-free grammar and show that it is safe to collapse cycles formed by edges labeled by transitive symbols.
- We propose an iterative-epoch framework for online cycle elimination, which aims to reduce the wasted edge generations caused by transitive cycles to the largest extent.
- We combine our iterative-epoch framework with both the standard CFL-reachability solver [Melski and Reps 2000] and a recent partially-ordered solver Pocr [Lei et al. 2022], yielding two algorithms Iea and Iea-Ocr.
- We evaluate the performance of Iea and Iea-Ocr on context-sensitive value-flow analysis for C/C++ and field-sensitive alias analysis for Java and show the promising performance of our iterative-epoch online cycle elimination technique.

This paper is divided into eight main sections. Section 2 presents a motivating example to showcase the key idea of our approach. Section 3 provides the necessary background and outlines the research problem. Section 4 details the approach of identifying collapsible cycles and the basic iterative-epoch framework. Section 5 discusses the combination of our iteration-epoch with existing CFL-reachability solvers. Section 6 is the experimental evaluation, followed by related works and conclusion in Sections 7 and 8.

2 MOTIVATING EXAMPLE

This section provides an example to illustrate our insight, motivate our approach, and show the benefits and the challenges.

Figure 1(a) is a C code fragment for dereference analysis. The analysis aims to figure out the dereference relations between pointers and variables. Figure 1(b) shows the context-free grammar (CFG) for the analysis and the graph abstracted from the code fragment in Figure 1(a). In the CFG, terminals a, b, c, d denote assignment, function call, return, and dereference instructions, respectively, and nonterminal A and S denote value flow and propagation of dereferenced values, respectively. S is the start symbol, i.e., S-relations are the relations that CFL-reachability result collects. In other words, the CFL-reachability problem is to determine the S-reachability relations among nodes in G. According to the CFG, S-relations can be transited by A-relations in both

```
int v0, *v1, *v2, *v3;
int *v4, *v5, *v6, **v7;
v4 = *v7;
v5 = foo(v4);
v6 = v5;
v4 = v6;
int* foo(v3) {
     v1 = v3;
     v0 = *v1;
     v2 = v1;
     v3 = v2;
     return v2;
}
```

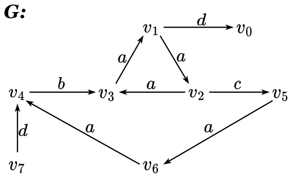
(a) C code fragment.

CFG:

$A ::= a \mid \varepsilon \mid b\ A\ c \mid A\ A$

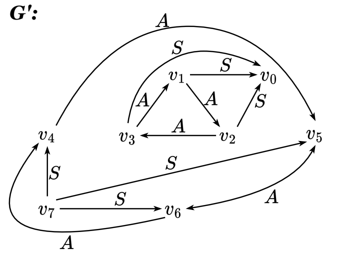
$S ::= d \mid A\ S \mid S\ A$

G:

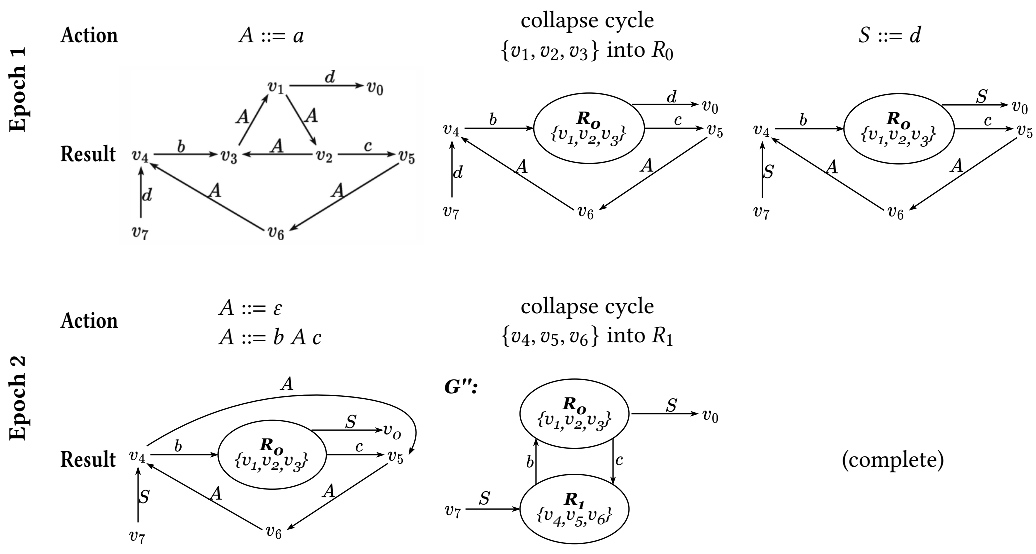


(b) A CFG and a directed graph G.

G′:



(c) G' transformed from G via standard CFL-reachability solving



(d) G'' is transformed from G via our technique within two epochs, with online cycle elimination embedded in each epoch.

Fig. 1. Motivating example for iterative-epoch online cycle elimination.

directions via $S ::= A\ S$ and $S ::= S\ A$. Namely, any two edges $v_i \xrightarrow{A} v_j$ and $v_j \xrightarrow{S} v_k$ will generate a new one $v_i \xrightarrow{S} v_k$ and any two edges $v'_i \xrightarrow{S} v'_j$ and $v'_j \xrightarrow{A} v'_k$ will generate a new one $v'_i \xrightarrow{S} v'_k$. Based on this observation, A is identified as the transitive symbol of the CFG.

Cycles and redundant edges. Solving CFL-reachability is to generate new edges from existing ones via the production rules and add the new edges to the graph to make explicit reachability relations. Figure 1(c) is the resulting graph with all the edges generated and added to G by applying the production rules of the CFG in Figure 1(b). We notice that there are two cycles comprised of A-edges in G', i.e., $v_1 \xrightarrow{A} v_2 \xrightarrow{A} v_3 \xrightarrow{A} v_1$ and $v_4 \xrightarrow{A} v_5 \xrightarrow{A} v_6 \xrightarrow{A} v_4$ generated by $A ::= A\ A$ and $A ::= b\ A\ c$. And there are $v_1 \xrightarrow{S} v_0$ and $v_7 \xrightarrow{S} v_4$ generated by $S ::= d$. Because of $S ::= A\ S$, there are

$v_2 \xrightarrow{S} v_0$ and $v_3 \xrightarrow{S} v_0$. Because of $S ::= S\ A$, there are $v_7 \xrightarrow{S} v_5$ and $v_7 \xrightarrow{S} v_6$. Namely, with respect to S-relations, v_1, v_2 and v_3 are equivalent reachable sources, and v_4, v_5 and v_6 are equivalent reachable targets. Thus, such reachability information can be recorded in an equivalent graph, i.e., G'' in Figure 1(d), with only three nodes, where R_0 represents v_1, v_2 and v_3, and R_1 represents v_4, v_5 and v_6. A comparison G'' of Figure 1(d) and G' of Figure 1(c) (which is solved without cycle elimination) shows that, if we collapse the cycles into representative nodes before applying the rule $S ::= S\ A$, we can avoid generating and adding four redundant S-edges whilst obtaining an equivalent result. Moreover, in a real-world program, the number of nodes in collapsible cycles can be up to tens of thousands or even more, and a large number of which cannot be found until performing the dynamic reachability analysis [Fähndrich et al. 1998; Pereira and Berlin 2009]. Therefore, online cycle elimination is desirable.

Our approach and its benefits. Figure 1(d) illustrates how our iterative-epoch framework performs cycle elimination during the dynamic CFL-reachability solving procedure. Given a transitive symbol A, the solving procedure undergoes two epochs, with each epoch repeatedly performing three actions: generating A-edges, collapsing A-cycles and generating other edges. Specifically, in Epoch 1, A-edges are generated from a-edges via $A ::= a$ and the A-cycle $\{v_1, v_2, v_3\}$ is detected and collapsed into R_0. In Epoch 2, an A-edge $v_4 \xrightarrow{A} v_5$ is generated from $v_4 \xrightarrow{b} R_0 \xrightarrow{c} v_5$ via $A ::= \varepsilon$ and $A ::= b\ A\ c$ and the A-cycle $\{v_4, v_5, v_6\}$ is detected and collapsed into R_1. After collapsing the cycle, there are no other new edges that can be generated in the graph, so the CFL-reachability solving is completed.

The benefit is that, by collapsing cycles before generating an edge in each epoch, we avoid generating redundant edges. In this example, by collapsing the cycle $\{v_1, v_2, v_3\}$, we avoid generating two redundant edges $v_2 \xrightarrow{S} v_0$ and $v_3 \xrightarrow{S} v_0$. By collapsing the cycle $\{v_4, v_5, v_6\}$, we avoid two redundant edges $v_7 \xrightarrow{S} v_5$ and $v_7 \xrightarrow{S} v_6$. In the literature, [Lei et al. 2022, 2023; Wang et al. 2017], offline cycle elimination was used in particular clients to reduce the graph size in the preprocessing stage. However, the offline technique can only collapse the a-cycle $\{v_1, v_2, v_3\}$. The cycle $\{v_4, v_5, v_6\}$ is only exposed in the dynamic solving process, which is beyond the capacity of the offline approach.

Challenges. Here, we re-illustrate the two challenges stated in the introduction using this motivating example. The first challenge is identifying collapsible cycles. In this example, the non-terminal A is regarded as the transitive symbol for collapsible A-cycles because of $S ::= S\ A$ and $S ::= A\ S$. However, in complex grammar with more production rules, this is not the case. We have to be thoughtful in choosing transitive symbols to guarantee a sound cycle elimination. The second challenge is how to perform cycle elimination. In this example, the cycles $\{v_1, v_2, v_3\}$ and $\{v_4, v_5, v_6\}$ are explicit at the beginning of each epoch, and there is no redundant edge generations caused by cycles. However, in real-world problems with larger graphs, cycles may not always emerge right after the last epoch's solving, and there may also be new cycles generated during each epoch's edge generation. Thus, we need to carefully choose the appropriate spot to perform cycle detection and collapsing.

3 PRELIMINARIES

This section introduces the relevant backgrounds and outlines the research problem this paper seeks to address.

$$\begin{array}{lll} A ::= a & B ::= b & S ::= d \\ A ::= B\ c & B ::= B\ A & S ::= S\ A \\ A ::= A\ A & & S ::= A\ S \end{array}$$

Fig. 2. Normalized version of the CFG in Figure 1(a).

3.1 Context-Free Language

A context-free language (CFL) is a series of strings generated by a context-free grammar (CFG). Specifically, a $CFG = \langle \Sigma, N, T, P, S \rangle$ consists of the following components:

$\Sigma = N \bigcup T$	is an *alphabet* containing all the symbols used in the CFL;
N	is a set of *non-terminal* symbols;
T	is a set of *terminal* symbols;
P	is a set of productions, each of which describes how the non-terminal on the left side produces the terminals/non-terminals on the right side;
$S \in N$	is the start symbol of the CFL, i.e., where the language starts generating.

Starting from S, all the terminal strings that belong to the CFL and can be generated by one or more productions of the CFG. For example, consider the context-free grammar with the start symbol S, terminal symbols $T = \{a, b\}$, and productions:

$$S ::= \ a\ S\ b \mid \varepsilon\ ,$$

where ε denotes an empty string. According to the production $S ::= \ a\ S\ b$, the start symbol S generates $a\ S\ b$. Then, by applying $S ::= \varepsilon$ to $a\ S\ b$, we get a string ab, which belongs to the CFL generated by the grammar. Besides, we can also apply $S ::= \ a\ S\ b$ again to $a\ S\ b$ to generate $aaSbb$, which then can produce a string $aabb$ belonging to the same CFL. In fact, any string $a^n b^n$, where n is a non-negative integer, belongs to the CFL.

3.2 CFL-Reachability

In general, given a context-free grammar *CFG* and an edge-labeled graph $G = \langle V, E \rangle$, a CFL-reachability instance $Reach\langle CFG, G\rangle$ is to determine whether particular node pairs are connected by a path whose edge labels sequentially form a string accepted by *CFG*. Specifically, for each edge $v_i \xrightarrow{X} v_j$ in G, $X \in \Sigma$. Initially, G contains only edges labeled with terminals.

Solving CFL-reachability follows a dynamic programming scheme called *summarization*, which is the reverse process of string generation. Given a path $p = v_0 \xrightarrow{Y_1} v_1 \xrightarrow{Y_2} \cdots \xrightarrow{Y_n} v_n$, if there exists a production $X ::= Y_1 Y_2 \cdots Y_n \in P$ whose right-hand side is exactly the sequence of edge labels of p, then p can be summarized into an edge $v_0 \xrightarrow{X} v_n$. If $v_0 \xrightarrow{X} v_n$ is not already in the graph, then it will be added to the graph to make the X-relation from v_0 to v_n explicit.

Formally, for two nodes $v_i, v_j \in V$, v_j is said to be X-*reachable* from v_i if $v_i \xrightarrow{X} v_j$ can be summarized from a path from v_i to v_j. Given specified sources $V_{src} \subseteq V$ and sinks $V_{snk} \in V$, CFL-reahability is to determine for each pair $(v_i, v_j) \in V_{src} \times V_{snk}$ whether v_j is S-*reachable* from v_i.

Solving CFL-Reachability. CFL-reachability is usually solved based on a standard dynamic programming algorithm [Melski and Reps 2000]. The algorithm requires the input grammar to be normalized, i.e., the right-hand side of each production has at most two symbols. For example, production $S ::= \ a\ S\ b$ needs to be converted into $A ::= \ a\ S$ and $S ::= \ A\ b$. Figure 2 is a normalized version of the CFG in Figure 1(a). For the convenience of discussion, we provide the details of the standard algorithm in Algorithm 1. The algorithm maintains a worklist W to record new edges.

Algorithm 1: Standard CFL-Reachability Algorithm.

```
1  Function Reach⟨$CFG, G$⟩
2      init();                                                        /* Lines 11-15 */
3      while $W \neq \emptyset$ do
4          select and remove an edge $v_i \xrightarrow{Y} v_j$ from $W$;
5          for each production $X ::= Y \in P$ do
6              if $v_i \xrightarrow{X} v_j \notin E$ then add $v_i \xrightarrow{X} v_j$ to $E$ and to $W$;
7          for each production $X ::= Y\ Z \in P$ do
8              SearchForward($X, Z, v_i, v_j, W$);                    /* Lines 19-21 */
9          for each production $X ::= Z\ Y \in P$ do
10             SearchBackward($X, Z, v_i, v_j, W$);                   /* Lines 16-18 */
11 Procedure init()
12     add all edges of $E$ to $W$;
13     for each production $X ::= \varepsilon \in P$ do
14         for each node $v_i \in V$ do
15             if $v_i \xrightarrow{X} v_i \notin E$ then add $v_i \xrightarrow{X} v_i$ to $E$ and to $W$ ;
16 Procedure SearchForward($X, Z, v_i, v_j, W$)
17     for each edge $v_j \xrightarrow{Z} v_k \in G$ do
18         if $v_i \xrightarrow{X} v_k \notin E$ then add $v_i \xrightarrow{X} v_k$ to $E$ and to $W$;
19 Procedure SearchBackward($X, Z, v_i, v_j, W$)
20     for each edge $v_k \xrightarrow{Z} v_i \in G$ do
21         if $v_k \xrightarrow{X} v_j \notin E$ then add $v_k \xrightarrow{X} v_j$ to $E$ and to $W$;
```

Whenever an edge is added to the graph, it is also added to the worklist (lines 6, 15, 18 and 21). The algorithm derives edges from paths consisting of only one edge in lines 5–6, and paths containing two edges using procedures `SearchForward` (lines 16–18) and `SearchBackward` (lines 19–21). In particular, we let `SearchForward` and `SearchBackward` accept a worklist W as a parameter, as our solution (Section 4.2) includes two worklists and we want to make sure which worklist the two procedures are operating on. The process of solving CFL-reachability iteratively processes the edges in the work list until a fixed point is reached, i.e., no new edge can be added to the graph. In this fixed point, all reachability relations are explicitly represented as edges in the graph.

3.3 Problem Formulation

Cycle elimination [Nuutila and Soisalon-Soininen 1994; Tarjan 1972] is a sophisticated technique to simplify the graphs for transitive closure. It has been extensively studied in improving the scalability of constraint-based pointer analysis (a.k.a., Andersen's analysis) [Andersen 1994; Fähndrich et al. 1998; Hardekopf and Lin 2007a; Pearce et al. 2003; Pereira and Berlin 2009].

This paper aims to improve the scalability of CFL-reachability analysis by applying cycle elimination to the dynamic solving process. Different from constraint-based pointer analysis, where the type of collapsible cycle is fixed (i.e., the cycles comprised of *copy*-edges [Hardekopf and Lin 2007a]), CFL-reachability has various grammars, and the types of collapsible cycles depend on the grammar. In some grammar, the types of collapsible cycles are obscure. Moreover, even if some cycles in the input graph can be collapsed by analyzing the grammar, realizing the full potential of cycle elimination in CFL-reachability is challenging as a large number of cycles are not detectable until we perform the online CFL-reachability analysis. Thus, we need to combine cycle detection and collapsing into online CFL-reachability solving.

We define the research problem as follows:

> Given a CFL-reachability instance $Reach\langle CFG, G\rangle$, improve the efficiency of reachability solving by detecting and collapsing cycles on G in the dynamic solving process.

4 ONLINE CYCLE ELIMINATION FOR CFL-REACHABILITY

This section is to solve the challenges raised in the introduction and motivating example. In Section 4.1, we formally define transitive symbol as the basis of identifying collapsible cycles. In our approach, whether a symbol (of the alphabet) is a transitive symbol depends fully on the input context-free grammar. In Section 4.2, we present our iterative-epoch framework for online cycle elimination. Our iterative-epoch framework divides the solving process into iterations, and performs cycle detection and elimination in a particular spot of each iteration in order to keep the number of redundant edges caused by cycles to a minimum. In particular, an iterative-epoch algorithm Iea is presented as the application of the iterative-epoch framework to the standard CFL-reachability solver [Melski and Reps 2000].

4.1 Transitive Symbols and Collapsible Cycles

In this part, we formally define *transitive symbols* under the context of normalized grammars and use transitive symbols to identify collapsible cycles in CFL-reachability. In general, the basic principle of a sound cycle collapsing is that the graph where the cycles have been collapsed should be able to produce the same CFL-reachability result as the original graph. Namely, collapsing the cycle does not actually change the result.

In particular, we define the transitive symbols for *directed* problems and *bi-directed* problems, respectively. The difference between them is that in a bi-directed problem, any symbol $X \in \Sigma$ has a corresponding reverse $\overline{X} \in \Sigma$, and any edge $v_i \xrightarrow{X} v_j \in G$ has a corresponding edge $v_j \xrightarrow{\overline{X}} v_i \in G$, whereas a directed problem does not subject to this constraint. The necessity of the separate definitions is that, in a bi-directed problem, for a group of nodes $\{v_1, ...v_k\}$, there is an X-cycle running through $\{v_1, ...v_k\}$ means that there is also an $\overline{X}$-cycle running through $\{v_1, ...v_k\}$. This property results in fewer limitations for transitive symbols in a bi-directed problem.

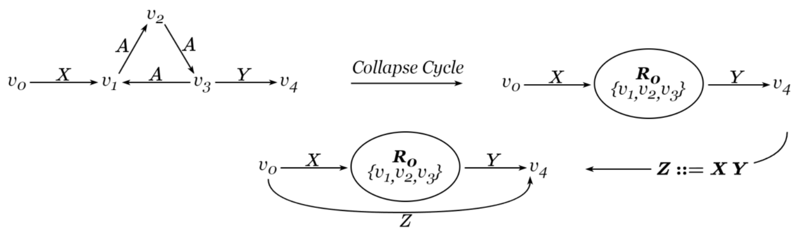
4.1.1 Transitive Symbols in Directed CFL-Reachability. In Figure 1(b) of our motivating example, A-cycles can be safely collapsed without affecting the CFL-reachability result. An important property is that nodes in such a cycle are equivalent with respect to the S-relation in the problem. Namely, given two nodes v_i, v_j in an A-cycle and an arbitrary node v_k in the graph, v_i is S-reachable from v_k iff v_j is S-reachable from v_k, meanwhile, v_k is S-reachable from v_i iff v_k is S-reachable from v_j. In the motivating example, such an equivalence property of A-cycles is guaranteed by the grammar (Figure 2), where all the non-terminals can be *transited* by A, i.e., $A ::= A\ A$, $B ::= B\ A$ and $S ::= S\ A$. In this paper, we use *A-transitivity* to call such a characteristic of a CFG (Definition 4.1).

Definition 4.1 (*A-Transitivity*). In a normalized CFG, given two symbols $X, A \in \Sigma$, X is *left-A-transitive* iff there is $X ::= A\ X \in P$; X is *right-A-transitive* iff there is $X ::= X\ A \in P$. In particular, X is *doubly-A-transitive* iff there are both $X ::= A\ X \in P$ and $X ::= X\ A \in P$.

However, A-transitivity is not sufficient to be the criteria of collapsible cycles. We can easily find counter-examples where all non-terminals are A-transitive while collapsing an A-cycle leads to an incorrect result. Consider the CFL-reachability instance in Figure 3, where all the non-terminals

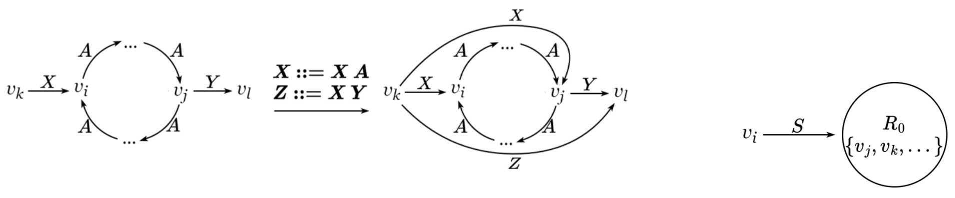
$X ::= A\ X$
$Y ::= Y\ A$
$Z ::= X\ Y$
$Z ::= Z\ A$

(a) A CFG.



(b) A graph segment.

Fig. 3. Incorrect cycle elimination.



(a) Before collapsing the A-cycle. **(b)** After collapsing the A-cycle.

Fig. 4. Collapsing an A-cycle, where A is a transitive symbol, does not change the S-edges with at least one endpoint belonging to the cycle.

are A-transitive in the CFG in Figure 3(a). In the graph segment of Figure 3(b), before collapsing the A-cycle $\{v_1, v_2, v_3\}$, there is no path making v_4 Z-reachable from v_0. After collapsing the A-cycle $\{v_1, v_2, v_3\}$, v_0 is Z-reachable from v_0 via the path $v_0 \xrightarrow{X} R_0 \xrightarrow{Y} v_4$. In this example, although X, Y and Z are all A-transitive, collapsing the A-cycle introduces an extra Z-reachability relation, which may lead to an incorrect result. However, if we replace $X ::= A\ X$ by $X ::= X\ A$ or $Y ::= Y\ A$ by $Y ::= A\ Y$ in the grammar of Figure 3(a), then v_0 will always be Z-reachable from v_0, no matter collapsing the A-cycle $\{v_1, v_2, v_3\}$ or not. This yields another observation: if the CFG contains $Z ::= X\ Y \in P$, there should be at least one of $X ::= X\ A \in P$ or $Y ::= A\ Y \in P$ to ensure that collapsing A-cycles will not lead to an incorrect result.

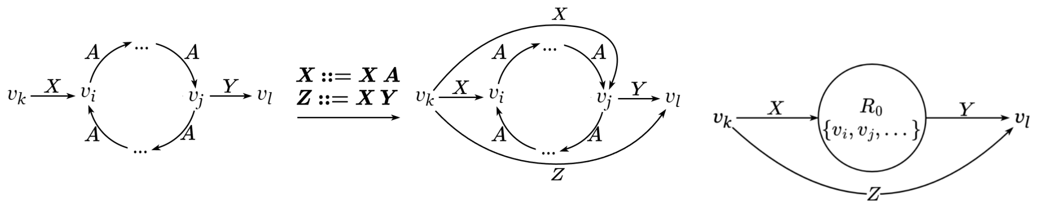
We define A-*intransitive combinations* as follows:

Definition 4.2 (*A-Intransitive Combination*)**.** In a normalized CFG, a production $Z ::= X\ Y \in P$ is a A-intransitive combination iff there is neither $X ::= X\ A \in P$ nor $Y ::= A\ Y \in P$.

The example in Figure 3 shows that, to ensure that the A-cycles are collapsible, there should be no A-intransitive combinations in the CFG. Combining the above discussions, we define transitive symbols as follows:

Definition 4.3 (*Transitive Symbol*)**.** In a normalized CFG, a symbol A is a transitive symbol iff the start symbol S is doubly-A-transitive (Definition 4.1) and the CFG does not contain any A-intransitive combination (Definition 4.2).

Theorem 4.1. *In a (directed) CFL-reachability instance Reach⟨CFG, G⟩, if A is a transitive symbol in CFG, cycles comprised of A-edges in G are collapsible.*



(a) Before collapsing the A-cycle. **(b)** After collapsing the A-cycle.

Fig. 5. Collapsing an A-cycle, where A is a transitive symbol, does not change any Z-edge whose both endpoints do not belong to the cycle.

Proof. To prove Theorem 4.1 is to prove that collapsing the A-cycles where A is a transitive symbol of the CFG will not change the CFL-reachability solution, i.e., the S-edges in the final graph. Obviously, the S-edges which are summarized from paths not containing any node in an A-cycle are never changed by collapsing the A-cycles. Thus, we only need to consider the S-edges summarized from paths containing at least one node in an A-cycle.

First, we consider the S-edges with at least one endpoint located in the A-cycle. When there is an S-edge $v_i \xrightarrow{S} v_j$ where v_j is in the A-cycle, since A is a transitive symbol, there is $S ::= S\ A$. Namely, for any other node v_k in the A-cycle, there will be $v_i \xrightarrow{S} v_k$ in the final graph, as depicted in Figure 4(a). Similarly, when there is an S-edge $v_i \xrightarrow{S} v_j$ where v_i is in the A-cycle, for any other node v_k in the A-cycle, there will be $v_k \xrightarrow{S} v_j$ in the final graph. Thus, collapsing the A-cycle will not affect the S-edges with at least one endpoint belonging to an A-cycle, as depicted in Figure 4(b).

Then, we consider the S-edges with no endpoint located in an A-cycle. For any four nodes v_i, v_j, v_k, v_l such that v_i and v_j are in an A-cycle and $v_k \xrightarrow{X} v_i \in G$, $v_j \xrightarrow{Y} v_l \in G$, since there is no A-intransitive combination, when there is $Z ::= X\ Y \in P$, there must also be either $X ::= X\ A$ or $Y ::= A\ Y$. Namely, $v_k \xrightarrow{X} v_j \in G$ or $v_i \xrightarrow{Y} v_l \in G$ will be generated. Thus, no matter collapsing the A-cycle or not, there is always $v_k \xrightarrow{Z} v_l \in G$, as depicted in Figure 5. Then we discuss how the consistency of such Z-edges guarantees the consistency of S-edges. First, the consistency of the S-edges generated by $S ::= Z \in P$ is self-evident. Then, for $S ::= Z\ Z' \in P$ and $S ::= Z'\ Z \in P$, the consistency of Z'-edges is guaranteed the same as Z-edges. So, the consistency of the S-edges is guaranteed.

Therefore, in summary, collapsing A-cycles satisfying the conditions of Theorem 4.1 will not change the S-edges in the graph. □

4.1.2 Transitive Symbols in Bi-Directed CFL-Reachability. In a bi-directed CFL-reachability problem, for a symbol $A \in \Sigma$, if there is $v_i \xrightarrow{A} v_j \in G$, there is $v_j \xrightarrow{\overline{A}} v_i \in G$. Thus, given a set of node $V' \subseteq V$, there is an A-cycle going through the nodes of V' means that there must also be an $\overline{A}$-cycle going through the nodes of V'. Because of this characteristic of bi-directed CFL-reachability, the transitive symbols in bi-directed problems are slightly different from those in directed problems:

Definition 4.4 (*Bi-A-Transitivity*). In a normalized bi-directed CFG, given three symbols $X, A, \overline{A} \in \Sigma$, X is *bi-left-A-transitive* iff there is $X ::= A\ X \in P$ or $X ::= \overline{A}\ X \in P$; X is *bi-right-A-transitive* iff there is $X ::= X\ A \in P$ or $X ::= X\ \overline{A} \in P$; and X is *bi-doubly-A-transitive* iff X is both *bi-left-A-transitive* and *bi-right-A-transitive.*

Definition 4.5 (*Bi-A-Intransitive Combination*). In a normalized bi-directed CFG, a production $Z ::= X\ Y \in P$ is a bi-A-intransitive combination iff there is neither $X ::= X\ A_1 \in P$ nor $Y ::= A_2\ Y \in P$ where $A_1, A_2 \in \{A, \overline{A}\}$.

Definition 4.6 (*Transitive Symbol in Bi-Directed CFL-Reachability*). In a normalized bi-directed CFG, a symbol A is a transitive symbol iff the start symbol S is bi-doubly-A-transitive (Definition 4.4) and the CFG does not contain any bi-A-intransitive combination (Definition 4.5).

Definitions 4.4 and 4.5 show that in a bi-directed problem, the condition for transitive symbols exist in pairs, e.g., A and $\overline{A}$. Thus, transitive symbols in bi-directed problems also exist in pairs. Namely, if A is a transitive symbol, then $\overline{A}$ must also be a transitive symbol. Similar to directed CFL-reachability, in a bi-directed CFL-reachability instance $Reach\langle CFG, G\rangle$, if A is a transitive symbol of *CFG*, cycles comprised of A-edges in G are collapsible. The proof is similar to that given in Section 4.1.1 and is not detailed here.

In real-world clients, collapsible cycles may be not explicit. However, once the CFG can be normalized into a form such that a transitive symbol A can be captured via Definition 4.3, we can safely collapse cycles comprised of A-edges in the graph.

4.2 IEA: Iterative-Epoch Algorithm for Online Cycle Elimination

In this part, we study the most appropriate strategy for online cycle elimination and propose our iterative-epoch framework.

4.2.1 Three-Stage Iterative-Epoch Framework for Online Cycle Elimination. In the standard CFL-reachability algorithm, edges are not processed in a specific order. As a consequence, a new cycle can be formed at any stage during the solving procedure, leading to redundant summarizations and edge additions by subsequent edge processing. To avoid this redundancy, an intuitive solution is to carry out cycle detection and collapse every time a new edge (labeled by a transitive symbol) is added to the graph. However, this is impractical because, despite the linear complexity of the cycle detection algorithm (i.e., SCC algorithm [Nuutila and Soisalon-Soininen 1994]), the number of edge additions in the online solving process can be up to quadratic with respect to the node number of the graph. Therefore, the problem becomes how to minimize the frequency of cycle elimination while simultaneously ensuring that redundant edge additions are reduced to the largest extent.

We propose an *iterative-epoch* framework to perform online cycle elimination in the sweet spot. Specifically, given a transitive symbol A, we separate the whole solving process into iterative epochs, with each epoch handling three tasks: generating A-edges from non-A-edges, detecting and collapsing A-cycles and generating other edges. To accomplish the three tasks, we categorize the productions into *A-generating productions* (Definition 4.7) and *non-A-generating productions*, and divide each epoch into three consecutive stages.

Definition 4.7 (*A-Generating Productions*). In a CFG where A is a transitive symbol, an A-generating production is in the form of $A ::= X$ or $A ::= X\ Y$ such that $X \neq A$ or $Y \neq A$.

Intuitively, an A-generating production can generate A-edges from non-A-edges in the graph. Relatively, the production not A-generating are collectively called non-A-generating productions.

The three stages of each epoch are as follows:

(1) Firstly, we only derive edges from A-generating productions. In this stage, all the A-edges that may introduce new A-cycles are generated and added to the graph.
(2) Secondly, we detect and collapse cycles comprised of A-edges using the strongly-connected-component (SCC) algorithm [Nuutila and Soisalon-Soininen 1994], whose complexity is linear with respect to the graph size.

Algorithm 2: Iea: Iterative-epoch algorithm for online cycle elimination.

```
Function Iea ($CFG, G$)
    init();                                                        // line 11, Algorithm 1
    while $W \neq \emptyset$ do
        GenerateTransitiveEdges();                                 // line 14
        CollapseTransitiveCycles();
        while $W_{NA} \neq \emptyset$ do
            select and remove an element $v_i \xrightarrow{Y} v_j$ from $W_{NA}$;
            for each non-$A$-generating production $X ::= Y \in P$ do
                if $v_i \xrightarrow{X} v_j \notin E$ then add $v_i \xrightarrow{X} v_j$ to $E$ and to $W$;
            for each non-$A$-generating production $X ::= Y\ Z \in P$ do
                SearchForward($X, Z, v_i, v_j, W$);                // line 16, Algorithm 1
            for each non-$A$-generating production $X ::= Z\ Y \in P$ do
                SearchBackward($X, Z, v_i, v_j, W$);               // line 19, Algorithm 1

Procedure GenerateTransitiveEdges()                                // $A$ is the transitive symbol
    while $W \neq \emptyset$ do
        select and remove an element $v_i \xrightarrow{Y} v_j$ from $W$;
        for each production $A ::= Y \in P$ do
            if $v_i \xrightarrow{A} v_j \notin E$ then add $v_i \xrightarrow{A} v_j$ to $E$ and to $W_{NA}$;
        for each $A$-generating (Definition 4.7) production $A ::= Y\ Z \in P$ do
            SearchForward($A, Z, v_i, v_j, W_{NA}$)                 // line 16, Algorithm 1
        for each $A$-generating production $A ::= Z\ Y \in P$ do
            SearchBackward($A, Z, v_i, v_j, W_{NA}$)                // line 19, Algorithm 1
        if there is any non-$A$-generating $X ::= Y \in P$ or $X ::= Y\ Z \in P$ or $X ::= Z\ Y \in P$ then
            add $v_i \xrightarrow{Y} v_j$ to $W_{NA}$;
```

(3) Lastly, we derive edges from non-A-generating productions. Since all A-cycles have already been collapsed, there will be no redundant edge caused by cycles generated in this epoch.

In each epoch, we perform cycle elimination immediately after the first stage that generates transitive A-edges from A-generating productions. This maximally avoids redundant edge generations caused by uncollapsed cycles. In another word, if we perform cycle elimination in any other spot of the epoch, there may be cycles causing redundant edges generated by productions with A on the right-hand side, e.g., $X ::= X\ A$ or $A ::= A\ A$.

We use two worklists to control the start and termination of each epoch. The first worklist is the innate worklist W of the standard algorithm, which is used to hold all unprocessed edges. Similarly, the whole process of CFL-reachability solving does not terminate until W is empty. The second worklist is called non-A-generating worklist and is denoted by W_{NA}, which is used to hold edges that are not processed by applying A-generating productions. While solving the item in W, the newly generated edges are added to W_{NA}; and while solving the items in W_{NA}, the newly generated edges are added to W. Then, after processing all the items in W_{NA}, we are able to determine whether to start a new epoch according to whether W is not empty.

4.2.2 Iterative-Epoch Algorithm. We present the basic iterative-epoch algorithm (Iea) for online cycle elimination in CFL-reachability, as seen in Algorithm 2. It directly applies the iterative-epoch framework (Section 4.2.1) into the standard algorithm (Algorithm 1) and extensively reuses the methods of the standard algorithm, e.g., `init`, `SearchForward` and `SearchBackward`.

Iea consists of two parts: initialization (line 2) and iterative epochs (lines 3–13). The initialization stage follows the initialization scheme of the standard algorithm (lines 11–15, Algorithm 1) to initialize the graph and the worklist. In this part, all the unprocessed edges, including all edges in the graph and all underlying self-cycles generated from empty productions (i.e., productions in the form of $X ::= \varepsilon$) are added to the main worklist W.

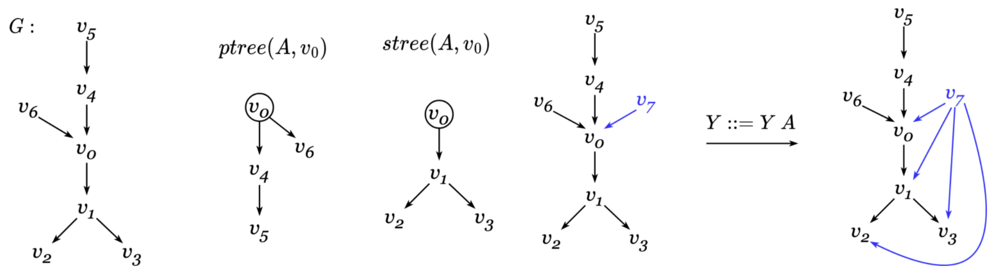
The iterative-epochs follow the three-stage scheme in Section 4.2.1. Stage (1) `GenerateTransitive-Edges`() (line 4) sequentially pops items $v_i \xrightarrow{Y} v_j$ from the main worklist W, and generates edges from the item (and its adjacent edges) via only A-generating productions. In this stage, any new A-edge generated in `SearchForward` (line 20) and `SearchBackward` (line 22) is added to the non-A-generating worklist W_{NA}. In particular, if the label Y of the popped item involves non-A-generating productions, which means that the popped item $v_i \xrightarrow{Y} v_j$ is not completely processed, the item itself will also be added to W_{NA} (lines 23–24), waiting for further processing.

Stage (2) `CollapseTransitiveCycles()` (line 5) detects and collapses A-cycles using the SCC algorithm [Nuutila and Soisalon-Soininen 1994]. After that is Stage (3) (lines 6–13), which processes the items in W_{NA} by generating new edges from each item and its adjacent edges via non-A-generating productions. In this stage, any generated new edge is added to W. After completing this stage, i.e., all the items in W_{NA} are popped and solved, if we get a non-empty W, we start a new epoch by jumping back to line 3. Otherwise, the solving terminates. It is noteworthy that in this stage, all the existing A-cycles are collapsed by `CollapseTransitiveCycles()` and there is not any new A-cycle created, since the edge derivations involving A-generating productions are not considered in this stage.

In summary, Iea still follows the summarization and dynamic programming schemes of the standard algorithm. This includes generating edges via the methods `SearchForward`, `SearchBackward`, etc., and iteratively processing the items in the worklist(s), adding new edges to the graph and to the worklist(s) until no new edges can be added to the graph. Iea differs from the standard algorithm in the following two aspects:

- It divides the solving into iterative epochs, with each epoch containing three stages: (1) generating and adding A-edges that may introduce new A-cycles in the graph, (2) detecting and collapsing A-cycles, and (3) generating and adding other edges. We should point out that the A-edges generated from $A ::= A\ A$ never lead to any new A-cycle, so we process this production (if it exists in the CFG) in the last stage.
- It uses two worklists, W and W_{NA}, to keep track of edges in the A-generating stage and non-A-generating stage. New edges generated in the A-generating stage are added to W_{NA}, and new edges generated in the non-A-generating stage are added to W.

Example 4.1. Let us illustrate Algorithm 2 using our motivating example (Figure 1(d)). After initializing the graph G and the worklist W (line 2), the iterative epochs (lines 3–13) starts. In Epoch 1, the algorithm first calls `GenerateTransitiveEdges()` (line 4) to generative A-edges via A-generating productions. In this stage, five A-edges are generated via $A ::= a$, as seen in the first subgraph of Figure 1(d). Then, the algorithm calls `CollapseTransitiveCycles()` (line 5) to collapse the A-cycle $\{v_1, v_2, v_3\}$ into R_0, as seen the second subgraph in Figure 1(d). Lastly, the algorithm generate other edges via non-A-generating productions (lines 6–13). In this stage, two S-edges are generated via $S ::= d$, as seen in the third subgraph of Figure 1(d). Note that when normalizing the CFG in Figure 1, the production $A ::= b\ A\ c$ is changed into $A ::= B\ c$ and $B ::= B\ A \mid b$ (Figure 2). So there is also an edge $v_4 \xrightarrow{B} R_0$ generated and added into the worklist



(a) A graph segment and the spanning trees of v_0. **(b)** Deriving edges from $Y ::= Y\ A$ using $stree(v_0)$.

Fig. 6. Pocr maintains transitive relations in spanning trees and use them to derive edges.

W, which is not shown in Figure 1(d). Since the worklist W is not empty, Epoch 2 starts. Similarly, the algorithm generates an A-edge $v_4 \xrightarrow{A} v_5$ and collapses the A-cycle $\{v_4, v_5, v_5\}$ into R_1, as seen in the fourth and the fifth subgraphs of Figure 1(d). Since there are no more new edges that can be generated, the algorithm terminates.

5 COMBINING ITERATIVE-EPOCH FRAMEWORK WITH EXISTING SOLVERS

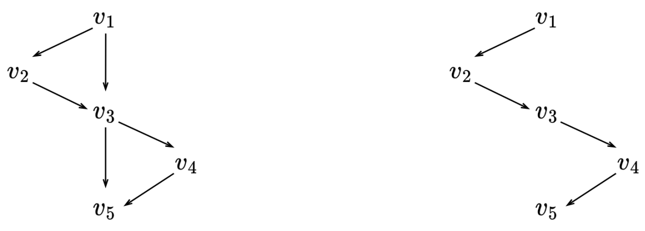
From the perspective of non-parallelized CFL-reachability solving, our iterative-epoch framework is well compatible with both the standard solver [Melski and Reps 2000] and the recent partially ordered solver [Lei et al. 2022]. The combination of the iterative-epoch framework with the standard solving is already presented in Section 4.2.2. In this section, we discuss how to combine the iterative-epoch framework with the partially ordered solver Pocr.

5.1 The Mechanism of Ordered CFL-Reachability Solving

Pocr captures the transitivity of doubly-recursive productions, i.e., $A ::= A\ A$, and performs a dynamic ordered edge generation to reduce redundant computation caused by such transitivity. Pocr uses spanning trees to record the "primary" A-edges (A-edges that are generated by A-generating productions) in a topological order for every node in the graph. When generating edges using productions containing A on the right-hand side, the solver traverses the spanning trees of the endpoints of the worklist item being processed to perform an ordered edge generation.

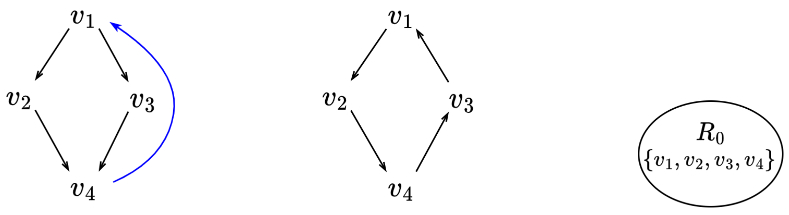
Example 5.1. For a symbol A such that $A ::= A\ A \in P$, Pocr maintains a predecessor tree $ptree(A, v_i)$ and a successor tree $stree(A, v_i)$ for each node $v_i \in V$ to collect the predecessors and successors of v_i connected by A-edges, as depicted in Figure 6(a). While processing a worklist item $v_j \xrightarrow{Y} v_i$ using a production $Y ::= Y\ A$, it traverses $stree(A, v_i)$ from the root to the leaves, sequentially deriving $v_j \xrightarrow{Y} v_k$, where v_k is a node of $stree(A, v_i)$, as depicted in Figure 6(b) where A-edges are colored in black and Y-edges are colored in blue. It avoids redundant derivations by stopping visiting the offsprings of v_k when $v_j \xrightarrow{Y} v_k$ is already in the graph. Similarly, when processing a worklist item $v_i \xrightarrow{Y} v_j$ using $Y ::= A\ Y$, it derive edges via traversing $ptree(A, v_i)$.

In a CFL-reachability problem, when the transitive symbol A (if there is one) also has a production $A ::= A\ A$, the partially ordered solver can be combined with our framework to further improve the performance of the online solving. The challenge of a sound and practical combination lies in the auxiliary data structure for transitive relations in Pocr, i.e., the spanning trees, which are naturally unsuitable for cycle elimination because cycles in the graph never manifest in the trees. In this



(a) A directed graph G. **(b)** The MEG of G.

Fig. 7. An example of minimum equivalence graph.



(a) Adding a new edge $v_4 \longrightarrow v_1$ to the graph G leads to a cycle. **(b)** The MEG of G. **(c)** The MEG is collapsed into a node via cycle elimination.

Fig. 8. Minimum equivalence graph is compatible with cycle elimination.

paper, we replace the spanning trees with a minimum equivalence graph, which is compatible with both ordered CFL-reachability solving and cycle elimination.

5.2 Minimum Equivalent Graph for Cycle Elimination

For a directed graph $G = \langle V, E\rangle$, its minimum equivalence graph $MEG = \langle V', E'\rangle$ is a subgraph of G such that $V' = V$, $E' \subseteq E$ and

(1) for any two nodes $v_i, v_j \in V$, there is a path from v_i to v_j in G iff there is a path from v_i to v_j in *MEG*, and
(2) removing any edge from E' will invalidate (1).

For example, Figure 7(a) is a directed graph, and Figure 7(b) is the MEG of Figure 7(a), where the redundant edges $v_1 \longrightarrow v_3$ and $v_3 \longrightarrow v_5$ are removed. We can easily observe that, for any two nodes v_i and v_j, if there is a path from v_i to v_j in Figure 7(a), there is also a path from v_i to v_j in Figure 7(b). On the other hand, removing any edge from Figure 7(b) will make its reachability property different from Figure 7(a). When applied to a CFL-reachability where there are transitive edges, we can maintain an MEG for the subgraph that is obtained from the main graph G by maintaining only the transitive edges, and use the MEG for ordered CFL-reachability solving.

Similar to spanning trees, a minimum equivalent graph also keeps transitive relations in order without redundant information, hence is suitable for ordered CFL-reachability solving. The difference is that a minimum equivalent graph keeps transitive relations in a graph, which is not only compatible with cycle elimination, but also memory saving. For example, in Figure 8(a), adding an edge $v_4 \longrightarrow v_1$ to G leads to a cycle, and the MEG of G is displayed in Figure 8(b), which has one

Algorithm 3: Ordered derivation for singly recursive rules using MEG.

```
1  Function OrderedSearchForward($v_i, v_j, X, A$)                    // For $X ::= X\ A$
2      for each $v_j \rightarrow v_k \in MEG_A$ do
3          if $v_i \xrightarrow{X} v_k \notin G$ then
4              add $v_i \xrightarrow{X} v_k$ to $G$ and to $W$;
5              OrderedSearchForward($v_i, v_k, X, A$);
6  Function OrderedSearchBackward($v_i, v_j, X, A$)                   // For $X ::= A\ X$
7      for each $v_k \rightarrow v_i \in MEG_A$ do
8          if $v_k \xrightarrow{X} v_j \notin G$ then
9              add $v_k \xrightarrow{X} v_j$ to $G$ and to $W$;
10             OrderedSearchBackward($v_k, v_j, X, A$);
```

edge fewer than Figure 8(a). While applying cycle elimination, both Figure 8(a) and Figure 8(b) yield the same result, as shown in Figure 8(c).

5.3 Ordered CFL-Reachability Solving with Online Cycle Elimination

The key to ordered CFL-reachability solving is to use the auxiliary data structure to help solve the productions involving the transitive symbol A. It includes two types of productions. The first one is the productions where A is on the right-hand side (but not on the left-hand side), which use existing A-edges to generate other edges. The second one is the productions where A is on the left-hand side, which generate and add A-edges to the graph.

Here, we show how to use a minimum equivalence graph for ordered CFL-reachability solving. For each transitive symbol A that has $A ::= A\ A \in P$, we maintain a minimum equivalence graph MEG_A, which is exactly the subgraph of the main graph G with all non-A-edges removed. When processing a worklist item $v_i \xrightarrow{X} v_j$ via productions like $X ::= X\ A$ (or $X ::= A\ X$), our solver searches in MEG_A in a topological from v_j (or v_i) to generate X-edges. The detail is presented in Algorithm 3. Similar to Pocr, our solver reduces redundant edges by stopping the traversal when it finds that the edge to be generated is already in the graph (lines 3 and 8).

When a new A-edge is generated (via A-generating productions) and added to the graph, our solver updates MEG_A, and generates and adds A-edges to G, via $A ::= A\ A$. The detail is presented in Algorithm 4. Specifically, when a new A-edge $v_i \xrightarrow{A} v_j$ is generated, the algorithm traverses the predecessors of v_i by `TravBackward` (line 5) and the successors of v_j by `TravForward` (line 12) to retrieve the information of the nodes v'_i and v'_j such that $v'_i \xrightarrow{A} v_i$ and $v_j \xrightarrow{A} v'_j$, then updates the new edges $v'_i \xrightarrow{A} v'_j$ in the main graph G and the worklist W_{NA} by `Update` (line 16). Notably, in the sub-procedure `TravBackward`, lines 6–8 is to remove the redundant A-edges from MEG_A and keep the number of edges in the MEG to a minimum. This removal requires the edges to be inserted to MEG_A to be not a back edge, i.e., adding the edge to MEG_A does not lead to any cycle.

The ordered CFL-reachability solver under the iterative-epoch framework (Iea-Ocr) is presented in Algorithm 5. Iea-Ocr extensively reuses the methods of the standard solver and Iea, including `SearchForward`, `SearchBackward` and `GenerateTransitiveEdges`. Compared with Iea, in each epoch of Iea-Ocr, the first stage includes the update of MEG (lines 5–6); the second stage (i.e., the cycle elimination stage) handles the cycles for both the main graph G and the MEG; and the third stage handles singly recursive productions (lines 13 and 17) and other productions separately,

Algorithm 4: Updating MEG and the main graph.

```
Function InsertMEGEdge($v_i, v_j$)
    TravBackward($v_i, v_j$);
    $isaBackEdge = (v_j \xrightarrow{A} v_i \in E)$;    // a boolean value denoting whether $v_i \xrightarrow{A} v_j$ is a back edge;
    add $v_i \xrightarrow{A} v_j$ to $MEG_A$;
Procedure TravBackward($v_i, v_j$)
    if not $isaBackEdge$ then
        for each $v_i \xrightarrow{A} v_k \in MEG_A$ do
            if $v_k \in succ(A, v_j)$ then remove $v_i \xrightarrow{A} v_k$ from $MEG_A$;    // remove a redundant edge ;
    TravForward($v_i, v_j$);
    for each $v_{i'} \xrightarrow{A} v_i \in MEG_A$ do
        if $v_{i'} \notin pred(A, v_j)$ then TravBackward($v_{i'}, v_j$);    // search backward ;
Procedure TravForward($v_i, v_j$)
    Update($v_i, v_j$);    // line 16
    for each $v_j \xrightarrow{A} v_{j'} \in MEG_A$ do
        if $v_{j'} \notin succ(A, v_i)$ then TravForward($v_i, v_{j'}$);    // search forward ;
Procedure Update($v_i, v_j$)
    add $v_i \xrightarrow{A} v_j$ to $E$ and to $W_{NA}$;
```

Algorithm 5: IEA-OCR: Iterative-epoch ordered CFL-reachability solver.

```
Function IEA ($CFG, G$)
    init();    // line 11, Algorithm 1
    while $W \neq \emptyset$ do
        GenerateTransitiveEdges();    // line 14, Algorithm 2
        for each $v_i \xrightarrow{A} v_j \in W_{NA}$ do
            InsertMEGEdge($v_i, v_j$);    // Algorithm 4
        CollapseTransitiveCycles();    // collapse cycles for both $G$ and MEG$_A$
        while $W_{NA} \neq \emptyset$ do
            select and remove an element $v_i \xrightarrow{Y} v_j$ from $W_{NA}$;
            for each non-$A$-generating production $X ::= Y \in P$ do
                if $v_i \xrightarrow{X} v_j \notin E$ then add $v_i \xrightarrow{X} v_j$ to $E$ and to $W$;
            for each non-$A$-generating production $X ::= Y\ Z \in P$ s.t. $\neg(X = Y = Z)$ do
                if $Z = A$ and $X = Y$ then
                    OrderedSearchForward($X, Z, v_i, v_j, W$);    // line 1, Algorithm 3
                else SearchForward($X, Z, v_i, v_j, W$);    // line 16, Algorithm 1 ;
            for each non-$A$-generating production $X ::= Z\ Y \in P$ do
                if $Z = A$ and $X = Y$ then
                    OrderedSearchBackward($X, Z, v_i, v_j, W$);    // line 6, Algorithm 3
                else SearchBackward($X, Z, v_i, v_j, W$);    // line 19, Algorithm 1 ;
```

where edge generations via singly recursive productions are kept in order using Algorithm 3, which is based on the MEG, while other productions are handled the same as the standard solver. Notably, the third stage does not need to handle doubly recursive productions (i.e., $A ::= A\ A$) because they are handled by `InsertMEGEdge` (Algorithm 4) in the first stage.

6 EXPERIMENT

We evaluate the performance of our online cycle elimination technique by comparing our iterative-epoch algorithms with existing techniques. Specifically, we compare the basic iterative-epoch solver Iea with the standard solver Std [Melski and Reps 2000], and the iterative-epoch ordered solver Iea-Ocr with the recent ordered solver Pocr [Lei et al. 2022]. Two sets of clients are studied. The first one is context-sensitive value flow analysis for C/C++ [Sui et al. 2014], and the second one is field-sensitive alias analysis [Sridharan et al. 2005] for Java. The grammars of both clients are widely used in evaluating the performance of CFL-reachability solvers [Lei et al. 2022; Wang et al. 2017]. Our evaluation focuses on the following three research questions:

RQ 1. What are the rates of edges and nodes reduced in the graph by online cycle elimination?
RQ 2. To what extent does online cycle elimination accelerate existing solvers?
RQ 3. To what extent does online cycle elimination reduce memory usage for existing solvers?

Our experiment shows the effectiveness of the iterative-epoch online cycle elimination technique in reducing redundant computations. On the one hand, by reducing 10.04% nodes and 49.23% edges in the solving process of value-flow analysis and 5.86% nodes and 43.67% edges in the solving process of alias analysis, Iea accelerates Std by 17.17× and 13.94×, respectively for value-flow analysis and alias analysis, with memory reductions of 48.8% and 45.0%. On the other hand, Iea-Ocr accelerates Pocr by 14.32× and 8.36×, respectively for value-flow analysis and alias analysis, with memory reductions of 55.2% and 57.8%.

6.1 Experimental Setup

The experimental platform utilized for conducting the experiments comprised an eight-core 2.60GHz Intel Xeon CPU with 128 GB memory, running Ubuntu 20.04.

Value-flow analysis. We perform context-sensitive value-flow analysis on sparse value-flow graphs (SVFGs) [Sui et al. 2014], which is built based on the result of Andersen's pointer analysis. Note that when constructing an SVFG, the analysis of mutable heaps is done in the preprocessing stage (i.e., the Andersen's analysis) in a conservative way. Thus, the value-flow analysis is sound as long as the Andersen's analysis is sound. The CFG for value-flow analysis is shown in Figure 9, where the start symbol of the CFG is A. The alphabet contains $call_i$ and ret_i to indicate a call and return, respectively, with a callsite index i; and a to indicate an assignment instruction; and A to indicate an interprocedural value flow. The normalized CFG is shown in Figure 9(b), with an explicit transitive symbol A.

Alias analysis. We perform field-sensitive alias analysis on program assignment graphs (PAGs) [Sridharan et al. 2005], and use the CFG shown in Figure 10. The start symbol of the CFG is *alias*. The alphabet contains terminals *new*, *assign*, *putField*[f] and *getField*[f] to denote allocation, assignment and field write and read. The normalized CFG is shown in Figure 9(b), where *flowTo* (also $\overline{flowTo}$) is the transitive symbol.

Benchmarks. For context-sensitive value-flow analysis, we use 10 C/C++ programs in the SPEC 2017 benchmark suite. For field-sensitive alias analysis, we use 10 Java programs from the DaCapo benchmark suite [Blackburn et al. 2006].

Setup and implementation. We implement our Iea, Iea-Ocr and the standard solver (Std) in C++. For the compared Pocr, we use the version provided in the artifact of the literature [Lei et al. 2022].

$$A ::= A\,A \mid call_i\ A\ ret_i \mid a \mid \varepsilon$$

(a) Context-free grammar.

$$\begin{aligned} A &::= A\,A \mid CA_i\ ret_i \mid a \mid \varepsilon \\ CA_i &::= CA_i\ A \mid call_i \end{aligned}$$

(b) Normalized grammar.

Fig. 9. CFG for context-sensitive value-flow analysis, A is the start symbol.

$$\begin{aligned} alias &::= \overline{flowTo}\ flowTo \\ flowTo &::= flowTo\ assign \mid new \\ &\quad \mid flowTo\ putField[f]\ alias\ getField[f] \\ \overline{flowTo} &::= \overline{assign}\ \overline{flowTo} \mid \overline{new} \\ &\quad \mid \overline{getField[f]}\ alias\ \overline{putField[f]}\ \overline{flowTo} \end{aligned}$$

(a) Context-free grammar. V is the start symbol.

$$\begin{aligned} alias &::= \overline{flowTo}\ alias \mid alias\ flowTo \mid \overline{new}\ new \\ flowTo &::= X\ getField[f] \mid assign \\ \overline{flowTo} &::= \overline{getField[f]}\ \overline{X} \mid \overline{assign} \\ X &::= putField[f]\ alias \\ \overline{X} &::= alias\ \overline{putField[f]} \end{aligned}$$

(b) Normalized grammar.

Fig. 10. CFG for Java field-sensitive alias analysis, V is the start symbol.

For the input graphs of our evaluation, the SVFGs of the SPEC 2017 C/C++ benchmarks are drawn from the bitcodes compiled using Clang-14.0.0 and linked by `wllvm` [1]. The PEGs of the DaCapo Java benchmarks are generated by converting the Java bytecode using Soot [Vallée-Rai et al. 1999]. The size and graph statistics of the benchmarks are listed in the columns from `Size/MB` to `#Edge` of Table 1 and Table 2.

Evaluated techniques. To study the real effectiveness of online cycle elimination in reducing the graph size, we perform offline graph simplification in the preprocessing stage and record the number of nodes and edges in each graph after offline graph simplification. In this way, the numbers of reduced nodes and edges by our online cycle elimination are exactly the ones that can only be reduced online. Our experiment includes two offline graph simplification techniques: offline cycle elimination (OSCC) and offline variable substitution (OVS) [Rountev and Chandra 2000]. Different from online cycle elimination that can reduce edges labeled by non-terminals, offline graph simplification techniques can only reduce edges labeled by terminals. In value-flow analysis, OSCC and OVS can be used to collapse a-cycles and a-edges, respectively. In alias analysis, OSCC and OVS can be used to collapse *assign*-cycles and *assign*-edges, respectively. To study the performance improvement of online solving by online cycle elimination, we compare Iea and Iea-Ocr with their original algorithms, i.e., Std (the standard algorithm) and Pocr.

6.2 RQ 1: Graph Simplification Performance

The columns `OSCC+OVS` of Tables 1 and 2 list the percentages of nodes and edges reduced by offline graph simplification (OSCC+OVS). With these data, we can calculate the numbers of nodes and edges of each graph at the beginning of online CFL-reachability solving. Table 3 presents the resulting data about the online cycle elimination of Iea and Iea-Ocr. According to the columns “#Epoch”, the iterative-epoch on average framework takes 9.7 epochs and 6.4 epochs to complete solving for value-flow analysis and alias analysis, respectively. Comparing the columns “CollapseTime/s” with “Runtime/s”, we see that online cycle elimination takes a rather small proportion of the overall running time, which is very efficient. On average, for value-flow analysis, online cycle elimination takes 9.22% and 11.92% of the running time of Iea and Iea-Ocr; for alias analysis, online cycle elimination takes 9.58% and 16.76% of the running time of Iea and Iea-Ocr.

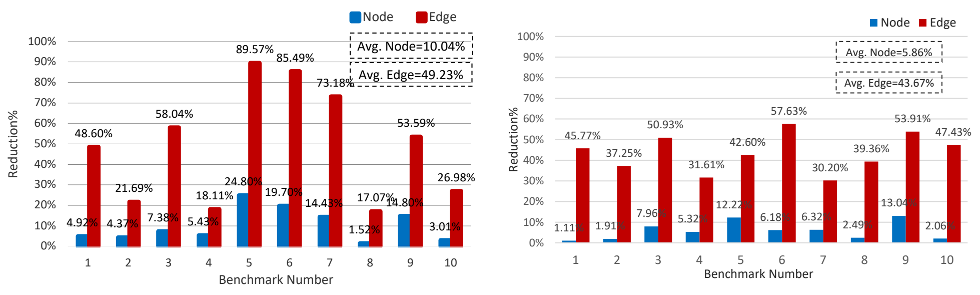
[1] https://github.com/travitch/whole-program-llvm.

Table 1. Benchmark info for value-flow analysis. #Node and #Edge refer to the total number of nodes and edges in the original graphs, respectively. Node% and Edge% denote the percentage of reduction in nodes and edges achieved through offline graph simplification techniques. OSCC stands for offline cycle elimination, while OVS represents offline variable substitution. MemoryUsage measures the amount of memory utilized during the solving process by two different methods, denoted as Std and Pocr, measured in gigabytes (GB).

Bench.	**Size/MB**	**SVFG**		**OSCC+OVS**		**Memory Usage/GB**		**Runtime/s**	
		#Node	**#Edge**	**Node%**	**Edge%**	**Std**	**Pocr**	**Std**	**Pocr**
1.xz	1.24	49395	62955	69.49%	63.46%	0.05	0.07	34.39	1.53
2.nab	1.41	55652	72366	72.30%	67.20%	3.33	4.34	225.94	52.35
3.leela	2.93	64466	89081	66.32%	54.64%	0.12	0.19	123.98	135.66
4.x264	4.68	207064	340217	67.92%	52.21%	4.77	6.46	17304.18	163.12
5.cactus	5.88	544480	1007989	59.04%	38.85%	-	40.28	-	639.87
6.povray	7.38	537775	1041687	60.37%	40.35%	-	55.87	-	2113.08
7.imagick	13.68	574089	842509	71.24%	62.12%	-	5.88	-	1510.90
8.parest	16.20	299718	407343	61.93%	50.95%	-	0.20	-	622.28
9.perlbench	18.69	697744	1662445	53.88%	32.46%	-	63.58	-	11054.37
10.omnetpp	21.81	664358	1857831	64.20%	31.26%	-	4.00	-	1480.80
Average				64.67%	49.35%	2.07	18.09	4422.13	249.09

Table 2. Benchmark info for alias analysis. The meanings of column headings are the same as Table 1

Bench.	**Size/MB**	**PAG**		**OSCC+OVS**		**Memory Usage/GB**		**Runtime/s**	
		#Node	**#Edge**	**Node%**	**Edge%**	**Std**	**Pocr**	**Std**	**Pocr**
1.avrora	1.71	80981	91532	31.89%	34.15%	0.23	0.57	54.11	21.54
2.biojava	3.54	96634	105421	29.15%	24.12%	0.36	0.45	89.14	87.45
3.h2o	4.13	21930	25012	23.12%	19.56%	0.56	3.58	189.15	121.86
4.batik	13.61	45752	85561	19.45%	26.56%	1.12	3.56	4508.01	815.21
5.fop	16.67	78991	101972	19.45%	24.56%	-	41.56	-	612.12
6.derby	19.94	78764	140235	31.56%	35.12%	-	32.15	-	754.12
7.jme	45.13	93861	130546	26.45%	29.14%	-	9.45	-	1105.88
8.cassandra	49.51	716674	95861	25.63%	29.14%	-	8.15	-	1305.24
9.pmd	56.85	628244	795621	35.21%	38.14%	-	30.89	-	1605.23
10.lucene	66.10	769161	562458	19.48%	23.89%	-	49.12	-	986.54
Average				26.19%	28.44%	0.57	17.99	1210.11	642.12



(a) Value-flow analysis. **(b)** Alias analysis.

Fig. 11. Reduction rates of nodes and summary edges by Iea on the graphs after offline simplification

Figure 11 shows the reduction rates of nodes and summary edges by online summary edges. Comparing Figure 11(a) with the columns OSCC+OVS of Table 1 and Figure 11(b) with the columns

Table 3. Results of Iea and Iea-Ocr. The left half is value-flow analysis and the right half is alias analysis. The columns "#Epoch" show the number of iterative-epoch in Iea. The columns "CollapseTime/s" shows the total time used for cycle elimination, measured in second. The columns "Runtime/s" shows the runtime of each approach, measured in second.

Bench.	**#Epoch**	**CollapseTime/s**		**Runtime/s**		**Bench.**	**#Epoch**	**CollapseTime/s**		**Runtime/s**	
		Iea	**Iea-Ocr**	**Iea**	**Iea-Ocr**			**Iea**	**Iea-Ocr**	**Iea**	**Iea-Ocr**
1.xz	10	0.15	0.09	1.59	0.27	1.avrora	6	0.15	0.13	2.98	1.65
2.nab	4	3.12	2.15	17.38	16.94	2.biojava	4	0.35	0.12	13.43	11.2
3.leela	8	1.56	0.07	6.48	5.42	3.h2o	5	0.12	0.89	16.83	15.28
4.x264	6	29.47	1.98	1159.02	22.96	4.batik	6	2.45	3.98	228.5	46.76
5.cactus	11	301.56	12.14	11607.02	29.77	5.fop	7	25.14	36.14	236.89	98.48
6.povray	12	38.95	2.87	5451.31	156.57	6.derby	7	90.12	29.48	229.69	245.72
7.imagick	13	245.25	2.56	2912.13	133.43	7.jme	6	61.32	84.15	916.03	148.88
8.parest	7	865.14	0.56	3510.11	37.32	8.cassandra	10	14.14	31.25	193.99	159.05
9.perlbench	13	178.17	54.12	27711.84	383.72	9.pmd	8	78.27	14.58	984.15	165.14
10.omnetpp	13	144.96	2.27	12156.35	140.76	10.lucene	7	251.1	39.47	1726.72	372.83
Average	9.7						6.4				

OSCC+OVS of Table 2, even after the offline graph simplification that has reduced a large proportion of nodes and edges in the initial graph, Iea is still able to further reduce a large number of summary edges generated during the online CFL-reachability solving process. On average, Iea reduces 10.04% nodes and 49.23% summary edges for value-flow analysis and reduces 5.86% and 43.67% for alias analysis. This result shows that collapsing a small number of cyclic nodes in the online solving process can significantly reduce the number of generated summary edges. This, in another way, implies that cycles created during the online solving can lead to numerous redundant edges generated and inserted, demonstrating the necessity of online cycle elimination.

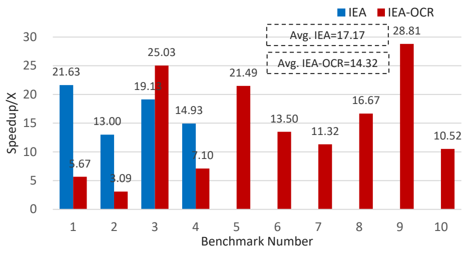
6.3 RQ 2: Speedups

Figure 12 displays the speedups of Iea over Std and Iea-Ocr over Pocr, where the blue bars denote the speedups of Iea and the red bars denote the speedups of Iea-Ocr. In general, both Std and Pocr can benefit from online cycle elimination, and our iterative-epoch framework can significantly accelerate the online CFL-reachability analysis.

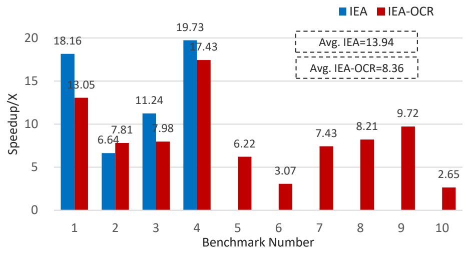
By observing the blue bars, Iea effectively accelerates Std by 17.17× for value-flow analysis and 13.94× for alias analysis. There are only four blue bars in both Figure 12(a) and Figure 12(b) because Std only successfully completes four of the ten benchmarks for both value-flow analysis and alias analysis. In contrast, according to the columns "Runtime/s" of Table 3, Iea successfully completes all the ten benchmarks.

By observing the red bars, both Figure 12(a) and Figure 12(b) have ten red bars, meaning that both Iea and Iea-Ocr successfully complete all the benchmarks. It is calculated that Iea-Ocr accelerates Pocr by 14.32× for value-flow analysis and 8.36× for alias analysis. Comparing the speedups of Iea and Iea-Ocr, we see that although Iea and Iea-Ocr reduce the same number of nodes and summary edges throughout the online solving, the speedup of Iea-Ocr (over Iea) is comparatively smaller. This can be attributed to the fact that Pocr already reduced a large number of edge generations through ordered solving, which also covers the redundancies caused by cycles.

By analyzing the outstanding benchmarks, programs with denser initial graphs tend to benefit more from online cycle eliminations with respect to speedup. Here, density denotes the quotient of the number of edges by the number of nodes in a graph. Let us see `leela` with `perlbench` in value-flow analysis and `biojava` with `batik` in alias analysis. According to the columns #Node

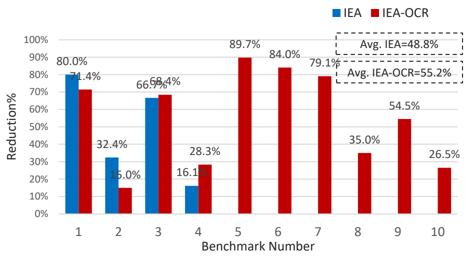


(a) Value-flow analysis.

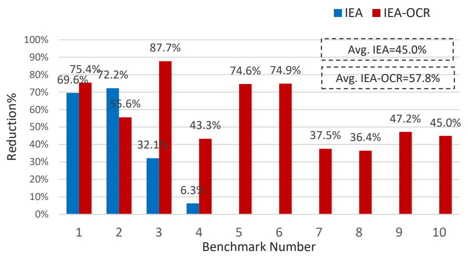


(b) Alias analysis.

Fig. 12. Speedups of IEA over STD and IEA-OCR over POCR.



(a) Value-flow analysis.



(b) Alias analysis.

Fig. 13. Memory usage reduction of IEA over STD and IEA-OCR over POCR.

and #Edge in Tables 1 and 2, the densities of the initial graphs of `perlbench` and `batik` are much larger than `leela` and `biojava`, respectively. Comparing Figures 11 and 12, although `perlbench` and `batik` have lower edge reduction rates than `leela` and `biojava`, their speedups are much larger. It is reasonable to speculate that CFL-reachability solving tends to spend more redundant computation in generating and inserting an edge when dealing with denser graphs. Thus, when applying cycle elimination, although denser graphs may have small edge reduction rates compared to sparser graphs, their speedups can still be larger.

6.4 RQ 3: Reductions of Memory Overhead

Figure 13 displays the reduction rates of memory overhead of IEA over STD and IEA-OCR over POCR. In general, iterative-epoch cycle elimination can effectively reduce the memory consumption of both STD and POCR for both clients. On average, IEA effectively reduces the memory overhead of STD by 48.8% and 45.0% for value-flow analysis and value-flow analysis, respectively. And IEA-OCR reduce the memory overhead of POCR by 55.2% and 57.8%, respectively, for value-flow analysis and value-flow analysis, which are even more significant. This can be attributed to the fact that the auxiliary data structure itself used in IEA-OCR can also save memory.

Comparing Figure 13 and Figure 11, the reduction rates of memory overhead (especially for POCR) are aligned with the reduction rates of summary edges. This is because most of the memory overhead in CFL-reachability solving is used to record summary edges. For larger benchmarks, i.e., benchmarks 6–10 for value-flow analysis, the reduction rate of memory usage is very close to the reduction rate of summary edge. And the largest recorded memory reduction rate 89.7% occurs in benchmark 5, which corresponds to the reduction rate of summary edge 89.6% in Figure 11(b).

This is because, in larger benchmarks, the proportions of memory usage taken by storing summary edges are larger, making reducing summary edges more effective to reduce memory overhead.

7 RELATED WORK

This section discusses related works.

7.1 Two Most Related Works

We first discuss the two existing works that are most related to our work.

IEA v.s. POCR. The recent work POCR [Lei et al. 2022] was proposed to reduce the redundant computations caused by "transitive relations" during the online solving process, using an ordered solving manner. The technique showed promising performance in clients where edges generated by doubly recursive productions dominate. The only similarity between IEA and POCR is that both of them capture the concept of transitivity to improve CFL-reachability solving. In fact, IEA and POCR work in essentially different aspects. First, the definitions of "transitive symbol" in IEA and "transitive relation" in POCR differ. In POCR, transitive relations need to be in the form of $A ::= A\ A$, whereas transitive symbol (Definition 4.3) do not have this restriction. Second, the overall mechanism of IEA is fundamentally different from that of POCR. Specifically, IEA improves the efficiency of CFL-reachability by dynamically eliminating cycles in the graph, whereas POCR enhances efficiency through ordered edge derivation. Third, POCR requires an auxiliary spanning tree data structure to facilitate ordered derivation, resulting in higher memory consumption compared to standard CFL-reachability. In contrast, IEA does not require any auxiliary data structure. Moreover, IEA can simplify not only the main graph of CFL-reachability (Section 4.2) but also the auxiliary data structure for ordered solving (Section 5.3). In other words, IEA is compatible with POCR, and they can be combined to achieve further better performance, as shown in our experiment.

IEA v.s. Wave propagation. We found that in constraint-based pointer analysis, there is a cycle elimination technique called Wave propagation [Pereira and Berlin 2009] that shares a similar idea to our iterative-epoch strategy. The similarity between IEA and Wave propagation is that they both divide the solving process into iterations and perform cycle detection and elimination in a certain spot of each iteration. It is interesting to point out that IEA performs online cycle elimination for CFL-reachability, while wave propagation was specially designed for constraint-based pointer analysis, which can be regarded as a client of CFL-reachability [Melski and Reps 2000]. From the perspective of methodology, for pointer analysis, there is only one kind of edges (i.e., *Copy*-edges) that can be added to the graph during solving, and it is easy to determine that cycles comprised of *Copy*-edges are collapsible [Hardekopf and Lin 2007a]. However, in CFL-reachability, there are multiple types of edges that will be added to the graph during solving. The main challenge is to determine the type of collapsible cycles. In this paper, we propose an approach to identify collapsible cycles based on the context-free grammar (Section 4.1).

7.2 Other Related Works

From a wider perspective, our approach is relevant to CFL-reachability performance improvement and graph simplification.

CFL-reachability performance improvement. Reducing the cubic time complexity of general CFL-reachability is difficult. To the best of our knowledge, the fastest algorithm [Chaudhuri 2008] treats CFL-reachability as an equivalent representation called recursive-state-machine reachability (RSM-reachability) and exhibits a slightly subcubic $O(n^3/\log^2 n)$ time complexity for bounded-stack RSMs, where n denotes the number of nodes of the input graph. So far, significant progress has only been made in handling special cases. One of the extensively studied relations in CFL-reachability is the Dyck relation [Chatterjee et al. 2018; Yuan and Eugster 2009; Zhang et al. 2013]. A recent work has reduced the complexity to nearly linear with respect to the input graph size [Chatterjee et al. 2018]. However, these techniques are only applicable to Dyck-reachability on bidirected graphs.

Researchers also studied the optimizing CFL-reachability from the perspective of its alternative forms, such as recursive state machines (RSMs) [Alur et al. 2005; Chaudhuri 2008] and pushdown automata (PDA) [Gauwin et al. 2019; Heizmann et al. 2017]. Different from graph simplification, these works tried to scale CFL-reachability by simplifying the automata (i.e., the context-free grammar) rather than the input graphs. For example, researchers have tried to model a program into a large RSM so that the value-flow analysis can be performed as an ordinary graph reachability problem in the RSM [Alur et al. 2005; Chaudhuri 2008]. In this way, a query in a demand-driven analysis can be done in linear time. However, this is not the typical CFL-reachability analysis, so it is not compared in this paper.

Reducing edge redundancy on the fly has also been incorporated into many existing techniques. Fähndrich et al. [1998] and Su et al. [2000] arbitrarily ordered the computations based on node indices in order to reduce repeated computations. Except for Pocr that is compared in our paper, another recent technique Graspan [Wang et al. 2017] use auxiliary structures to classify "new" and "old" edges to avoid the need to recompute the established edges.

Graph simplification. Due to the difficulty in reducing time complexity, many existing approaches improve the performance of CFL-reachability from more practical perspectives. In existing works, offline cycle elimination and edge contraction, which were not originally designed for CFL-reachability, have been adopted to improve the performance of CFL-reachability for particular clients [Lei et al. 2022; Wang et al. 2017]. For CFL-reachability, a recent offline graph simplification technique [Li et al. 2020] was proposed to remove the non-Dyck-contributing edges for interleaved-Dyck reachability problems. Notably, all the aforementioned techniques are offline techniques, whereas our approach focuses on online graph simplification.

From the perspective of cycle elimination [Nuutila and Soisalon-Soininen 1994; Tarjan 1972], it is a mature technique to simplify the graphs for transitive closure problems and is extensively adopted in constraint-based pointer analysis [Andersen 1994; Fähndrich et al. 1998; Hardekopf and Lin 2007a; Pearce et al. 2007; Pereira and Berlin 2009]. Besides wave propagation, there are two famous cycle elimination techniques for constraint-based point-to analysis. The study of cycle elimination in Andersen's pointer analysis can be dated back to the 1990's when a partial online cycle elimination [Fähndrich et al. 1998] was proposed. The algorithm sorts nodes with indices and performs cycle elimination when an edge from a higher-indexed node to a lower-indexed node is detected. Lazy cycle detection (LCD) and hybrid cycle detection (HCD) [Hardekopf and Lin 2007a] are two popular methods, among which LCD collapses cycles online based on the insight that nodes in cycles have the same points-to-set, while HCD detects and collapses cycles in the preprocessing stage and can be used as a reference to collapse cycles in solving.

8 CONCLUSION

The paper introduces an iterative-epoch online cycle elimination framework to improve the dynamic CFL-reachability solving, and applies the framework to the standard CFL-reachability solver and the recent ordered solver, yielding to new solvers Iea and Iea-Ocr. Experimental results on context-sensitive value-flow analysis for C/C++ and field-sensitive alias analysis for Java demonstrate the promising performance of our online cycle elimination technique. Specifically, Iea significantly reduces 10.04% nodes and 49.23% edges in the solving process of value-flow analysis and 5.86% nodes and 43.67% edges in the solving process of alias analysis, and accelerates the standard solver by 17.17× and 13.94×, respectively, for value-flow analysis and alias analysis, with memory reductions of 48.8% and 45.0%. Besides, IEA-OCR accelerates POCR by 14.32× and 8.36×, respectively, for value-flow analysis and alias analysis, with memory reductions of 55.2% and 57.8%. Such results show the importance of online cycle elimination in CFL-reachability. In addition to determining transitive symbols for collapsible cycles offline (Section 4.1), our study also reveals the possibility of identifying transitive patterns online, representing an interesting future topic.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their reviews and suggestions. This research is supported by Australian Research Grants DP210102409, DP240103194 and FT220100391, and by a generous Aspire Gift Grant from Google.

REFERENCES

Rajeev Alur, Michael Benedikt, Kousha Etessami, Patrice Godefroid, Thomas Reps, and Mihalis Yannakakis. 2005. Analysis of recursive state machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 27, 4 (2005), 786–818. https://doi.org/10.1007/3-540-44585-4_18

Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph. D. Dissertation. Citeseer. https://www.cs.cornell.edu/courses/cs711/2005fa/papers/andersen-thesis94.pdf

Stephen M Blackburn, Robin Garner, Chris Hoffman, Asjad M Khan, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications.* ACM, 169–190. https://doi.org/10.1145/1167515.1167488

Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. 2018. Optimal Dyck reachability for data-dependence and alias analysis. *Proc. ACM Program. Lang.* 2, POPL (2018), 30:1–30:30. https://doi.org/10.48550/arXiv.1910.00241

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. https://doi.org/10.1145/1328897.1328460

Manuel Fähndrich, Jeffrey S Foster, Zhendong Su, and Alexander Aiken. 1998. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation.* ACM, EECS Department University of California, Berkeley, 85–96. https://doi.org/10.1145/277652.277667

Olivier Gauwin, Anca Muscholl, and Michael Raskin. 2019. Minimization of visibly pushdown automata is NP-complete. *arXiv preprint arXiv:1907.09563* (2019). https://doi.org/10.48550/arXiv.1907.09563

Ben Hardekopf and Calvin Lin. 2007a. 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.* ACM, University of Texas at Austin, 290–299. https://doi.org/10.1145/1273442.1250767

Ben Hardekopf and Calvin Lin. 2007b. Exploiting pointer and location equivalence to optimize pointer analysis. In *Static Analysis: 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007. Proceedings 14.* Springer, Springer Berlin Heidelberg, Berlin Heidelberg 2007, 265–280. https://doi.org/10.1007/978-3-540-74061-2_17

Matthias Heizmann, Christian Schilling, and Daniel Tischner. 2017. Minimization of visibly pushdown automata using partial Max-SAT. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 461–478. https://doi.org/10.1007/978-3-662-54577-5_27

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

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

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

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.* ACM, Georgia Institute of Technology, 780–793. https://doi.org/10.1145/3492428

David Melski and Thomas Reps. 2000. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science* 248, 1-2 (2000), 29–98. https://doi.org/10.1145/258994.259006

Esko Nuutila and Eljas Soisalon-Soininen. 1994. On finding the strongly connected components in a directed graph. *Inform. Process. Lett.* 49, 1 (1994), 9–14. https://doi.org/10.1016/0020-0190(94)90047-7

David J Pearce, Paul HJ Kelly, and Chris Hankin. 2003. Online cycle detection and difference propagation for pointer analysis. In *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE, 3–12. https://doi.org/10.1023/B:SQJO.0000039791.93071.a2

David J Pearce, Paul HJ Kelly, and Chris Hankin. 2007. Efficient field-sensitive pointer analysis of C. *TOPLAS* 30, 1 (2007), 4. https://doi.org/10.1145/1290520.1290524

Fernando Magno Quintao Pereira and Daniel Berlin. 2009. Wave propagation and deep propagation for pointer analysis. In *2009 International Symposium on Code Generation and Optimization*. IEEE, IEEE, UCLA, 126–135. https://doi.org/10.1109/CGO.2009.9

Jakob Rehof and Manuel Fähndrich. 2001. Type-base flow analysis: from polymorphic subtyping to CFL-reachability. *ACM SIGPLAN Notices* 36, 3 (2001), 54–66. https://doi.org/10.1145/373243.360208

Thomas Reps. 1995. Shape analysis as a generalized path problem. In *Proceedings of the 1995 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. ACM, University of Wisconsin, 1–11. https://doi.org/10.1145/215465.215466

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*. ACM, University of Wisconsin, 49–61. https://doi.org/10.1145/199448.199462

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

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. https://doi.org/10.1145/1103845.1094817

Zhendong Su, Manuel Fähndrich, and Alexander Aiken. 2000. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 81–95. https://doi.org/10.1145/325694.325706

Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM. https://github.com/SVF-tools/SVF. In *CC '16*. 265–266. https://doi.org/10.1145/2892208.2892235

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

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

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press. https://dl.acm.org/doi/10.5555/781995.782008

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. https://doi.org/10.1145/3093336.3037744

Mihalis Yannakakis. 1990. Graph-Theoretic Methods in Database Theory. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM, AT&T Bell Laboratories, 230–242. https://doi.org/10.1145/298514.298576

Hao Yuan and Patrick Eugster. 2009. An efficient algorithm for solving the dyck-cfl reachability problem on trees. In *European Symposium on Programming*. Springer, 175–189. https://doi.org/10.1007/978-3-642-00590-9_13

Qirun Zhang, Michael R Lyu, Hao Yuan, and Zhendong Su. 2013. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 435–446. https://doi.org/10.1145/2491956.2462159

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*. ACM, Computer Science Department Cornell University, 197–208. https://doi.org/10.1145/1328897.1328464

Received 21-OCT-2023; accepted 2024-02-24