Scaling CFL-Reachability-Based Alias Analysis: Theory and Practice

ZHANG, Qirun

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Doctor of Philosophy
in
Computer Science and Engineering

The Chinese University of Hong Kong

September 2013

Thesis/Assessment Committee

Professor Yufei Tao (Chair)

Professor Michael Rung-Tsong Lyu (Thesis Supervisor)

Professor Lap Chi Lau (Committee Member)

Professor Zhendong Su (External Examiner)

Abstract of thesis entitled:

Scaling CFL-Reachability-Based Alias Analysis: Theory and Practice

Submitted by ZHANG, Qirun

for the degree of Doctor of Philosophy

at The Chinese University of Hong Kong in September 2013

Alias analysis is a fundamental static analysis problem which aims at determining if two pointer variables can refer to the same memory location. Alias analysis is usually a prerequisite for many static analyses. The precision of alias analysis leaves a great impact on many subsequent static analyses. Alias analysis can be formulated as a context-free language (CFL) reachability problem on edge-labeled bidirected graphs.

Solving *all-pairs* CFL-reachability is expensive. For a graph with n nodes and m edges, the traditional dynamic programming style algorithm exhibits an $O(n^3/\log n)$ time complexity. It is also well-known that CFL-reachability-based alias analysis solving *all-pairs* CFL-reachability does not scale in practice.

This thesis makes both theoretical and practical contributions

on scaling CFL-reachability-based alias analysis.

On the theoretical end, we present several fast algorithms for solving *all-pairs* CFL-reachability for alias analysis. In particular, for alias analysis for Java, we present a new Dyck-CFL-reachability algorithm with $O(n + m \log m)$ time complexity. When the input graph is restricted to a tree, we present an $O(n)$ time Dyck-CFL-reachability algorithm. For alias analysis for C, we present an efficient algorithm with $O(n(m + M))$ time complexity, where M denotes the number of memory alias edges which is very sparse with $M = O(n)$ in practice. Moreover, if the pointer usage in C is restricted to be well-typed, we present an $O(n(m+\tilde{M}))$ time alias analysis algorithm, where $\tilde{M}$ denotes the maximum memory alias edges on one layer.

On the practical end, we present the implementation of our algorithms and conduct extensive experiments on real-world applications. In practice, our CFL-reachability-based alias analysis scales extremely well. The performance compared to the state-of-the-art alias analyses for both Java and C indicates that our algorithms achieve orders of magnitude speedup and consume less memory. In particular, our CFL-reachability-based alias analysis for C can analyze the *latest* Linux kernel in under 80 seconds.

論文題目 : 擴展基於 CFL-Reachability 的別名分析

作者 : 章启潤

學校 : 香港中文大學

學系 : 計算機科學與工程學系

修讀學位 : 哲學博士

摘要 :

別名分析是一個傳統的靜態分析問題。別名分析對很多實際的靜態分析工具而言是不可或缺的，其精度也極大的影響到靜態分析工具的準確性。別名分析可以被表示成一個圖上的上下文無關文法可到達性（CFL-reachability）問題。

解決圖上的所有點之間的 CFL-reachability 問題的代價非常高。對於一個有 n 個點和 m 條邊的圖來說，傳統的動態規劃算法需要$O(n^3/\log n)$的時間複雜度。此外，大家廣泛熟知的，基於解決所有點間 CFL-reachability 的別名分析算法並不能應用到實際的程序分析中去。

這篇論文從理論和實際兩方面擴展優化了基於CFL-reachability的別名分析。

在理論方面，我們提出了快速的算法來解決別名分析中的所有點間的CFL-reachability問題。比如，對於Java上的別名分析，我們提出了一個 $O(n + m \log m)$ 時間複雜度的Dyck-CFL-reachability 算法。當輸入的圖是樹的時候，我們提出了一個 $O(n)$ 時間的算法。對於C上的別名分析，我們提出了一個有效的 $O(n(m + M))$ 時間複雜度的算法， 其中的內存別名 M 在實際中非常的稀疏（$M = O(n)$）。其次，當C中的指針使用具有一致類型的限制時，我們提出了一個 $O(n(m + \tilde{M}))$ 時間的別名分析算法。

在實際方面，我們實現了論文中的算法並且在很多實際程序上做了實驗。我們發現我們的算法在實際中的性能卓越。在與最新的Java和C的別名分析算法的比較中，我們的算法能夠取得上千倍的性能提高並且消耗更少的內存。特別的，我們的基於CFL-reachability的C的別名分析可以在80秒內處理完最新的Linux內核。

Acknowledgement

I would like to express my sincerest gratitude to many people who assisted me in the research presented in this thesis. First of all, I would like to thank my advisor Michael Lyu, for his guidance during my study at CUHK. His support and encouragement always showed me the light when I was low and in need of help. Moreover, I appreciate the invaluable help and support from Hao Yuan. His technical insight and advice led to many exciting results. Most importantly, this thesis would not have been possible without the inspiration of his early Dyck-CFL-reachability result on bidirected trees. Furthermore, I count myself fortunate to work with Zhendong Su at UC Davis, who has been more than a mentor to me. His great vision and enthusiasm not only prompted many stimulating discussions but also enlightened me of exploring new ideas in programming language research.

My four years studying at CUHK has been a memorable experience of me. I enjoyed my staying with the teachers and stu-

dents here. Among many others, I would like to thank Lap Chi Lau and his students in all my endeavors, for the discussions on graph reachability problems. The most gratitude goes to Wu-jie Zheng, who has been helping me to establish both serious scientific attitude and optimistic personality. Over the years, I was very happy to share office with a few talented guys, Yu Kang, Chao Zhou, Guang Ling, Chen Cheng, Shouyuan Chen, Hongyi Zhang. Life would not have been nearly as enjoyable without these fantastic officemates. When I was staying at UC Davis, I also thank Linfeng Liu, Zhongxian Gu, Dara Hazeghi, Vu le, Martin Velez, Ke Wang for many constructive discussions on both life and research. In particular, my special appreciation extends to Liang Xu and Fangqi Sun. Words can't express how much I appreciate your kindness and hospitality.

Last, but not least, I am deeply indebted to my parents and grandparents, for their unwavering love of me, for helping me get where I am today, and for their tireless support all through my life. And of course, my heartfelt thanks and love must go to Yin Lu, my wife, for her words and acts of encouragement during my Ph.D. study. I consider myself the luckiest in the world to have her in my life, who offers me both unconditional love and tremendous support.

To my family

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software defects cause severe economic loss. According to a well-publicized study by the National Institute of Standards and Technology (NIST), software defects cost the U.S. economy an estimated $59.5 billion annually [83]. A more recent study by Cambridge University reports that the global economic cost of software defects has risen to $312 billion annually [1]. A common rule of thumb in software defect reduction is to detect and eliminate defects early when it is cheaper to do so [15].

Static analysis is the process of assessing source code without executing the program, which can be applied at early stages before release [22]. Research has demonstrated that static analysis tools can significantly reduce software defects and about 60% of software faults could have been detected using static analysis tools [87]. However, precise static analysis is quite difficult [49].

From a programming language prospective, the main source of difficulties is due to the use of *indirection* [36]. The *indirection* supported in various programming languages provides flexibility to developers but poses significant difficulties to static analysis tools. In particular, pointer dereferences in C and object references in Java introduce *data-flow indirection*. Function pointers in C, virtual method dispatch in Java and closures in functional languages such as haskell introduce *control-flow indirection*.

Alias analysis is one kind of memory disambiguation methods, which aims at resolving the *indirection* [33]. It takes a computer program as input and computes the pairs of pointer variables that may refer to the same memory locations. Therefore, the *indirected* access to a memory location can be resolved by checking the alias pairs *w.r.t.* the pointer variable that represents the memory location. Alias analysis plays a pivotal role in static analysis tools [33, 75], since alias information is usually a prerequisite for most subsequent analyses. In the literature, alias analysis has been formulated as a context-free language (CFL) reachability problem on directed graphs [67, 80, 92, 97].

Unfortunately, traditional CFL-reachability-based alias analysis does not scale well in practice. For instance, traditional CFL-reachability-based alias analysis takes more than 8 hours to

compute the *all-pairs* alias information on the current[1] release of Linux kernel, which is certainly far too time consuming. On the other hand, modern software often undergoes significant changes in the life cycle [53]. The code bases of modern software are getting larger, with observed exponential growth patterns [28]. As a result, the scalability of CFL-reachability-based alias analysis draws much attention.

The main focus of this thesis is to develop new techniques to scale CFL-reachability-based alias analysis for both C and Java programs. From a theoretical perspective, we propose a set of algorithms with improved time complexities for solving the *all-pairs* CFL-reachability problem for alias analysis. From a practical perspective, we contributes several strategies to boost the CFL-reachability implementations of alias analysis. Together, these techniques could achieve orders of magnitude speedup and consume less memory.

1.1 Alias Analysis

An alias occurs at some program point when two variables refer to the same memory location. Given two pointers `p` and `q`, alias analysis determines if they might point to the same memory loca-

[1] As of March 2013.

tions during program execution. Alias analysis is a fundamental static analysis problem [40]. The aliasing information obtained by an alias analysis is a prerequisite for many compiler optimizations and static analysis tools (*e.g.*, software verifiers [27], data race detectors [63], and static slicers [42]).

1.1.1 Alias Analysis via CFL-Reachability

The context-free language (CFL) reachability problem is a generalization of the traditional graph reachability problem [67, 93]. Many program analyses have been formulated as CFL-reachability problems, such as interprocedural data flow analysis [70], program slicing [69], shape analysis [66], type-based flow analysis [62, 65], and pointer analysis [54, 78, 80, 90, 92, 97]. Two central ingredients of CFL-reachability formulation are a context-free grammar which depicts the idea to solve the client problem, and an edge-labeled directed graph which abstracts the information from the input problem. The solution to the given problem is represented as the CFL-reachability between nodes in the edge-labeled directed graph.

Many state-of-the-art alias analyses [54, 78, 80, 90, 92, 97] are formulated using CFL-reachability. In particular, alias analysis for Java has been formulated as a bidirected Dyck-CFL-

```
char *a, *b;
char c[10], d[11];

...
strcpy(a, b);
```

Figure 1.1: An example to illustrate buffer overflow.

reachability problem on Symbolic Points-to Graph (SPG) [90, 92]. In the SPG, the nodes represent symbolic memory locations of objects, and the edges with labels represent different field accesses of an object. On the other hand, alias analysis for C has also been formulated as a CFL-reachability problem on Pointer Expression Graph (PEG) [97]. In the PEG, the nodes represent pointer variables, and the labeled edges represent the pointer dereferences and assignments. Two pointer variables are aliases if and only if there is a path joining the two corresponding nodes in the graph, and the string concatenating edge labels from the path can be derived from the start symbol from the given CFG.

1.1.2 Client Applications

The alias information is also quite useful for subsequent analysis. We proceed to give two client applications of alias analysis.

Bug Detection

Static analysis tools could be applied to detect potential bugs in programs. Let us consider buffer overflow in C programs as an example. Buffer overflow occurs when the data to write is larger than the buffer size. Buffer overflow vulnerability has been a severe threat. According to the ICS-CERT Advisories [3] in 2013, more than 16% vulnerabilities are primarily associated with buffer overflow[2].

Due to *indirection* in C language, a buffer can be accessed via pointers. A static analysis tool may first consult pointer aliasing information before going further to detect out-of-bound buffer accesses. We give an illustrative example to introduce the scenario. Let us consider the C code snippet in Figure 1.1. The last line of the code snippet uses the unsafe `strcpy()` function to copy the buffer pointed by `b` to the buffer pointed by `a`. If at this point, `b` points `d[11]` and `a` points to `c[10]`, a buffer overflow vulnerability occurs. Without prior alias information of `a` and `b`, it is not possible for a static analysis tool to detect the bug.

```
1:  int a, b, *c;
2:  a = 1;
3:  c = &a;
4:  *c = 2;
5:  b = a;
```

Figure 1.2: An example to illustrate constant propagation.

Compiler Optimization

Constant propagation is a classical compiler optimization [6]. The goal of constant propagation is to collect the variables that are constant at each program point and propagate the values through the program paths.

Let us consider the code snippet in Figure 1.2. Due to the assignment at line 3, `*c` and `a` are aliased. Without alias information, the value of `a` is 1 at line 5. As a result, variable `b` gets the constant 1. With the alias information, we know variable `a` is *indirectly* accessed through `*c` at line 4. Consequently, variable `b` gets the correct constant.

1.2 Scalability Challenges

The challenges of scaling CFL-reachability-based alias analysis lie in both theory and practice.

[2] As of April 2013.

1.2.1 Improving Asymptotic Complexities

Traditional *all-pairs* CFL-reachability algorithm exhibits an $O(n^3)$ worst-case time complexity [67, 93], which is commonly known as "the cubic bottleneck" in flow analysis [37]. Applied on alias analysis for Java, the concerned CFL is restricted to a subclass called Dyck language that generates properly matched parentheses. For Dyck language of size k (*i.e.*, k kinds of parentheses), traditional Dyck-CFL-reachability algorithm runs in $O(k^3n^3)$.

Improving the cubic time complexity is very hard. When the input graph is an chain, the CFL-reachability problem can be thought of as a CFL parsing problem, for which the best time complexity is Boolean matrix multiplication time [52]. As a result, any breakthrough in CFL-reachability may lead to faster algorithms for CFL parsing [67]. Various enhancements have been proposed to improve the cubic complexity. For instance, Kodumal and Aiken described a specialized set constraint reduction for Dyck-CFL-reachability on graphs and obtained an $O(kn^3)$ Dyck-CFL-reachability algorithm. Yuan and Eugster [94] proposed an efficient Dyck-CFL-reachability algorithm in $O(n \log n \log k)$ time on bidirected trees. Only recently, Chaudhuri showed that the well-known Four Russians' Trick [11] could be employed to speed up in the CFL-reachability algorithm to immediately yield

an $O(n^3/\log n)$ algorithm [20]. Similar techniques were used in Rytter's work [73] for CFL parsing.

1.2.2 Boosting Practical Implementations

A worst-case cubic algorithm does not scale well on real-world applications. The best example would be the original Andersen's pointer analysis algorithm [10]. It took years' effort to make Andersen's pointer analysis [32, 35, 38, 60, 71, 82] to scale. Similarly, straightforward implementations of *all-pairs* are ill-suited for handling large-scale applications in practice. Thus far, the key to scale the CFL-reachability-based alias analysis is to make the analysis demand-driven, aiming at solving the *single-source-single-sink* CFL-reachability problem [80, 92, 97]. On the other hand, the practical benefits of the subcubic CFL-reachability algorithm [20] remains unclear, since there is no subcubic implementation for alias analysis to date.

1.3 Thesis Contributions

This thesis makes both theoretical and practical contributions to scale CFL-reachability-based alias analysis. In particular, Tables 1.1 and 1.2 summarize our main algorithmic results for analysis on Java and C respectively. The principal contributions of

Bidirectness	**Input**	**Time**	**Space**	**Reference**
Bidirected	Tree	$O(n \log n \log k)$	$O(n \log n)$	[94]
General	Graph	$O(k^3n^3)$	$O(kn^2)$	[70, 93]
General	Graph	$O(kn^3)$	$O(kn^2)$	[47]
General	Graph	$O(kn^3/\log n)$	$O(kn^2)$	[20]
Bidirected	Tree	$O(n)$	$O(n)$	Algorithm 4
Bidirected	Graph	$O(n + m \log m)$	$O(n + m)$	Algorithm 5
General	Graph	$O(n(m + S))$	$O(n^2)$	Algorithm 9

Table 1.1: Dyck-CFL-reachability algorithms for alias analysis for Java on SPG.

PEG type	**Time**	**Space**	**Reference**
General	$O(n^3)$	$O(n^2)$	[97]
General	$O(n^3/\log n)$	$O(n^2)$	[20]
Well-typed	$O(n(m + \tilde{M}))$	$O(n^2)$	Algorithm 15
General	$O(n(m + M))$	$O(n^2)$	Algorithm 11

Table 1.2: CFL-reachability algorithms for alias analysis for C on PEG.

this thesis are detailed as follows:

- For alias analysis for Java on bidirected trees, we give an algorithm that runs in $O(n)$ time and $O(n)$ space, which answers the Dyck-CFL-reachability queries for any node pair in $O(1)$ time. This result improves the $O(n \log n \log k)$ time and $O(n \log n)$ space algorithm proposed by Yuan and Eugster [94].

- For alias analysis for Java on bidirected graph with n nodes and m edges, we give an algorithm that runs in $O(n + m \log m)$ time and $O(n+m)$ space, which answers the Dyck-

CFL-reachability queries for any node pair in $O(1)$ time. This result improves the traditional subcubic time result in the literature [20].

- For alias analysis for C on the Pointer Expression Graph (PEG), we give an *all-pairs* CFL-reachability algorithm that runs in $O(n(m + M))$ time and $O(n^2)$ space, where M denotes the memory alias pairs in the final graph. Furthermore, if the given PEG is well-typed, we also give an *all-pairs* CFL-reachability algorithm that runs in $O(n(m+\tilde{M}))$ time with $O(n^2)$ space, where $\tilde{M}$ denotes the maximum memory alias pairs on one layer. The numbers of m and M are typically very sparse (*i.e.*, both m and M are observed to be $O(n)$), which implies our algorithms have a quadratic time complexity in practice.

- We apply our Dyck-CFL-reachability algorithms to a state-of-the-art context-insensitive alias analysis for Java [92]. Experimental results show that our algorithm achieves orders of magnitude speedup on real-world benchmarks. For C, we also present the implementation of the first subcubic CFL-reachability-based alias analysis, and conduct large-scale experiments on the *latest stable* releases of popular C programs used from the pointer analysis literature. The

performance of the subcubic alias analysis solving *all-pairs* CFL-reachability is extremely well in practice. In particular, our alias analysis algorithm is able to analyze the 10M SLOC Linux kernel in less than 80 seconds on commodity hardware.

1.4 Thesis Organization

This thesis is organized as follows. Chapter 2 gives explanations of various terminology relevant to alias analysis and CFL-reachability. Chapter 3 describes our fast Dyck-CFL-reachability algorithms. Chapter 4 applies our Dyck-CFL-reachability algorithms on a state-of-the-art alias analysis for Java and presents experimental results. Then, Chapter 5 describes our fast CFL-reachability algorithms for alias analysis for C. Chapter 6 presents the first subcubic CFL-reachability implementation and compares various CFL-reachability algorithms on *latest stable* releases of C programs. Finally, Chapter 7 briefly surveys related work, and Chapter 8 concludes.

□ **End of chapter.**

Chapter 2

Background

In this chapter, we present a brief introduction to CFL-reachability-based alias analysis. The aim of this chapter is to give the sufficient background materials to understand the main results in the forthcoming chapters. In particular, Section 2.1 introduces the definitions and the dimensions of pointer analysis. Section 2.2 discusses the two traditional approaches to alias analysis via points-to analysis. Section 2.3 presents the introduction to CFL-reachability. Finally, Section 2.4 briefly mentions two CFL-reachability formulations for alias analysis that our fast algorithms adopt in later chapters.

2.1 Pointer Analysis

Pointer analysis is a static analysis which aims to determine the pointer usage in imperative languages such as C and Java. In

```
int *a,*b;
int *c,*d,*e;
  a = malloc();
  b = a;
  c = &b;
  d = *c;
  *c = e;
```

Figure 2.1: An simple example illustrates the pointer usage in C.

general, pointer analysis is composed of points-to analysis and alias analysis. This thesis follows the work of Hind [40] to make the distinction between points-to analysis and alias analysis.

2.1.1 Points-to Analysis

In imperative languages, a pointer variable can be assigned with the memory location of other variables. For Example, in C language, a pointer variable of type `int*` can be used to denote the address of an integer variable. In Java, a variable can point to the heap location returned by `new` operator.

A points-to analysis typically computes a set of memory locations that each pointer variable may point to during the executing of a program. Let $loc(x)$ and $pt(x)$ be the memory location and points-to set of variable `x` respectively. We say pointer variable `x` *may* point to variable `y` iff $loc(y) \in pt(x)$.

For example in Figure 2.1, pointer `c` *may* points to `b` and pointer `a` *may* points to an abstract location on heap. In C

language, the memory location can be obtained via either the *address-of* operator (*e.g.*, `&x`) or dynamic memory allocation (*e.g.*, `malloc()`). A *dereference* expression `*x` obtains the *lvalue* [45] of variable `x`. In Java language, the memory location is typically obtained via the `new` operator. Moreover, the references to objects play the similar rule as the pointers in C, *e.g.*, calling the methods of a class such as `A.a()` can be thought of as dereferencing the class variable `A`.

2.1.2 Alias Analysis

In general, an alias occurs at some program point if two pointer variables point to the same memory location. The two pointer variables are called aliases. The pair of aliases can also be called aliasing pair, alias pair or, simply, aliasing.

A alias analysis typically computes all alias pairs during the executing of a program. In C language, aliasing may occur due to the pointer assignments (*e.g.*, `**a = *b`), array indexing (*e.g.*, `a[i]`) and union accesses (*e.g.*, `a.b`). In Java, aliasing may occur due to the field access (*e.g.*, `A.a`) and method invoking (*e.g.*, `A.a()`) of a particular class variable `A`.

For example in Figure 2.1, pointer variables `d` and `e` are aliases due to the assignments associated with `*c`. Note that the aliasing

relation is reflexive, *i.e.*, `a` aliases with `a`. It is also symmetric, *i.e.*, `a` aliases with `b` implies `b` aliases with `a`. However, it is not transitive [97].

2.1.3 Analysis Dimensions

In the pointer analysis literature, some dimensions are used to classify different pointer analyses. Many papers on pointer analysis use these dimensions to place their work, *e.g.*, the work of Ryder [72].

Flow-Sensitivity

Flow-sensitivity determines whether the analysis takes the control flow into consideration. The control flow of a program defines the execution order of program statements. Typically, it is depicted using a directed graph known as a control flow graph. The nodes in the graph represent either basic blocks or predicates, where predicates usually represent branching statements (*e.g.*, `if` statements) which split the control flow and basic blocks denote the set of program statements without predicates.

A flow-sensitive analysis respects the control flow of a program and computes separate solutions for each program point. However, a flow-insensitive analysis ignores the control flow and computes a single solution of the entire program or procedure.

```
1:    a = &x;
2:    a = &y;
```

Figure 2.2: An example illustrates the flow-sensitivity of pointer analysis.

Namely, a flow-insensitive analysis assumes the program statements can be executed in *any* order. The most general approach towards a flow-sensitive analysis is to solve a system of data-flow equiations *w.r.t.* the control flow graph [46, 58].

Example 1 *Let us consider the example in Figure 2.2. The elementary example contains two lines of code without branching statements. A flow-insensitive points-to analysis computes the solution* $pt(a) = \{loc(x), loc(y)\}$ *for the whole code snippet. A flow-sensitive points-to analysis computes* $pt(a) = \{loc(x)\}$ *at line 1 and* $pt(a) = \{loc(y)\}$ *at line 2. Note that the value of* `a` *is overwritten at line 2. A flow-sensitive analysis should be able to analyze it by performing* strong updates *of memory locations.*

Context-Sensitivity

Context-sensitivity determines whether the analysis takes the procedure invocation into consideration. A typical program could be composed of various procedures (*a.k.a.* functions or methods in different terminologies). A procedure can be possibly invoked by another procedure for multiple times on different call sites.

A context-sensitive analysis respects the calling context of a

```
1:  int* id(int* x){ return x; }
2:  int main(){
3:    int *a, *b,*c, *d;
4:    b = id(a);
5:    d = id(c);
6:    return 0;
7:  }
```

Figure 2.3: An example illustrates the context-sensitivity of pointer analysis.

procedure and computes separate solutions for each call site. Namely, a context-sensitive analysis keeps track of the calling context of a procedure while analyzing it. However, a context-insensitive analysis merges different calling contexts together.

Example 2 *Let us consider the example in Figure 2.3. The* `id()` *procedure simply returns a pointer variable according to its parameter. A flow-insensitive alias analysis merges two procedure calls at lines 4 and 5 and obtains four alias pairs:* $\langle b, a\rangle$, $\langle b, c\rangle$, $\langle d, a\rangle$ *and* $\langle d, c\rangle$. *In contrast, a context-sensitive alias analysis distinguishes the two call sites and obtains two alias pairs:* $\langle b, a\rangle$ *and* $\langle d, c\rangle$.

Field-Sensitivity

Field-sensitivity deals with how aggregates (*e.g.*, structures and objects) are handled by the analysis. Unlike the primitive variables, an aggregate can be accessed via different fields. For example, the `Data` object in JDK contains various fields such as `Day`,

```
1:  typedef struct{ int *f; int *g; } T;
2:  T a, b;
3:  int x, y, z;
4:  a.f = &x;
5:  a.g = &y;
6:  b.g = &z;
```

Figure 2.4: An example illustrates the field-sensitivity of pointer analysis.

`Month` and `Year`.

There are three classical treatments on field-sensitivity. A *field-insensitive* analysis discards the field information that models all field accesses as the accesses to a single representative variable. A *field-sensitive* analysis respects the field information and models each field access as a unique access to the aggregate. Finally, a *field-based* analysis [38] models each field access as the access to all aggregates that contain this field.

Example 3 *Let us consider the example in Figure 2.4. Aggregate* `T` *contains two fields. A field-insensitive points-to analysis computes two points-to sets:* $pt(a) = \{loc(x), loc(y)\}$ *and* $pt(b) = \{loc(z)\}$. *A field-based points-to analysis computes two points-to sets:* $pt(T.f) = \{loc(x)\}$ *and* $pt(T.g) = \{loc(y), loc(z)\}$. *Finally, a field-sensitive points-to analysis computes three points-to sets:* $pt(a.f) = \{loc(x)\}$, $pt(a.g) = \{loc(y)\}$ *and* $pt(b.g) = \{loc(z)\}$.

2.1.4 Computability and Complexity

Despite the different analysis dimensions, precise pointer analysis is a computationally hard problem. In this section, we briefly introduce some computability and complexity results.

To begin with, let us discard the impact of procedure calls (*i.e.*, be context-insensitive and intra-procedural) and consider the pointer analysis within a single procedure. When the dynamic memory allocation is allowed, the precise flow-sensitive pointer analysis with non-scale variables (*i.e.*, with structures and arrays) is undecidable [49, 64]. Specifically, Landi [49] first established the undecidable result by showing a reduction of the halting problem. Later, Ramalingam [64] provided a simpler proof by reducing the pointer analysis problem to the Post's Correspondence problem [61]. The undecidability result holds even if the programs are restricted with only scalar variables (*i.e.*, primitive variables) [17]. When dynamic memory allocation is disallowed, the configuration space of pointers are finite and the precise flow-sensitive pointer analysis becomes pspace-complete. When the programs are further restricted with at most two levels of dereferences (*e.g.*, `int **`), the problem is in **P**.

Then, we shift the discussion to the flow-insensitive analysis. When dynamic memory allocation is disallowed, the precise

flow-insensitive pointer analysis is **NP**-hard with arbitrary levels of dereferences [41]. The hardness result was established by showing a reduction of the Hamiltonian path problem. Furthermore, if the pointer variables are restricted to be scalars with well-defined types, the problem is in **P** [17].

These computability and complexity results reflect the general setting of pointer analysis problem. In practice, any pointer analysis must approximate the exact solution and explore the trade-off between precision and performance. Flow-sensitive analysis offers the precision, but is more expensive to compute compared with the flow-insensitive analysis. Moreover, it can be inferred that taking the context-sensitive and field-sensitive into consideration makes the analysis more precise but computationally more expensive.

This thesis focuses on flow- and context-insensitive pointer analysis. In the next section, we introduce two traditional flow-insensitive pointer analyses.

2.2 Traditional Approaches to Alias Analysis

Given two variables p and q, traditional approach to determine whether p and q are aliases is to perform a points-to analysis and check whether $pt(p) \cap pt(q)$ is non-empty. In this section, we de-

scribe two popular flow-insensitive points-to analysis approaches in the literature.

Before performing the actual analysis, traditional points-to analysis needs to pre-process the given program. All assignment statements manipulating pointer variables are normalized into a canonical form that consists of four kinds for statements:

$$\begin{array}{rcl} S & ::= & p \coloneqq \&q \\ & | & p \coloneqq q \\ & | & *p \coloneqq q \\ & | & p = *q \end{array}$$

Specially, multiple uses of dereferences are replaced with a sequence of statements by introducing new temporaries. For example, a statement like `**p = q` is normalized into `*p = temp` and `*temp = q`. Note that the normalization pass causes some precision loss, since it introduces additional points-to or alias pairs to the original program [13, 17, 41]. The key distinction between the two approaches lies in the way that they process these assignment statements.

Statement	Constraint	Name
`p = &q`	$loc(q) \in pt(p)$	[AddrOf]
`p = q`	$pt(p) = pt(q)$	[Copy]
`*p = q`	$\forall a \in pt(p).pt(a) = pt(q)$	[Store]
`p = *q`	$\forall a \in pt(q).pt(p) = pt(a)$	[Load]

Figure 2.5: Constraints for flow-insensitive unification-based points-to analysis.

2.2.1 Unification-based Analysis

Unification-based (*a.k.a.* Steensgaard-style) pointer analysis [81] unifies the memory locations and computes a single representative location for each normalized statements. Specifically, unification-based pointer analysis processes each normalized statement using equality constraints described in Figure 2.5. For example, given a Copy statement `p = q`, the analysis merges the two points-to sets $pt(p)$ and $pt(q)$, and outputs a single representative set.

The most appealing feature of unification-based analysis is that the algorithm can be efficiently implemented using a Union-Find data structure [24]. Namely, "merging two points-to sets" can be implemented using the `Union()` operation. Note that in the Union-Find algorithm, there is no need to enumerate each element in a points-to set when processing the Store and Load constraints. The unification-based algorithm runs very fast in practice since each statement only needs to be processed once. As a result, the time complexity of unification-based pointer

```
int *a,*b;
int *c,*d,*e;
  a = &b;
  a = &c;
  d = &e;
  d = a;
```

Figure 2.6: An example for unification-based and inclusion-based pointer analysis.

Statement	Constraint	Name
`p = &q`	$loc(q) \in pt(p)$	[AddrOf]
`p = q`	$pt(q) \subseteq pt(p)$	[Copy]
`*p = q`	$\forall a \in pt(p).pt(q) \subseteq pt(a)$	[Store]
`p = *q`	$\forall a \in pt(q).pt(a) \subseteq pt(p)$	[Load]

Figure 2.7: Constraints for flow-insensitive inclusion-based points-to analysis.

analysis is $O(n\,\alpha(n))$, where n denotes the number of variables and $\alpha(n)$ is the inverse Ackermann's function [24].

Example 4 *Figure 2.6 shows a C code snippet. The first two statements generate the constraint* $pt(a) = \{loc(b), loc(c)\}$. *Similarly, the third statement generates* $pt(d) = \{loc(e)\}$. *The last statement makes* $pt(d)$ *and* $pt(a)$ *identical by performing* `Union()` *operation on the corresponding points-to sets* , i.e., $pt(d) = pt(a) = \{loc(b), loc(c), loc(e)\}$.

2.2.2 Inclusion-based Analysis

Inclusion-based (*a.k.a.* Andersen-style) pointer analysis [10] generates subset constraints descried Figure 2.7 and propagates these constraints among assignment statements. Due to the inclusions,

new subset constraints may alter the current sets. Therefore, the actual points-to sets are obtained by computing the fixed-point solution of all subset constraints.

Inclusion-based pointer analysis offers a more precise solution than unification-based pointer analysis. However, it is more expensive to compute. For example, in order to compute the fixed-point on the [LOAD] statements, each element a in $pt(q)$ must be enumerated to generate new subset constraints. As we shall discuss in Section 2.3.2, the time complexity of inclusion-based pointer analysis is $O(n^3)$. Over the yeas, a lot of enhancements have been proposed to scale the inclusion-based pointer analysis.

Example 5 *Let us consider the example in Figure 2.6 again. Inclusion-based analysis generates the same constraints as unification-based analysis on the first three statements. However, for inclusion-based analysis, the last statement makes $pt(a)$ a subset of $pt(d)$. Therefore, $pt(d)$ is now $\{loc(b), loc(c), loc(e)\}$. Note that inclusion-based analysis is more precise in the sense that $loc(e)$ is not in $pt(a)$ in any execution order.*

2.3 CFL-Reachability

Context-free language (CFL) reachability [67, 93] is an extension to standard graph reachability. Let $CFG = (\Sigma, N, P, S)$ be a

context-free grammar with alphabet Σ, nonterminal symbols N, production rules P and start symbol S. Given a context-free grammar $CFG = (\Sigma, N, P, S)$ and a directed graph $G = (V, E)$ with each edge $(u, v) \in E$ labeled by a terminal $\mathcal{L}(u, v)$ from the alphabet Σ or ε, each path $p = v_0, v_1, v_2, \ldots, v_m$ in G *realizes* a string $R(p)$ over the alphabet by concatenating the edge labels in the path in order, *i.e.*, $R(p) = \mathcal{L}(v_0, v_1)\mathcal{L}(v_1, v_2) \ldots \mathcal{L}(v_{m-1}, v_m)$. Let X be a nonterminal, we define X-paths as follows:

Definition 1 (X-path) *A path $p = u, \ldots, v$ in G is an X-path if the* realized *string $R(p)$ can be derived from the nonterminal symbol $X \in N$, represented as a summary edge (u, X, v).*

The CFL-reachability problem is to determine if there is an S-path from node u to v in G, where S is the start symbol. In particular, the CFL-reachability problem has four variants:

(1) *The all-pairs S-path problem:* For every pair of nodes u and v, is there an S-path in G from u to v?

(2) *The single-source S-path problem:* Given a source node u, for all nodes v, is there an S-path in G from u to v?

(3) *The single-target S-path problem:* Given a target node v, for all nodes u, is there an S-path in G from u to v?

(4) *The single-source-single-sink problem:* Given two nodes u and v, is there an S-path in G from u to v?

2.3.1 Traditional CFL-Reachability Algorithm

In the literature, there is a popular dynamic-programming algorithm [70, 93] for solving the *all-pairs* CFL-reachability problem. It is described in Algorithm 1, where W denotes a worklist, (u, A, v) denotes the directed edge (u, v) with label $\mathcal{L}(u, v) = A$, and $\textsc{Out}(u, A)$ denotes the set of all outgoing A-edges of u, *i.e.*, $\textsc{Out}(u, A) = \{v \mid (u, A, v)\}$. The main algorithm has two steps: (1) *CFG Normalization* — The underlying CFG must be converted to a normal form, similar to the Chomsky Normal Form. When the grammar is in the normal form, all production rules are of the form $A \to BC$, $A \to B$ or $A \to \varepsilon$, where A is nonterminal, B and C are terminals or nonterminals, and ε denotes the empty string; and (2) *"Filling in" New Edges* — In order to compute the S-paths, new edges are added to the graph. For example, lines 11-14 describe that for the production rule $A \to BC$ and edge (i, B, j), all outgoing edges of node j are considered. If there is an outgoing edge (j, C, k), a new summary edge (i, A, k) is added to G if it is not in the current graph. The algorithm terminates if there are no more new edges to be added.

Algorithm 1: CFL-Reachability Algorithm.

```
Input  : Edge-labeled directed graph G = (V, E); normalized
         CFG = (Σ, N, P, S);
Output: the set of summary edges;
1  add E to W ;
2  foreach production A → ε ∈ P do
3      foreach node v ∈ V do
4          if (v, A, v) ∉ G then
5              insert (v, A, v) to G and to W ;
6  while W ≠ ∅ do
7      (i, B, j) ← SELECT-FROM(W) ;
8      foreach production A → B ∈ P do
9          if (i, A, j) ∉ G then
10             insert (i, A, j) to G and to W ;
11     foreach production A → BC ∈ P do
12         foreach k ∈ OUT(j, C) do
13             if (i, A, k) ∉ G then
14                 insert (i, A, k) to G and to W ;
15     foreach production A → CB ∈ P do
16         foreach k ∈ IN(i, C) do
17             if (k, A, j) ∉ G then
18                 insert (k, A, j) to G and to W ;
```

2.3.2 Complexity

Both of the inclusion-based pointer analysis and CFL-reachability problems have cubic time complexity in the worst case [35, 67, 79]. The inclusion-based pointer analysis works on a constraint graph where each node represents a pointer variable and each edge represents set inclusion. In the worst case, there are $O(n^2)$ inclusions in the graph. In essence, the inclusion-based analysis algorithm computes dynamic transitive closure, which immedi-

ately yields its $O(n^3)$ complexity.

The situations in CFL-reachability is similar. The running time of Algorithm 1 is dominated by line 12 and line 16. When each item is removed from the worklist, it takes time $O(n)$ to generate new items. In the worst case, there can be $O(n^2)$ items in the worklist. As as result, the overall algorithm takes time $O(n^3)$ in the worst case.

The worst case complexity of both problems is hard to improve. Only recently, Chaudhuri shows that the well-known Four Russians' Trick [11] can be employed at lines 12-13 and lines 16-17 in the CFL-reachability algorithm to yield a subcubic algorithm with an $O(n^3/\log n)$ time complexity [20]. When the concerned CFL is restricted to the Dyck language that generates matched pairs of parentheses, an algorithm with $O(n+m \log m)$ time complexity exists [95].

2.4 Alias Analysis via CFL-Reachability

Alias analysis for C and Java has been formulated as a CFL-reachability problem, with precision equivalent to an inclusion-based points-to analysis. The advantage of CFL-reachability-based alias analysis is that the alias information can be directly computed without first obtaining each variable's points-to set.

This thesis follows two CFL-reachability formulations for alias analysis. Specifically, alias analysis for Java has been formulated as a Dyck-CFL-rechability problem on symbolic points-to graph (SPG) [90, 92]. And alias analysis for C has been formulated as a CFL-reachability problem on pointer expression graph (PEG) [97].

In the following chapters, we propose a set of fast algorithms for scaling CFL-reachability-based alias analysis. Specifically, in Chapters 3 and 4, we present fast Dyck-CFL-reachability algorithms for alias analysis for Java. In Chapters 5 and 6, we present fast CFL-reachability algorithms for alias analysis for C. Moreover, Chapter 5 also offers a CFL-reachability-based pointer analysis algorithm on the well-typed C subset.

□ **End of chapter.**

Chapter 3

Fast Dyck-CFL-Reachability Algorithms

3.1 Introduction

When the underlying CFL is restricted to a Dyck language which generates matched parentheses, the CFL-reachability problem is referred to as Dyck-CFL-reachability. Although a restricted version of CFL-reachability, Dyck-CFL-reachability can express "almost all of the applications of CFL-reachability" in program analysis [47]. Specifically, alias analysis for Java has been formulated as a Dyck-CFL-Reachability problem on symbolic points-to graph [80, 90, 92].

Solving Dyck-CFL-reachability of size k (*i.e.*, k kinds of parentheses) is expensive in practice. The traditional dynamic programming style CFL-reachability algorithm [70, 93] runs in $O(k^3n^3)$

time. Only recently, the first subcubic algorithm was proposed, reducing the cubic time complexity by a factor of $\log n$ [20]. Scaling Dyck-CFL-reachability-based analyses on real-world applications is challenging. Various enhancements have been proposed, such as leveraging demand-driven properties in specific analyses [80, 92, 97], making use of a specialized reduction to set constraints [47], and approximating the client problems [78, 80]. However, all existing Dyck-CFL-reachability algorithms relying on the dynamic programming scheme exhibit a subcubic time complexity. When the underlying graphs are restricted to bidirected trees, Yuan and Eugster proposed an algorithm with $O(n \log n \log k)$ time complexity [94].

In this chapter, we focus on the Dyck-CFL-reachability problem, as detailed in Section 3.2.2. The bidirected Dyck-CFL-reachability is particularly suitable for pointer analysis. All state-of-the-art demand-driven pointer analyses [74, 78, 80, 92, 97] are formulated by extending Dyck-CFL-reachability and compute on edge-labeled bidirected graphs. Specifically, matched parentheses derived from Dyck-CFL-reachability can be used to capture field accesses (*i.e.*, `load`/`store`) in Java [78, 80, 90, 92] and indirections (*i.e.*, references/dereferences) in C [97]. The bidirectness of graphs is also a prerequisite for CFL-reachability for-

mulations of pointer analyses as discussed by Reps [67]. Namely, edges in the original graph need to be augmented with reverse edges (*a.k.a.* barred edges). Otherwise, two nodes may not be reachable even via standard graph reachability.

This chapter proposes three fast algorithms for solving the Dyck-CFL-reachability on trees and graphs respectively. The key insight behind our bidirected algorithms is the observation of an equivalence property on bidirected structures that has not been fully utilized in previous work. We exploit this property to obtain asymptotically much faster algorithms by safely collapsing nodes that belong to the same equivalence class. Moreover, the key insight behind our general Dyck-CFL-reachability algorithm is to dynamically maintain the transitive closure *w.r.t.* rule $S \to SS$, which improves the complexity by making it sensitive to the S-edges in the final graph.

The chapter is structured as follows. Section 3.2 reviews the background material on Dyck-CFL-reachability. Section 3.3 discusses the equivalence property and a naïve all-pairs Dyck-CFL-reachability algorithm. Sections 3.4 and 3.5 present our fast algorithms for bidirected Dyck-CFL-reachability on trees and graphs respectively. Finally, Section 3.6 presents the algorithm for solving general Dyck-CFL-reachability.

3.2 Preliminaries

This section reviews basic background on Dyck-CFL-reachability and defines its bidirected variants. We also include the traditional subcubic solution for reference and completeness.

3.2.1 Dyck-CFL-Reachability

The Dyck-CFL-reachability is defined similarly to CFL-reachability described in Section 2.3, by restricting the underlying CFL to a Dyck language, which generates strings of properly balanced parentheses. Consider an alphabet Σ over the set of opening parentheses $A = \{a_1, a_2, \ldots, a_k\}$ and the set of their matching closing parentheses $\bar{A} = \{\bar{a}_1, \bar{a}_2, \ldots, \bar{a}_k\}$. The Dyck language of size k (*i.e.*, k kinds of parentheses) is defined by the following context-free grammar:

$$S \rightarrow SS \mid a_1 S \bar{a}_1 \mid ... \mid a_k S \bar{a}_k \mid \varepsilon$$

where S is the start symbol and ε is the empty string. Specially, we say node v is *Dyck-reachable* from node u iff there exists an S-path from u to v, where S is the start symbol in the Dyck grammar above. We call such a path joining nodes u and v a *Dyck-path*.

3.2.2 Bidirected Dyck-CFL-Reachability

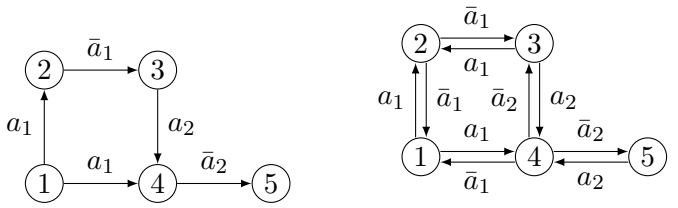
In Sections 3.4 and 3.5, we focus on the bidirected Dyck-CFL-reachability problems, which require the underlying graph to be bidirected and edge-labeled. For any directed edge (u, v) in the graph that is not labeled by ε, if it is labeled by an opening parenthesis $a_i \in A$, there must be a reverse edge (v, u) which is labeled by a matching closing parenthesis $\bar{a}_i \in \bar{A}$, and vice versa. Formally, we have the following definition.

Definition 2 (Bidirected Dyck-CFL-Reachability) *Given a bidirected graph $G = (V, E)$ and a Dyck language of size k, the labels of directed edges in the graph must satisfy the following constraints:*

- $\forall u, v \in V$, *if* $\mathcal{L}(u, v) = \varepsilon$, $\mathcal{L}(v, u)$ *must be* ε*;*
- $\forall u, v \in V$, *if* $\mathcal{L}(u, v) = a_i$, $\mathcal{L}(v, u)$ *must be* $\bar{a}_i$*;*
- $\forall u, v \in V$, *if* $\mathcal{L}(u, v) = \bar{a}_i$, $\mathcal{L}(v, u)$ *must be* a_i*.*

The bidirected Dyck-CFL-Reachability and its four variants are defined similarly as Dyck-CFL-Reachability.

The Dyck-CFL-reachable node pairs (u, v) can be defined as a binary relation $\mathbb{D}$.



(a) The directed graph case.

(b) The bidirected graph case.

Figure 3.1: Example graphs illustrating a directed graph and its corresponding bidirected graph.

Definition 3 (Dyck-CFL-Relation) *Given a bidirected graph $G = (V, E)$, we call a binary relation $\mathbb{D}$ on $V \times V$ a Dyck-CFL-relation iff for all $(u, v) \in \mathbb{D}$, v is* Dyck-reachable *from u in G.*

We give an example to illustrate the Dyck-CFL-reachability and the bidirected Dyck-CFL-reachability problems.

Example 6 *Consider the two graphs in Figure 3.1. The graph to the left shows a directed graph for Dyck-CFL-reachability, and the one to the right is its bidirected counterpart. In both graphs, the* realized *string $R(p)$ of the path $p = 1, 2, 3, 4, 5$ is "$a_1\bar{a}_1a_2\bar{a}_2$", with properly matched parentheses. Therefore, node 5 is* Dyck-reachable *from node 1. However, the path $1, 4, 5$ is not a valid* Dyck-path.

The bidirected Dyck-CFL-reachability formulation has wide applications in pointer analysis. For pointer analysis problems, the directed edges in the underlying graph must be augmented

with reverse edges (*a.k.a.* barred edges) [67], otherwise, two nodes may not be reachable from each other even by standard graph reachability. All existing CFL-reachability formulations for pointer analysis require the underlying graph to be bidirected. In addition, many pointer analyses employ Dyck-CFL-reachability to match certain properties, such as field accesses (*i.e.*, `load`/`store`) in Java [78, 80, 90, 92] and indirections (*i.e.*, references/dereferences) in C [97], which naturally satisfy the requirements of bidirected Dyck-CFL-reachability.

3.3 Dyck-CFL-Relation

3.3.1 An Equivalence Property

We first study an equivalence property of Dyck-CFL-relations $\mathbb{D}$ on bidirected trees and graphs, which has not been fully utilized in previous work. Since trees are simply graphs without cycles, we use the more general term "graph" to illustrate the equivalence property. A binary relation $\sim \subseteq B \times B$ on a set B is an equivalence relation iff it is reflexive, symmetric and transitive. Specifically,

- $\sim$ is reflexive if $\forall a \in B,\ a \sim a$;
- $\sim$ is symmetric if $\forall a, b \in B,\ a \sim b \implies b \sim a$; and

- $\sim$ is transitive if $\forall a, b, c \in B,\ a \sim b \wedge b \sim c \implies a \sim c$.

For a given bidirected graph $G = (V, E)$, we consider the Dyck-CFL-relation $\mathbb{D}$ over $V \times V$. Based on the definition of relation $\mathbb{D}$, node $v \in V$ is *Dyck-reachable* from node $u \in V$ iff $(u, v) \in \mathbb{D}$. We list below the properties of relation $\mathbb{D}$ on bidirected graphs:

- *Relation* $\mathbb{D}$ *is reflexive:* This is because the start symbol S in the Dyck grammar is nullable (*i.e.*, it generates the empty string ε). Therefore, $(u, u) \in \mathbb{D}$ for all $u \in V$.

- *Relation* $\mathbb{D}$ *is symmetric:* One can identify a symmetric relation by showing it is equal to its inverse. For the bidirected graphs, the *realized* string $R(p)$ on a path p from node u to v is the reverse of $R(p')$ on the reverse path p' from v to u. It is easy to show $R(p)$ is generated by the Dyck grammar iff $R(p')$ is generated by the Dyck grammar with a simple induction on the path length. As a result, if v is *Dyck-reachable* from u (*i.e.*, $(u, v) \in \mathbb{D}$), u is also *Dyck-reachable* from v (*i.e.*, $(v, u) \in \mathbb{D}$).

- *Relation* $\mathbb{D}$ *is transitive:* That is, for any three nodes $u, v, w \in V$ in graph $G = (V, E)$, if v is *Dyck-reachable* from u (*i.e.*, $(u, v) \in \mathbb{D}$) and w is *Dyck-reachable* from v (*i.e.*, $(v, w) \in \mathbb{D}$),

w is *Dyck-reachable* from u (*i.e.*, $(u, w) \in \mathbb{D}$). It is immediate that the *realized* string $R(p_1)$ for any path p_1 connecting node u and v can be derived from the start symbol S in the Dyck grammar. Similarly, the *realized* string $R(p_2)$ for any path p_2 connecting nodes v and w is also generated from the Dyck grammar. Consequently, the concatenated string $R(p_1)R(p_2)$ is generated by the Dyck grammar because of the rule $S \to SS$. Hence, the path p_1p_2 from node u to w is also a *Dyck-path*.

The discussions above lead to the following lemma.

Lemma 1 *The Dyck-CFL-relation* $\mathbb{D}$ *on a bidirected graph is an equivalence relation.*

The key insight in our algorithms is that the equivalence property can be exploited to obtain asymptotically much faster algorithms. All nodes in the Dyck-CFL-relation $\mathbb{D}$ are equal to the other nodes in the graph, and thus nodes that belong to the same equivalence class can be safely collapsed to a single representative node. For example, in Figure 3.1(b), node 3 is *Dyck-reachable* from 1, thus, they can be collapsed into a single representative node $\{1, 3\}$ indicating that they are in the same equivalence class. Similarly, node 5 can be collapsed to the representative node $\{1, 3\}$ as well. Finally, we have a representative

node $\{1, 3, 5\}$ reflecting the fact that the three nodes are *Dyck-reachable* from each other in the graph.

3.3.2 A Naïve Approach

We proceed to give a naïve all-pairs Dyck-CFL-reachability algorithm by collapsing the nodes in the graph that are in the Dyck-CFL-relation $\mathbb{D}$. Let $a_i\langle u, v\rangle$ denote the directed edge (u, v) labeled by $a_i \in A$. We note that while collapsing two *Dyck-reachable* nodes x and y in the graph, there always exists a node z such that $a_i\langle x, z\rangle = a_i\langle y, z\rangle$. For example, in Figure 3.1(b), we have $a_1\langle 1, 2\rangle = a_1\langle 3, 2\rangle$. Without loss of generality, given a bidirected graph $G(V, E)$, the naïve algorithm can work on a directed graph $G'(V', E')$ by removing all edges labeled by closing parentheses from the original graph, *i.e.*, $V' = V$ and $a_i\langle u, v\rangle \in E'$ iff $a_i\langle u, v\rangle \in E$ for all labeled edges in E'. The basic idea of the naïve approach is to explicitly maintain a list W of nodes. For every item z popped from W, we pick two incoming neighbors x and y whose edges are labeled by the same opening parenthesis *i.e.*, $\exists a_i\langle x, z\rangle = a_i\langle y, z\rangle$, and then collapse x and y since they are *Dyck-reachable* via z. Due to the collapsing between nodes, E' may possibly contain *multiple edges*. The whole algorithm terminates if W is empty.

The naïve algorithm is given in Algorithm 2, where `Eq_nodes`$[v]$ denotes the equivalence set of node v and `Set`$[v]$ denotes the equivalence set number that node v belongs to. The procedure HAS-SAME-IN(v) traverses all incoming neighbors of node v, and returns true if there exist two neighbors u_1 and u_2 such that $a_i\langle u_1, v\rangle = a_i\langle u_2, v\rangle$. In Algorithm 2, line 1 transforms the given graph G to G', and lines 2-5 initialize W and `Eq_nodes[`v`]`. Lines 10-26 collapse node y to x *w.r.t.* node z, and remove y. The detailed procedure on collapsing y to x is given in Section 3.5.1. Finally, lines 29-31 assign the equivalence set number to each node v, such that any query can be answered in $O(1)$ time.

Complexity Analysis. The time complexity of the naïve algorithm is $O(kn^2)$. We begin by analyzing the maximum number of steps that the "while" loop on line 6 can be executed. We note that Algorithm 2 adds items to W only through lines 5 and 25. On line 25, item x can be added to W for at most $n-1$ times, since line 26 can be executed for at most $n-1$ times. On line 5, W is initialized with n items. Therefore, the worklist W can be filled with at most $2n-1$ items by Algorithm 2. In the "while" loop, only line 28 removes an item from W, thus, the "else" part of the "if" statement can be executed for at most $2n-1$ times. Since the "then" part of the same "if" statement can be executed for

Algorithm 2: A naïve Dyck-CFL-reachability algorithm.

Input : Edge-labeled directed graph $G = (V, E)$
Output: $\texttt{Set}[v]$ for all $v \in V$

```
1  transform the input graph G to G' = (V', E')
2  initialize W to be empty
3  foreach v ∈ V' do
4      Eq_nodes[v] = {v}
5      if HAS-SAME-IN(v) then add v to W
6  while W ≠ ∅ and |V'| > 1 do
7      let z be the front node from W
8      if z ∈ V' and HAS-SAME-IN(z) then
9          let x, y be two nodes such that ∃a_i⟨x, z⟩ = a_i⟨y, z⟩
10         Eq_nodes[x] = Eq_nodes[y] ∪ Eq_nodes[x]
11         foreach a_i ∈ A do
12             if a_i⟨y, y⟩ ∈ E' then
13                 if a_i⟨x, x⟩ ∉ E' then add a_i⟨x, x⟩ to E'
14                 remove a_i⟨y, y⟩ from E'
15         foreach a_i ∈ A do
16             foreach w ∈ V' do
17                 if a_i⟨w, y⟩ ∈ E' then
18                     if a_i⟨w, x⟩ ∉ E' then
19                         add a_i⟨w, x⟩ to E'
20                     remove a_i⟨w, y⟩ from E'
21                 if a_i⟨y, w⟩ ∈ E' then
22                     if a_i⟨x, w⟩ ∉ E' then
23                         add a_i⟨x, w⟩ to E'
24                     remove a_i⟨y, w⟩ from E'
25         add x to W if x ∉ W and HAS-SAME-IN(x)
26         remove y from V'
27     else
28         remove z from W
29 foreach v ∈ V' do
30     foreach u ∈ Eq_nodes[v] do
31         Set[u] = v
```

at most $n - 1$ times, the "while" loop can be executed for at most $(n - 1) + (2n - 1) = 3n - 2 = O(n)$ times. For each item

z popped from W in the "while" loop, lines 8-28 take $O(kn)$ time to process. Specifically, the procedure HAS-SAME-IN(v) on lines 8 and 25 takes $O(kn)$ time to traverse all neighbors of node v, and the two "foreach" loops on lines 15 and 16 are bounded by $|A| = k$ and $|V'| = n$ respectively. Therefore, Algorithm 2 takes $O(kn^2)$ time. The space complexity is $O(n+m)$, since the input graph can be stored using FDLL to be introduced in Section 3.5.1 with $O(m)$ space and the worklist W takes $O(n)$ space. Putting everything together, we have the following theorem:

Theorem 1 *Algorithm 2 pre-processes the input graph in $O(kn^2)$ time and $O(n+m)$ space to answer any online bidirected Dyck-CFL-reachability query in $O(1)$ time.*

In the following two sections, we describe two improved algorithms. They share the same insight with the the naïve approach, which have better time complexities on bidirected trees and graphs respectively. Specifically, our tree algorithm in Section 3.4 uses a single tree walk to find all equivalence sets because trees do not contain cycles. Our graph algorithm in Section 3.5 employs improved data structures to track nodes in W and to merge edges on x and y.

3.4 Dyck-CFL-Reachability Algorithm on Bidirected Trees

This section presents our algorithm for solving the all-pairs Dyck-CFL-reachability problem on bidirected trees. Its time and space complexities are $O(n)$ and $O(n)$ respectively, and it answers any reachability query in $O(1)$ time. We remind the reader that the previous best result on bidirected trees [94] has $O(n \log n \log k)$ time and $O(n \log n)$ space complexities. First, we describe a linear-sized data structure to store the all-pairs reachability information. We then show how to utilize the equivalence property to solve the all-pairs Dyck-CFL-reachability problem using a single walk on trees.

3.4.1 The Stratified-Sets Representation

In our algorithm, the all-pairs Dyck-CFL-reachability information is stored in disjoint sets. Two nodes u and v are *Dyck-reachable* from each other in the tree iff they belong to the same set. In other words, each disjoint set C corresponds to an equivalence class described by relation $\mathbb{D}$, *i.e.*, $u, v \in C$ iff $(u, v) \in \mathbb{D}$. We name the disjoint set representation in our main algorithm as Stratified-Sets.

The Stratified-Sets consist of several disjoint sets span-

ning over different layers. Each disjoint set stores the nodes that are *Dyck-reachable* from each other in the bidirected tree. The layers are indexed by an integer i. Note that the layer information is only used for providing a better explanation. The layer index i grows downward, *i.e.*, layer i is the upper layer in any two adjacent layers i and $i+1$. The disjoint sets on the same layer i have no edges directly connecting each other. For any two adjacent layers i and $i+1$, there exists at least one edge connecting two disjoint sets C from layer i and C' from layer $i+1$. Specially, the connecting edge is labeled by $\mathcal{L}(u,v) \in A$, respecting the fact that there exist $u \in C$ and $v \in C'$ such that (u,v) is a directed edge in the tree with the same label $\mathcal{L}(u,v) \in A$. Note that there can be at most k edges connecting the set C' with the distinct sets from the upper layer i. However, more than k edges are possible for connecting the set C with the distinct sets from the lower layer $i+1$. Figure 3.2(b) shows an example Stratified-Sets representation, where there are seven sets spanning four layers.

The Stratified-Sets representation is implemented using three ingredients: one integer variable `curset`, two integer arrays $\texttt{Set}[v]$ and $\texttt{Up}[\texttt{Set}[v]][a_i]$. $\texttt{Set}[v]$ records the equivalence set number that node v belongs to, and $\texttt{Up}[\texttt{Set}[v]][a_i]$ stores the equivalence

set number of the set from the upper layer that is connected to $\texttt{Set}[v]$ *w.r.t.* the edge labeled by the opening parenthesis $a_i \in A$. The STRATIFIED-SETS uses the integer variable `curset` to keep track of the *current* total number of disjoint sets. Due to the `Up` array, the tree algorithm does not need the layer information explicitly. The STRATIFIED-SETS implementation also permits three operations: $\texttt{Init}(v)$, $\texttt{Find}(v)$ and $\texttt{Add}(v, e)$ described in Procedure 3. The functioning of procedures $\texttt{Init}(v)$ and $\texttt{Find}(v)$ is fairly straightforward. The procedure $\texttt{Init}(v)$ takes a node v as input, assigns it to a new set indexed by `curset` in STRATIFIED-SETS, and increases the `curset` count. $\texttt{Find}(v)$ returns the equivalence set number that node v belongs to.

We detail the description of procedure $\texttt{Add}(v, e)$ to illustrate the idea of collapsing nodes in relation $\mathbb{D}$. We use $\texttt{Add}(v, e)$ to insert the node v to the STRATIFIED-SETS with regard to the edge $e = (u, v)$ and the edge label $\mathcal{L}(u, v)$ in the tree. Node v is added to STRATIFIED-SETS by either assigning it to an new set (lines 3 and 9) or collapsing it to an existing set (line 13). Consider the example input tree in Figure 3.2(a), node 3 and edge $(2, 3)$ are processed by $\texttt{Add}(v, e)$. The resulting STRATIFIED-SETS is in Figure 3.2(b). Node 3 is assigned to a new set on layer 3. The new set is then linked with the set containing node 2 on layer

2 respecting the fact that $\mathcal{L}(2,3) = a_1$. Then, node 4 and edge $(3,4)$ are processed. Node 4 is collapsed to the set on layer 2 that contains node 2 respecting the facts that $\mathcal{L}(3,4) = \bar{a}_1$ and node 4 is *Dyck-reachable* from node 2 (*i.e.*, $(2,4) \in \mathbb{D}$). Formally, if the edge label $\mathcal{L}(u,v)$ in the tree is an opening parenthesis $a_i \in A$, v is assigned to a new set indexed by `curset` in STRATIFIED-SETS. This new set is then linked with the set returned by `Find`(u) on the upper layer as described by lines 2-5 . If the edge label is a closing parenthesis $\bar{a}_i \in \bar{A}$, we simply collapse node v to the equivalence set that is connected via a matched opening parenthesis $a_i \in A$ from u's upper layer. The equivalence set is indexed by `Up`[`Find`(u)][a_i] as described by line 13. Lines 9-11 indicate that, for node u whose link node does not exist, we assign node v to a new set indexed by `curset` and link the set returned by `Find`(u) to the new set from the upper layer.

Note that the `Up` array used in Procedure 3 is indeed a map: $(Num \to A) \to Num$, where Num denotes the domain of the set numbers. For each set in STRATIFIED-SETS, line 8 in Procedure 3 needs to find a particular edge a_i from $O(k)$ link edges in `Up`$[s]$, where $s \in Num$. The time taken to search such $O(k)$ edges depends on the actual implementation of the `Up` array. For example, if the `Up` array stores such $O(k)$ edges for each set s us-

ing a binary search tree, the lookup for an a_i edge in $\texttt{Up}[s]$ takes $O(\log k)$ time as mentioned in Yuan and Eugster's work [94]. In our algorithm, we implement the $\texttt{Up}$ array using the FDLL data structure illustrated as Example 8 in Section 3.5.1, thus a lookup takes *expected* $O(1)$ time. The space required is $O(m)$ since there are m edges in a tree, where $m = n - 1$. Therefore, the time complexity of Procedure 3 is $O(1)$, and the space complexity of the $\texttt{Up}$ array is $O(n)$.

3.4.2 Main Algorithm

This section presents the main algorithm. The key idea is to operate on the linear-sized Stratified-Sets data structure to build the all-pairs Dyck-CFL-reachability information during a single tree walk.

The goal of our algorithm is to assign nodes u and v to the same set in Stratified-Sets, for all $(u, v) \in \mathbb{D}$. The overall algorithm takes two steps:

(1) *Initializing a leaf node:* In this step, we pick an arbitrary leaf node v from the tree and invoke the $\texttt{Init}(v)$ procedure to initialize the given node v.

(2) *Processing each encountered edge:* For each edge (u, v) with label $\mathcal{L}(u, v)$ encountered during the tree walk, we process

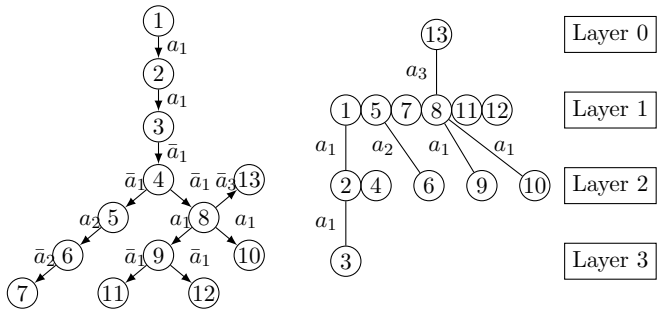
Procedure 3: Add(v, e) to add a node v to STRATIFIED-SETS according to the directed edge $e = (u, v)$.

```
1  if L(u,v) ∈ A then
2      let a_i = L(u,v)
3      Set[v] = curset
4      Up[curset][a_i] = Find (u)
5      curset ++
6  if L(u,v) ∈ Ā then
7      let ā_i = L(u,v)
8      if Up[Find(u)][a_i] does not exist then
9          Set[v] = curset
10         Up[Find(u)][a_i] = curset
11         curset ++
12     else
13         Set[v] = Up[Find(u)][a_i]
```

the edge *w.r.t.* the edge label and insert the node v to STRATIFIED-SETS according to the `Add`(v, e) procedure.

The complete algorithm is shown as Algorithm 4. In the main algorithm, lines 1-6 initialize the relevant data structures, and lines 7-14 describe a standard depth-first search (DFS) starting at node v. For a given bidirected tree $T = (V, E)$ with n nodes, DFS takes $O(n)$ time. For every node v, the `Add`(v, e) procedure takes $O(1)$ time. The space required by Algorithm 4 depends on the STRATIFIED-SETS representation, which is essentially implemented using the `Up` array. Therefore, the space complexity is $O(n)$.

Example 7 *We consider the bidirected tree in Figure 3.2(a), where reverse edges are omitted for brevity. Algorithm 4 outputs*



(a) Input tree. (b) STRATIFIED-SETS representation.

Figure 3.2: A running example for Dyck-CFL-reachability on trees.

the STRATIFIED-SETS *in Figure 3.2(b). The* STRATIFIED-SETS *representation contains seven disjoint sets:* $\{1, 5, 7, 8, 11, 12\}$, $\{2, 4\}$, $\{6\}$, $\{9\}$, $\{10\}$, $\{3\}$ *and* $\{13\}$. *The nodes in the same set are* Dyck-reachable *from each other in the tree. Note that the layer information is only used for providing a better explanation. Algorithm 4 uses the* `Up` *array for finding a set from the upper layer in* STRATIFIED-SETS.

After constructing the STRATIFIED-SETS, any Dyck-CFL-reachability query (u, v) can be answered in $O(1)$ time by simply checking whether the indices returned by $\texttt{Find}(u)$ and $\texttt{Find}(v)$ are the same. Putting everything together, we have the following theorem.

Theorem 2 *The bidirected Dyck-CFL-reachability problem on trees can be pre-processed in* $O(n)$ *time and* $O(n)$ *space to answer any online query in* $O(1)$ *time.*

Algorithm 4: Dyck-CFL-reachability algorithm on trees.

Input : Edge-labeled bidirected tree $T = (V, E)$
Output: the STRATIFIED-SETS

```
1  initialize the Up array to be empty
2  foreach v ∈ V do
3      visited[v] = false
4      Set[v] = 0
5  stack.push(a leaf node v)
6  curset = 0
7  Init (v)
8  while stack is not empty do
9      v = stack.pop
10     if not visited[v] then
11         visited[v] = true
12         foreach unvisited neighbor u of v do
13             stack.push(u)
14             Add (u, e(v, u))
```

3.5 Dyck-CFL-Reachability Algorithm on Bidirected Graphs

In this section, we study the Dyck-CFL-reachability problems on bidirected graphs. Computing Dyck-CFL-reachability on graphs is harder than that on trees because graphs may contain cycles. Consequently, the algorithm introduced in Section 3.4 based on the STRATIFIED-SETS representation cannot be directly applied to graphs.

Although Dyck-CFL-reachability on bidirected graphs is more complicated, the Dyck-CFL-relation $\mathbb{D}$ shares the same equivalence properties as for trees. For bidirected graphs, we utilize the

idea of edge merging to collapse the node pairs $(u, v) \in \mathbb{D}$. For a bidirected graph with n nodes and $2m$ edges, our algorithm processes the given graph in $O(n + m \log m)$ time with $O(n + m)$ space, and can answer any Dyck-CFL-reachability query over any pair of nodes (u, v) in $O(1)$ time.

3.5.1 Basic Idea

As the naïve approach, given a bidirected graph $G(V, E)$, our algorithm works on the same directed graph $G'(V', E')$ by removing all edges labeled by closing parentheses from the original graph, *i.e.*, $V' = V$ and $a_i\langle u, v\rangle \in E'$ iff $a_i\langle u, v\rangle \in E$ for all labeled edges in E'. Therefore, G' has n nodes and m edges. The key idea behind our algorithm is to collapse any node pair (u, v) connected by a *Dyck-path* in the graph because such pairs (u, v) are in the Dyck-CFL-relation $\mathbb{D}$. As in the naïve approach, E' may possibly contain *multiple edges* due to the collapsing between nodes.

To make the above idea more concrete, we consider the example in Figure 3.3. Figure 3.3(a) shows the original path in G which contains two non-trivial sets of nodes that are *Dyck-reachable* from each other: $\{1, 5, 7\}$ and $\{2, 4\}$. Figure 3.3(b) shows the corresponding reduced path in G'. In such directed

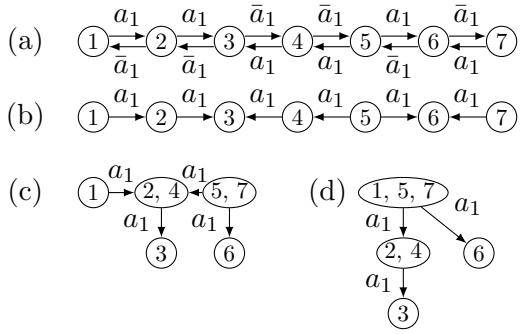


Figure 3.3: A Dyck-path example.

cases, node 5 is "connected" to node 7 via node 6, and $a_1\langle 5,6\rangle = a_1\langle 7,6\rangle$. Since nodes 5 and 7 are *Dyck-reachable* from each other, the two edges $a_1\langle 5,6\rangle$ and $a_1\langle 7,6\rangle$ should be merged to collapse nodes 5 and 7 into a single representative node $\{5,7\}$ in Figure 3.3(c). We define a node like node 6 to be the *merging node*. Formally, we have the following definition.

Definition 4 (Merging Node) *If a node $v \in V$ has at least two incoming edges labeled by $a_i \in A$, we say node v merges a_i edges. We define a node as a* merging node *iff it merges some $a_i \in A$ edges.*

Like the naïve algorithm in Algorithm 2, our main algorithm fulfills the following two tasks:

(1) *Merge edges for each merging node:* For any merging node in the graph G', the algorithm finds the incoming nodes with the same labeled edges, and merges the two edges to collapse

the nodes. In the path in Figure 3.3(b), nodes 3 and 6 are two merging nodes. The relevant incoming edges should be merged.

(2) *Track the new merging nodes:* During edge merging, new merging nodes can be introduced into the graph G'. The algorithm tracks all new merging nodes in order to perform another edge merging. For the same example in Figure 3.3(c), collapsing nodes 2 and 4 generates a new representative node $\{2, 4\}$, which is also a merging node. Its corresponding edges should also be merged. The final output is shown in Figure 3.3(d).

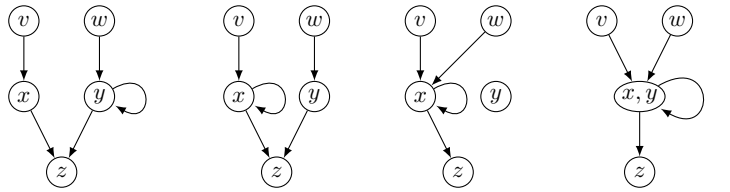
In the naïve approach, nodes x and y are arbitrarily picked on line 9. Merging edges from y to x by enumerating all neighbors of y exhaustively takes $O(kn)$ time. Moreover, node tracking is achieved by simply traversing all neighbors of node z on line 8, which takes $O(kn)$ time as well. When the given graph is sparse, we can use additional data structures to improve the two tasks of merging edges and tracking nodes. Next, we describe them in detail.

Merging Edges

For the directed graph G', edge merging in G' picks a merging node z that has two incoming edges (x, z) and (y, z) with $a_i\langle x, z\rangle = a_i\langle y, z\rangle$, and collapses nodes x and y. Specifically, if we choose to merge edge (y, z) to edge (x, z), all edges connecting y and its neighbor w should be deleted from G'. Node w is made a neighbor of x by inserting the relevant edges to G'. Finally, node y is removed from G', because it has been collapsed to x.

The order of edge merging is important. The naïve method in Algorithm 2 performs edge merging by collapsing nodes x and y arbitrarily with regard to the merging node z. If one adopts this approach, the time complexity can be $O(kn^2)$. When the graphs are sparse, it is possible to do edge merging faster. To this end, we employ a technique that is similar to the weighted-union heuristic (also known as the "union-by-size" heuristic) used in the disjoint sets data structures [24]. Namely, for each edge merging operation, we always collapse the node with a smaller degree to the node with a larger degree. Our new method bounds the total numbers of edge merging for *each edge* to $O(\log m)$. We provide a detailed complexity analysis in Section 3.5.3.

Taking the naïve approach in Algorithm 2 as an example, we discuss the process of edge merging. Specifically, our handling



(a) Before edge merging. (b) Collapsing self loops. (c) Collapsing all neighbors. (d) After edge merging.

Figure 3.4: Steps in edge merging.

of edge merging from node y to node x has three phases, as illustrated in Figure 3.4. We assume that the degree of node x is larger than that of y, and all irrelevant edges are omitted in Figure 3.4. Figure 3.4(a) shows the original graph before edge merging. Figure 3.4(b) illustrates the handling in the first phase. If y has a self loop, the self loop is removed and added to x if x does not already have one (lines 11-14 in Algorithm 2). Second, we consider all neighbors of node y. As in Figure 3.4(c), for all shared neighbors z of x and y, those edges between z and y are removed; for those neighbors w that only belong to y, new edges between w and x are inserted (lines 15-24 in Algorithm 2). Finally, Figure 3.4(d) shows that node y is removed from the graph and degrees for relevant nodes in edge merging are updated.

Tracking Nodes

During edge merging, the in-degrees of merging nodes may change. We need to explicitly maintain a list of merging nodes whose in-

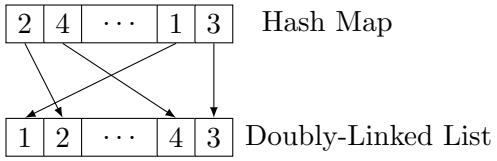


Figure 3.5: The illustration of the FDLL data structure.

degrees *w.r.t.* an opening parenthesis are at least 2. This section describes the design of our data structure to effectively maintain this information.

In the naïve algorithm, tracking nodes is achieved using a worklist W, which is typically implemented using a list. In our improved algorithm, we uses a doubly-linked list (DLL) and a hash map. We name the data structure FAST-DOUBLY-LINKED-LIST (FDLL). It is important to note that traditional list W used by the naïve approach on line 25 takes $O(n)$ time to find an element while FDLL takes *expected* $O(1)$ time. Figure 3.5 depicts an example FDLL. All elements in the FDLL are stored using a DLL and a hash map. The hash map associates each element with its position in the DLL, represented by arrows in Figure 3.5, to help quickly locate every element in the DLL. The *insertion* and *pop* operations on FDLL are nearly identical to DLL (or list), both of which take $O(1)$ time. Moreover, the FDLL supports two additional operations in *expected* $O(1)$ time:

- *Query:* Querying the membership of a particular element in

FDLL is the same as querying the relevant membership in the hash map, which can be done in *expected* $O(1)$ time.

- *Deletion:* According to the hash map, the position of the element to be deleted can be found in *expected* $O(1)$ time. Thus removing the element at the specified position in the DLL can also be done in $O(1)$ time. Finally, the hash map entry associated with that element is erased in $O(1)$ time.

Example 8 *In the tree algorithm, the* `Up` *array can be implemented using FDLL to support the* expected $O(1)$ *time lookup. Recall that the* `Up` *array is indeed a map:* $(Num \to A) \to Num$*. For each set* $s \in Num$*, we use an FDLL* $A[s]$ *to store the set of opening parentheses* a_i*, such that there exists an* a_i *labeled edge connecting set* s *and the set from the upper layer. For each* $a_i \in A[s]$*, we use another FDLL* $U[s\_i]$ *to store the corresponding set from the upper layer. For example, to look up an edge* a_3 *of set* 2*, we query whether* $a_3 \in A[2]$*. If such an* a_3 *exists,* $U[2\_3]$ *keeps the corresponding set number. The two lookups both take* expected $O(1)$ *time.*

3.5.2 Main Algorithm

We now present our algorithm solving bidirected Dyck-CFL-reachability on graphs in Algorithm 5, combining the ideas of

merging edges and tracking nodes.

Before delving into the main algorithm, we first illustrate the use of FDLL in our main algorithm. For each node v in the input graph $G' = (V', E')$, the set of opening parentheses of v's incoming edges is represented using an FDLL, denoted as $A_{\text{in}}[v]$. For each $a_i \in A_{\text{in}}[v]$, we use the FDLL $\texttt{In}[v\_i]$ to store all v's incoming neighbors. We denote the size of $\texttt{In}[v\_i]$ as $\texttt{In}[v\_i]\texttt{.size()}$. Node v merges edge a_i iff $\texttt{In}[v\_i]\texttt{.size()} > 1$. We represent $\texttt{Out}[v\_i]$ and $A_{\text{out}}[v]$ similarly. Finally, the worklist FDLL_w is also represented using an FDLL.

Algorithm 5 uses the array $\hat{D}[v]$ to represent the *total degree* of node v, which is initialized to v's degree in the original graph. Notations $\texttt{eq\_nodes}[v]$ and $\texttt{Set}[v]$ are defined in the same way as the naïve approach. The functioning of the algorithm proceeds as follows. On line 1, the original bidirected graph G is preprocessed to obtain the directed graph $G' = (V', E')$. From lines 2-5, the equivalence set $\texttt{Eq\_nodes}[v]$ is initialized with node v and the FDLL_w is initialized with v_i indicating node v merges a_i. The FDLL_w in the main algorithm is used to implement the idea of node tracking. The main algorithm then proceeds to handle edge merging as follows:

- Lines 7-11: The algorithm pops one z_i from the FDLL_w,

then chooses its two neighbors x and y for edge merging. Specifically, node y with a smaller *total degree* is collapsed to node x with a larger *total degree*.

- Lines 12-18: The self loop on node y is handled as described in Figure 3.4(b). During edge merging, if a node v becomes a non-merging node for a_i (*i.e.*, `In[`v_i`].size()` < 2), v_i is removed from the FDLL$_w$. Similarly, when node v becomes a merging node that merges a_i edges, v_i is inserted to the FDLL$_w$. Note that when an edge is inserted/removed from E', all of `In[`v_i`]`, $A_{\text{in}}[v]$, `Out[`v_i`]` and $A_{\text{out}}[v]$ need to be updated accordingly.

- Lines 19-31: All incoming and outgoing neighbors of y are handled as described in Figure 3.4(c). The update on FDLL$_w$ for merging nodes is similar to the handling of self loop.

- Line 32: Node y is removed from V'. Note that we do not need to remove z_i because it was once w_i.

The edge merging ends when the FDLL$_w$ is empty, *i.e.*, there are no merging nodes in the final graph. Lines 33-35 indicate that all nodes u in the equivalence set `Eq_nodes`$[v]$ are enumerated to associate `Set`$[u]$ with v.

After the main algorithm terminates, `Set`$[v]$ stores the equiv-

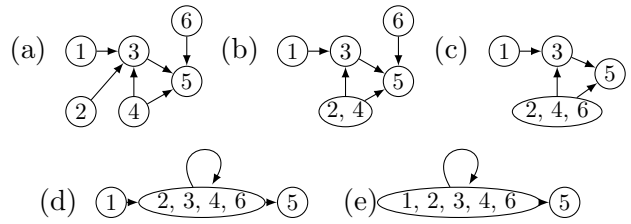


Figure 3.6: A running example for bidirected Dyck-CFL-reachability on graphs.

alence set number that node v belongs to. Similar to the tree case, any Dyck-CFL-reachability query (u, v) can be answered in $O(1)$ time by simply checking the equivalence set numbers of nodes u and v.

Example 9 *Figure 3.6 shows an example. All edges are labeled by* $a_1 \in A$*. The original edges labeled by* $\bar{a}_1 \in \bar{A}$ *are removed first. Nodes* 3 *and* 5 *are two merging nodes. In the first iteration, nodes* 2 *and* 4 *are collapsed by edge merging. Then node* 6 *is collapsed as well. Finally, node* 1 *is collapsed due to its self loop in the graph. In the final graph, all of the nodes in the original graph are distributed into two disjoint sets.*

Algorithm 5: Dyck-CFL-reachability algorithm on graphs.

Input : Edge-labeled bidirected graph $G = (V, E)$
Output: $\texttt{Set}[v]$ for all $v \in V$

```
1  transform the input graph G to G' = (V', E')
2  foreach v ∈ V' do
3      Eq_nodes[v] = {v}
4      foreach a_i ∈ A_in[v] do
5          add v_i to FDLL_w if In[v_i].size() > 1
6  while FDLL_w ≠ ∅ do
7      let z_i be the front item from FDLL_w
8      let x and y be two front nodes from In[z_i]
9      let x denote the node such that D̂[x] ⩾ D̂[y]
10     D̂[x] = D̂[x] + D̂[y]
11     Eq_nodes[x] = Eq_nodes[y] ∪ Eq_nodes[x]
12     foreach a_i ∈ A_in[y] do
13         if a_i⟨y, y⟩ ∈ E' then
14             if a_i⟨x, x⟩ ∉ E' then
15                 add a_i⟨x, x⟩ to E'
16                 add x_i to FDLL_w if x_i ∉ FDLL_w and In[x_i].size() > 1
17             remove a_i⟨y, y⟩ from E'
18             remove y_i from FDLL_w if y_i ∈ FDLL_w and In[y_i].size() < 2
19     foreach a_i ∈ A_in[y] do
20         foreach w ∈ In[y_i] do
21             if a_i⟨w, x⟩ ∉ E' then
22                 add a_i⟨w, x⟩ to E'
23                 add x_i to FDLL_w if x_i ∉ FDLL_w and In[x_i].size() > 1
24             remove a_i⟨w, y⟩ from E'
25             remove y_i from FDLL_w if y_i ∈ FDLL_w and In[y_i].size() < 2
26     foreach a_i ∈ A_out[y] do
27         foreach w ∈ Out[y_i] do
28             if a_i⟨x, w⟩ ∉ E' then
29                 add a_i⟨x, w⟩ to E'
30             remove a_i⟨y, w⟩ from E'
31             remove w_i from FDLL_w if w_i ∈ FDLL_w and In[w_i].size() < 2
32     remove y from V'
33 foreach v ∈ V' do
34     foreach u ∈ Eq_nodes[v] do
35         Set[u] = v
```

3.5.3 Algorithm Correctness and Complexity Analysis

This section discusses the correctness and complexity of our proposed algorithm. First, we establish its correctness.

Theorem 3 (Correctness) *Algorithm 5 correctly finds all* Dyck-paths *in the input graph.*

Proof. It is clear that any *Dyck-path* reported by Algorithm 5 is indeed a *Dyck-path* due to the observed equivalence property (Lemma 1). Thus, our proof focuses on the other direction, that is Algorithm 5 finds all *Dyck-path*s in the input graph.

Any trivial *Dyck-path* generated by rule $S \to \varepsilon$ is handled correctly, because every node v in the graph is marked as *Dyck-reachable* from itself due to line 3 in Algorithm 5. Dyck grammar essentially generates the properly matched parentheses. Therefore, the length of any non-trivial *Dyck-path* in the graph is even. We prove by induction on the length $|p|$ of any non-trivial *Dyck-path.*

Base case. $|p| = 2$. Let the *Dyck-path* be $p = v_1 v_2 v_3$, with the *realized* string $R(p) = \mathcal{L}(v_1, v_2)\mathcal{L}(v_2, v_3) = a_i \bar{a}_i$. Because the graph is bidirected, we have $\mathcal{L}(v_3, v_2) = \mathcal{L}(v_1, v_2) = a_i$. Nodes v_1 and v_3 are collapsed due to the merging node v_2.

Inductive step. Suppose Algorithm 5 correctly finds all non-

trivial *Dyck-paths* of length $|p|$ in the graph. According to the Dyck grammar, any non-trivial *Dyck-path* of length $|p| + 2$ is generated by the following two rules:

- $S \rightarrow a_i S \bar{a}_i$ indicates that the new S-path is generated by prepending an open parenthesis and appending by a matching closing parenthesis. Let the path be $p = v_1 v_2 \dots v_3 v_4$, where $\mathcal{L}(v_1, v_2) = a_i$ and $\mathcal{L}(v_3, v_4) = \bar{a}_i$. The *realized* string $R(v_2 \dots v_3) = S$ indicates that nodes v_2 and v_3 are *Dyck-reachable*, where the *Dyck-path* joining v_2 and v_3 is of length $|p|$. According to the induction hypothesis, they have been collapsed into a single representative node. Such a node is the merging node for v_1 and v_4. Thus, v_1 and v_4 are collapsed according to Algorithm 5.
- $S \rightarrow SS$ indicates that the new S-path is composed of two S-paths. Let the path be $p = v_1 \dots v_2 \dots v_3$, where $R(v_1 \dots v_2) = R(v_2 \dots v_3) = S$. Note that the length of any non-trivial *Dyck-path* is at least 2. Since the new S-path is of length $|p| + 2$, the lengths of both path $v_1, \dots, v_2$ and path $v_2, \dots, v_3$ are less than or equal to $|p|$. According to the induction hypothesis, both v_1, v_2 and v_2, v_3 are collapsed into a single representative node.

As a result, v_1 and v_3 are collapsed into the same representation node as well.

□

Next we analyze the complexity of Algorithm 5. Note that the *total degree* $\hat{D}[v]$ used in our algorithm is different from the degree $D[v]$ of node v respecting the fact that the *total degree* $\hat{D}[v]$ admits duplicated edges. For example, in Figure 3.4(c), $\hat{D}[x] = 4+2 = 6$, but $D[x] = 5$, because edge (y, z) is duplicated with (x, z) according to the final representative node $\{x, y\}$ in Figure 3.4(d). Therefore, the *total degree* $\hat{D}[v]$ never decreases during edge merging. Our algorithm processes the merging node z and collapses its neighbor y with a smaller $\hat{D}[y]$ to node x with a larger $\hat{D}[x]$. Nodes y and x are collapsed into a single representative node $\{x, y\}$. As a result, the *total degree* of y after merging is the same as the *total degree* of x according to line 10, namely, $\hat{D}[x] = \hat{D}[x] + \hat{D}[y]$. On line 9, we have $\hat{D}[x] \geqslant \hat{D}[y]$. Combining the analysis of the two lines, the following lemma is immediate:

Lemma 2 *For each edge merging in Algorithm 5, $\hat{D}[y]$ is doubled.*

Let m denote the number of edges in graph G', *i.e.*, $2m = \sum_{v \in V} D[v]$. Due to the duplicated edges, $D[v]$ may decrease during edge merging. Similarly, $\hat{m}$ denotes the *total number* of edges in graph G' *w.r.t.* $\hat{D}[v]$ that admits duplicated edges, *i.e.*, $\hat{m} = \sum_{v \in V} \hat{D}[v]$. For each edge (x, z) in graph G', we say it is "moved" if either the *total degree* $\hat{D}[x]$ or $\hat{D}[z]$ is doubled according to Lemma 2. For all $v \in V$, we have $\hat{D}[v] \leqslant \hat{m} \leqslant 2m$, because in the worst case, all nodes are collapsed into a single representative node (with m duplicated edges) in the final graph. Combined with Lemma 2, an edge is "moved" at most $(\log \hat{D}[x] + \log \hat{D}[z] \leqslant 2 \log 2m)$ times. The "while" loop on line 6 in Algorithm 5 takes time proportional to the number of times that an edge is "moved". Therefore, the total running time for the "while" loop is $O(m \log m)$. For lines 33-35, it takes $O(n)$ time to enumerate all nodes u in the equivalence set $\texttt{Eq\_nodes}[v]$ and to associate $\texttt{Set}[u]$ with v. Therefore, the time complexity of our algorithm is $O(n+m \log m)$. At the beginning, each node is associated with two FDLLs $\texttt{In}[v\_i]$ and $\texttt{Out}[v\_i]$. In each iteration of edge merging, there is one node removed from G', decreasing the number of edges m in G'. As a result, $O(n + m)$ space is required for processing the graph. Putting all these together, we have the following theorem:

Theorem 4 *Algorithm 5 pre-processes the input graph in $O(n + m \log m)$ time and $O(n + m)$ space to answer any online bidirected Dyck-CFL-reachability query in $O(1)$ time.*

In practice, since the graphs in client analyses are typically sparse, Algorithm 5 performs in $O(n \log n)$ time. When the graphs are dense, we can use the naïve approach in Algorithm 2 for pre-processing. As a result, we have the following theorem:

Theorem 5 *The bidirected Dyck-CFL-reachability problem on graphs can be pre-processed in $O(min\{kn^2, n + m \log m\})$ time and $O(n + m)$ space to answer any online query in $O(1)$ time.*

3.6 Dyck-CFL-Reachability Algorithm

In this section, we focus on the general Dyck-CFL-Reachability problem. Similar to the bidirected counterpart described in Sections 3.4 and 3.5, given a digraph $G = (V, E)$, we also consider the binary Dyck-CFL-relation $\widetilde{\mathbb{D}}$ over $V \times V$. Node $v \in V$ is *Dyck-reachable* from node $u \in V$ iff $(u, v) \in \widetilde{\mathbb{D}}$. Note that the *directed* Dyck-CFL-relation $\widetilde{\mathbb{D}}$ is different from the *bidirected* Dyck-CFL-relation $\mathbb{D}$ since $\widetilde{\mathbb{D}}$ is not symmetric. Consequently, the *directed* Dyck-CFL-relation $\widetilde{\mathbb{D}}$ is not an equivalence relation and the two algorithms described in ddsd can not be applied.

3.6.1 Basic Idea

We revisit the Dyck grammar to illustrate the basic idea of the Dyck-CFL-Reachability algorithm. The Dyck grammar actually consists of two kinds of rules:

$$S \rightarrow SS \mid \varepsilon \tag{3.1}$$

$$S \rightarrow a_1 S \bar{a}_1 \mid ... \mid a_k S \bar{a}_k \tag{3.2}$$

where S is the start symbol and ε is the empty string. We note that rules 3.1 describe the transitivity and reflexivity on $\widetilde{\mathbb{D}}$. It is fairly straightforward to realize that the Dyck-CFL-relation $\widetilde{\mathbb{D}}$ generated by rules 3.1 can be thought of as the standard reachability relation in a directed graph, *i.e.*, node v is reachable from itself and for all nodes $x, y, z \in V$, whenever y is reachable from x and z is reachable from y then z is reachable from x. As a result, maintaining the Dyck-CFL-relation $\widetilde{\mathbb{D}}$ generated by rules 3.1 is exactly the same as maintaining the transitive closure of a directed graph. Rules 3.2 depict the pairs of matched parentheses in $\widetilde{\mathbb{D}}$. For matching parentheses, we can use a dynamic programming style algorithm to compute all pairs of opening parentheses and their matched closing parentheses, which is quite similar to the traditional CFL reachability algorithm.

To sum up, the basic idea of our Dyck-CFL-Reachability al-

gorithm is to employ a dynamic programming style algorithm for *matching parentheses* (handling rules 3.2). For each summary edge (u, S, v), our algorithm searches the incoming opening parentheses of u and the matched outgoing closing parentheses of v to generate new S-edges. Since the parentheses considered by our algorithm are exactly the edges in the original graph, the running time required for *matching parentheses* is $O(mn)$, where m and n represent the number of edges and nodes in the given graph. Then we adopt an efficient data structure for *maintaining transitive closure* based on all S-edges (handling rules 3.1). The data structure explicitly maintains the transitive closure in $O(n)$ amortized time [43]. Let S denote the number of S-edges in the final graph. The running time required for *maintaining transitive closure* is $O(Sn)$. As a result, the whole algorithm computes the all-pairs Dyck-CFL-Reachability in $O(n(m + S))$ time.

3.6.2 Maintaining Transitive Closure

Given a directed graph $G = (V, E)$, the *transitive closure* of G is also a directed graph with the same set of nodes V but has an directed edge from node u to v iff there is a path from u to v in G. In this section, we review an existing incremental algorithm for

Procedure 6: Init() to initialize the key data structures.

```
1 for i ← 1 to n do
2 |   initialize T(i) as an empty tree
3 |   for j ← 1 to n do m_ij ← 0
```

maintaining the transitive closure of a directed graph a directed graph [43]. The incremental algorithm plays a pivotal role in our Dyck-CFL-Reachability algorithm. It starts with an empty graph that undergoes a sequence of edges insertions. Specifically, it permits the following operations:

- *Insert*(u, v)*:* insert a directed edge between nodes u and v in the graph.
- *Query*(u, v)*:* check whether v is reachable from u in the graph.

Note that the directed edges concerned with the insertion operation are completely arbitrary. The incremental algorithm supports each edge insertion in $O(n)$ amortized time and each query in $O(1)$ time. It also explicitly maintains the transitive closure, which yields an $O(n^2)$ space complexity.

Key Data Structures

The incremental algorithm operates on a matrix M representing the *transitive closure* and *spanning trees* associated with each

node. We give the necessary definitions as follows.

A directed graph is called a *directed acyclic graph* (DAG) if it does not contain any cycle. Given a directed graph $G = (V, E)$, a *spanning tree* $T(x)$ rooted at node $x \in V$ is a DAG, such that:

(1) the root node $x \in V$ does not have any incoming edge;

(2) each node $n \in T(x) \setminus \{x\}$ has exactly one incoming edge;

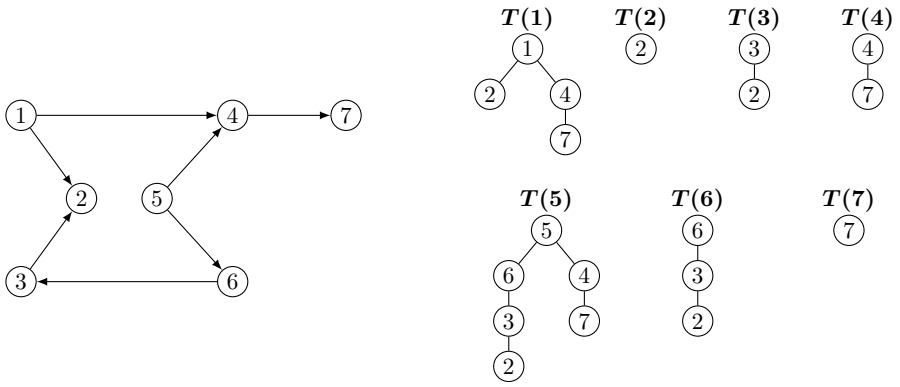
(3) there is a unique path from root x to $n \in T(x) \setminus \{x\}$ iff n is reachable from x in G.

The incremental algorithm also maintains an $n \times n$ matrix M, each entry m_{ij} is defined as follows:

$$m_{ij} = \begin{cases} 1, & \text{if node } j \text{ is reachable from } i \text{ in } G, \\ 0, & \text{otherwise.} \end{cases}$$

The matrix M explicitly represents the adjacency matrix of the transitive closure of G.

The key data structures are initialized by Procedure 6. The running time required for the `init()` procedure is clearly $O(n^2)$. However, the time complexity can be easily reduced to $O(n)$ by initializing each matrix entry m_{ij} the first time when it is accessed [5, pp. 71].



(a) The input graph. (b) The spanning trees.

Figure 3.7: Example graphs illustrating a directed graph and its corresponding bidirected graph.

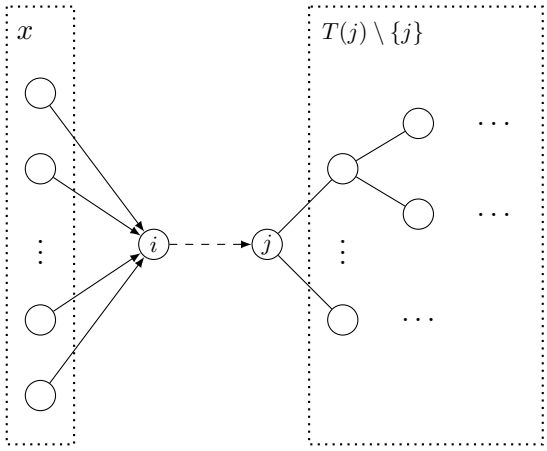
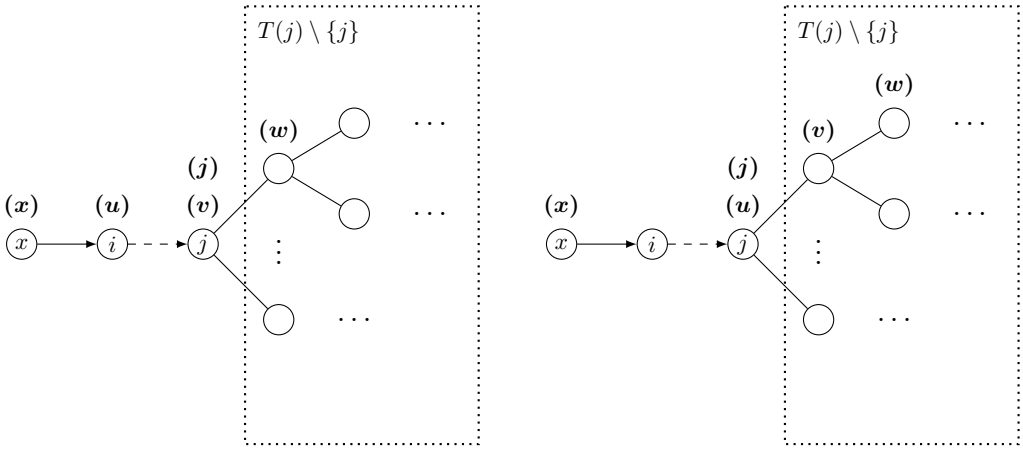


Figure 3.8: The reachability information after adding a new edge (i, j).

Example 10 *We show an example in Figure 3.7. The graph is shown in Figure 3.7(a), and the spanning trees are shown in Figure 3.7(b).*

Main Procedures

We precede to give the main procedures for maintaining the transitive closure. Procedure 7 describes the procedure for handling



(a) Calling procedure `Meld`(). (b) Recursively calling procedure `Meld`().

Figure 3.9: Situations calling recursive procedure `Meld`().

the insertion operation. Let the directed edge considered be (i, j). The edge provides new reachability information only if node j is not previously reachable from i, as indicated by line 1. Figure 3.8 shows the new reachability information introduced by edge (i, j), *i.e.*, each node x that previously reaches i should reach all nodes in the *spanning tree* associated with j. In Procedure 7, lines 2-3 searches all such nodes x and updates the reachability information only if x does not previous reaches j, *i.e.*, $m_{xi} \neq 0$ and $m_{xj} == 0$.

The updating of reachability information is handled by procedure `Meld`() shown in Procedure 8. The new reachability information between node x and $n \in T(j)$ is updated by pruning a unique copy of $T(j)$ and inserting it into $T(x)$. Specifically, Procedure 8 involves the following two steps:

Procedure 7: Add(i, j) to insert an edge (i, j).

```
1 if m_ij == 0 then          // there are no previous path from i to j
2     for x ← 1 to n do
3         if m_xi ≠ 0 and m_xj == 0 then
4             // the edge (i, j) gives rise to a new path from x to j
5             Meld (x, j, i, j)
```

(1) recursively pruning a unique copy of $T(j)$ by eliminating the nodes that are already in $T(x)$ (lines 4-6);

(2) linking the nodes in the unique copy of $T(j)$ to $T(x)$ and updating the reachability matrix (lines 1-3).

Note that line 3 in Procedure 8 inserts the summary edge (x, S, v) to the graph G and the worklist W used by the Dyck-CFL-Reachability algorithm. Figure 3.9 further illustrates the functionality of the recursive procedure `Meld`(). Given an directed edge (i, j), the procedure `add`(i, j) calls `Meld`(x, j, i, j). In the subsequent recursive calls, u represents the parent of v in $T(j)$, and every child w of v is considered. If node w is already reachable from x (*i.e.*, $m_{xw} \neq 0$), the `Meld`() procedure returns since all children of w are reachable from x (*i.e.*, they are also the children of x in $T(x)$).

Example 11 *Figure 3.10 shows an example of updating the reachability information after inserting edge* $(2, 5)$ *in Figure 3.7(a). The first row shows the spanning trees before inserting edge* $(2, 5)$.

Procedure 8: Meld(x, j, u, v) to merge trees.

```
1 insert v in T(x) as a child of u
2 m_xv ← 1
3 insert (x, S, v) to G and to W
4 foreach child w of v ∈ T(j) do
5 |   if m_xw == 0 then
6 |   |   Meld (x, j, v, w)                    // update by means of
```

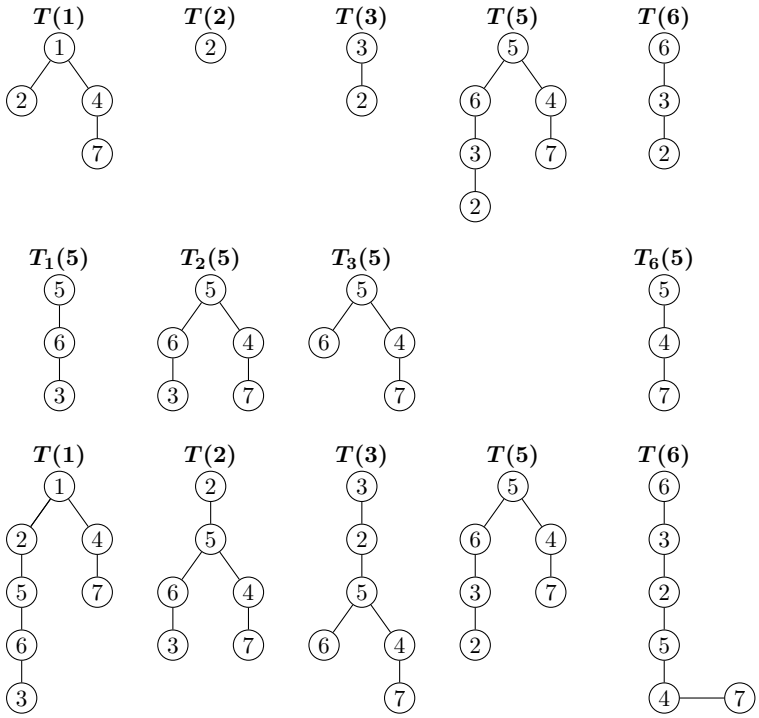


Figure 3.10: Updating spanning trees.

The second row shows the pruned spanning tree $T_x(5)$, where node x previously reach node 2 and all nodes $v \in T_x(5)$ are not in $T(x)$. Finally, the last row shows the spanning trees after the edge insertion.

3.6.3 Matching Parentheses

In this section, we describe our Dyck-CFL-Reachability algorithm. The Dyck language contains k kinds of parentheses and is generated by a Dyck grammar with the start symbol S. Our algorithm takes a graph whose edges are labeled by parentheses as input and computes the *all-pairs* Dyck-CFL-reachability in $O(n(m + S))$ time, where S, m and n represent the number of S-paths in the final graph, the number of edges and nodes in the original graph respectively.

Our algorithm follows the popular dynamic programming style [70, 93] for solving *all-pairs* CFL-reachability. As aforementioned, our algorithm generates the new matched parentheses (*i.e.*, rules 3.2) by searching the incoming opening parentheses on node u and the matched closing outgoing parentheses on node v for each summary (u, S, v) in the graph. Then our algorithm concatenates the existing matched parentheses (*i.e.*, rules 3.1) by adopting the data structures for maintaining the transitive closure on all S-paths.

Generating New Matched Parentheses

Three kinds of summary edges are considered by our algorithm, *i.e.*, the opening parentheses edges (u, A_i, v), the closing paren-

theses edges $(u, \bar{A}_i, v)$ and the *Dyck*-paths (u, S, v). At the beginning, each labeled edge in the given graph is transformed into a corresponding summary edge (*i.e.*, (u, a_i, v) becomes (u, A_i, v) and $(u, \bar{a}_i, v)$ becomes $(u, \bar{A}_i, v)$).

A worklist W is used to maintain all summary edges. The main algorithm processes all summary edges in W, detailed as follows:

- *Opening parentheses edges* (u, A_i, v)*:* the main algorithm searches all outgoing neighbors w of v, such that $(v, \bar{a}_i, u)$ exists in the original graph. Then new summary edges representing the *Dyck*-paths (u, S, w) are inserted to the worklist W;

- *Closing parentheses edges* $(u, \bar{A}_i, v)$*:* similarly, the main algorithm searches all incoming neighbors w of u, such that (w, a_i, u) exists in the original graph. Then new summary edges representing the *Dyck*-paths (w, S, v) are inserted to the worklist W;

- *Dyck-paths* (u, S, v)*:* the main algorithm searches all incoming neighbors w of u, such that (w, a_i, u) exists in the original graph. Then new summary edges (w, A_i, v) representing the opening parentheses are inserted to the worklist W. Similarly, the main algorithm also search all outgoing neighbors

w of v to insert new closing parentheses edges $(u, \bar{A}_i, w)$ to W.

To sum up, when a summary edge representing opening or closing parenthesis is popped from W, the new *Dyck*-path representing the new matched parentheses is generated with respect to rules 3.2 in the Dyck grammar. When a *Dyck*-path (u, S, v) is popped from W, the main algorithm propagates the parentheses from nodes u and v by inserting the corresponding new summary edges into W.

Concatenating Existing Matched Parentheses

Note that the current algorithm discussed so far does not handle the transitivity and reflexivity on *Dyck*-path described by rules 3.1. Transitional CFL-reachability algorithm [70, 93] handles them by searching all incoming and outgoing S-edges for each summary edge popped from the worklist W. If one adopt the same strategy, the time complexity of the main algorithm is $O(n^3)$. Next, we introduce an improved strategy by using the incremental algorithm discussed in Section 3.6.2 for maintaining the transitive closure on all S-edges.

To apply the incremental algorithm, we need to modify the handling of summary edges in the current algorithm. Specifi-

cally, for each newly generated S-edges described in Section 3.6.3 (*i.e.*, during the processing of the summary edges representing opening and closing parentheses), we insert them into the data structures maintained by the incremental algorithm discussed in Section 3.6.2. Namely, the transitive closure maintained by the incremental algorithm consists of all S-edges in the input graph. The data structures then return all new S-edges according to the transitivity and insert them into the worklist W as described on line 3 in Procedure 8.

Main Algorithm

We now present our Dyck-CFL-Reachability algorithm. Given an edge-labeled graph, the algorithm computes the set of all *Dyck*-paths in the graph.

Algorithm 9 shows the main algorithm. It is a worklist algorithm that propagates the Dyck-CFL-Reachability information among summary edges. Lines 1-3 initialize the worklist W with all summary edges derived from the parentheses original graph. The summary edges are also inserted to the original graph. For each summary edge (i, B, j) popped from W, the algorithm processes it according to Sections 3.6.3 and 3.6.3 as follows:

- Lines 6-10: the algorithm handles the summary edges rep-

Algorithm 9: Dyck-CFL-Reachability Algorithm.

Input : Edge-labeled directed graph $G = (V, E)$;
Output: the set of summary edges;

```
1  initialize W to be empty
2  foreach (i, a_i, j) ∈ E do insert (i, A_i, j) to G and to W
3  foreach (i, ā_i, j) ∈ E do insert (i, Ā_i, j) to G and to W
4  while W ≠ ∅ do
5      (i, B, j) ← SELECT-FROM(W)
6      if B == Ā_i then
7          foreach k ∈ IN(i, a_i) do
8              if (k, S, j) ∉ G then
9                  Add (k, j)
10                 insert (k, S, j) to G and to W
11     if B == A_i then
12         foreach k ∈ OUT(j, ā_i) do
13             if (i, S, k) ∉ G then
14                 Add (k, j)
15                 insert (i, S, k) to G and to W
16     if B == S then
17         foreach k ∈ IN(i, a_i) do
18             if (k, A_i, j) ∉ G then
19                 insert (k, A_i, j) to G and to W
20         foreach k ∈ OUT(j, ā_i) do
21             if (i, Ā_i, k) ∉ G then
22                 insert (i, Ā_i, k) to G and to W
```

resenting closing parentheses (*i.e.*, $(i, \bar{A}_i, j)$). All incoming neighbors k of node i with summary edge (k, A_i, i) are considered. If a new *Dyck*-path (*i.e.*, (k, S, j)) is generated, it is then inserted to the graph and the data structure.

- Lines 11-15: the algorithm handles the summary edges representing opening parentheses, which similar to the handling

of the closing parentheses.

- Lines 16-22: the algorithm handles all generated S-edges (i, S, j). Lines 17-19 search all incoming opening parentheses (k, a_i, i), and insert the summary edge (k, A_i, j) representing an opening parenthesis to the graph. Similarly, lines 20-22 handles the closing parentheses.

Algorithm 9 terminates when there is no new S-edge to be inserted. To query if a node j is *Dyck*-reachable from node i, we can simply test whether the summary edge (i, S, j) exists.

3.6.4 Algorithm Correctness and Complexity Analysis

The correctness of Algorithm 9 can be established *w.r.t.* the Dyck grammar. Lines 6-15 match every parenthesis depicted in rule 3.2. Moreover, by calling to the `Add()` procedure on lines 9 and 14, all new transitive S-edges depicted in rule 3.1 are correctly generated and inserted into the graph. On the other hand, lines 16-22 generate new summary edges representing the opening and closing parentheses for matching *w.r.t.* rule 3.2, *i.e.*, for each S-edge (i, S, j), all summary edges (k, A_i, j) and $(k, \bar{A}_i, j)$ are inserted to the graph where A_i represents $a_i S$ and $\bar{A}_i$ represents $S\bar{a}_i$ respectively.

Then, we analyze the complexity of Algorithm 9. For each

summary (i, B, j) popped from the worklist W, the **foreach** loops at lines 7, 12, 17 and 20 search exactly the incoming and outgoing edges in the *original* graph. For a graph with n nodes and m edges, the algorithm takes $O(mn)$ time for searching. The `Add`() procedure is called at lines 9 and 14 iff there is a new S-edge generated. As a result, the procedure `Add`() is called for $|S|$ times where $|S|$ denotes the number of S-edges in the *final* graph. The procedures `Add`() and `Meld`() perform the same work as the previous dynamic graph reachability algorithm [43]. Therefore, the amortized running time for each edge insertion (*i.e.*, each call to `Add`()) is $O(n)$. Combined the analysis, the running time of Algorithm 9 is $O(n(m + S))$. The space complexity is clearly $O(n^2)$ due to the use of the reachability matrix. Finally, we have the following theorem:

Theorem 6 *The general Dyck-CFL-reachability problem on graphs can be pre-processed in* $O(n(m+S))$ *time and* $O(n^2)$ *space to answer any online query in* $O(1)$ *time, where* S *is the number of* Dyck*-paths in the final graph.*

□ **End of chapter.**

Chapter 4

Application: Scaling an Alias Analysis for Java

To demonstrate the practical applicability of our fast Dyck-CFL-reachability algorithms, we leverage a recent demand-driven *context-sensitive* alias analysis for Java [92] formulated using CFL-reachability. Dyck-CFL-reachability is used to formulate its *context-insensitive* variant. The analysis is demand-driven in the sense that it solves the *single-source-single-target* Dyck-CFL-reachability problem. We show that our fast algorithms for *all-pairs* Dyck-CFL-reachability applies directly to this *context-insensitive* alias analysis.

4.1 Demand-driven Alias Analysis for Java

4.1.1 Symbolic Points-to Graph

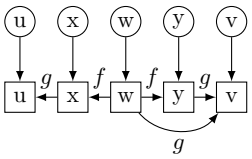
The underlying graph representation of the alias analysis is called the Symbolic Points-to Graph (SPG) [90, 92]. It extends the locally-resolved points-to graph representation [80] by introducing additional symbolic nodes as placeholders for abstract heap objects. The SPG contains three kinds of nodes: *variable nodes* $v \in \mathcal{V}$ representing variables, *allocation nodes* $o \in \mathcal{O}$ representing allocations for `new` expressions, and *symbolic nodes* $s \in \mathcal{S}$ representing abstract heap objects. It also consists of the following three types of edges:

- edges $v \rightarrow o_i \in \mathcal{V} \times \mathcal{O}$ to represent that variable v points to object o_i;
- edges $v \rightarrow s_i \in \mathcal{V} \times \mathcal{S}$ to represent that variable v points to an abstract heap object.
- edges $o_i \xrightarrow{f} o_j \in (\mathcal{O} \cup \mathcal{S}) \times \text{Fields} \times (\mathcal{O} \cup \mathcal{S})$ to represent that field f of o_i points to o_j.

A Java program's SPG is constructed in three steps. First, symbolic nodes are introduced for each procedure parameter, method invocation and field access. Second, the set of abstract

```
x = w.f;
w.f = y;
u = x.g;
v = y.g;
v = w.g;
```

(a) A code snippet.



(b) Its SPG.

Figure 4.1: An example of alias analysis with the SPG.

heap locations $\mathcal{O} \cup \mathcal{S}$ that a variable may point to[1] is computed. The relevant points-to edges are inserted to the SPG. Third, the field access edges $o_i \xrightarrow{f} o_j$ are added with regard to field loads and stores. The SPG also includes the barred edges (*i.e.* $o_j \xrightarrow{\bar{f}} o_i$ edges) implicitly.

4.1.2 Context-Insensitive Alias Analysis

The context-insensitive alias analysis computes the aliasing relation over variables within a method. In the analysis, the method invocation edges (*i.e.*, entry and exit edges) are of no interest. Specifically, the *memory aliasing* between the allocation or symbolic nodes that variable nodes x and y may points-to indicates the aliasing relation between x and y. The memory alias relation defined in [92] over $(\mathcal{O} \cup \mathcal{S}) \times (\mathcal{O} \cup \mathcal{S})$ is described by the

[1] In the original work that using SPG [90, 92], the *flowsTo* edges are used. A *flowsTo* edge is obtained on the flow graph by computing a regular language reachability. An abstract heap object *flowsTo* a variable if it is in the points-to set of that variable.

following context-free grammar:

$$\begin{aligned} memAlias \;\rightarrow\; & \bar{f}_1 \;\; memAlias \;\; f_1 \mid \ldots \mid \bar{f}_k \;\; memAlias \;\; f_k \\ & \mid memAlias \;\; memAlias \mid \varepsilon \end{aligned}$$

Note that the alias analysis based on *memAlias* reachability is a simplification of the *alias* reachability presented by Sridharan *et al.* [78, 80]. The field edges between abstract symbolic nodes in an SPG approximate the field loads and stores in the flow graph [78, 80]. The approximation may lead spurious aliasing as detailed by Xu *et al.* [90, Section 4]. However, the experimental results show that the overall performance is better than that proposed by Sridharan *et al.* [78, 80] in practice. The precision loss is insignificant enough compared to the performance gains.

Example 12 *Consider the example in Figure 4.1. The Java code snippet (left) and its SPG (right) are shown. In the SPG, the boxes denote symbolic nodes, and the circles denote variable nodes. The reverse edges (*a.k.a. *barred edges) are omitted for brevity. Note that the Dyck-CFL-reachability formulation used in the client alias analysis represents the barred edges as the opening parentheses. There are two pairs of memAlias nodes:* (x, y) *and* (u, v)*, because the realized strings of the two joining paths are "*$\bar{f}f$*" and "*$\bar{g}\bar{f}fg$*" respectively, which can be generated*

from the memAlias grammar. However, the node pair (x, v) *is not memAlias because the realized strings of two possible joining paths are "*$\bar{f}g$*" and "*$\bar{f}fg$*", such that the parentheses along the paths are not properly matched.*

4.1.3 Applying Our Fast Algorithms

Since the CFL used to describe the context-insensitive memory aliasing is a Dyck-CFL with k kinds of parentheses, the two Dyck-CFL-reachability algorithms presented in this paper can be directly applied. Note also that this alias analysis is demand-driven in the sense that the original algorithm solves the "single-source-single-target" Dyck-CFL-reachability problem, because solving "all-pairs" reachability is considered computationally much more expensive in these analyses. Both our algorithms are intended to solve the "all-pairs" Dyck-CFL-reachability problem. Next, we show how our "all-pairs" algorithm performs in practice.

4.2 Empirical Evaluation

In this section, we compare the traditional CFL-reachability algorithm with our proposed algorithm for solving the *all-pairs* Dyck-CFL-reachability problem on graphs for standard, real-

world Java benchmarks. The input graphs are generated from the *context-insensitive* alias analysis for Java described in Section 4.1. The results show that our algorithm outperforms the traditional CFL-reachability algorithm by several orders of magnitude.

4.2.1 Experimental Setup

Benchmark Selection. The benchmark suite used in our evaluation is the DaCapo suite [2]. We include the entire DaCapo-2006-10-MR2 suite which consists of 11 benchmarks with five additional large benchmarks form the DaCapo-9.12bach suite. Table 4.1 describes the benchmarks. For each benchmark, columns 2 and 3 list the numbers of methods and statements in intermediate representations of the underlying analysis infrastructure, respectively.

Graph Collection. We have used the same code as Xu *et al.* [90] and Yan *et al.* [92] to generate the Symbolic Points-to Graphs (SPGs). The analysis is built on top of the Soot program analysis framework for Java [84].

All benchmarks are processed with the nightly-build version[2] of Soot. To measure scalability, we use the latest release of JDK

[2]As of 2012-10-23.

1.6 (version 1.6u37) as the base analysis library for Soot. The five large benchmarks from DaCapo-9.12bach are processed with the help of Tamiflex [14] for reflection resolution.

Implementation. We implemented the proposed graph algorithm to compare with the traditional CFL-reachability algorithm. Both algorithms are implemented in C++ with extensive use of the Standard Template Library (STL). The FDLL data structure described in Section 3.5 is implemented using STL `unordered_map` and `list`. The underlying graphs are represented using adjacency lists implemented with FDLL.

Our code is compiled using gcc-4.6.3 with the "-O2" optimization flag. Both algorithms take the same SPG as input. Their outputs are verified to ensure the consistency and correctness . All experiments are conducted on a Dell Optiplex 780 machine with Intel Core2 Quad Q9650 CPU and 8 GB RAM, running Ubuntu-12.04.

4.2.2 Time and Memory Consumption

Table 4.2 shows the performance comparison of the two algorithms over our benchmark set. Column 4 and 5 list the numbers of nodes and edges in each SPG respectively. Column 6 lists the aliasing pair counts. Column 7 shows the number of

different kinds of parentheses (*i.e.*, the size of each Dyck grammar) in each SPG. The remaining columns list the time and memory consumption of the traditional CFL-reachability algorithm versus that of our algorithm. We denote our algorithm as FAST-DYCK.

The results indicate that our algorithm significantly improves over the traditional CFL-reachability algorithm. We observe that the running time of our algorithm grows very slowly *w.r.t.* the growth of the number of nodes. For example, the running time of the CFL-reachability algorithm on "jython09" is 30 times over that on "xalan06". While, it is only 4 times for our algorithm on the same benchmarks. We also note that our algorithm consumes less memory than the traditional CFL-reachability algorithm.

4.2.3 Discussion

Understanding the Asymptotic Behavior. The SPGs generated from the benchmarks are very sparse — there are fewer edges than nodes across all SPGs. This is expected for the client alias analysis and is consistent with the information in the original papers [90, 92]. For sparse graphs with $m = O(n)$, the asymptotic complexity of our algorithm is $O(n \log n)$.

Moreover, in the traditional CFL-reachability algorithm, the

Benchmark	**#Mtds**	**#Stmts**	**SPG**			
			#Nodes	**#Edges**	**#*S*-pair**	**#para**
antlr	9904	170402	16735	13878	19385	1087
bloat	11818	206857	20320	16224	23080	1197
chart	25184	448984	44584	36329	50670	2948
eclipse	10447	181101	17527	14411	20335	1182
fop	23643	431569	39977	31515	45837	2724
hsqldb	9177	156265	15015	12693	17615	998
jython	12802	216068	21615	17381	24487	1240
luindex	9668	164598	16098	13336	18716	1071
lusearch	10196	175354	17003	14195	19911	1117
pmd	11167	193375	18167	14958	20843	1168
xalan	9181	155180	15030	12645	17608	996
batik	22938	404097	40273	32052	46225	2565
eclipse	18741	354818	37531	31889	54471	2221
jython	41518	642242	63516	49005	85552	2855
sunflow	22346	385873	39321	31339	45161	2484
tomcat	25123	441606	45966	37338	63414	3013

Table 4.1: Benchmark programs.

grammar rules should be scanned for each iteration for inserting new summary edges. Specifically, for the Dyck language of size k, each edge popped from the worklist (line 7) in Algorithm 1 needs to be compared with the $O(k)$ rules in the given grammar. However, in our algorithms, the above is unnecessary. It takes *expected* $O(1)$ time to find a relevant edge labeled by a matched parenthesis in both our tree algorithm and graph algorithm due to the use of the FDLL.

Benchmark	Time		Memory	
	CFL	**fast-dyck**	**CFL**	**fast-dyck**
antlr	37.42	0.041	29.68	20.21
bloat	43.09	0.048	35.09	23.89
chart	253.06	0.119	76.75	52.02
eclipse	42.26	0.042	30.97	21.19
fop	219.53	0.101	67.99	46.08
hsqldb	33.39	0.038	27.10	18.22
jython	49.57	0.052	37.20	25.32
luindex	35.15	0.040	28.64	19.45
lusearch	40.22	0.043	30.34	20.73
pmd	40.28	0.046	32.00	21.90
xalan	32.93	0.038	26.93	18.21
batik	206.50	0.100	68.77	46.60
eclipse	366.39	0.103	70.82	44.54
jython	947.49	0.163	112.14	72.18
sunflow	196.23	0.096	67.22	45.57
tomcat	622.36	0.124	83.98	53.56

Table 4.2: Performance comparison: time in seconds and memory in MB.

Understanding the Memory Consumption. Both our algorithm and the traditional CFL-reachability algorithm demand moderate amount of memory for the client alias analysis. The memory cost for representing the input graphs in both algorithm is similar. The traditional CFL-reachability algorithm needs more iterations to compute the graph closure than those in our algorithm, therefore, it requires more space as well.

Note that we only used the cubic CFL-reachability algorithm (without applying the Four Russians' Trick) in our comparison. The subcubic CFL-reachability algorithm demands non-trivial

memory for storing the input graphs in our client application. For instance, given a medium-sized graph from our client analysis with 15000 nodes and 1000 parentheses, the subcubic algorithm needs about 26.2 GB memory to store the graph. It is an interesting topic to scale the subcubic CFL-reachability algorithm on real-world analysis.

Interpreting the Alias Analysis. In the *field-sensitive, context-insensitive alias analysis* for Java, the aliasing pairs are typically sparse. All benchmarks in our evaluation have $O(n)$ aliasing pairs (the #S-pair column in Table 4.2). This indicates that for real-world applications, most of the variables are not aliases. We have also observed from the experiments that the length of an aliasing path is small; almost all of the aliasing paths are simple paths without cycles. This observation is consistent with the state-of-the-art demand-driven analyses [80, 92, 97].

Demand-Driven vs. Exhausted. We now discuss perhaps one of the most interesting implications from our study. We have noticed that the performance of our all-pairs algorithm for *field-sensitive, context-insensitive* alias analysis is extremely fast. Such an exhaustive analysis with small time and memory cost is particularly suitable for application scenarios that need client analyses to be

able to respond instantly, such as just-in-time (JIT) optimizations and interactive development environments (IDEs). Compared to demand-driven analyses, our exhaustive alias analysis can answer any query in $O(1)$ time.

In practice, the two algorithms introduced in this paper can be combined to achieve better performance. For a connected component of the SPG encountered during analysis, it is straightforward to check whether the component is a graph or a tree by counting the number of nodes and edges. Furthermore, one can design an effective analysis switching between our tree and graph algorithms to achieve even better performance.

□ **End of chapter.**

Chapter 5

Fast CFL-Reachability Algorithms

5.1 Introduction

Programs written in C make extensive use of pointers. Determining pointer aliases is one of the fundamental static analysis problems, since alias information is usually a prerequisite for most subsequent analyses. Given two pointer variables, the general approach to alias analysis is to check whether the intersection of their points-to sets is non-empty [40]. Alias analysis for C has been formulated as a context-free language (CFL) reachability problem on Pointer Expression Graphs (PEGs) [97], with precision equivalent to an inclusion-based (*i.e.*, Andersen-style) points-to analysis [10]. The advantage of CFL-reachability-based alias analysis is that the alias information can be directly com-

puted without first obtaining each variable's points-to set.

In general, the CFL-reachability problem has several variants [67]. The *single-source-single-sink* variant concerns CFL-reachability between only two nodes, while the *all-pairs* variant considers CFL-reachability over all nodes in the graph. Solving *all-pairs* CFL-reachability is considerably more expensive than the *single-source-single-sink* variant. The traditional *all-pairs* CFL-reachability algorithm exhibits an $O(n^3)$ time complexity, where n denotes the number of nodes in the given graph [70, 93]. Consequently, straightforward implementations are ill-suited for handling large-scale applications in practice. Thus far, the key to scale the CFL-reachability-based alias analysis is to make the analysis *demand-driven*, aiming at solving the *single-source-single-sink* CFL-reachability problem [78, 80, 92, 97]. When the concerned CFL is restricted to a Dyck language, an improved analysis for Java solving *all-pairs* Dyck-CFL-reachability is proposed recently [95]. However, no *all-pairs* CFL-reachability-based alias analysis for C is known to date. Moreover, a subcubic CFL-reachability algorithm has been proposed [20], but its practical benefits remain unclear.

In this chapter, we present a highly scalable alias analysis for C. To the best of our knowledge, this is the first *all-pairs* CFL-

reachability-based alias analysis. Our principal contribution is an efficient algorithm that solves the *all-pairs* CFL-reachability problem formulated in an existing alias analysis for C using PEG [97]. The main novelty of our alias analysis algorithm is to compute the CFL-reachability summaries based on original edges in the graph and summary edges that describe only memory aliases, while the traditional CFL-reachability algorithm computes all summary edges. We also utilize the Four Russians' Trick [11] — a key enabling technique in the subcubic CFL-reachability algorithm [20] — in our alias analysis. We have implemented our subcubic alias analysis and conducted extensive experiments on the *latest stable* releases of widely-used C programs from the pointer analysis literature[1]. The results demonstrate that our alias analysis solving *all-pairs* CFL-reachability performs extremely well in practice. In particular, it can analyze the latest Linux kernel, which has over 10M source lines of code (SLOC), in less than 80 seconds.

Moreover, we also study the algorithmic complexity of flow- and context-insensitive inclusion-based pointer analysis on well-typed C. Within this domain, the precise pointer analysis problem is shown to be in **P** [17]. In this work, we follow the definition on the well-typeness introduced in the work of Chakar-

[1] As of March 2013.

avarthy [17]. We show that, for the well-typed C, there exist asymptotically faster inclusion-based pointer analysis algorithms.

To sum up, this chapter proposes two fast CFL-reachability algorithms for alias analysis using PEGs. Given a PEG with n nodes and m edges, we give an efficient algorithm for solving the *all-pairs* CFL-reachability in $O(n(m + M))$ time, where M denotes the number of memory alias pairs in the final graph. On average, our CFL-reachability algorithm is 2-3 orders faster than the traditional CFL-reachability algorithm on large real-world applications. When the input C program is restricted to be well-typed, we also present an algorithm with $O(n(m + \tilde{M}))$ time and $O(n^2)$ space complexities for processing, after which both points-to and alias analysis queries can be answered in $O(1)$ time, where $\tilde{M}$ denotes the maximum memory alias pairs on one layer. In the literature, the PEGs are typically quite sparse with $m = O(n)$, which implies that our graph algorithm has a quadratic time complexity in practice.

The rest of this chapter is structured as follows. Section 5.2 introduces the CFL-reachability formulation for alias analysis for C. Section 5.3 presents our subcubic alias analysis algorithm. Section 5.4 describes our alias analysis algorithm for well-typed

C.

5.2 The Zheng-Rugina Alias Analysis Formulation

Our algorithm solves the *all-pairs* CFL-reachability formulated by Zheng and Rugina [97] for *demand-driven* alias analysis for C. This section briefly reviews their formulation and discusses the advantages of using PEGs for alias analysis.

5.2.1 Pointer Expression Graphs

The input to our algorithms is a bidirected graph, known as a *Pointer Expression Graph* (PEG) [97]. A PEG represents the given C program in a canonical form that consists of sets of pointer assignments. The pointer analysis based on PEGs is flow-insensitive, therefore, control flow between pointer assignments is irrelevant. PEGs model the core C-style pointer language shown in Figure 5.1. PEGs also handle additional C language features (*e.g.*, arrays, structures, and pointer arithmetics), as detailed by Zheng and Rugina [97, Section 6.1].

There are three basic ingredients in the core language in Figure 5.1: memory addresses $a \in \mathit{Addresses}$, pointer expressions $e \in \mathit{Expressions}$ and pointer statements $s \in \mathit{Statements}$. Memory addresses model the symbolic addresses of variables, and can

$$
\begin{array}{lcl}
a \in \mathit{Addresses} & ::= & a_{var} \mid a_{heap} \\
e \in \mathit{Expressions} & ::= & *e \mid a \\
s \in \mathit{Statements} & ::= & *e_1 \coloneqq e_2
\end{array}
$$

Figure 5.1: Core syntax of C pointers

be obtained via either the address-of operator (*e.g.*, `&x`) or memory allocation (*e.g.*, `malloc()`), denoted as a_{var} and a_{heap} respectively. Pointer expressions model the behavior of the indirection operator (*e.g.*, `*x`) in C. Pointer variables are allowed by arbitrary pointer dereferences. Finally, pointer statements model program statements that manipulate pointers.

A PEG $G = (V, E)$ is a graph representation that depicts the canonical form of all pointer statements from the input C program. In a PEG, each node $v \in V$ represents a pointer expression e. A PEG also contains two kinds of edges:

- *Pointer dereference edges (d-edges)*: For each pointer deference $*e$, there is a directed edge from e to $*e$ labeled by d. Let the nodes representing e and $*e$ be u and v. We denote such labeled edges as $(u, d, v) \in E$.

- *Pointer assignment edges (a-edges)*: For each assignment statement $*e_1 \coloneqq e_2$, there is a directed edge from e_2 (as node u) to $*e_1$ (as node v) labeled by a. We denote it as $(u, a, v) \in E$.

$$M ::= \bar{d}\, V\, d \tag{5.1}$$
$$V ::= (M?\ \bar{a})^*\ M?\ (a\ M?)^* \tag{5.2}$$

Figure 5.2: CFL-reachability formulation of alias analysis for C.

For example, the top-level pointer variables are represented as nodes without outgoing d-edges, while the address-taken variables are represented as nodes without incoming d-edges and incoming a-edges. Specially, for each d-edge and a-edge in the PEG, there always exist a corresponding reverse edge labeled by $\bar{d}$ and $\bar{a}$ in the opposite direction, *i.e.*, $\forall(u, d, v), (u, a, v) \in E$, we have $(v, \bar{d}, u), (v, \bar{a}, u) \in E$. We call the corresponding edges $\bar{d}$-edges and $\bar{a}$-edges respectively. Moreover, we denote the set of d-edges and $\bar{d}$-edges as $\mathcal{D}$-edges. $\mathcal{A}$-edges are defined similarly. Note that the bidirectedness accomplished by introducing the reverse edges is a prerequisite for CFL-reachability-based formulations of pointer analysis [67].

5.2.2 Memory Aliases and Value Aliases

In a PEG, the alias analysis problem is formulated by the *CFG* shown in Figure 5.2, using EBNF notation. *CFG* can be represented using recursive state machines [9]. The equivalent recursive state machines of Zheng-Rugina formulation are adopted in Figure 5.3.

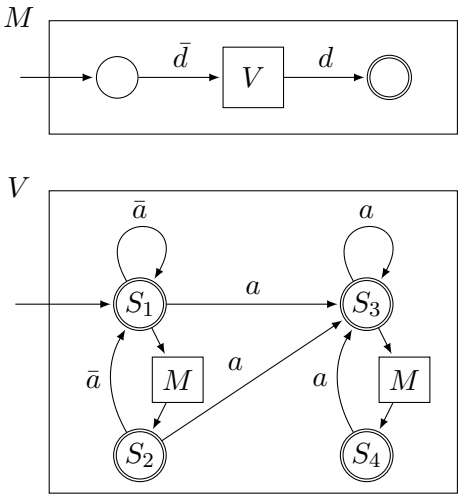


Figure 5.3: The recursive state machines.

The formulation distinguishes two kinds of aliases:

- Memory aliases (M): two pointer variables are memory aliases if they denote the same memory location.
- Value aliases (V): two pointer variables are value aliases if they are evaluated to the same pointer value.

According to the grammar, nodes u and v in the PEG are aliases if there exist an M-path or V-path between them. Moreover, the memory aliases and value aliases, represented as summary edges (u, M, v) and (u, V, v), can be considered a binary relation on all node pairs. Following the discussion by Zheng and Rugina [97], we summarize the properties on the M and V relations as follows:

- V is *nullable*, M is not *nullable*;

$$
\begin{array}{rcl}
M & ::= & DV\ d \\
DV & ::= & \bar{d}\ V \\
V & ::= & MAM\ \ AM_s \\
MAM & ::= & MA_s\ M_q \\
M_q & ::= & \varepsilon \\
M_q & ::= & M \\
MA_s & ::= & \varepsilon \\
MA_s & ::= & MA_s\ MA \\
MA & ::= & M_q\ \bar{a} \\
AM_s & ::= & \varepsilon \\
AM_s & ::= & AM_s\ AM \\
AM & ::= & a\ M_q
\end{array}
\qquad\qquad
\begin{array}{rcl}
M & ::= & DV\ d \\
DV & ::= & \bar{d}\ V \\
S_1 & ::= & S_1\ \bar{a} \\
S_2 & ::= & S_1\ M \\
S_1 & ::= & S_2\ \bar{a} \\
S_3 & ::= & S_2\ a \\
S_3 & ::= & S_1\ a \\
S_3 & ::= & S_3\ a \\
S_4 & ::= & S_3\ M \\
S_3 & ::= & S_4\ a \\
V & ::= & S_1 \\
V & ::= & S_2 \\
V & ::= & S_3 \\
V & ::= & S_4 \\
S_1 & ::= & \varepsilon
\end{array}
$$

Figure 5.4: Two normal forms (CFL1 and CFL2) of the CFL used in alias analysis for C.

- Both V and M are *symmetric*;
- V is *reflexive*, M is *reflexive* for non-address-taken variables;
- Neither V nor M is *transitive*.

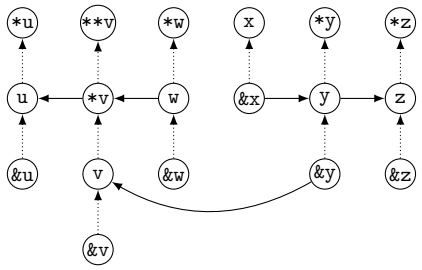
As aforementioned, the traditional CFL-reachability algorithm uses a "normal form" for the given *CFG* in Figure 5.2. We consider two such normal forms in Figure 5.4. In the figure, the form to the left (CFL1) is converted directly from the original grammar using a standard procedure for translating EBNF. For example, "$(M?\,\bar{a})^*\,M?$" is translated into the rule "$MAM ::= MA_s\ M_q$", where the subscripts s and q denote the star and question marks respectively. The form to the right (CFL2) is converted directly from the recursive state machines in Figure 5.3. For example, the state transition $\delta(S_1, a) = S_3$ is translated into the rule "$S_3 ::= S_1\ a$". Finally, we give an analysis example summarizing the discussions for illustration.

Example 13 *Figure 5.5 gives an example of alias analysis using the PEG. The C code snippet (left) and its PEG (right) are shown. In the PEG, the dotted edges represent the d-edges and the solid edges represent the a-edges. The reverse edges (*i.e.*, $\bar{a}$-edges and $\bar{d}$-edges) are omitted for brevity. In the PEG, nodes* `*v` *and* `y` *are memory aliases because the realized string $R(p)$ of*

```
int x;
int *u, *w, *y, *z;
int **v;

u = *v;
*v = w;
v = &y;
y = &x;
z = y;
```

(a) A code snippet.



(b) The corresponding PEG.

Figure 5.5: An example of pointer analysis with the PEG.

path $p = *\mathtt{v}, \mathtt{v}, \&\mathtt{y}, \mathtt{y}$ *is "*$\bar{d}\bar{a}d$*", which can be generated from* M *in Figure 5.2. Similarly, nodes* `u` *and* `&x` *are value aliases since the realized string "*$\bar{a}d\bar{a}d\bar{a}$*" can be generated from* V*. Note that* V *is not transitive. In the PEG, notes* `w` *and* `u`*, nodes* `u` *and* `&x` *are both value aliases. However, nodes* `w` *and* `&x` *are not value aliases since the realized string "*$ad\bar{a}d\bar{a}$*" can not be generated from* V*.*

5.2.3 Advantages of PEG

Alias analysis for C via CFL-reachability on PEGs has several advantages over traditional pointer analysis formulated as a dynamic transitive closure problem. We discuss some of the advantages below.

The most attractive feature is that PEGs depict the complex pointer assignments (e.g., `**x = ***y`) directly without introducing temporaries. As discussed in Section 2.2, traditional inclusion-

based pointer analyses have to transfer pointer statements into one of the four forms in Figure 2.7. The transformation also causes some precision loss, since it introduces additional points-to or alias pairs to the original program [13, 17, 41]. However, in the PEG representation, we do not need the temporaries. The pointer assignment is directly represented as an a-edge (u, a, v), where u and v represent `***y` and `**x` respectively.

In a PEG, the pointer variables are partitioned into different connected components. It is impossible for a variable in one connected component to be aliased with the variables from other connected components. For traditional inclusion-based pointer analysis formulated as a dynamic transitive closure problem, it is not straightforward to distinguish the connected components because new edges could be inserted to the graph during graph closure [34]. In the literature, there has been work that heuristically determines the components and performs analysis on each component independently [44, 96]. However, the idea of connected component decomposition on PEGs is quite natural and can be done using a simple depth-first search (DFS). Consequently, the CFL-reachability algorithm can work on each connected component of smaller size to achieve better performance.

Finally, the traditional approach to alias analysis is to perform

an inclusion-based points-to analysis and then check the intersection of every variable pair's points-to sets. Both points-to analysis and alias analysis have been formulated as CFL-reachability problems on PEGs. Specifically, the alias result can be computed independently, with precision equal to an inclusion-based points-to analysis [97].

5.3 Alias Analysis Algorithm

In this section, we present our alias analysis algorithm for C. Our algorithm takes PEGs as input and computes *all-pairs* CFL-reachability formulated by Zheng and Rugina [97]. We begin by illustrating the basic idea of our algorithm in Section 5.3.1. We then describe our technique for CFL-reachability summary propagation in Section 5.3.2, and give the main alias analysis algorithm in Section 5.3.3. Finally, we describe how to apply the Four Russians' Trick to our *all-pairs* CFL-reachability algorithm in Section 5.3.4.

5.3.1 Basic Idea

Our alias analysis algorithm for C is a worklist-based algorithm that follows the traditional dynamic programming scheme for solving *all-pairs* CFL-reachability. Each worklist item represents

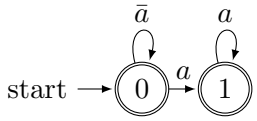
a reachability summary edge (u, X, v) between nodes u and v. The main algorithm exploits the following two facts from the original CFL-reachability formulation. Consider the summary edges describing memory aliases and value aliases respectively in the PEG,

Fact 1 *Each M-path is generated by prepending a $\bar{d}$-edge and appending a d-edge to a V-edge.*

Fact 2 *Each V-path is generated by a path whose $R(p) = \overline{a}^* a^*$, injected with zero or more non-consecutive M-paths.*

Fact 1 immediately follows from rule (5.1) in the grammar in Figure 5.2. In rule (5.2), if we substitute every occurrence of M with the empty string ε, V becomes a regular language $\bar{a}^* a^*$. Moreover, the M nonterminals injected in the V nonterminal are not consecutive because the M relation is not transitive. Therefore, they are separated by at least one $\bar{a}$-edge or a-edge, as described in Fact 2. According to the two observed facts, we introduce the following lemma on value aliases,

Lemma 3 *For each V-path joining two nodes representing non-top-level variables, there exist an M-path joining the nodes representing the corresponding dereferenced pointer variables.*



State \ Input	a	$\bar{a}$
0	1	0
1	1	$\times$

Figure 5.6: The finite automata used in the chain case.

Proof. Let the two nodes be u and v. Since neither of them is a top-level variable, the corresponding nodes representing dereferenced pointer variables always exist. Let the two nodes be u' and v', such that they are connected via d-edges (u, d, u') and (v, d, v') respectively. Following from Fact 1, if there exist (u, V, v), there must also exist (u', M, v'). □

When the input graph is a chain, we can use a stack-based algorithm to compute the *single-source-single-sink* reachability. Specifically, the matched parentheses d and $\bar{d}$ can be simulated using the stack. The parentheses are properly matched iff the stack is not any empty both during the processing and at the end of processing. For the regular language $\bar{a}^* a^*$ observed in Fact 2, the a and $\bar{a}$ symbols can be accepted using a finite automata shown in Figure 5.6. Note that due to the CFG in Figure 5.2, for every M-path $p = u, u', \ldots, v', v$, there always exists an enclosed V-path $p' = u', \ldots, v'$ such that $\mathcal{L}(u, u') = \bar{d}$ and $\mathcal{L}(v', v) = d$. Therefore, the finite automata in Figure 5.6 can be used on each layer recursively. As a result, the CFL-reachability for

Algorithm 10: Stack-based algorithm to compute the V-reachability between nodes u and v in the chain case.

Input : A path $p = (u, u_1, \ldots, v_1, v)$ from a tree T;
Output: true or false indicating if v is V-reachable from u;

```
1  w ← u and State [u] ← 0 ;
2  while w ≠ v do
3      x ← w and y ← Succ(x);
4      switch L(x, y) do
5          case d
6              if Stack is empty return false;
7              State [y] ← Stack.pop() ;
8              break;
9          case d̄
10             Stack.push(State [x]);
11             break;
12         case a ;
13         case ā
14             State [y] ← δ(State [x], L(x, y)) ;
15             if State [y] = × return false ;
16             break;
17     w ← y ;
18 if the Stack is empty return true , else, return false ;
```

pointer analysis can be computed by combining the stack and finite automata together.

The stack-based algorithm for computing the V-reachability between nodes u, and v in the chain case is given in Algorithm 10. For brevity, we omit the computation on Pt-reachability and M-reachability since they share the same insight. Algorithm 10 associates every node v in the path p with a state $\mathsf{State}[v]$. On the same layer, δ denotes the state transit of the finite automata in Figure 5.6. The stack is used to handle the properly matched

parentheses due to d and $\bar{d}$ edges. Specifically, when an opening parenthesis $\mathcal{L}(x, y) = \bar{d}$ is encountered, $\mathsf{State}[x]$ is pushed to the stack. When the matched closing parenthesis $\mathcal{L}(x, y) = d$ is encountered, the state information is restored to $\mathsf{State}[x]$ by popping the stack. Finally, v is V-reachable from u iff the stack is empty when v is reached, as described at line 18 in Algorithm 10.

The basic idea of our algorithm is to generate M and V reachability summaries *w.r.t.* the matched pairs of $\mathcal{D}$-edges in the PEG. To this end, our algorithm first propagates the reachability summaries to find the rightmost d-edge in each M-edge. Then, the reachability summaries are propagated in the opposite direction to find the leftmost matched $\bar{d}$-edge. We name this procedure as *two-phase propagation*. Specifically, for each memory alias summary edge (u, M, v) popped from the worklist, the reachability information is propagated along $\mathcal{A}$-edges and M-edges connected to nodes u and v (Fact 2). New summary edges representing value aliases are inserted to the worklist. Similarly, for each value alias summary edge popped from the worklist, we look for matched pairs of $\mathcal{D}$-edges using the *two-phase propagation* to generate new memory alias summary edge (Fact 1). When the memory alias result is obtained, the value aliases between non-top-level variables are obtained as well due

to Lemma 3. For top-level variables, we perform one additional *two-phase propagation* to compute the alias pairs.

It is important to note that for each summary edge (u, X, v) popped from the worklist, our algorithm computes the new CFL-reachability summaries based on the neighbors of u and v in the original graph and the neighbors connected by edges that describe memory aliases in the current graph. However, the traditional CFL-reachability algorithm considers the neighbors connected by more summary edges in the current graph. Next, we give a concrete example to illustrate the basic idea.

Example 14 *Consider the PEG in Figure 5.5. We describe the major steps to compute the memory aliases between nodes* $*\mathtt{v}$ *and* $\mathtt{y}$ *as below. The summary edge popped from the worklist is* $(\mathtt{v}, M, \mathtt{v})$*. In phase one, the reachability information is propagated to find the "rightmost"*[2] *d-edge* $(\mathtt{v}, d, *\mathtt{v})$*. Then phase two propagation starts. After the "leftmost"* $\bar{d}$*-edge* $(\mathtt{y}, \bar{d}, \&\mathtt{y})$ *is encountered, the summary edge* $(\mathtt{y}, M, *\mathtt{v})$ *representing the memory aliases is inserted into the PEG.*

[2]The left and right are relative to summary edge $(\mathtt{y}, M, *\mathtt{v})$ rather than the actual left and right in Figure 5.5.

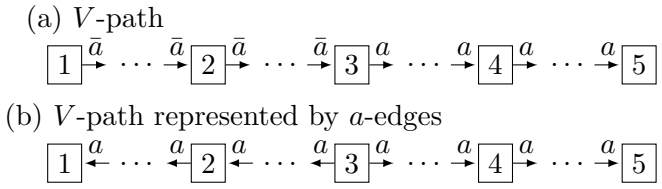


Figure 5.7: The positions of M in V.

Step	***Current edge***	***New edge***	***Phase***
1	—	$(\mathtt{v}, M, \mathtt{v})$	—
2	$(\mathtt{v}, M, \mathtt{v})$	$(\mathtt{v}, D'_1, *\mathtt{v})$	*Phase one*
3	$(\mathtt{v}, D'_1, *\mathtt{v})$	$(\&\mathtt{y}, D_1, *\mathtt{v})$	*Phase two*
4	$(\&\mathtt{y}, D_1, *\mathtt{v})$	$(\mathtt{y}, M, *\mathtt{v})$	*Phase two*

5.3.2 Propagating CFL-Reachability Summaries

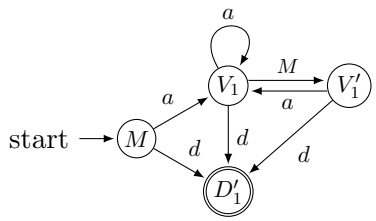
The above *two-phase propagation* focuses on M-edges and $\mathcal{A}$-edges, since the matched pair of $\mathcal{D}$-edges are the two endpoints for the propagation. We first discuss the relation between M-edges and $\mathcal{A}$-edges in a V-path. Due to Fact 2, the realized string of a V-path can be considered as $\bar{a}^* a^*$ without any M-edges. Although M may be injected in the various positions within V, all positions can be distinguished as five unique positions described in Figure 5.7(a). Since the PEG is bidirected with reverse edges, Figure 5.7(b) represents the same V-path using only a-edges. Moreover, positions 1 and 5 are two endpoints of

the V-path. If there exist additional edges, it must be a matched pair of $\mathcal{D}$-edges to continue the propagation.

Let us now consider the M-edges (u, M, v) in each of the five positions. Note that due to the reflexivity of M on non-address-taken variables, for each edge (u, M, v), we have (v, M, u) for all u and v. For the address-taken variables, we represent the M-edges implicitly in the worklist to start the propagation without inserting them to the graph. When the M-edge (u, M, v) is in positions 3 and 4, the reachability information can always be propagated though node v to position 5 via a-edges and M-edges. Position 5 indicates that the first phase propagation is completed, since it is one endpoint of the V-path. When edge (u, M, v) is in position 2, we can use the reflexive edge (v, M, u) to propagate the reachability summary though node u to position 1 via a-edges and M-edges. Position 1 is exactly the same as position 5 if the reflexive edge (v, M, u) is considered. We summarize the above discussion as the following lemma:

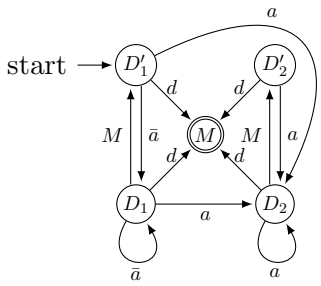
Lemma 4 *For each edge (u, M, v), it suffices to consider the a-edges of u or v to initiate the first phase propagation.*

It is important to note that the five positions in Figure 5.7 describe both M positions in state machine V in Figure 5.3. Specifically, the left M position in state machine describes M



Input State	a	d	M
M	V_1	D'_1	$\times$
V_1	V_1	D'_1	V'_1
V'_1	V_1	D'_1	$\times$

Figure 5.8: Phase one propagation.



Input State	a	$\bar{a}$	d	M
D'_1	D_2	D_1	M	$\times$
D_1	D_2	D_1	M	D'_1
D'_2	D_2	$\times$	M	$\times$
D_2	D_2	$\times$	M	D'_2

Figure 5.9: Phase two propagation.

in the language $\bar{a}^*$, which is depicted by positions 1, 2 and 3 in Figure 5.7. Similarly, the right M position describes M in the language a^* following $\bar{a}^*aa^*$, which is depicted by positions 4 and 5. Without loss of generality, we explain the two-phase propagation *w.r.t.* the five positions in Figure 5.7.

Phase One Propagation. In this phase, we use the finite state machine in Figure 5.8 to propagate the reachability summary for each M-edge (u, M, v). If one of the nodes u and v is the endpoint in the V-path, the phase two propagation starts immediately using the other node. Otherwise, due to the reflexivity of M and Lemma 4, let v be the node with an outgoing neighbor v',

such that $\mathcal{L}(v, v') = a$. In the first phase, the CFL-reachability summary is propagated to the right. According to positions 3, 4 and 5 in Figure 5.7, node v' may encounter arbitrary a-edges and M-edges to the right during the propagation. We depict it as state V_1 in Figure 5.8. Moreover, one additional state V_1' is needed to respect the fact that the Ms are non-consecutive (Fact 2). Finally, when the rightmost d-edge is encountered, states M, V_1 and V_1' transit to D_1' and phase-two propagation starts.

Phase Two Propagation. In this phase, we use the finite state machine in Figure 5.9 to propagate the reachability summary in the opposite direction for each D_1'-edge (u, D_1', v). Similarly, according to positions 3, 4 and 5 in Figure 5.7, node u may encounter some $\mathcal{A}$-edges and M-edges to the left during the propagation. Specifically, due to both the symmetry of V and Fact 2, all $\mathcal{A}$-edges encountered to the left can be described by $\bar{a}^* a^*$. Therefore, two states D_1 and D_2 are required to accept the regular language. As before, two additional states D_1' and D_2' are required to respect the fact that the Ms in V are non-consecutive. Finally, when the leftmost $\bar{d}$-edge is encountered[3], a new M-edge is generated and the two-phase propagation completes.

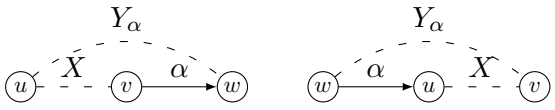
[3]The leftmost $\bar{d}$-edge is treated as a d-edge when the right-to-left direction is considered.

5.3.3 Alias Analysis Algorithm

The main algorithm for computing *all-pairs* memory aliases is given in Algorithm 11. It is a worklist-based algorithm that follows the traditional dynamic programming scheme for solving *all-pairs* CFL-reachability. The algorithm takes a PEG as input, and proceeds in two major steps:

- *Initialization.* The worklist W is initialized on lines 1-4. All nodes are considered. The non-address-taken node u has an incoming d-edge (u, d, v). Due to the reverse edges, the realized path $R(p)$ for $p = u, v, u$ is $\bar{d}d$, which describes a memory aliases location. The resulting M-edge is inserted into the graph. Note that the M-edges for address-taken variables in the initialization phase do not need to be inserted to the PEG explicitly.

- *Reachability summary propagation.* When a reachability summary edge (u, X, v) popped from the worklist, the reachability information is propagated using the *two-phase propagation*. Specifically, we use the FIND-TRANSITION procedure to look for the relevant transitions in the corresponding phase. For example, in the phase one propagation on lines 8-15, for each outgoing neighbor w of v connected via edge



(a). Phase one. (b) Phase two.

Figure 5.10: Adding summary edges in two phases.

(v, α, w), the FIND-TRANSITION procedure returns state Y_α according to the transition table in Figure 5.8. The summary edge (u, Y_α, w) is then inserted to the PEG depicted in Figure 5.10. The phase-two propagation on lines 17-24 is handled similarly.

The algorithm terminates when the worklist W is empty. All summary edges describing memory aliases are presented in the final PEG.

Computing Value Aliases. Based on the memory alias result, we can compute the value alias reachability by reusing some of the summary edges in the current graph. Due to Lemma 3, the value aliases between any non-top-level variables can be obtained by removing the matched $\mathcal{D}$-edge pair of any existing M-path. In order to compute the value aliases for top-level variables, we can perform an additional two-phase propagation if one of the endpoints is a top-level variable. The algorithm is exactly the same. Due to the space constraints, we omit the value alias

Algorithm 11: Computing Memory Aliases.

Input : PEG $G = (V, E)$;
Output: the set of summary edges;

```
1  foreach v ∈ V do
2      insert (v, M, v) to W;
3      if v has incoming d-edges then
4          insert (v, M, v) to G;
5  while W ≠ ∅ do
6      (u, X, v) ← SELECT-FROM(W) ;
7      /* Phase 1 propagation.                                   */
8      if X = M or X = V_1 or X = V_1' then
9          foreach α ∈ {a, d, M} do
10             Y_α ← FIND-TRANSITION(X, α) ;
11             if Y_α == × then continue;
12             foreach w ∈ OUT(v, α) do
13                 if (u, Y_α, w) ∉ G then
14                     insert (u, Y_α, w) to G and to W ;
15                     if Y_α == M then insert (u, Y_α, w) to G and to W ;
16     /* Phase 2 propagation.                                   */
17     if X = D_1 or X = D_1' or X = D_2 or X = D_2' then
18         foreach α ∈ {a, ā, d, M} do
19             Y_α ← FIND-TRANSITION(X, α) ;
20             if Y_α == × then continue;
21             foreach w ∈ IN(u, α) do
22                 if (w, Y_α, v) ∉ G then
23                     insert (w, Y_α, v) to G and to W ;
24                     if Y_α == M then insert (w, Y_α, v) to G and to W ;
```

algorithm.

Complexity Analysis. The worst-case complexity of Algorithm 11 is $O(n(m + M))$, where M denotes the number of M-edges, and n and m denote the numbers of nodes and edges in the original graph respectively. For each summary edge (u, X, v), Algo-

rithm 11 traverses its neighbors connected via $\mathcal{A}$-edges, $\mathcal{D}$-edges and M-edges. Let k and Δ_v denote the grammar size and the degree of node v concerning these three kinds of edges respectively. From Figure 5.8 and Figure 5.9, we have $k = 7$ since there are 7 kinds of summary edges. The total number of steps required are $7 \cdot \Sigma_{(u,v)}\Delta_v = 7 \cdot \Sigma_u(\Sigma_v\Delta_v)$. Therefore, the worst-case time complexity is $O(n(m + M))$.

Connected Component Decomposition. The worst-case time complexity depends on the number of nodes in the PEG. Therefore, in practice, we can reduce n by decomposing the original PEG into connected components. Since the nodes in two connected components are unreachable, computing the reachability on those smaller components yields the same results as computing on the original PEG. The connected component decomposition can be done using a simple linear-time depth-first search on the PEG. We further investigate the practical benefits that this optimization brings in the evaluation section.

5.3.4 Saving a Logarithmic Factor

The Four Russians' trick [11] is known as a popular technique for speeding up set operations under the random access machine model with uniform cost criterion. The original paper proposed

an $O(\log\ d)(n^3/\log\ n)$ algorithm for finding the transitive closure of a directed graph with n nodes and diameter d. The technique has been applied in various contexts. Examples include shortest path problems [18], Boolean matrix multiplication [5, Ch. 6], and k-clique problems [85], to name just a few.

In particular, this technique has also been adopted for fast recognition of context-free languages [73] as well as reachability problems in recursive state machines [20], resulting in a logarithmic speedup. We can apply this technique to Algorithm 11 directly. We first recall some preliminaries. We begin by assuming the RAM model has word length $\theta(\log\ n)$ and constant-time bitwise operations on words. Let U denote a universe of n elements. A subset of U can be represented as a bit vector (*a.k.a.* characteristic vector) of length n by representing each element as a single bit. The characteristic vector is then stored in $O(\lceil n/\log\ n\rceil)$ words each with $\theta(\log\ n)$ bits. Following the work of Chaudhuri [20], we refer to the resulting data structure as *fast set*, which permits the following two operations:

- *insert(X, i)*: insert an element i into *fast set* X.
- *diff(X, Y)*: compute the set difference between *fast set* X and Y and return a list of all the resulting elements.

Lemma 5 *Given* fast sets $X, Y \subseteq \{1, \ldots, n\}$, *and* $i \in \{1, \ldots, n\}$,

(i) insert *(*X*,* i*) takes* $O(1)$ *time;*

(ii) diff *(*X*,* Y*) takes* $O(\lceil n/\log n \rceil + v)$ *time, where* v *is the number of elements in the result set.*

Proof. (i) is obvious by determining the position of i in relevant word of X and then performing the bitwise or operation. (ii) follows in two steps. First, we perform the bitwise operations on the words comprising X and Y, resulting in $Z = X \setminus Y$. This takes $O(\lceil n/\log n \rceil)$ time under the assumed RAM model. Then, we list all the elements in Z by repeatedly finding and turning of the most significant bit, this takes time $O(v)$, where v is the number of elements. If this operations are not directly supported, we can precompute the answers to all words (or pairs of words) with $O(n)$ preprocessing time and subsequently perform table lookups. □

In our algorithm, we can represent the In and Out sets using *fast sets*. Therefore, lines 12-13 in Algorithm 11

$$\textbf{foreach } w \in \textsc{Out}(v, \alpha) \textbf{ do}$$
$$\textbf{if } (u, Y_\alpha, w) \notin G \textbf{ then}$$

can be changed using the *fast-sets* operations as

$$\textbf{foreach } w \in \mathit{diff}(\textsc{Out}(v, \alpha), \textsc{Out}(u, Y_\alpha)) \textbf{ do}.$$

Similarly, lines 21-22 can be changed to

$$\textbf{foreach } w \in \mathit{diff}(\textsc{In}(u, \alpha), \textsc{In}(v, Y_\alpha)) \textbf{ do}.$$

The new algorithm takes $O(n/\log n)$ time to traverse the three kinds of edges for each node n. As a result, the total time complexity is $O(n(n \cdot n/\log n)) = O(n^3/\log n)$.

5.4 Well-Typed Alias Analysis Algorithm

In this section, we describe the algorithm for solving the well-typed pointer analysis problem on PEGs. Let $\tilde{M}$ denote the maximum memory alias pairs on one layer. Given a well-typed PEG with n nodes and m edges, our algorithm processes the graph in $O(n(m + \tilde{M}))$ time with $O(n^2)$ space, after which any points-to and alias query can be answered in $O(1)$ time.

Our algorithm first pre-processes the input PEG in $O(m)$ time and collects necessary information (*e.g.*, layer information and pointer deference information) which is required by the main algorithm. As discussed in Section 5.3.1, the realized string $R(p)$ eliminating M-edges for each V-path is $\bar{a}^* a^*$, which is a regular language. We describe the finite state automate for regular language $\bar{a}^* a^*$ in Figure 5.6. The high-level idea of our main pointer

analysis algorithm is to compute the all-pairs points-to and alias reachability layer by layer, in a bottom-up manner. In the following sections, we begin by introducing the pre-processing pass. Then, we describe the algorithm for handling bottom-layer variables. With bottom-layer reachability computed, we further illustrate the main algorithm for the whole PEG by connecting reachability information of two adjacent layers. Finally, we give the complexity and correctness analysis.

5.4.1 Pre-Processing

The pre-processing pass is actually an $O(m)$ time graph traversal procedure starting at an arbitrary node. The procedure achieves the goals as follows:

- *Obtaining layer information.* For the starting node u, we safely assign it to layer $|n|$, *i.e.*, $l(u) = |n|$. For any edge (u, d, v) encountered during the graph traversal, we assign it to layer $l(v) = l(u) + 1$. Similarly, we assign node v to layer $l(v) = l(u) - 1$ according to edge $(u, \bar{d}, v)$. The layer information remains the same for any $\mathcal{A}$-edge or $\bar{\mathcal{A}}$-edge. After the traversal completes, the layer information $l(v)$ is then adjusted to range from 0 to k, denoted as *bottom* and *top*, respectively.

- *Testing well-typeness.* If the PEG is not well-typed, there exist some cross-layer edges that make the layer information inconsistent, *i.e.*, $l(u) \neq l(v)$ for some $\mathcal{A}$-edges or $\bar{\mathcal{A}}$-edges, $l(v) \neq l(u)+1$ for some $\mathcal{D}$-edges, or $l(u) \neq l(v)+1$ for any $\bar{\mathcal{D}}$-edges. The pre-processing procedural returns immediately if the PEG is not well-typed.

- *Mapping node information.* During the graph traversal, we construct two hash tables H_e : *Expressions* $\rightarrow n$ and H_a : *Addresses* $\rightarrow n$ for mapping each pointer expression to the corresponding node in the PEG. Moreover, each node v is assigned to set $V[l(v)]$ and each edge $(u, \bar{d}, v)$, (u, a, v) or $(u, \bar{a}, v)$ is assigned to set $E[l(v)]$.

5.4.2 Handling Bottom-Layer Variables

After pre-processing, the nodes in the input PEG are stored using disjoint sets $V[k]$, where k denotes the layer information. Our algorithm starts with nodes at the bottom layer (*i.e.*, $v \in V[0]$). We compute the all-pairs reachability among bottom-layer variables by formulating it as an incremental *Dynamic path problem* [16, 43]. Specifically, an incremental *Dynamic path problem* instance starts with an empty graph that undergoes a sequence of edge insertions. Note that the order of edges to be inserted

is completely arbitrary. For the bottom layer, there are only $\mathcal{A}$-edges and $\bar{\mathcal{A}}$-edges among nodes. Our algorithm handles each edge insertion in *amortized* $O(n)$ time.

Key Data Structures

In order to cope with the finite automata in Figure 5.6, we pair each node $v \in V$ with a state $q \in \{0, 1\}$, denoted as node v_q. Each edge label $\mathcal{L}(u, v)$ in the PEG corresponds to a one-step state transition from state r in u_r to state q in v_q, *i.e.*, $\delta(r, \mathcal{L}(u, v)) = q$. Specifically, the state transitions between u_r and v_q *w.r.t.* $\mathcal{A}$- and $\bar{\mathcal{A}}$-edges are depicted in Table 5.1. We say node v_q is reachable from u_r iff there exists a path $p = u_r, w_i, x_j \ldots, y_k, z_l, v_q$ such that $\delta(r, \mathcal{L}(u, w)) = i$, $\delta(i, \mathcal{L}(w, x)) = j$, ..., $\delta(k, \mathcal{L}(y, z)) = l$, $\delta(l, \mathcal{L}(z, v)) = q$.

The main algorithm operates on two key data structures: an $2n \times 2n$ reachability matrix $\boldsymbol{M}$ and a *reachability spanning tree* $T(v_q)$ associated with each node v_q. The reachability matrix $\boldsymbol{M}$ depicts whether two nodes u_r and v_q are reachable. For nodes u_r and v_q, the entry $m_{u_r v_q}$ in $\boldsymbol{M}$ is defined as follows:

$$m_{u_r v_q} = \begin{cases} 1, & v_q \text{ is reachable from } u_r, \\ 0, & \text{otherwise.} \end{cases}$$

Procedure 12: Init() to initialize the key data structures.

```
1 for u ← 1 to n do
2     initialize T(u_1) and T(u_2) as two empty trees
3     for v ← 1 to n do
4         m_{u_q v_r} ← 0, where q, r ∈ {0, 1}
5         m_{u_0 v_0} ← 1 and m_{u_1 v_1} ← 1
```

On the other hand, the *reachability spanning tree* $T(v_q)$ keeps a list of u_r that are reachable from v_q. Formally, the trees are defined as follows:

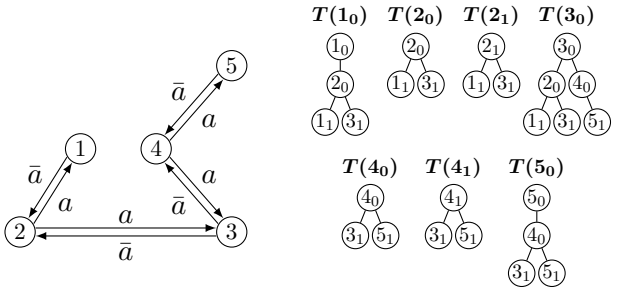
(1) the root node in $T(v_q)$ does not have any incoming edge, and each of all non-root nodes has exactly one incoming edge;

(2) a node u_r is in $T(v_q)$ iff $m_{v_q u_r}$ is 1;

(3) in any spanning trees $T(v_q)$, node w_s is a descendant of node u_r only if $m_{u_r w_s}$ is 1.

The key data structures are initialized by Procedure 12. The running time required for the `init()` procedure is clearly $O(n^2)$. However, the time complexity can be easily reduced to $O(n)$ by initializing each matrix entry $m_{u_r v_q}$ the first time when it is accessed [5, pp. 71].

Example 15 *Let us consider the example in Figure 5.11. Figure 5.11(a) shows the input PEG, and Figure 5.11(b) shows all non-empty reachability spanning trees. In the graph, there is a*



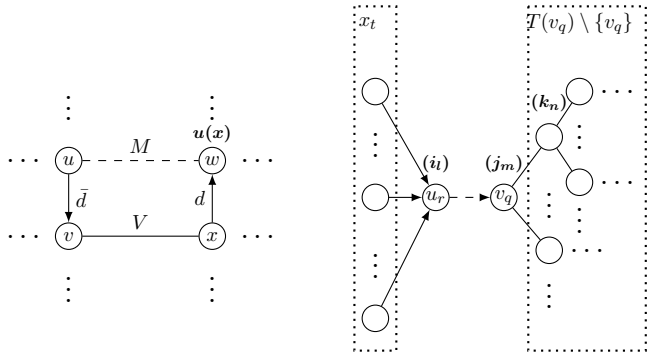
(a) The input PEG. (b) The reachability spanning trees.

Figure 5.11: An example PEG with bottom-layer variables and corresponding reachability spanning trees.

V-path from node 1 *to* 3 *with realized string* $\bar{a}a$*. Therefore, node* 3_1 *is in* $T(1_0)$ *indicating that* 3_1 *is reachable from* 1_0*.*

Main Procedures

The processing of each edge insertion is handled by `Add()` in Procedure 13 and `Mix()` in Procedure 14. In particular, procedure `Add()` determines the reachability entries $m_{u_r v_q}$ and reachability trees $T(x_t)$ to be updated. And procedure `Mix()` is a recursive procedure that performs the actual updating. The *flag* marks whether there is an M-path connecting x_t and j_m. For the bottom layer, the *flag* is always set to be 0 since there is no M-path. Note that array V_M and routine ADJUST() in the two procedures are used for handling two adjacent layers to be considered in Section 5.4.3. We can safely discard their impact when discussing the bottom layer.



(a) Updating reachability information.

(b) Nodes processed by procedure `Mix`().

Figure 5.12: Updating reachability information among bottom-layer variables.

The outcome of inserting a new edge (u, a, v) or $(u, \bar{a}, v)$ impacts on both reachability matrix entries $m_{u_r v_q}$ and the corresponding reachability trees $T(x_t)$. First, according to the label $\mathcal{L}(u, v)$ of the edge, the corresponding reachability matrix entries $m_{u_r v_q}$ need to be updated *w.r.t.* Table 5.1. Furthermore, updating the $m_{u_r v_q}$ affects the corresponding reachability trees $T(x_t)$ as described in Figure 5.12(a). Specifically, the updated matrix entry $m_{u_r v_q}$ may cause any node x_t that reaches u_r possibly reaches some nodes w_t in $T(v_q)$. On line 3, procedure `add`() employs a routine SearchTable5.1() to determine the $m_{u_r v_q}$ and $T(x_t)$ *w.r.t.* Table 5.1. Taking Figure 5.11(a) as an example, inserting an edge $(1, a, 4)$ updates the matrix entries $m_{1_1 4_1}$ and $m_{1_0 4_1}$. In the PEG, node 2_0 previously reaches 1_1. Therefore, 2_0 should reach all nodes in $T(4_1)$.

Procedure `Add`$(u, \mathcal{L}, v)$ then calls `Mix`(x_t, v_q, i_l, j_m) to update the

Edge inserted	$m_{u_r v_q}$ updated	$T(x_t)$ affected
(u, a, v)	$m_{u_1 v_1}$	$T(x_0)$
	$m_{u_0 v_1}$	$T(x_0)$
	$m_{u_1 v_1}$	$T(x_1)$
$(u, \bar{a}, v)$	$m_{u_0 v_0}$	$T(x_0)$

Table 5.1: Reachability information updated according to $\mathcal{A}$- and $\bar{\mathcal{A}}$-edges.

key data structures, where x_t denotes the node that reaches u_r in the PEG. Procedure `Mix`(x_t, v_q, i_l, j_m) recursively searches $T(v_q)$ *w.r.t.* edge (i_l, j_m) and updates the new reachability information between nodes x_t and $j_m \in T(v_q)$ by pruning a unique copy of $T(v_q)$ and inserting it into $T(x_t)$. Specifically, it involves the following two steps:

- recursively pruning a unique copy of $T(v_q)$ by eliminating the nodes that are already in $T(x_t)$ (lines 4-5);
- linking the nodes in the unique copy of $T(v_q)$ to $T(x_t)$ and updating the reachability matrix (lines 2-3).

In the subsequent recursive calls, i_l represents the parent of j_m in $T(v_q)$, and every child k_n of j_m is considered. If node k_n is already reachable from x_t (*i.e.*, $m_{x_t k_n} = 1$), the procedure `Mix`() returns since all children of k_n are reachable from x_t (*i.e.*, they are also in $T(x_t)$).

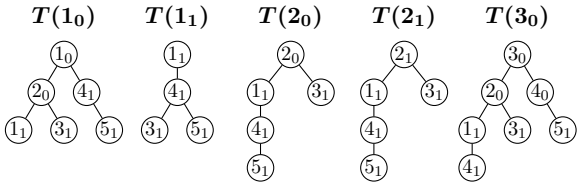
Example 16 *Figure 5.13 gives the result of inserting an edge* $(1, a, 4)$ *to the PEG in Figure 5.11. The pruned copies of* $T(4_1)$

Procedure 13: Add$(u, \mathcal{L}, v)$ to insert an edge $(u, \mathcal{L}, v)$.

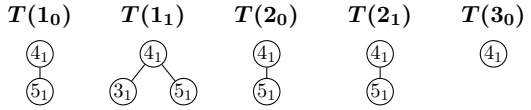
```
1  foreach x ∈ V[l(v)] do
2      if L(u,v) == a or L(u,v) == ā then
3          x_t, u_r, v_q ← SearchTable5.1(L(u,v))
4          if m_{x_t u_r} == 1 then
5              if m_{x_t v_q} ≠ 1 then Mix (x_t, v_q, u_r, v_q, 0)
6              if v_q ∈ V_M[x_t] then
7                  Adjust(x_t, u_r, v_q)
8                  Mix (x_t, v_q, u_r, v_q, 1)
9      if L(u,v) == d̄ then
10         foreach q ∈ {0,1}, r ∈ {0,1}, s ∈ {0,1} do
11             if m_{v_q x_r} == 1 and u(x) exists then
12                 w ← u(x)
13                 if m_{u_s w_s} ≠ 1 then
14                     V_M[u_s] ← V_M[u_s] ∪ {w_s}
15                     insert w_s as a child of T(u_s)
16                     m_{u_s w_s} ← 1
```

Procedure 14: Mix$(x_t, v_q, i_l, j_m, flag)$ to merge trees.

```
1  if flag == 0 then
2      insert j_m in T(x_t) as a child of i_l
3      m_{x_t j_m} ← 1
4      foreach child k_n of j_m ∈ T(v_q) do
5          if m_{x_t k_n} ≠ 1 then Mix (x_t, v_q, j_m, k_n, 0)
6          if k_n ∈ V_M[x_t] then
7              Adjust(x_t, j_m, k_n)
8              Mix (x_t, v_q, j_m, k_n, 1)
9  if flag == 1 then
10     foreach k_n ∈ V_M[j_m] do
11         if m_{x_t k_n} ≠ 1 then Mix (x_t, v_q, j_m, k_n, 0)
12         if k_n ∈ V_M[x_t] then
13             Adjust(x_t, j_m, k_n)
14             Mix (x_t, v_q, j_m, k_n, 1)
```



(a) New reachability trees.



(b) The pruned unique copy of $T(4_1)$ for each x_t.

Figure 5.13: The updated reachability spanning trees after inserting $(1, a, 4)$ in Figure 5.11.

by procedure `Mix`() *are given in Figure 5.13(b). Then, each* $T(x_t)$ *is updated by inserting the corresponding* $T(4_1)$*. Finally, all updated reachability trees* $T(x_t)$ *are given in Figure 5.13(a).*

5.4.3 Main Algorithm: A Bottom-Up Approach

Having the reachability among bottom-layer variables computed, this section presents the main pointer analysis algorithm. Given a well-typed PEG, our algorithm propagates both alias and points-to reachability information from bottom layer to top layer.

Connecting Layers

In the PEG, the layers are connected via $\mathcal{D}$- and $\bar{\mathcal{D}}$-edges. Let us safely assume that the reachability summaries on layer $k-1$ has been computed. We then focus on propagating the summaries to the upper layer k.

The key idea for connecting two layers on the PEG is to generate new M-paths at layer k *w.r.t.* any old V-paths at layer $k-1$. On the same layer, any two reachable nodes are joined via only three kinds of paths, *i.e.*, V-, M- and Pt-paths. The CFG in Figure 5.2 ensures that both nonterminals M and Pt are derivable from nonterminal V, *i.e.*, there exists (u, V, v) for all $(u, M, v), (u, Pt, v)$. Therefore, for each summary edge (u, V, v) at layer $k-1$, there should be a new summary edge (u', M, v') at layer k if both u' and v' exist and $\mathcal{L}(u, u') = \mathcal{L}(v, v') = d$. In our algorithm, we store such summary edge (u', M, v') using array V_M, *i.e.*, $v' \in V_M[u']$ for all (u', M, v').

The `Add()` procedure in Procedure 13 propagates the reachability summaries from layer $k-1$ to layer k by taking advantage of the $\bar{\mathcal{D}}$-edges (*i.e.*, $(u, \bar{d}, v)$, where $l(u) = k$ and $l(v) = k-1$) connecting any two adjacent layers. Specifically, on lines 9-16, the procedure searches each node x that are reachable from v at layer $k-1$. If the dereferenced node w of x exists at layer k (*i.e.*, $(w, \bar{d}, x)$), it is immediate that the new path $p = u, v, \ldots, x, w$ is a new M-path at layer k. Finally, procedure `Add()` updates the corresponding matrix entries and spanning trees.

The memory alias relation (*i.e.*, M-edge) is not transitive. On lines 10-14 of `Mix()` procedure, we set *flag* $= 1$ to cope with

the non-transitivity. Note that the `Add()` and `Mix()` procedures insert a new node j_m to the reachability spanning tree $T(x_t)$ iff there is a new $\mathcal{A}$- or $\bar{\mathcal{A}}$-edge processed. As a result, the two procedures never introduce consecutive M-edges to the reachability spanning trees. On the other hand, if x_t reaches j_m via an M-path, x_t does not reach the sub trees rooted at k_n where j_m reaches k_n via an M-path. However, the newly processed $\mathcal{A}$- or $\bar{\mathcal{A}}$-edge makes k_n reachable from x_t. We handle this by using array V_M and routine ADJUST(). Specifically, array $V_M[x_t]$ keeps all nodes v_q that are M-reachable from x_t. Routine ADJUST(x_t, u_r, v_q) moves v_q to be a child of u_r in $T(x_t)$ and remove v_q from $V_M[x_t]$, it is called only if there is a new $\mathcal{A}$- or $\bar{\mathcal{A}}$-edge processed. Consequently, $v_q \in T(x_t)$ iff there is a V-path joining them.

Pointer Analysis Algorithm

Our pointer analysis algorithm for well-typed PEG is given in Algorithm 15. It takes a well-typed PEG $G = (V, E)$ as input and outputs the reachability matrix for answering any points-to or alias analysis query.

The functioning of the main algorithm proceeds as follows. On line 1, the main algorithm pre-processes the input PEG and

Algorithm 15: Pointer analysis algorithm for well-typed C.

Input : Edge-labeled bidirected PEG $G = (V, E)$;
Output: the reachability matrix $\boldsymbol{M}$

```
1 run pre-process pass described in Section 5.4.1
2 Init ()
3 for k ← bottom to top do
4 |   foreach (i, a, j), (i, ā, j) ∈ E_k do Add (i, L(i, j), j)
5 |   foreach (i, d, j) ∈ E_k do Add (i, L(i, j), j)
```

collects the necessary information. The key data structures are initialized on line 2. On lines 3-5, the reachability matrix is computed in a bottom-up manner. Specifically, each $\mathcal{A}$-edge and $\bar{\mathcal{A}}$-edge at layer k is handled first. The reachability information between adjacent layers is propagated from layer k to $k+1$ by processing $\mathcal{D}$-edges.

Answering Pointer Analysis Query

We then discuss how to use the reachability matrix $\boldsymbol{M}$ to answer the pointer analysis queries. The detailed procedure for answering points-to and alias analysis query is given in Procedure 16.

Points-to Query Given two pointer expressions p and q, we locate the representative nodes in the PEG using $H_e(p)$ and $H_a(q)$. Then we check the layer information by comparing $l(H_e(p))$ and $l(H_a(q))$. If both nodes are on the same layer, we look up the reachability matrix entry $m_{u_0 v_0}$. Node u and v are Pt-reachable

Procedure 16: Query(p, q, flag) to answer the pointer analysis query.

```
1  u ← H_e(p)
2  if flag is alias then v ← H_a(q)
3  else v ← H_e(q)
4  if l(u) ≠ l(v) then return false
5  if flag is alias and (m_{u0v0} or m_{u0v1} or m_{u1v1} is 1) then
6  |  return true
7  else if flag is pt and m_{u0v0} == 1 then
8  |  return true
9  else
10 |  return false
```

in the PEG iff v_0 is reachable from u_0.

Alias Query Similarly, to answer the alias query *w.r.t.* p and q, we first check the layer information by comparing $l(H_e(p))$ and $l(H_e(q))$. If both nodes are on the same layer, we look up $m_{u_0v_0}$, $m_{u_0v_1}$ and $m_{u_1v_1}$ entries. If one of them is 1, node v are M-reachable from u in the PEG. Finally, the query procedure returns true.

5.4.4 Algorithm Correctness and Complexity Analysis

This section discusses the correctness and the complexity of the proposed algorithm. We begin with establishing the correctness theorem.

Correctness

The correctness of the whole algorithm is proved by an induction on layers.

First, let us consider the V-reachability at bottom layer with only $\mathcal{A}$- and $\bar{\mathcal{A}}$-edges. Any trivial V-path is correctly handled by Algorithm 15, since the `Init`() procedure called at line 2 marks each v_q as V-reachable from itself, where $q \in \{0, 1\}$. We prove the correctness by induction on path length $|p|$ of any non-trivial V-path.

- *Base case.* $|p| = 1$. Every (u, a, v) and $(u, \bar{a}, v)$ is inserted by procedure `Add`() *w.r.t.* the state information in Table 5.1, *i.e.*, reachability between u_r and v_q is correct for all $\mathcal{A}$- and $\bar{\mathcal{A}}$-edges.

- *Inductive step.* Suppose Algorithm 15 correctly finds all V-paths of length $|p-1|$, any non-trivial V-path of length $|p|$ is generated according to the three cases as follows:

  - Case $u, v, \ldots, v'$, where $p' = v, \ldots, v'$ and $|p'| = p - 1$. Let (u, X, v) be the new $\mathcal{A}$- or $\bar{\mathcal{A}}$-edge processed by Algorithm 15, where $X \in \{a, \bar{a}\}$. According to the inductive hypothesis, path p' is correctly handed by Algorithm 15. As a result, all descendants of v are inserted

in $T(v_q)$. The procedure Add() called at line 4 recursively traverses $T(v_q)$ and inserts all unique descendants to $T(u_q)$. Therefore, node v'_q is inserted to $T(u_q)$ and the corresponding reachability matrix entry is updated as well.

- Case $u', \ldots, u, v$, where $p' = u', \ldots, u$ and $|p'| = p - 1$. On line 1 of procedure Add(), all nodes u' currently reaches u are traversed. Therefore, when a new edge (u, X, v) is inserted, procedure Add() correctly finds the right $T(u'_q)$ to insert v_q and updates the corresponding reachability entry. Finally, the new V-path between u' and v is generated.
- Case $u', \ldots, u, v, \ldots, v'$, where $p_1 = u', \ldots, u$, $|p_1| \leqslant |p - 1|$ and $p_2 = v, \ldots, v'$,$|p_2| \leqslant |p - 1|$. This case can be thought of as a combination of the previous two cases. When a new edge (u, X, v) is inserted, procedure Add() correctly finds u' that reaches u, and recursively traverses $T(v_q)$ to insert the unique descendant v' into $T(u'_q)$.

Then, we assume the *all-pairs* V-reachability on layer $l - 1$ is correctly computed, we discuss computing V-reachability on layer l. The following claims hold:

- All $\mathcal{D}$-edges in the PEG are processed by Algorithm 15 since line 5 considers all corresponding reverse $\bar{\mathcal{D}}$-edges. The handling of all $\bar{\mathcal{D}}$-edges is done on lines 9-16 in procedure `Mix`(). Specifically, procedure `Mix`() correctly generates (u', M, v') on layer l for all summary edge (u, V, v) on layer $l-1$. In other words, all non-consecutive M-paths on layer l are generated for any potential new V-path on layer l.

- On layer l, the reachability information is initially empty. For any new M-edges (u', M, v') generated, node u'_q is inserted to $T(v'_q)$ respecting the fact that v'_q is reachable from u'_q. This step essentially simulates the stack in Algorithm 10 for the chain case, *i.e.*, state q is pushed at node u' and it is popped at node v'. After initializing the reachability information on layer l, the other $\mathcal{A}$- and $\bar{\mathcal{A}}$-edges on layer l are handled similarly to the bottom layer. Therefore, the reachability information on layer l is correctly handled by Algorithm 15.

- The V-reachability between nodes u and v on layer l is computed despite any M-path connecting them. If v_q is M-reachable from u_q (*i.e.*, $v_q \in V_M[u_q]$) and the *flag* is 1, routine ADJUST() adjust v_q's position in $T(u_q)$ and remove it from $V_M[u_q]$. If the *flag* is 0, during the next recursive call

to procedure Mix(), v_q is inserted to $T(u_q)$ again respecting the fact that u_q is V-reachable from v_q. This maintains the invariants of reachability spanning trees *w.r.t.* the definition in Section 5.4.2.

The correctness on computing V-paths and Pt-paths is essentially the same as M-paths discussed above. Finally, we have the following theorem:

Theorem 7 (Correctness) *Given the CFG in Figure 5.2, Algorithm 15 correctly computes the* all-pairs *CFL-reachability information on well-typed PEG.*

Complexity

Then we discuss the time complexity of Algorithm 15. We note that Algorithm 15 calls procedure Mix() to handle all edges in the PEG. Procedure Mix() handles all $\bar{\mathcal{D}}$-edges on lines 9-16. As discussed in Section 5.4.4, all M-edges in the PEG are generated after processing $\bar{\mathcal{D}}$-edges. It is straightforward that the time spent on all $\bar{\mathcal{D}}$-edges in the PEG is $O(n)$.

On the other hand, all $\mathcal{A}$- and $\bar{\mathcal{A}}$-edges are handled by procedures Add() and Mix(). Due to the Fact 2, without considering M-paths on the same layer, the realized string for any V-path is $\bar{a}^*a^*$, which is a regular language. If the *flag* is set to be 0,

the two procedures `Add`() and `Mix`() handles each $\mathcal{A}$- and $\bar{\mathcal{A}}$-edge essentially the same as the previous work on dynamic regular language path problem [16]. The key distinction is the situation when *flag* is set to be 1. In that case, our algorithm maintains the invariants *w.r.t.* the definition of reachability spanning trees by computing V-reachability between nodes u and v on layer l despite the M-edge between them. The distinction introduces additional work which is bounded by $O(n\tilde{M})$, where $\tilde{M}$ denotes the maximum memory alias pairs on one layer. This is because line 7 in Procedure 13 and lines 7 and 13 in Procedure 14 remove an M-edge immediately after it has been examined and M-edges are generated only by processing $\bar{\mathcal{D}}$-edges. As a result, the two procedures handle all edges in the PEG in $O(n(m + \tilde{M}))$ time. The space complexity is $O(n^2)$ due to the use of reachability matrix. Combined the analysis, we have the following theorem,

Theorem 8 (Complexity) *Given a well-typed PEG with n nodes and m edges, the inclusion-based pointer analysis problem can be computed in $O(n(m+\tilde{M}))$ time with $O(n^2)$ space to answer any online pointer analysis query in $O(1)$ time.*

□ **End of chapter.**

Chapter 6

Application: Scaling an Alias Analysis for C

To evaluate the effectiveness of the proposed CFL-reachability algorithm, we apply the CFL-reachability-based alias analysis on the *latest stable* releases of widely-used C programs from the pointer analysis literature. All algorithms used in our evaluation solve *all-pairs* CFL-reachability formulated by Zheng and Rugina [97]. The results demonstrate that the alias analysis based on our *all-pairs* CFL-reachability algorithm performs extremely well in practice. For instance, it can analyze the Linux kernel in about 80 seconds. In particular, we design two sets of experiments to realize various aspects of the performance speedup:

- We use the CFL1 normal form in Figure 5.4 and investigate the practical benefits of the subcubic CFL-reachability algo-

Program	SLOC	#Procs	PEG		#Temps
			# Nodes	# Edges	
Gdb-7.5.1	1,828K	10,536	649,564	1,219,788	6,472
Emacs-24.2	254K	3,626	687,691	1,290,200	2,424
Insight-6.8-1a	1,742K	10,507	787,289	1,494,168	7,898
Gimp-2.8.4	702K	17,842	872,681	1,675,546	13,876
Ghostscript-9.07	851K	12,211	1,198,753	2,368,086	6,806
Wine-1.5.25	2,306K	70,923	4,652,983	8,472,950	25,064
Linux-3.8.2	10,601K	138,095	12,807,645	23,398,670	69,840

Table 6.1: Benchmark applications. The SLOC is reported by `sloccount` counting only C code.

rithm. We also evaluate the impact of connect component decomposition in CFL-reachability-based alias analyses for C.

- We then apply the CC decomposition and compare the performance of three subcubic CFL-reachability algorithms for alias analysis. Besides our own algorithm, the other two are traditional subcubic CFL-reachability algorithms using the two different normal forms described in Figure 5.4. We also summarize our experiences of scaling *all-pairs* CFL-reachability algorithms.

6.1 Experimental Setup

Benchmark Selection.

The set of C programs used in our evaluation is described in Table 6.1. For each program, we list its number of source lines of code, its number of procedures, the size of its Program Expression Graph (PEG), and the number of temporaries introduced in the traditional inclusion-based pointer analysis. The subject programs were obtained from the official websites. Gdb is the GNU debugger; Emacs is a text editor; Insight is a GUI for Gdb; Gimp is an image processing application; Ghostscript is a PostScript interpreter and PDF generator; Wine is a Windows emulator; and Linux is the kernel of the Linux operating system.

PEG Generation.

All C programs used in our evaluation are first processed by Gcc-4.6.3 for generating aliasing information for each procedure. The aliasing information is stored as constraint files represented in four standard forms described in Figure 2.7. We developed a Perl script to generate the PEGs from the constraint files. In particular, the temporaries introduced in transforming the source code to the four standard forms are eliminated, represented as the "#Temp" column in Figure 6.1. We observe that

the number of temporaries is relatively small compared to the number of pointer variables (*i.e.*, the "#Nodes" in the PEGs).

Implementation.

Both our analysis algorithm and the traditional CFL-reachability algorithms are implemented in C++ with extensive use of the Standard Template Library (STL). The Four Russians' Trick used in the subcubic CFL-reachability algorithms is implemented with a combination of Gcc's own built-in functions operating on bitvectors. Those built-in functions are each translated to a single CPU instruction. For example, the bit scan reverse (`bsr`) instruction can locate the first set bit, clear the bit and return its position in one CPU instruction, which can be considered as constant time. For more bitwise tricks, we refer the reader to Warren's book [86] and Andersen's bit twiddling hacks page[1].

All of the executables are compiled with Gcc-4.6.3 with "-O2" optimization. The algorithms take the same PEGs as input. Their outputs are verified to ensure consistency and correctness. Note that our algorithm computes the memory aliases (*i.e.*, the M-edges) and value aliases (*i.e.*, the V-edges) in two phases. All experiments were conducted on a Dell Optiplex 780 desktop with Intel Core2 Quad Q9650 CPU and 8GB RAM, running

[1] `http://graphics.stanford.edu/~seander/bithacks.html`

Program	Time			Memory		
	Cubic	**Sub**	**Sub_CC**	**Cubic**	**Sub**	**Sub_CC**
Gdb-7.5.1	2405.62	10.83	8.36	150.54	102.60	75.48
Emacs-24.2	54781.90	128.03	106.60	1095.50	1908.16	900.75
Insight-6.8-1a	665.06	8.65	5.98	116.29	149.57	75.52
Gimp-2.8.4	662.27	8.19	6.30	56.84	51.04	29.91
Ghostscript-9.07	4209.84	28.67	22.15	256.40	271.51	184.47
Wine-1.5.25	8234.29	64.07	38.02	451.28	1448.21	103.09
Linux-3.8.2	31997.00	160.81	119.03	236.44	197.98	93.00

Table 6.2: The performance of the cubic and subcubic alias analysis algorithms using the CFL1 formal form: time in seconds and memory in MB.

Ubuntu-12.04.

Performance of the Subcubic CFL-Reachability Algorithm

First, we present, for the first time in the literature, the performance of the subcubic CFL-reachability algorithm in practice. We use the CFL1 normal form in Figure 5.4 to compare the time and memory consumption between the cubic and subcubic CFL-reachability algorithms. We also evaluate the practical benefits of applying connected component decomposition in CFL-reachability-based alias analysis.

6.1.1 Time Consumption

Table 6.2 shows the time and memory consumption of both the cubic and subcubic algorithms computing *all-pairs* CFL-reachability using CFL1 normal form. The time and memory

consumption is collected differently. Specifically, the running time columns in Table 6.2 report the *accumulated* running times on all PEGs. On the other hand, the memory consumption columns in Table 6.2 report the *maximum* memory amount in the project, since the analysis is intraprocedure and the memory can be freed before processing the next procedure. The "Subcubic+CC" columns present the performance of applying connected component decomposition for alias analysis (discussed in Section 6.1.3).

From the running time columns in Table 6.2, we can see that the traditional cubic *all-pairs* CFL-reachability algorithm does not scale well. For example, the cubic algorithm takes about 10 minutes to complete on Gimp, which is already the best running time result of all programs used in our study. This result explains why there has been no practical *all-pairs* CFL-reachability-based pointer analysis. We note that the subcubic algorithm brings tremendous speedup in practice. Specifically, the subcubic algorithm using the Four Russians' Trick is more than 183.2 times faster than the cubic algorithm on average. The Linux kernel project takes the subcubic algorithm the longest time to complete. However, it is still within 3 minuets, which is already quite acceptable for a large-scale project like that. Note take it typ-

ically takes more than 30 minutes to compile the Linux kernel (without executing `make` in parallel) on the desktop used for our experiments.

6.1.2 Memory Consumption

The actual memory consumption of the cubic algorithm is slightly different from the subcubic algorithm. Despite the memory taken by the iterative computation, most of the memory is taken by the underlying data structures used to represent the graph. Specifically, the cubic algorithm typically uses an adjacency list to store all nodes in the graph. It can be observed from Table 6.1 that the PEGs are quite sparse in practice, with $m = O(n)$ where n and m represent the number of nodes and edges respectively. As a result, the space required to store the PEGs in the cubic algorithm is $O(m) = O(n)$.

On the other hand, the space required to store the PEGs in the subcubic algorithm is $O(n^2)$, because each node needs a bitvector for representing the summary edges of all nodes in the graph. Moreover, all the terminals and non-terminals should be considered to initialize the corresponding bitvectors. For instance, CFL1 contains 4 terminals and 9 non-terminals. The amounts of optimal space required to represent the largest PEG

Program	#CC	PEGs in Proc.		PEGs in CC	
		Max.	Avg.	Max.	Avg.
Gdb-7.5.1	66,154	5,162	61.65	4,350	9.82
Emacs-24.2	67,608	22,654	189.66	15,690	10.17
Insight-6.8-1a	75,373	6,273	74.93	4,350	10.45
Gimp-2.8.4	79,572	3,599	48.91	2,693	10.97
Ghostscript-9.07	87,768	8,573	98.17	7,025	13.66
Wine-1.5.25	537,370	20,008	65.61	5,106	8.66
Linux-3.8.2	1,449,718	7,205	92.75	4,755	8.83

Table 6.3: Connected component information on the benchmark programs.

in Wine with 20,008 nodes and Emacs with 22,654 nodes are 620 MB and 795MB respectively. However, only 41MB is required to store the largest PEG in Gdb with 5162 nodes. From the memory columns in Table 6.2, we can observe that both the cubic and subcubic CFL-reachability algorithms demand similar amounts of memory for the largest PEG. For Emacs and Wine, the subcubic algorithm consume 1.7 times and 3.2 times more memory respectively, since the two program contains larger PEG.

6.1.3 Impact of CC Decomposition

The scalability of the subcubic CFL-reachability algorithm depends on the size of the input graph, as observed in a recent work by Zhang *et al.* [95]. For instance, the 8GB RAM desktop used in our experiments can only afford to store a PEG with at most 72,705 nodes. Therefore, it is infeasible to feed the

Program	Max PEG in Proc. and CC
Gdb-7.5.1	`regex_byte_regex_compile()`
Emacs-24.2	`dbusbind_xd_append_arg()`
Insight-6.8-1a	`tclExecute_TclExecuteByteCode()`*
Gimp-2.8.4	`scale-region_scale()`
Ghostscript-9.07	`gxclrast_clist_playback_band()`
Wine-1.5.25	`image_convert_pixels()`*
Linux-3.8.2	`altera_altera_execute()`

Table 6.4: Procedures that contain the Max PEG in each benchmark program. Only in Insight and Wine, the Max PEGs are in the CCs belong to different procedures `regex_byte_regex_compile` and `int21_DOSVM_Int21Handler`. In the remaining bechmarks, the Max PEG is in the CC of the same procedure.

whole-program PEGs described in Table 6.1 for the alias analysis. However, since the alias analysis is context-insensitive, the PEG from each procedure can be processed independently. Table 6.3 shows that each program's PEGs typically have less than 200 nodes on average, which can be effectively handled by the subcubic algorithm in practice.

The CC decomposition can reduce the size of each PEG. The cost of CC decomposition is negligible, since a simple linear-time DFS through the PEGs is sufficient. Table 6.3 also shows the number of connect components, the maximum and average sizes of PEGs from both procedures and connected components. Table 6.4 gives the procedure name of the largest PEG. As expected, the size of PEGs in connected components is about 9 times smaller than the size in procedures.

Program	Time				Memory			
	CFL1	CFL2	Our		CFL1	CFL2	Our	
			M	$V+M$			M	$V+M$
Gdb-7.5.1	8.36	16.16	1.89	5.69	75.48	68.59	44.05	56.85
Emacs-24.2	106.60	222.30	39.34	90.80	900.75	783.76	539.86	659.16
Insight-6.8-1a	5.98	12.27	0.87	3.31	75.52	63.68	44.13	56.31
Gimp-2.8.4	6.30	14.13	0.75	3.01	29.91	27.02	18.25	22.58
Ghostscript-9.07	22.15	46.47	5.07	16.23	184.47	171.36	112.22	140.79
Wine-1.5.25	38.02	84.68	5.66	21.19	103.09	96.05	60.29	76.14
Linux-3.8.2	119.03	244.51	21.95	73.45	93.00	80.22	52.08	66.09

Table 6.5: The performance of various subcubic CFL-reachability-based alias analyses: time in seconds and memory in MB.

Reducing the input graph size further improves the performance of the underlying alias analysis. As shown in Table 6.2, the "subcubic+CC" approaches bring additional 1.4 times speedup in running time and uses 3.5 times less memory than subcubic approaches on average. We note that the overall performance of each CFL-reachability algorithm depends on the largest PEGs encountered during analysis. Emacs and Wine have PEGs that are larger than those in other programs. Therefore, the two programs require more memory in subcubic algorithm. When the CC composition is applied, the underlying subcubic CFL-reachability algorithm consumes 2.1 times and 14.0 times less memory on these two programs.

Program	#Orig. Edges	#V-Edges	#M-Edges
Gdb-7.5.1	1,219,788	12,904,372	356,075
Emacs-24.2	1,290,200	49,011,799	758,925
Insight-6.8-1a	1,494,168	12,560,471	432,657
Gimp-2.8.4	1,675,546	16,809,343	518,511
Ghostscript-9.07	2,368,086	35,910,829	705,121
Wine-1.5.25	8,472,950	79,613,731	2,769,135
Linux-3.8.2	23,398,670	234,930,383	6,272,658

Table 6.6: Number of original edges vs. number of alias edges.

Program	#Final Edges			
	CFL1	CFL2	Our	
			M	$M+V$
Gdb-7.5.1	29,961,321	46,675,387	8,578,778	26,771,729
Emacs-24.2	112,772,537	183,419,945	41,373,828	103,516,235
Insight-6.8-1a	27,573,822	43,826,051	6,651,681	22,948,332
Gimp-2.8.4	33,151,263	57,113,620	7,013,137	26,882,903
Ghostscript-9.07	76,022,402	127,231,348	20,997,042	74,873,044
Wine-1.5.25	161,447,212	272,708,579	35,592,825	124,527,418
Linux-3.8.2	482,622,025	818,188,002	108,039,628	435,178,287

Table 6.7: The graph densities for each algorithm.

6.2 Performance of Subcubic CFL-Reachability-Based Alias Analysis

In this section, we apply the CC decomposition and compares our proposed CFL-reachability algorithm for alias analysis with the traditional subcubic CFL-reachability algorithm using different normal forms. The comparisons between various algorithms is given in Table 6.5. CFL1 and CFL2 are two normal forms (described in Figure 5.4), which are used in the traditional CFL-

reachability algorithm. Specifically, the M column of our approach indicates the running time of computing only memory aliases, while the "$M + V$" column represents the running time of computing both memory aliases and value aliases.

All three subcubic CFL-reachability algorithms are working on small PEGs after the CC decomposition and have $O(n^3/\log\ n)$ time and $O(n^2)$ space complexities. The subcubic CFL-reachability algorithm using the CFL1 normal form runs faster than that using CFL2. However, the subcubic CFL-reachability algorithm based on the CFL2 normal form consumes less memory since it has less non-terminals. Among the three variants, our algorithm runs the fastest with least memory consumption over all programs in our evaluation. In particular, our memory alias algorithm runs 5.6 times faster than CFL1 and consumes 1.5 times less memory than CFL2, and the whole alias analysis algorithm runs 1.6 times faster than CFL1 and consumes 1.2 times less memory than CFL2.

In order to understand the performance speedup better, we calculate the number of alias edges in Table 6.6 and the graph densities in Table 6.7. The columns in Table 6.6 represent the number of original edges, and the number of memory and value alias edges. And the columns in Table 6.7 give all summary edges

in the final graph respectively for each program. We first note that the number of edges $m \ll n^2$ for all programs, which implies that the alias analysis graphs are unlikely to be dense in practice. We also observe that our subcubic CFL-reachability algorithm computes the least number of summary edges among the three variants. Specifically, the number of final summary edges in our memory alias algorithm and whole alias analysis algorithm is 4.0 times and 1.2 times fewer than the CFL1 algorithm and 6.6 times and 1.9 times fewer than the CFL2 algorithm respectively.

Our subcubic CFL-reachability algorithm computes the memory aliases based on only the original edges and memory alias edges in the final graph. The number of memory alias summary edges are 3.1 times fewer than the number of original edges. On the other hand, the number of other reachability summary edges is far greater than the number of original edges. For example, the V-edges and total final edges in CFL1 are 14.5 times and 31.7 times more. In practice, our subcubic CFL-reachability algorithm performs better than traditional subcubic CFL-reachability algorithms reflecting the fact that it computes far fewer reachability summary edges.

6.3 Discussions

Finally, we discuss the main findings of our design and implementation of the subcubic CFL-reachability algorithms.

Staged CFL-Reachability

Perhaps the most interesting finding in our study is that it is possible to design a staged CFL-reachability algorithm which is faster than the traditional CFL-reachability algorithm. In particular, different stages focus on *all-pairs* reachability for different summary edges. We show that our memory alias algorithm computes *all-pairs* memory aliases which is 5.6 times faster then traditional subcubic CFL-reachability algorithm in practice. Moreover, our value alias algorithm depends on the results on memory aliases. In practice, some client analyses may only be interested in some of the summary edges. For example, the Zheng and Rugina points-to formulation [97] only concerns the memory alias edges. In such cases, the staged CFL-reachability algorithm may be much more efficient than the traditional CFL-reachability algorithm.

The CFL-reachability algorithm can use the results from different stages to bootstrap each other. Specifically, the later stages can reuse some of the existing summary edges obtained

in previous stages. For example, in our value alias algorithm, we can reuse all summary edges obtained from the memory alias algorithm. The value alias algorithm only needs to perform additional propagations over top-level variables. The key to enable a staged CFL-reachability algorithm is to exploit the dependencies between the relevant nonterminals in the *CFG*.

CFL-Reachability via Partial Summary Edges

Although the worst-case time complexity of the three algorithms in Table 6.5 is $O(n^3/\log n)$, the performance in practice mainly depends on the number of summary edges the underlying algorithm computes. The input normal forms to the CFL-reachability algorithm have a non-trivial impact on the algorithm's practical performance. In our study, the traditional CFL-reachability algorithm using the CFL1 normal form is 2.1 times faster than that using the CFL2 normal form. Choosing an appropriate input normal form for the CFL-reachability algorithm is crucial, since the traditional CFL-reachability algorithm relies on all summary edges to propagate reachability information.

In practice, it is possible to design a more efficient CFL-reachability algorithm using only some of the nonterminal edges. For example, in our memory alias algorithm, each worklist item

uses only the original edges and M-edges to propagate reachability information. As shown in Table 6.6, the memory edges are quite sparse. Propagating information through those sparse edges in our memory alias algorithm yields more than 5.6 times speedup.

Limitations of the Subcubic Algorithm

The Four Russians' Trick is the key technique to scale the *all-pairs* CFL-reachability algorithm. In particular, it improves the worst-case complexity of the traditional CFL-reachability algorithm and brings tremendous speedup in practice. However, the subcubic CFL-reachability algorithm has two sources of limitations.

The first source of limitation is the size of the input graph. As aforementioned, the 8GB RAM desktop used in our experiments can only afford to store a graph with at most 72,705 nodes. In order to make the subcubic algorithm scale, we need to reduce the size of the input graphs, since the algorithm has a quadratic space complexity. As in our alias analysis, the input graphs are the PEGs of each procedure rather than the whole-program PEG, which makes the analysis feasible. Moreover, connected component decomposition can further reduce the graph

size, which brings 1.4 times speedup.

The second source of limitation is the size of the input grammar. The subcubic algorithm needs to allocate the space for storing the summary of all nonterminals and terminals from the input grammar. Reducing the grammar size may have practical benefits in saving required memory. It is possible to exploit properties on the input grammar and represent the grammar using fewer nonterminal. For example, our analysis algorithm uses the fewest number of nonterminals among the three subcubic algorithms, thus consumes the least amount of memory (Table 6.5).

□ **End of chapter.**

Chapter 7

Related Work

This thesis contributes to scale CFL-reachability-based alias analysis for both C and Java. In this chapter, we survey two strands of closely related work: CFL-reachability and alias analysis. Moreover, since points-to analysis is the traditional approach to alias analysis, we summarize the most important developments in the points-to analysis literature.

7.1 CFL-Reachability

The CFL-reachability framework was initially proposed by Yannakakis [93] for Datalog chain query evaluation. Later, it has been used to formulate interprocedural dataflow analysis [70], program slicing [69], shape analysis [66], container analysis for Java [91], type-based flow analysis [30, 62, 65], and pointer analysis [54, 74, 78, 80, 90, 95, 97]. The seminal work by Reps

offers insights of formulating program analysis problems using CFL-reachability [67]. CFL-reachability has shown to be closely related to set constraints [47, 55]. Moreover, the work done by Kodumal and Aiken has demonstrated a set constraints for CFL-reachability intersected with a regular language [48].

The central theme in the CFL formulations is that many program analyses have the balanced-parentheses property that can be captured by Dyck-CFL-reachability [47]. The CFL and Dyck-CFL-reachability problems are also studied in the context of recursive state machines [9], visibly pushdown languages [8] and streaming XML [7]. Specially, when the recursive state machines are restricted to allow a constant number of entry/exit nodes per module, reachability is solvable in linear-time.

It is well-known that CFL-reachability-based algorithms have cubic worst-case complexity, commonly known as "the cubic bottleneck in flow analysis" [37]. Finding more efficient algorithms for CFL-reachability is a difficult problem as any breakthrough in CFL-reachability may lead to faster algorithms for CFL parsing [67]. Chaudhuri showed that the well-known Four Russians' Trick [11] could be employed to speed up in the original CFL-reachability algorithm to immediately yield a subcubic algorithm [20]. Similar techniques were used in Rytter's work [73]

for CFL parsing. Besides the subcubic result, Kodumal and Aiken [47] described a specialized set constraint reduction for Dyck-CFL-reachability on graphs and Yuan and Eugster [94] proposed an efficient Dyck-CFL-reachability algorithm on bidirected trees.

This thesis introduces asymptotically faster algorithms for Dyck-CFL-reachability. Moreover, we also give a fast algorithm for solving a specific CFL-reachability instance with application to alias analysis for C.

7.2 Alias Analysis

Alias analysis has been extensively studied in the literature. Its goal is to decide if two pointer variables may point to the same memory location during program execution. The problem is first formulated by Choi *et al.* [23] and Landi and Ryder [50]. For more background on alias analysis, we refer readers to a comprehensive survey maintained by Wu [4].

Although precise alias information is quite helpful for subsequent analysis [26, 75], deciding aliasing is commonly known as a computationally hard problem [41, 64]. Approximations must be made for any practical alias analysis. Various techniques have been proposed to scale alias analyses, such as improving the un-

derlying points-to analysis [32, 35], making the analysis demand-driven [38, 92], and using novel data structures [88]. Two most notable challenges for any practical alias analysis are modeling the context-sensitivity and recursively data structures. In the literature, various graph-based approaches has been proposed to resolve the context sensitivity [19, 31, 51]. On the other hand, many analysis handles the recursive data structures by limiting the maximum accesses path [21, 29, 89].

Most state-of-the-art alias analyses have been formulated as CFL-reachability on edge-labeled graphs [54, 74, 78, 80, 90, 92, 97]. The CFL-reachability-based analyses do not need the points-to analysis to obtain the points-to sets first. Specifically, Zheng and Rugina proposed a CFL-reachability formulation on pointer expression graph (PEG) for C [97]. For Java, Yan *et al.* [92] proposed an CFL-reachability formulation on symbolic points-to graph (SPG) to approximate the precise CFL-reachability-based alias analysis described by Sridharan *et al.* [78, 80] and Xu *et al.* [90].

This thesis adopts the CFL-reachability formulations on PEG [97] and SPG [92] for C and Java respectively. We show that our fast algorithms help dramatically speed up the *context-insensitive* alias analysis on both formulations. Moreover, all of the men-

tioned state-of-the-art alias analyses in this section, except for the work of Xu *et al.* [90], are *demand-driven*, which solving *single-source-single-sink* CFL-reachability problem. Our alias analysis algorithms solve the *all-pairs* CFL-reachability problem and scale to large, real-world applications.

7.3 Points-to Analysis

The goal of points-to analysis is to compute a set of memory locations that a pointer variable may point to during program execution. Points-to analysis has been recognized as a traditional approach to alias analysis because aliasing relation can be decided by consulting the points-to sets of the two variables. We refer the readers to Hind's survey paper [40] on a large body of points-to analysis work.

Like alias analysis, precise points-to analysis is also a computationally hard problem. In this thesis, we focus on flow- and context-insensitive analysis. In this domain, equality-based (Steensgaard-style) analysis [81] and inclusion-based (*i.e.*, Andersen-style) points-to analysis [10] are two important approximations. Compared with equality-based analysis, inclusion-based analysis is more precise and also more expensive [79]. A recent paper [13] concludes that "while better algorithms for the precise flow-

insensitive analysis are still of theoretical interest, their practical impact for C programs is likely to be negligible."

Traditional inclusion-based points-to analysis has been formulated as a dynamic transitive closure problem, with a cubic time complexity in the worst-case [35, 79]. Over the decade, many enhancements have been proposed to scale the inclusion-based pointer analysis, including online cycle elimination [32], projection merging [82], off-line variable substation [71], improved dynamic transitive closure algorithms [35, 38, 59], using better data structures for points-to sets [12, 88, 98], adopting probabilistic approaches [76], making the analysis paralleled [56, 57] or demand-driven [39, 80], just to name a few [25, 44, 60, 77].

Inclusion-based points-to analysis for C has been formulated as a CFL-reachability problem by Reps [67]. Sridharan *et al.* also proposed a CFL-reachability formulation for points-to analysis for Java [80]. Several work has been done to improve the performance of CFL-reachability-based points-to analysis, *e.g.*, speeding up computing summary edges using improved strategies [54, 74], applying algorithmic tricks [20] and rolling out infeasible paths [90] during computation. On the other hand, the CFL-reachability-based points-to analysis is interesting in its own right. Compared with traditional inclusion-based anal-

ysis which has been formulated as a dynamic transitive closure problem, the CFL-reachability instance works on a fixed edge-labeled graph such that no new edges are inserted during computation. In Chapter 5, we exploited the properties of the underlying CFL on both alias analysis and points-to analysis, and proposed asymptotically faster algorithms for solving the *all-pairs* CFL-reachability.

□ **End of chapter.**

Chapter 8

Conclusion

In this chapter, we summarize the main results presented in this thesis. We then discuss some possible interesting directions as future work.

8.1 Thesis Summary

In this thesis, we have contributed to both theoretical and practical developments in scaling CFL-reachability-based alias analysis. From a theoretical perspective, we have proposed a set of asymptotically faster algorithms for solving *all-pairs* CFL-reachability formulated in two existing state-of-the-art alias analyses. Our algorithms improve the traditional (sub) cubic CFL-reachability algorithm. From a practical perspective, we have implemented our CFL-reachability-based alias analysis for both Java and C. The evaluation has demonstrated that our algo-

rithms perform extremely well in practice and consume less memory.

In particular, Chapters 3 and 4 have proposed two fast algorithms for solving Dyck-CFL-reachability in $O(n)$ time for bidirected trees and in $O(n + m \log m)$ time for bidirected graphs respectively, and a fast algorithm in $O(n(m + S))$ time for solving general *all-pairs* Dyck-CFL-reachability. The key insights behind our algorithms are that the bidirected Dyck-relation is an equivalence relation and solving the general Dyck-CFL-reachability problem can be benefited from a known result on dynamically maintaining the transitive closure. We have also applied our bidirected Dyck-CFL-reachability algorithm to a *context-insensitive* alias analysis for Java on SPG [92]. The experimental results have shown that our graph algorithm help bring orders of magnitude speedup on DaCapo benchmarks.

Later, in Chapters 5 and 6, we have also presented an efficient subcubic algorithm for solving the *all-pairs* CFL-reachability formulated in an existing alias analysis for C on PEG [97]. When the given PEG is restricted to be well-typed, we have given an asymptotically fast algorithm in $O(mn)$ time. Moreover, we have also presented the design and implementation of the first subcubic CFL-reachability-based alias analysis for C. To evalu-

ate its scalability, we have conducted extensive experiments on the *latest stable* releases of the most popular C programs from the pointer analysis literature. Our results have shown that the proposed CFL-reachability-based alias analysis scales extremely well. In particular, it can analyze the latest Linux kernel in under 80 seconds.

8.2 Future Work

There are several interesting directions of improving CFL-reachability-based alias analysis which we would like to explore in the future.

First, in this thesis, we have considered the CFL-reachability formulation for *context-insensitive* alias analysis. The CFL-reachability formulation can not be directly adopted for *context-sensitive* alias analysis, since the problem is undecidable [68]. Currently, typical *context-sensitive* CFL-reachability-based alias analyses [78, 90, 92] adopt a regular language to approximate the procedure calls/returns. It remains an open problem whether asymptotically faster algorithms exist for the *context-sensitive* CFL-reachability-based alias analysis.

Second, in Chapter 3, we have demonstrated that the bidirected Dyck-CFL-reachability problem on trees can be solved much faster than the counterpart on graph. It is interesting to

see whether the same claims hold for the CFL-reachability problem considered in Chapter 5 (*i.e.*, the inclusion-based pointer analysis on the tree PEGs). Moreover, it should also be interesting to carry out some empirical work to study the impact of tree PEGs on real-world C programs. If most of the PEGs in practice are trees, fast tree algorithms can significantly improve the performance of CFL-reachability-based pointer analysis.

Last, the time complexities of both the general Dyck-CFL-reachability algorithm in Chapter 3 and the general CFL-reachability-based alias analysis algorithm in Chapter 5 are output-sensitive, *i.e.*, they are related to S- and M-edges in the final graph. The time complexities of the two algorithms are very hard to improve. For example, the alias analysis for C presented in Chapter 5 is a variant of Andersen's pointer analysis [10], for which the (sub) cubic worst-case time complexity has been exhibited for many years. It is indeed a breakthrough if one of the two general algorithms can be improved to $O(mn)$ time.

□ **End of chapter.**

Bibliography

[1] Cambridge university study states software bugs cost economy $312 billion per year. http://undo-software.com/content/press-release-8.

[2] DaCapo benchmark suite. http://dacapobench.org/.

[3] The industrial control systems cyber emergency response team. http://ics-cert.us-cert.gov/.

[4] Survey of alias analysis. http://www.cs.princeton.edu/~jqwu/Memory/.

[5] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[6] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, second edition, 2006.

[7] R. Alur. Marrying words and trees. In *PODS*, pages 233–242, 2007.

[8] R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC)*, pages 202–211, 2004.

[9] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. W. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Transactions on Programming Languages and Systems*, 27(4):786–818, 2005.

[10] L. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[11] V. Arlazarov, E. Dinic, M. Kronrod, and I. Faradzev. On economic construction of the transitive closure of a directed graph. *Soviet Mathematics Doklady*, 11:1209–1210, 1970.

[12] M. Berndl, O. Lhoták, F. Qian, L. J. Hendren, and N. Umanee. Points-to analysis using bdds. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, pages 103–114, 2003.

[13] S. Blackshear, B.-Y. E. Chang, S. Sankaranarayanan, and

M. Sridharan. The flow-insensitive precision of andersen's analysis in practice. In *Proceedings of the 18th International Static Analysis Symposium (SAS)*, pages 60–76, 2011.

[14] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 241–250, 2011.

[15] B. W. Boehm and V. R. Basili. Software defect reduction top 10 list. *IEEE Computer*, 34(1):135–137, 2001.

[16] A. L. Buchsbaum, P. C. Kanellakis, and J. S. Vitter. A data structure for arc insertion and regular path finding. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 22–31, 1990.

[17] V. T. Chakaravarthy. New results on the computability and complexity of points-to analysis. In *Proceedings of the 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 115–125, 2003.

[18] T. M. Chan. All-pairs shortest paths for unweighted undirected graphs in *o(mn)* time. In *Proceedings of the Sev-*

enteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 514–523, 2006.

[19] R. Chatterjee, B. G. Ryder, and W. Landi. Relevant context inference. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 133–146, 1999.

[20] S. Chaudhuri. Subcubic algorithms for recursive state machines. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 159–169, 2008.

[21] B.-C. Cheng and W. mei W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 57–69, 2000.

[22] B. Chess and J. West. *Secure programming with static analysis*. Addison-Wesley Professional, 2007.

[23] J.-D. Choi, M. G. Burke, and P. R. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL*, pages 232–245, 1993.

[24] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009.

[25] M. Das. Unification-based pointer analysis with directional assignments. In *PLDI*, pages 35–46, 2000.

[26] M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. In *Proceedings of the 8th International Static Analysis Symposium (SAS)*, pages 260–278, 2001.

[27] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 57–68, 2002.

[28] A. Deshpande and D. Riehle. The total growth of open source. In *Proceedings of the 4th International Conference on Open Source Systems (OSS)*, pages 197–209, 2008.

[29] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, pages 230–241, 1994.

[30] C. Earl, I. Sergey, M. Might, and D. V. Horn. Introspective

pushdown analysis of higher-order programs. In *ICFP*, pages 177–188, 2012.

[31] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, pages 242–256, 1994.

[32] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 85–96, 1998.

[33] R. Ghiya, D. M. Lavery, and D. C. Sehr. On the importance of points-to analysis and other memory disambiguation methods for c programs. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 47–58, 2001.

[34] B. Hardekopf. personal communication, 2012.

[35] B. Hardekopf and C. Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the ACM SIGPLAN 2007 Confer-*

ence on Programming Language Design and Implementation (PLDI), pages 290–299, 2007.

[36] B. C. Hardekopf. *Pointer analysis: building a foundation for effective program analysis.* PhD thesis, The University of Texas at Austin, 2009.

[37] N. Heintze and D. A. McAllester. On the cubic bottleneck in subtyping and flow analysis. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 342–351, 1997.

[38] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 254–263, 2001.

[39] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 24–34, 2001.

[40] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT*

Workshop on Program Analysis For Software Tools and Engineering (PASTE), pages 54–61, 2001.

[41] S. Horwitz. Precise flow-insensitive may-alias analysis is NP-Hard. *ACM Transactions on Programming Languages and Systems*, 19(1):1–6, 1997.

[42] S. Horwitz, T. W. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.

[43] G. F. Italiano. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48(3):273–281, 1986.

[44] V. Kahlon. Bootstrapping: A technique for scalable flow and context-sensitive pointer alias analysis. In *PLDI*, pages 249–259, 2008.

[45] B. W. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1988.

[46] U. P. Khedker, A. Sanyal, and B. Sathe. *Data Flow Analysis - Theory and Practice*. CRC Press, 2009.

[47] J. Kodumal and A. Aiken. The set constraint/CFL reachability connection in practice. In *Proceedings of the ACM*

SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI), pages 207–218, 2004.

[48] J. Kodumal and A. Aiken. Regularly annotated set constraints. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, pages 331–341, 2007.

[49] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, 1992.

[50] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *PLDI*, pages 235–248, 1992.

[51] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation (PLDI)*, pages 56–67, 1993.

[52] L. Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *Journal of the ACM*, 49(1): 1–15, 2002.

[53] M. M. Lehman and F. N. Parr. Program evolution and its impact on software engineering. In *Proceedings of the 2nd*

International Conference on Software Engineering (ICSE), pages 350–357, 1976.

[54] Y. Lu, L. Shang, X. Xie, and J. Xue. An incremental points-to analysis with cfl-reachability. In *Proceedings of the 22nd International Conference on Compiler Construction (CC)*, pages 61–81, 2013.

[55] D. Melski and T. W. Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science*, 248(1-2):29–98, 2000.

[56] M. Méndez-Lojo, A. Mathew, and K. Pingali. Parallel inclusion-based points-to analysis. In *OOPSLA*, pages 428–443, 2010.

[57] M. Méndez-Lojo, M. Burtscher, and K. Pingali. A gpu implementation of inclusion-based points-to analysis. In *PPOPP*, pages 107–116, 2012.

[58] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2005.

[59] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis of c. *ACM Transactions on Programming Languages and Systems*, 30(1), 2007.

[60] F. M. Q. Pereira and D. Berlin. Wave propagation and deep propagation for pointer analysis. In *Proceedings of the 7th International Symposium on Code Generation and Optimization (CGO)*, pages 126–135, 2009.

[61] E. L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52(4):264–268, 1946.

[62] P. Pratikakis, J. S. Foster, and M. Hicks. Existential label flow inference via CFL reachability. In *Proceedings of the 13th International Static Analysis Symposium (SAS)*, pages 88–106, 2006.

[63] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Practical static race detection for C. *ACM Transactions on Programming Languages and Systems*, 33(1):3, 2011.

[64] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16 (5):1467–1471, 1994.

[65] J. Rehof and M. Fähndrich. Type-base flow analysis: from polymorphic subtyping to CFL-reachability. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 54–66, 2001.

[66] T. W. Reps. Shape analysis as a generalized path problem. In *Proceedings of the 1995 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation (PEPM)*, pages 1–11, 1995.

[67] T. W. Reps. Program analysis via graph reachability. *Information & Software Technology*, 40(11-12):701–726, 1998.

[68] T. W. Reps. Undecidability of context-sensitive data-independence analysis. *ACM Transactions on Programming Languages and Systems*, 22(1):162–186, 2000.

[69] T. W. Reps, S. Horwitz, S. Sagiv, and G. Rosay. Speeding up slicing. In *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering (FSE)*, pages 11–20, 1994.

[70] T. W. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 49–61, 1995.

[71] A. Rountev and S. Chandra. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the 2000*

ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 47–56, 2000.

[72] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *Proceedings of the 12th International Conference on Compiler Construction (CC)*, pages 126–137, 2003.

[73] W. Rytter. Fast recognition of pushdown automaton and context-free languages. *Information and Control*, 67(1-3): 12–22, 1985.

[74] L. Shang, X. Xie, and J. Xue. On-demand dynamic summary-based points-to analysis. In *CGO*, pages 264–274, 2012.

[75] M. Shapiro and S. Horwitz. The effects of the presision of pointer analysis. In *Proceedings of the 4th International Static Analysis Symposium (SAS)*, pages 16–34, 1997.

[76] J. D. Silva and J. G. Steffan. A probabilistic pointer analysis for speculative optimizations. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 416–425, 2006.

[77] L. Simon. Optimizing pointer analysis using bisimilarity. In

Proceedings of the 16th International Static Analysis Symposium (SAS), pages 222–237, 2009.

[78] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI)*, pages 387–400, 2006.

[79] M. Sridharan and S. J. Fink. The complexity of andersen's analysis in practice. In *SAS*, pages 205–221, 2009.

[80] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 59–76, 2005.

[81] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 32–41, 1996.

[82] Z. Su, M. Fähndrich, and A. Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium*

on Principles of programming languages (POPL), pages 81–95, 2000.

[83] G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, 2002.

[84] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing java bytecode using the Soot framework: Is it feasible? In *Proceedings of the 9th International Conference on Compiler Construction (CC)*, pages 18–34, 2000.

[85] V. Vassilevska. Efficient algorithms for clique problems. *Information Processing Letters*, 109(4):254–257, 2009.

[86] H. S. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[87] J. Whaley. *Context-sensitive pointer analysis using binary decision diagrams*. PhD thesis, Stanford University, 2007.

[88] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144, 2004.

[89] R. P. Wilson and M. S. Lam. Efficient context-sensitive

pointer analysis for c programs. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 1–, 1995.

[90] G. Xu, A. Rountev, and M. Sridharan. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP)*, pages 98–122, 2009.

[91] G. H. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 160–173, 2010.

[92] D. Yan, G. H. Xu, and A. Rountev. Demand-driven context-sensitive alias analysis for Java. In *Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA)*, pages 155–165, 2011.

[93] M. Yannakakis. Graph-theoretic methods in database theory. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 230–242, 1990.

[94] H. Yuan and P. T. Eugster. An efficient algorithm for solving

the Dyck-CFL reachability problem on trees. In *Proceedings of the 18th European Symposium on Programming (ESOP)*, pages 175–189, 2009.

[95] Q. Zhang, M. R. Lyu, H. Yuan, and Z. Su. Fast algorithms for dyck-cfl-reachability with applications to alias analysis. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 435–446, 2013.

[96] S. Zhang, B. G. Ryder, and W. Landi. Program decomposition for pointer aliasing: A step toward practical analyses. In *SIGSOFT FSE*, pages 81–92, 1996.

[97] X. Zheng and R. Rugina. Demand-driven alias analysis for C. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 197–208, 2008.

[98] J. Zhu and S. Calman. Symbolic pointer analysis revisited. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*, pages 145–157, 2004.