Towards a Scalable Framework for Context-Free Language Reachability

Nicholas Hollingum and Bernhard Scholz

The University of Sydney, NSW 2006, Australia
nhol8058@uni.sydney.edu.au, scholz@it.usyd.edu.au

Abstract. Context-Free Language Reachability (CFL-R) is a search problem to identify paths in an input labelled graph that form sentences in a given context-free language. CFL-R provides a fundamental formulation for many applications, including shape analysis, data and control flow analysis, program slicing, specification-inferencing and points-to analysis. Unfortunately, generic algorithms for CFL-R scale poorly with large instances, leading research to focus on ad-hoc optimisations for specific applications. Hence, there is the need for scalable algorithms which solve arbitrary CFL-R instances.

In this work, we present a generic algorithm for CFL-R with improved scalability, performance and/or generality over the state-of-the-art solvers. The algorithm adapts Datalog's semi-naïve evaluation strategy for eliminating redundant computations. Our solver uses the quadtree data-structure, which reduces memory overheads, speeds up runtime, and eliminates the restriction to normalised input grammars. The resulting solver has up to 3.5x speed-up and 60% memory reduction over a state-of-the-art CFL-R solver based on dynamic programming.

Keywords: program analysis, context-free language reachability, semi-naïve evaluation, quad-trees, matrix multiplication.

1 Introduction

The **Context-Free Language Reachability** (CFL-R) problem has been researched extensively since it was initially identified by Yannakakis [29]. In a pleasing symmetry to our own work, he viewed the problem as a means of solving a sub-class of Datalog queries via CFL-R. Not limited to logic programming though, CFL-R soon proved to be useful for diverse computational tasks, from formal security analysis [8] to a wide range of program analysis problems [22].

The importance of CFL-R as a program analysis framework cannot be understated. CFL-R encompasses shape analysis [22], data- [23] and control-flow [27], set-constraints [15] [18], specification-inferencing [3], object-flow [31], and a plethora of context, flow and field sensitive and insensitive alias [33] [17] [35] [25] [28] analyses, to name a few. We attribute this extensive utility to the fact that such analyses rely on dynamic reachability queries for program graphs, which is a common enough problem to deserve its own complexity class [11], and is expressed naturally

B. Franke (Ed.): CC 2015, LNCS 9031, pp. 193–211, 2015.
DOI: 10.1007/978-3-662-46663-6_10

by CFL-R. The continuous stream of CFL-R research since the 90s indicates that it will remain an important problem into the future.

Unfortunately, the promise of CFL-R cannot be reached by the current state-of-the-art solvers. The original dynamic-programming algorithm, due to Melski and Reps [18], has cubic time-complexity. More recently, Chaudhuri [5] improved previous work slightly with an $\mathcal{O}(\frac{n^3}{\log n})$ algorithm utilising the Four Russians' Trick, which offers a relatively minor speedup at the cost of a significant memory increase. The complexity issue, coined the "cubic bottleneck" [11], spurs research in restricted sub-classes of CFL-R [31] [33] which have better time complexities, but limited applicability. To provide a general CFL-R framework for a large range of applications, our work restricts neither the graph nor the grammar classes.

It is unlikely [21] that algorithmic improvement will be made to the current $\mathcal{O}(\frac{n^3}{\log n})$ lower-bound. This work, therefore, focuses on improving performance in practice, by removing redundant computations and memory inefficiencies which occur for the current state-of-the-art solvers. The new approach has worse theoretical properties, but achieves better practical runtime performance by adapting efficient machinery from a similar problem-space. Thus, in a reversal of Yannakakis' formulation, we turn to Datalog as a scaffolding for the development of a new CFL-R algorithm. Redundant computations, which occur in the Melski-Reps algorithm, can be eliminated by an intelligent evaluation strategy. We specialise the machinery used in Datalog engines, called **semi-naïve evaluation**, to the CFL-R context. The adapted algorithm is implemented on top of an efficient **quadtree** binary-relation representation. Together, the quadtree-based semi-naïve algorithm achieves more efficient memory usage, improves the practical runtime performance, especially for sparse problems, and obviates the need for an expensive grammar-normalisation operation.

We outline our contributions as follows:

- We specialise the semi-naïve evaluation strategy from the Datalog context to CFL-R. This leads to a new algorithm with fewer redundant calculations, and performant behaviour for non-normalised input graphs.
- We present quadtrees as a vehicle for representing the input graph and evaluating the solution. Quadtrees provide further advantages to our approach, since they give new-information-tracking for free and have efficient memory utilisation for sparse problems.
- We experimentally verify the advantages of the new technique, showing up to 3.5x speedup and 60% memory reduction, on a Java points-to analysis problem.

Our paper is organised as follows: Section 2 provides background material about the CFL-R problem, and introduces the current state-of-the-art solvers. An explanation of our contributions is presented in Section 3, which includes our adaptation of the semi-naïve evaluation strategy, the explanation of quadtrees, and the reasons that normalisation is unnecessary. Section 4 presents our experimental findings, specifically on the superior memory and runtime performance of our approach. We survey the related literature in Section 5, and conclude our findings and plans for future work in Section 6.

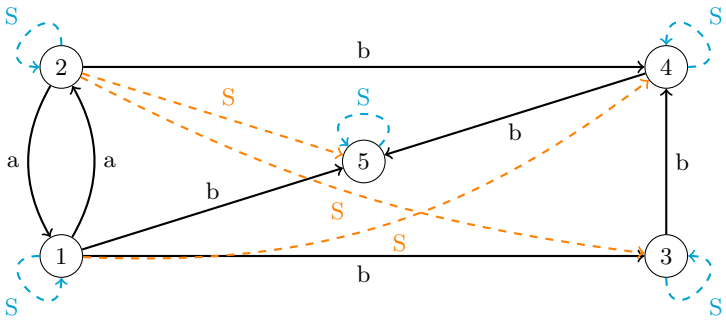


Fig. 1. Running CFL-R Example, its grammar is $\mathcal{P} = \{[S \to aSb], [S \to \epsilon]\}$. Dashed edges represent solutions to the CFL-R problem, lowercase letters are terminal symbols, and the non-terminal start-symbol is S.

2 Context-Free Language Reachability

We use the standard terminology to define the CFL-R problem as a 6-tuple $\mathcal{L} = (\Sigma, \mathcal{N}, \mathcal{P}, S, V, E)$, of terminals Σ, non-terminals $\mathcal{N}$, production rules $\mathcal{P}$, a start symbol S, vertices V, and edges E. For notational convenience we say $E \subseteq V \times V \times (\Sigma \cup \mathcal{N})$, such that an edge is a triple $(u, v, X) \in E$, denoting that vertex u is connected to v via an edge labelled with the terminal or non-terminal X. We will refer to the elements of an edge triple as its source, destination and label respectively. Henceforth, let n count the number of vertices in the input problem, and k the sum of the left and right-hand sides of all the production rules.

CFL-R is a generalisation of graph reachability and context-free recognition. Informally, we search a graph for those paths between vertices, whose labels concatenate to form a sentence in the context-free language. We use the standard notions [13] of production expansion and sentences here, so a sentence in the language must be reachable by finitely many production-rule expansions beginning with the start symbol. In this way the CFL-R problem can express both transitive reachability (according to a grammar $[S \to a^*]$) and context-free language recognition (reachability in a line graph). Figure 1 illustrates a CFL-R problem. Solutions to the problem are displayed with dashed lines, and summarise paths in the graph which traverse some (possibly zero) "a" labelled edges, followed by the same number of "b" edges.

We make use of two extensions to aid expressivity of the grammars. A non-terminal symbol may be parametric (written A_f), which is simply a stand-in for the distinct non-terminal A_f to make the grammar's presentation concise. Also, for notational convenience, symbols in the right-hand-side of the production may indicate the transposed relation, using $\overline{\texttt{overline}}$. In a CFL-R instance, this refers to reverse edges, which could be tracked by their own productions [22], but this would lengthen the grammar. The rule $[A \to B\overline{C}D]$ matches paths which travel backwards along the C edge.

Algorithm 1. Generalised worklist-based CFL-R algorithm

```
1: procedure WORKLIST(L = (Σ, N, P, S, V, E))
2:     for all v ∈ V, [X → ϵ] ∈ P do
3:         add (v, v, X) to E
4:     end for
5:     W ← E
6:     while W ≠ ∅ do
7:         remove (u, v, Y) from W
8:         for all [Z → X_0 ... X_a Y X_b ... X_L] ∈ P do
9:             B ← {u}
10:            for i = a to 0 do
11:                B ← {n : (n, u', X_i) ∈ E, u' ∈ B}
12:            end for
13:            F ← {v}
14:            for j = b to L do
15:                F ← {n : (v', n, X_j) ∈ E, v' ∈ F}
16:            end for
17:            W ← W ∪ ({(u', v', Z) : u' ∈ B, v' ∈ F} \ E)
18:            E ← E ∪ W
19:        end for
20:    end while
21: end procedure
```

The state-of-the-art scalable algorithm for CFL-R is due to Melski and Reps [18]. The reader should note that Chaudhuri has introduced an improvement [5] using a fast-set representation. However, the fast-set representation requires at least $\Theta(kn^2)$ memory, such that even the smallest benchmark used in our experimental evaluation (cf. Section 4) would require over 138GB. The difficulty of using Chaudhuri's approach for Java benchmarks was similarly observed in [33]. A modified version of the Melski-Reps algorithm is shown in Algorithm 1. The $\mathcal{O}(kn^2)$-sized worklist must, for each edge, check $\mathcal{O}(k)$ production rules in an attempt to find L-length paths containing the dequeued edge. Extending the path in general requires finding $\mathcal{O}(kn^2)$ edges that join temporary B or F nodes to new nodes, resulting in a worst-case runtime complexity of $\mathcal{O}(Lk^3n^4)$. Typically, the worklist algorithm requires the production rules to be normalised to a **binary normal-form** [16], which creates new non-terminals that break up productions with more than two symbols on their right-hand-side. Our modification to the algorithm allows it to work (albeit inefficiently) with non-normalised grammars. Importantly, though, the complexity becomes the expected $\mathcal{O}(k^3n^3)$ [18] when the grammar is normalised, because the path is only expanded once, from a single node, requiring $\mathcal{O}(kn)$ work instead of $\mathcal{O}(Lkn^2)$. The cubic-time required by this algorithm is well understood in the literature to be the bottleneck for many program analyses [11].

3 Novel CFL-R Algorithm

3.1 Semi-naïve Evaluation

The CFL-R algorithm due to Melski and Reps [18], as shown in Algorithm 1, introduces inefficiencies. Consider Figure 1: the Melski-Reps algorithm would discover the $(2, 5, S)$ edge up to nine times, since there are three potential paths ($\langle 2, 1, 5\rangle$, $\langle 2, 1, 2, 4, 5\rangle$ and $\langle 2, 1, 2, 1, 3, 4, 5\rangle$), and each one can be expanded from either the "a", "b" or "S" edge. The actual number of times it is discovered depends on which order edges are dequeued from the worklist, so the evaluation is also chaotic. This issue has been solved in the Datalog context before by a bottom-up strategy known as **semi-naïve evaluation** [2].

Datalog is a declarative programming model which derives information from base facts according to expansion rules. Facts are written $\texttt{A}(1, 5)$, meaning that the relation `A` contains a pairing between 1 and 5. A rule composes relations, so that $\texttt{S}(\texttt{u}, \texttt{x})$ `:-` $\texttt{A}(\texttt{u}, \texttt{v}), \texttt{S}(\texttt{v}, \texttt{w}), \texttt{B}(\texttt{w}, \texttt{x})$ implies new `S` relations can be derived by stringing `A`, `S`, and `B` relations together. The example 1 could be translated to a Datalog program in this fashion.

When Datalog begins bottom-up evaluation, the relations are empty. Facts are inserted into the relations, and the bodies of the rules are evaluated to obtain new knowledge. Since Datalog relations are bounded, a fixed-point will be reached after a finite number of iterations, which constitutes the solution. A naïve implementation of the bottom-up strategy, would iterate over the bodies of clauses several times with the same knowledge over and over, rediscovering already-known relations many times. Yet worse, if a relation is already stable in an iteration (i.e. more iterations do not obtain more knowledge), the relation is re-computed in all subsequent iterations. To overcome the problem of re-computation, the semi-naïve evaluation was introduced.

The semi-naïve evaluation strategy is two-fold. Firstly, it uses the new information discovered in the previous iteration (or, initially, the base facts) to derive new information for the current iteration, called the Δ relation. Secondly, it only derives new information for relations whose dependant relations (those appearing in the rule body) have stabilised, or reflexive-transitively depend on it (such as recursive rules), until that relation stabilises. For a more in-depth presentation of semi-naïve evaluation, refer to [2].

Yannakakis, in his seminal work [29], introduced the fundamental relationship between Datalog and CFL-R, i.e., clauses in chain-rule format become productions:

$$[X \rightarrow Y_1\, Y_2 \ldots Y_k] \Leftrightarrow \texttt{X}(\texttt{a}, \texttt{c})\ \texttt{:-}\ \texttt{Y}_1(\texttt{a}, \texttt{b}_1), \texttt{Y}_2(\texttt{b}_1, \texttt{b}_2), \ldots, \texttt{Y}_\texttt{k}(\texttt{b}_{\texttt{k}-1}, \texttt{c})$$

and facts $\texttt{X}(\texttt{u}, \texttt{v})$ become labelled edges $(u, v, X) \in E$ in the CFL-R input graph. In this way, labels in the CFL-R problem and binary relations in the Datalog formulation are conceptually identical. Yannakakis' original intention was to convert sub-classes of Datalog to CFL-R, obtaining an efficient solving vehicle. For our new algorithm, we use the semi-naïve evaluation as a scaffolding, and translate CFL-R instances to Datalog programs in the reverse direction of

Yannakakis' reduction. We specialise Datalog's semi-naïve evaluation to obtain a new algorithm for CFL-R, which is efficient by virtue of avoiding redundant computations.

Firstly, the evaluation strategy must only use recently discovered information to determine new information. We record a label's new information in a Δ relation, which is updated during the evaluation of production rules, and zeroed after it has been used, avoiding redundancy. The semi-naïve strategy also specifies that we should not compose label relations if the labels on which they depend have not stabilised. The correct order for these micro-fixed-point calculations is deduced from the **dependency graph** $G = ((\Sigma \cup \mathcal{N}), \{(X, Y) : [X \rightarrow ...Y...] \in \mathcal{P}\})$, whose nodes are labels and whose edges express dependencies between two labels. The left-hand-side label in a production depends on all the labels on the right. In the case the dependency is cyclic, we simply iteratively evaluate the production rules until all inter-dependant relations reach a fixed-point.

The semi-naïve-based algorithm is presented in Algorithm 2. This algorithm assumes a binary-relation representation, where the label Y has a relational representation $\mathbf{Y} = \{(a, b) : (a, b, Y) \in E\}$. Instead of using relational algebra operations [2] for evaluating the body of a CFL-R clause, we observe that the CFL-R clauses resemble cascaded equi-joins, which are further reduced to relational compositions, i.e.,

$$\begin{aligned}
&\{(a, c) : (a, b_1) \in \mathbf{Y_1} \wedge (b_1, b_2) \in \mathbf{Y_2} \wedge \ldots \wedge (b_{k-1}, c) \in \mathbf{Y_k}\} && = \\
&\{(a, c) : \mathbf{Y_1}(a, b_1) \bowtie \mathbf{Y_2}(b_1, b_2) \bowtie \ldots \bowtie \mathbf{Y_k}(b_{k-1}, c)\} && = \\
&\mathbf{Y_1} \circ \mathbf{Y_2} \circ ... \circ \mathbf{Y_k}
\end{aligned}$$

where $\mathbf{P} \circ \mathbf{Q} = \{(r, t) : (r, s) \in \mathbf{P}, (s, t) \in \mathbf{Q}\}$.

3.2 Quadtrees

The Datalog formulation from Section 3.1 performs relational composition, set-difference and union operations (Algorithm 2, Line 14). We therefore require a data-structure with low space and runtime overheads for these operations. In this work the **quadtree** representation of Boolean matrices is chosen as a suitable data structure. Quadtrees have better time-complexity operations than, for example, adjacency lists [18], and smaller memory utilisation than a dense-matrix representation [5].

Initially, we examine Boolean matrices as a vehicle for relational composition. A binary relation A can be represented as a Boolean matrix $\hat{A}$ whose elements are defined by:

$$\hat{A}_{ij} = \begin{cases} 1, & \text{if } (i, j) \in \mathbf{A} \\ 0, & \text{otherwise} \end{cases}$$

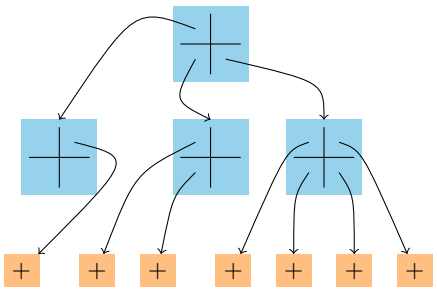
Assuming a one-to-one mapping between the domains of the relation and the indices of the matrix, the well-known identity $\widehat{A \circ B} = \hat{A} \cdot \hat{B}$ can be established, permitting the computation of CFL-R using matrix calculus. Used in this way, Boolean-matrix relational-composition would increase the complexity

Algorithm 2. Semi-naïve CFL-R algorithm using quadtrees(c.f. Section 3.2)

```
1: procedure QUADTREE(L)
2:     ⟨C_1, ..., C_q⟩ ← reverse_topological_strongly_connected_comps(P)
3:     for all [X → ϵ] ∈ P do
4:         X ← X ∪ {(v, v) : v ∈ V}
5:     end for
6:     for all X ∈ Σ ∪ N do
7:         ΔX ← X                          ▷ All Δs initialised to the problem state.
8:     end for
9:     for C_i = C_1 to C_q do
10:        while ∃Z ∈ C_i s.t. |ΔZ| > 0 do
11:            Temp ← ΔZ
12:            ΔZ ← 0
13:            for all [Y → X_0, ..., Z, ..., X_L] ∈ P do
14:                ΔY ← ΔY ∪ (X_0 ∘ ... ∘ Temp ∘ ... ∘ X_L) \ Y
15:                Y ← Y ∪ ΔY
16:            end for
17:        end while
18:    end for
19: end procedure
```

of the solver from $\mathcal{O}(k^3n^3)$ to $\mathcal{O}(k^3n^2BMM(n))$, where the time complexity of Boolean-Matrix-Multiplication, $BMM(n)$, is roughly $\mathcal{O}(n^{2.3})$ [7]. This time bound is derived from Algorithm 2, which loops Line 10 at most kn^2 times, and propagates the chosen delta to at most k non-terminals each loop, requiring k matrix multiplications for each propagation.

To minimise the overhead imposed by the matrix-formulation we turn to a quadtree representation. Quadtrees are a well-known matrix representation in the field of computer graphics, and they have some useful theoretical properties. Figure 2 shows a quadtree and the Boolean matrix it represents. For the CFL-R application, we are interested in the time requirements for set-difference and multiplication operations, as well as the memory requirements of quadtrees. Our



$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

Fig. 2. Quadtree representation of a 4x4 Boolean matrix

intuition here relies on the fact that quadtrees perform very well for sparse matrices, let m be the number of set bits in the matrix, i.e. $m < n^2$.

Lemma 1. *The quadtree requires* $\mathcal{O}(\min(n^2, m \log n))$ *space to store.*

The absolute size of the quadtree is bounded above by $\mathcal{O}(n^2)$. Consider the 1-matrix with side-length n, it is clearly maximal, as all nodes have the maximum number of children. Its quadtree has n^2 leaves, each representing a single 1 element, with each layer having $\frac{1}{4}$ as many nodes as the layer below it. The total number of nodes is at most:

$$\sum_{f=0}^{\infty} n^2 \frac{1}{4^f} = \frac{n^2}{1 - \frac{1}{4}} = \frac{4n^2}{3}$$

For a more practical bound, we say that m bits are set. In this case, each set bit requires at most $\log n$ nodes joining it to the root of the tree, thus no more than $m \log n$ nodes are needed for m set bits. This count is bounded above by the known n^2 limit, requiring $\min(n^2, m \log n)$ nodes.

Corollary 1. *The time complexity of the elementary operations: union, intersection, set-difference and deep-copy, is also* $\mathcal{O}(\min(n^2, m \log n))$.

Multiplication of two Boolean matrices is defined intuitively for quadtrees:

$$\left(\begin{array}{c|c} A_0 & A_1 \\ \hline A_2 & A_3 \end{array}\right) \cdot \left(\begin{array}{c|c} B_0 & B_1 \\ \hline B_2 & B_3 \end{array}\right) = \left(\begin{array}{c|c} A_0B_0 \cup A_1B_2 & A_0B_1 \cup A_1B_3 \\ \hline A_2B_0 \cup A_3B_2 & A_2B_1 \cup A_3B_3 \end{array}\right) \tag{1}$$

Unfortunately, it is difficult to quantify the expected runtime of the recursive algorithm derived from that definition. In this paper we prove an upper-bound on the runtime, and provide an intuition as to the expected runtime for fixed m.

Lemma 2. *Multiplication of two quadtrees requires* $\mathcal{O}(n^3)$ *time.*

Via application of the master theorem for recurrence relations. The recursive algorithm for Equation 1 requires eight sub-multiplications and four sub-unions, and recurses to a depth of $\log_2 n$. We know the unions have $\mathcal{O}(n^2)$ complexity, from Corollary 1, hence the recurrence equation of this system is:

$$I_x = 8I_{x-1} + 4x^2 + 1 \Rightarrow \mathcal{O}(8^{\log_2 n}) = \mathcal{O}(n^3)$$

Note that though this worst-case complexity does occur in practice (for two complete matrices), the time required can vary substantially. Furthermore, the computational load for matrices with m set-bits is difficult to reason about. Adverse arrangements of inputs with $m = \mathcal{O}(n)$ can incur output matrices with all or none of their bits set.

An intuition on the average-case complexity for fixed m arises by examining the case of balanced quadtrees. We call this a **J-tree** (jellyfish), because its nodes have maximal children towards the root (the bell) and at most one child below the bell (the tentacles).

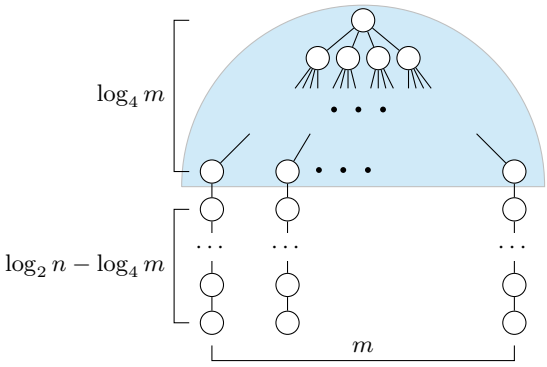


Fig. 3. A J-tree, showing the distinction between bell (upper semi-circle) and tentacles. Nodes in the bell have maximal branching factor, whilst tentacle nodes have at most one child. The height of a quadtree is always $\log_2 n$, so the height of the tentacle section is that height less the $\log_4 m$-high bell.

Lemma 3. *Multiplication of two J-trees requires* $\mathcal{O}(m^{\frac{3}{2}} + m \log n)$ *time.*

The structure of the J-tree allows us to break up the multiplication into the bell and tentacle components according to the depth of the recursion. We see that if the recursion depth is above $\log_4 m$ then the nodes of both trees typically have 4 children, which imposes the most computational work. Conversely, if we are below a recursion depth of $\log_4 m$ then the nodes have 1 child, and very little computational work is required. To reason about the necessary computations we will analyse work done above the $\log_4 m$ cutoff, the bell, separately from the tentacles below.

The bell's computation is slightly different to that from Lemma 2. Instead the recurrence is to a depth of $\log_4 m$, which yields $\mathcal{O}(8^{\log_2 m^{\frac{1}{2}}}) = \mathcal{O}(m^{\frac{3}{2}})$. This is a true upper bound for J-trees, as nodes with fewer than 4 children impose strictly less work.

The tentacle nodes all have 1 child, so at most one of the eight sub-multiplications are necessary and none of the unions. There are m tentacles on each J-tree, since there are m set bits, thus the recursion's breadth is also m. Unlike for the bell, the two single-child nodes of the input will incur at most one sub-multiplication and no sub-unions, because each node only has one child. Each tentacle is $log_2 n - log_4 m$ nodes long, with one subroutine-call per node, hence in total the m tentacles require $m \log n$ subroutine calls or fewer. Each call simply checks which of the sub-matrices it needs to recurse to and makes the call in constant time. Hence the total work required is the $\mathcal{O}(m^{3/2})$ work for the bell, and $\mathcal{O}(m)$ lots of $\mathcal{O}(\log n)$ work for each tentacle, totalling $\mathcal{O}(m^{3/2} + m \log n)$.

This section has shown the favourable properties of quadtrees, which makes them useful to our adapted semi-naïve Algorithm 2. Quadtree multiplication is a means of performing the relational composition operation $\circ$, and had a favourable $\mathcal{O}(\min(m \log n, n^2))$ memory footprint. The typical quadtree operations have

favourable time-complexities in the worst and average case, and we intuit that multiplication itself takes $\mathcal{O}(m^{3/2} + m \log n)$ time, making it very efficient for sparse problems.

3.3 Notes on Normalisation

Conventional algorithms for CFL-R require the input grammar to be normalised, either to Chomsky normal-form, or the less restrictive binary normal-form. Normalisation increases the size of the grammar, typified by the Chomsky requirement that right-hand sides contain exactly two non-terminals, causing the number of non-terminals to double and the number of rules to expand quadratically.

From a complexity-theoretic standpoint, normalisation is acceptable, since it is computationally cheap, and the size of the grammar is often not a component of the algorithm's time complexity. In practice, liberal expansion of the grammar incurs large overhead in the memory requirements of the algorithm. Chaudhuri's sparse-set method [5] typifies this, as it requires $\Theta(kn^2)$ memory, requiring terabytes of RAM for typically sized problems.

In this work, we choose to remove the requirement that grammars be pre-normalised. As Section 2 showed, the time complexity of Algorithm 1 increases without normalisation. The Melski-Reps formulation's inner loop can no longer rely on the fixed-form of grammar rules. Searching the graph for paths whose labels exactly match the right-hand-side of a production can naïvely require $k(kn^2)$ steps, making the complexity $\mathcal{O}(k^4n^4)$, clearly worse than $\mathcal{O}(k^3n^3)$ when the grammar is normalised. The adapted semi-naïve method, Algorithm 2, composes binary relations via matrix multiplication. Its presentation already allows for arbitrarily long production rules, and therefore retains the $\mathcal{O}(k^3n^2BMM(n))$ running time.

From a theoretical standpoint, normalising the grammar makes no difference to our quadtree-based-semi-naïve formulation. Shortening the length of the matrix multiplication chains directly increases the number of such chains that must be evaluated. Indeed, normalisation may even be considered an unnecessary overhead, since despite having the same computational complexity, a normalised grammar imposes excess memory requirements by retaining the edge and Δ information for intermediate nonterminals.

4 Experimental Results

For the experimental evaluation of our new CFL-R algorithm we use a case study of Java based points-to analysis. We evaluate our algorithm in comparison to the Melski-Reps worklist algorithm [18]. Specifically we are interested in the execution time, memory utilisation, and sensitivity towards grammar normalisation for both approaches.

The competing implementations will be referred to as the worklist and quadtree methods, and refer to Algorithms 1 and 2 (using quadtrees) respectively. Both algorithms were implemented in C++ making partial use of the STL library. The

source code of both algorithms is available online [12]. Our experimental evaluation is performed on a 32 core Intel Xeon E5-2450 at 2.1GHz, with 128GB ram.

Case Study. Our CFL-R case study is a points-to analysis expressed in CFL-R for Java. As a benchmark suite we choose the Dacapo benchmarks Version 2006-10. For extracting the labelled input graphs from the Java source code in the Dacapo benchmarks, we have used the DOOP extractor [4]. The extractor produces a set of relations representing input programs in relational format. The DOOP relations are sufficient to generate a labelled input graph for a context-insensitive, flow-insensitive, and field-sensitive points-to analysis, with minimal textual preprocessing. A vertex in the input problem is either a program variable or an object creation site (i.e. representing an object to the program analysis). The input edges are labelled with the following terminals:

- **New:** relating program variables to their object creation sites
- **Assign:** relating a source variable with a destination variable of an assignment statement
- **PutField$_f$, GetField$_f$:** for each field f, relating base variables of field loads/stores to the program variables that load/store information from/into the object of the base variable
- **GetInstanceField, PutInstanceField:** self-edges identifying base variables of field loads/stores
- **Cast$_t$:** relating program variables to the variables they are t-cast from
- **IsHeap$_t$:** self-edge identifying all polymorphic types t of an object

The Java points-to analysis uses the grammar shown in Figure 4. The result of the CFL-R algorithm produces output edges labelled with the following non-terminals:

- **VPT:** relating program variables to heap objects that they may point to
- **Alias:** relating two program variables if they may reference the same heap object
- **DAssign:** relating program variables which are assigned indirectly by field store and load
- **GetInstanceVPT, PutInstanceVPT:** relating the heap objects to the subset of variables which point to them that are derived by field load/stores

The CFL-R grammar is an extension of the grammar presented by Sridharan et al. in [25]. Field sensitivity is ensured by the $DAssign$ production, which is equivalent to

$$[\mathit{flowsTo} \rightarrow \mathit{flowsTo}\ \mathit{putField}_f\ \mathit{alias}\ \mathit{getField}_f]$$

from the Sridharan et al. formulation. We have extended the grammar to capture a type-safe casting with rule $[VPT \rightarrow Cast_t\ VPT\ IsHeap_t]$. Types are encoded in the input graph by a self edge $(h, h, IsHeap_t)$ for all types t which the h object can take. For performance reason, we compute $Alias$ relation only for base variables of field loads/stores.

$$
\begin{aligned}
[\mathit{Alias} &\rightarrow \mathit{GetInstanceVPT}\ \overline{\mathit{PutInstanceVPT}}] && \\
[\mathit{DAssign} &\rightarrow \mathit{PutField}_f\ \mathit{Alias}\ \mathit{GetField}_f] && \text{for all fields } f \\
[\mathit{GetInstanceVPT} &\rightarrow \mathit{GetInstanceField}\ \mathit{VPT}] && \\
[\mathit{PutInstanceVPT} &\rightarrow \mathit{PutInstanceField}\ \mathit{VPT}] && \\
[\mathit{VPT} &\rightarrow \mathit{New}] && \\
[\mathit{VPT} &\rightarrow \overline{\mathit{Assign}}\ \mathit{VPT}] && \\
[\mathit{VPT} &\rightarrow \overline{\mathit{DAssign}}\ \mathit{VPT}] && \\
[\mathit{VPT} &\rightarrow \mathit{Cast}_t\ \mathit{VPT}\ \mathit{IsHeap}_t] && \text{for all types } t
\end{aligned}
$$

Fig. 4. The parameterised grammar for field-sensitive context-insensitive Java points-to analysis used in our experiments. We adapt the grammar by Sridharan et al. from [25]. The parameters f and t take arbitrary values depending on the fields and types in the input problem.

Problem Sizes. Table 1 shows the relationship between the problem size and the intermediate and output relation sizes. The n, m(avg) and m(max) columns respectively show: the number of vertices in the input graph, the number of edges (averaged across all relations) after running the CFL-R algorithm, and the maximum number of edges of all labels. Unparameterised nonterminals are counted as-is, but parameterised nonterminals are counted together, so that the *Load* relation is the sum of the sizes of all Load_{field} sub-relations. We also chart the associated sizes of the quadtrees (total number of nodes in the tree) in a similar fashion. As we have shown in Section 3.2, the quadtree's size is bounded by $\mathcal{O}(m \log n)$ nodes, yet the actual sizes are significantly better, the Norm. column shows the normalised fraction $QT_{max}/(m_{max} \log n)$, and improves our intuition on the quadtrees practical size.

The Labels and Labels-nf columns of Table 1 show the number (input and normalised, respectively) of labels in the problem. They show that normalisation imposes a 35%-43% increase in the number of labels. Here we acknowledge that this is an artefact of the grammar we are using. Nevertheless, the fact that our method obviates the need for normalisation still proves useful, as it will be no worse, and definitely can improve memory efficiency.

The sparse nature of the problem is difficult to see from the tables, and is better shown in Figure 5. Here we show the logarithmic index of m in terms of n, knowing that the theoretical maximum is two (i.e. an edge between every pair of nodes). In practice (at least for points-to) m is on average less than n^1, and even in the worst cases is only $n^{1.14}$ in the `jython` benchmark. This validates our intuition from Section 3.2, that CFL-R problems are highly sparse. We observe a pattern of increasing log-indices according to the size of the problem, which we attribute to over-approximation in the analysis as the potential for more edges increases.

Table 1. Statistical information for the Dacapo benchmarks (ordered by problem size n). Includes: the number of Labels ($|\Sigma \cup \mathcal{N}|$) for the input and normalised (-nf) grammar, the problem size (n) and the size of the average and maximum output set-bits (m) and quadtree node-count (QT). Norm shows the ratio between the expected quadtree size $\mathcal{O}(m \log n)$ and the maximum's actual size.

Benchmark	Labels	Labels-nf	n	m (avg)	m (max)	QT (avg)	QT (max)	Norm.
luindex	1653	2301	22699	3413.62	12993	16391	60986	0.32
pmd	1755	2399	32295	12833.23	54660	41645.92	143505	0.18
antlr	1095	1520	32927	5916.85	21891	25573.31	93252	0.28
eclipse	2257	3172	33912	4748.08	21313	23861.54	99813	0.31
bloat	1900	2650	40989	9068.77	57829	35733.54	177415	0.2
xalan	2449	3436	46780	8606.54	42461	38492.15	155194	0.24
chart	2914	4166	49893	8717.85	39753	40075.77	160375	0.26
fop	3495	4742	53851	8591.54	38802	41334.15	160827	0.26
hsqldb	2817	3974	63281	24412.38	200762	86334.77	592910	0.19
jython	3351	4588	78639	40349.77	383917	135013.23	1102813	0.18

Table 2. Absolute runtime (s) of points-to analysis for the Dacapo benchmarks by Worklist and Quadtree implementations of the solver with (-nf) and without grammar normalisation

Benchmark	Worklist	Worklist-nf	Quadtree	Quadtree-nf
luindex	2.518	2.05	0.726	0.862
pmd	25.348	18.462	8.806	8.436
antlr	3.218	2.614	0.968	1.038
eclipse	7.326	5	1.794	2.042
bloat	12.61	9.548	2.724	2.99
xalan	14.81	11.554	3.9	4.296
chart	17.392	13.782	4.266	4.602
fop	13.828	10.756	3.318	3.812
hsqldb	47.282	36.132	12.218	12.636
jython	96.688	73.264	24.72	24.208

Runtime. We first compare the execution time for the standard worklist algorithm. Table 2 records the absolute runtime (in seconds) of the Dacapo benchmarks for the Worklist and Quadtree implementations, with (-nf) and without grammar-normalisation.

Since we are interested in the performance and scaling behaviour of the quadtree implementation, Figure 6 plots the relative speedup of those implementations normalised to Worklist-nf. Observe that quadtrees universally outperform the worklist, with an average 2.93x speedup. The largest speedup occurs for the `bloat` benchmark, at 3.51x, and the smallest is `pmd`, with 2.10x. It is interesting that the extreme speedups/slowdowns do not occur with the largest and smallest benchmarks, `jython` and `luindex` respectively, which show 2.96x and 2.82x speedups. This is strong evidence that, although the worst-case complexity of CFL-R via

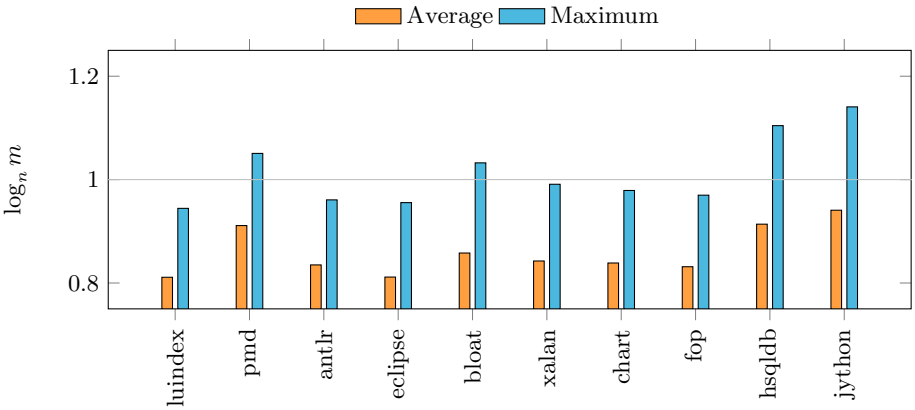


Fig. 5. Graphical plot for sparsity ($\log_n m$) from Table 1, showing that the average and maximum set-bits (m) range from $n^{0.8}$ to $n^{1.2}$ for all benchmarks

quadtrees is much worse, in practice it scales in the same manner as the worklist algorithm.

In support of our choice not to normalise the grammar, Figure 6 shows the effects of normalisation. We see the expected speed-increase for the worklist algorithm (which has historically been presented exclusively for normalised grammars). We also see virtually no change in execution times for the quadtree implementation, which intrinsically performs the work of the normalised grammar via intermediate matrices.

Memory Consumption. The peak memory consumption of the Worklist and Quadtree implementations in binary normal-form (-nf) and as-is, is recorded in Table 3. To assist understanding, relative memory usage against the normalised Worklist-nf implementation is plotted in Figure 7. We observe clear trends in the memory usage both between the implementations, and according to the normalisation of the grammar.

Firstly, the quadtree implementation clearly has a smaller memory footprint. Comparing the more favourable normalised worklist results against the non-normalised quadtree, we see universally less usage of memory, averaging to 0.39x. The greatest reduction occurs for `hsqldb`, one of the larger benchmarks, and the least for `antlr`, a smaller one. Furthermore, the smallest and largest benchmarks (`luindex` and `jython`) have 0.48x and 0.43x memory consumption respectively. Our results indicate that memory reduction does not seem to scale with problem size, but is more likely dependant on problem-specific information (such as the order of information).

We are also interested in how normalisation impacts memory usage. The worklist algorithm clearly benefits from normalisation in the expected manner, where

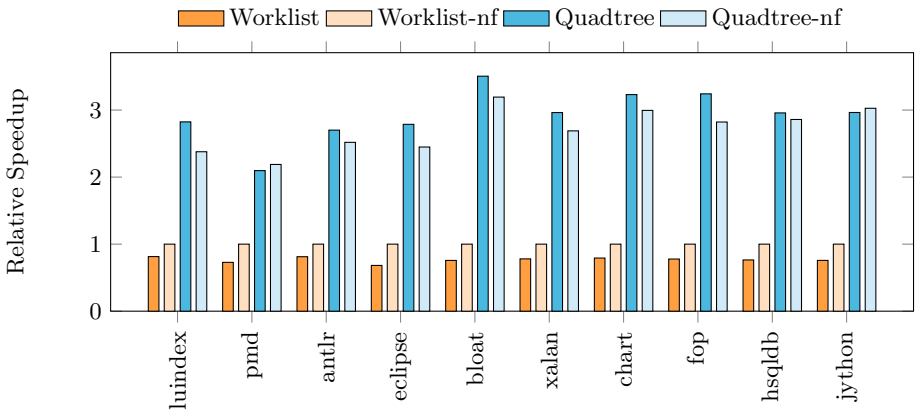


Fig. 6. Relative speedup of the Worklist, Quadtree and normalised Quadtree-nf against the normalised Worklist-nf implementation. Larger values show faster runtimes

Table 3. Absolute memory usage (MB) of points-to analysis for the Dacapo benchmarks by Worklist and Quadtree implementations of the solver with (-nf) and without grammar normalisation

Benchmark	Worklist	Worklist-nf	Quadtree	Quadtree-nf
luindex	149.86	72.5	34.87	35.13
pmd	347.88	163.29	53.73	89.31
antlr	149.43	80.07	51.46	51.71
eclipse	401.09	151.27	53.04	53.3
bloat	375.48	164.59	61.46	76.93
xalan	513.27	214.72	73.7	77.05
chart	568.72	239.61	79.84	81.65
fop	614.94	255.34	82.3	84.39
hsqldb	853.08	433.1	109.55	164.72
jython	868.43	415.32	177.75	277.91

memory consumption drops on average 0.46x. This result is not particularly interesting, since the worklist algorithm is not the focus of our research, however the reader should note that this memory drop must be attributed to the large intermediate-result set that is computed for longer rule chains. In comparison, we see a slight increase in memory consumption when normalising the grammar for the quadtree algorithm. Unlike other metrics, the experiments do show that the larger benchmarks `jython` and `hsqldb` show the largest increases, more than 1.5x. This result is to be expected, and motivates the ideas of Section 3.3, which is that maintaining intermediate results permanently becomes problematic for particularly large problems.

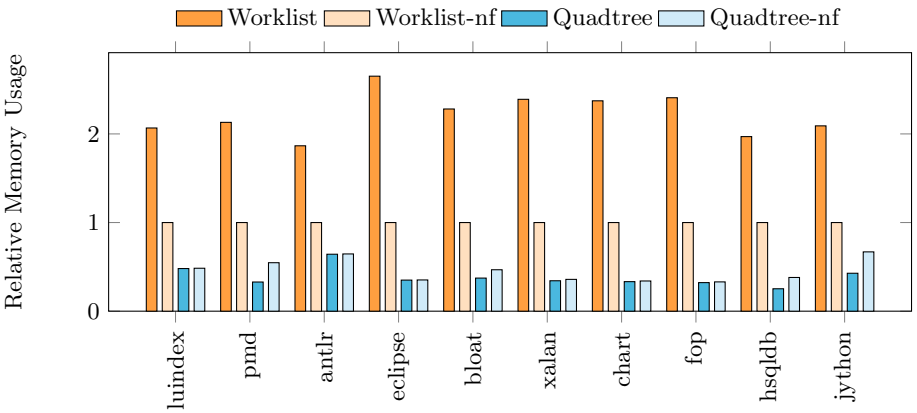


Fig. 7. Relative memory usage of the Worklist, Quadtree and normalised Quadtree-nf against the normalised Worklist-nf implementation. Smaller values show a reduced memory footprint

5 Related Work

Recognition of context free languages is one of the oldest formalisms in theoretical computer science. The first efficient algorithms for recognition were proposed independently by Cocke [6], Younger [30] and Kasami [14], and subsequently improved by Valiant [26] and generalised by Okhotin [20].

The reachability variant of CFL was formalised by Yannakakis [29] as a data-flow evaluation strategy. Our work relies on reversing this encoding, so that we can apply data-flow techniques to a new context. It was later, through Reps [22] [23] [18] [24] [21], that the problem was popularised as as a vehicle for solving a range of computational problems.

In particular, CFL-R has been identified as a viable solver for many analyses. Notable ones are: shape analysis [22], constant propagation [24], control-flow analysis [27], set-constraint solving [18] [15], but particularly points-to analysis. There is much demand in the literature for fast and scalable points-to analysis [10] [19]. CFL reachability is valuable in this context because it provides a queryable "as-needed" framework, useful in incremental [17] or demand-driven [35] [25] [28] contexts. For this reason, our work uses the points-to benchmarks as a case-study for viable CFL-R algorithms.

Another line of research focuses on improving CFL-R algorithms. Very fast algorithms have been developed for restricted cases, particularly Dyck-grammars and bi-directed graphs:

$$\mathcal{P} = \{[S \to \epsilon], [S \to SS], [S \to A_f S \hat{A}_f]\}$$

$$\forall u, v \in V \wedge A_f, \hat{A}_f \in \Sigma : (u, v, A_f) \in E \Leftrightarrow (v, u, \hat{A}_f) \in E$$

Here f is a parameter which can take any value according to the input being solved. Yuan and Eugster first formulated an efficient Dyck-reachability algorithm

for bi-directed trees in [31]. Their work was later improved and extended by Zhang et al. in [33], which is able to solve bi-directed graphs in $\mathcal{O}(n + m \log m)$ and trees in $\mathcal{O}(n)$. In that work, the authors noticed that when a graph is bi-directed, Dyck-reachability forms an equivalence relation, whose equal members can be collapsed to representative nodes. Successively stratifying the intra-reachable sets in this way grants the significantly reduced time complexities which they reported. Unsurprisingly, there are few problem contexts in which the graph is naturally bi-directed. Introducing reverse edges for every parenthesis label leads the analysis to report an over-approximation of the actual reachable sets, which can still be useful depending on the problem context. The work in [33] is therefore of greatest use as a fast pre-processing step for a more precise and expensive analysis.

Unfortunately, results by Heintze and McAllester [11] and Reps [21] imply that the Dyck results are unlikely to generalise. Indeed, only Chaudhuri [5], using the Four Russians' Trick, has been able to improve on the long-standing cubic-time algorithm. As was stated in Section 2, we are unable to use Chaudhuri's advancement, since the memory required is excessive, though work by Zhang et al. has found a means of adapting it for C points-to analysis whilst retaining subcubic runtime [34].

Matrix multiplication has been used in the CFL context since Valiant [26]. Much of the theory surrounding matrices is concerned with efficient computations and representations of matrices in the natural domain, most famously the Coppersmith and Winograd algorithm for fast matrix-matrix multiplication [7]. For our CFL-R context, we are concerned with Boolean matrices, which are typically sparse. Long-standing algorithms for sparse matrix multiplication [9] have been improved recently by Yuster and Zwick [32]. This paper favours the quadtree representation, which was shown to be efficient both for memory and computations by Abdali and Wise in [1].

6 Conclusions

In this paper we present a radically different approach to the evaluation of CFL-R problems. Our work draws from well-researched ideas in the Datalog community, and applies them to a new context. We have successfully adapted the semi-naïve evaluation strategy of Datalog by using the memory-efficient quadtree representation both as a means of tracking Δ-information and as a relational-composition vehicle. The algorithm we develop has theoretical advantages over the traditional Melski-Reps approach [18], by eliminating many redundant calculations, and Chaudhuri's subcubic approach [5], by making efficient use of memory. Our advances have been implemented as a CFL-R solver, and compared experimentally with the current scalable state-of-the-art solver. The experimentation shows that our CFL-R algorithm brings up to 3.5x speedup and 60% memory reduction. Going forward, we intend to fully understand the average and worst-case runtime of quadtree-based semi-naive evaluation, to characterise the nature of real-world CFL-R problems, and to assess the viability of alternate data structures within the semi-naïve framework.

Acknowledgments. We would like to thank Andrew Santosa, Chenyi Zhang, Lian Li, Paul Subotic and Paddy Krishnan from Oracle Labs, Brisbane, Australia. This research is support by ARC Grant DP130101970.

References

1. Abdali, S.K., Wise, D.S.: Experiments with quadtree representation of matrices. In: Gianni, P. (ed.) ISSAC 1988. LNCS, vol. 358, pp. 96–108. Springer, Heidelberg (1989)
2. Abiteboul, S., Hull, R., Vianu, V. (eds.): Foundations of Databases: The Logical Level, 1st edn. Addison-Wesley Longman Publishing Co., Inc., MA (1995)
3. Bastani, O., Anand, S., Aiken, A.: Specification inference using context-free language reachability. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 553–566. ACM (2015)
4. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2009, pp. 243–262. ACM, New York (2009)
5. Chaudhuri, S.: Subcubic algorithms for recursive state machines. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, pp. 159–169. ACM, New York (2008)
6. Cocke, J.: Programming languages and their compilers. Courant Institute Math. Sci., New York, USA (1970)
7. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. Journal of Symbolic Computation 9(3), 251–280 (1990); Computational algebraic complexity editorial
8. Dolev, D., Even, S., Karp, R.M.: On the security of ping-pong protocols. Information and Control 55(1-3), 57–68 (1982)
9. Gustavson, F.G.: Two fast algorithms for sparse matrices: Multiplication and permuted transposition. ACM Trans. Math. Softw. 4(3), 250–269 (1978)
10. Hardekopf, B., Lin, C.: The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. SIGPLAN Not. 42(6), 290–299 (2007)
11. Heintze, N., McAllester, D.: On the cubic bottleneck in subtyping and flow analysis. In: Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science, LICS 1997, pp. 342–351. IEEE (1997)
12. Hollingum, N.: Source code for worklist and semi-naive cfl-r algorithms (October 2014), `http://sydney.edu.au/engineering/it/~nhol8058/cfl/`
13. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation, International edn. Pearson Education International Inc., Upper Saddle River (2003)
14. Kasami, T.: An efficient recognition and syntax-analysis algorithm for context-free languages. Technical report, DTIC Document (1965)
15. Kodumal, J., Aiken, A.: The set constraint/cfl reachability connection in practice. In: Proceedings of the ACM SIGPLAN, Conference on Programming Language Design and Implementation, PLDI 2004, pp. 207–218. ACM, New York (2004)
16. Lange, M., Leiß, H.: To cnf or not to cnf? an efficient yet presentable version of the cyk algorithm. Informatica Didactica 8, 2008–2010 (2009)
17. Lu, Y., Shang, L., Xie, X., Xue, J.: An incremental points-to analysis with cfl-reachability. In: Jhala, R., De Bosschere, K. (eds.) Compiler Construction. LNCS, vol. 7791, pp. 61–81. Springer, Heidelberg (2013)
18. Melski, D., Reps, T.: Interconvertibility of a class of set constraints and context-free-language reachability. Theoretical Computer Science 248(1–2), 29–98 (2000)

19. Mendez-Lojo, M., Burtscher, M., Pingali, K.: A gpu implementation of inclusion-based points-to analysis. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2012, pp. 107–116. ACM, New York (2012)
20. Okhotin, A.: Fast parsing for boolean grammars: A generalization of valiants algorithm. In: Gao, Y., Lu, H., Seki, S., Yu, S. (eds.) DLT 2010. LNCS, vol. 6224, pp. 340–351. Springer, Heidelberg (2010)
21. Reps, T.: On the sequential nature of interprocedural program-analysis problems. Acta Informatica 33(5), 739–757 (1996)
22. Reps, T.: Program analysis via graph reachability. Information and Software Technology 40(11-12), 701–726 (1998)
23. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1995, pp. 49–61. ACM, New York (1995)
24. Sagiv, M., Reps, T., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. In: Mosses, P.D., Nielsen, M., Schwartzbach, M. (eds.) TAPSOFT 1995. LNCS, vol. 915, pp. 651–665. Springer, Heidelberg (1995)
25. Sridharan, M., Gopan, D., Shan, L., Bodík, R.: 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 2005, pp. 59–76. ACM, New York (2005)
26. Valiant, L.G.: General context-free recognition in less than cubic time. Journal of Computer and System Sciences 10(2), 308–315 (1975)
27. Vardoulakis, D., Shivers, O.: Cfa2: A context-free approach to control-flow analysis. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 570–589. Springer, Heidelberg (2010)
28. Yan, D., Xu, G., Rountev, A.: Demand-driven context-sensitive alias analysis for java. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA 2011, pp. 155–165. ACM, New York (2011)
29. Yannakakis, M.: Graph-theoretic methods in database theory. In: Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 1990, pp. 230–242. ACM, New York (1990)
30. Younger, D.H.: Recognition and parsing of context-free languages in time n^3. Information and control 10(2), 189–208 (1967)
31. Yuan, H., Eugster, P.: An efficient algorithm for solving the dyck-cfl reachability problem on trees. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 175–189. Springer, Heidelberg (2009)
32. Yuster, R., Zwick, U.: Fast sparse matrix multiplication. ACM Trans. Algorithms 1(1), 2–13 (2005)
33. Zhang, Q., Lyu, M.R., Yuan, H., Su, Z.: 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 2013, pp. 435–446. ACM, New York (2013)
34. Zhang, Q., Xiao, X., Zhang, C., Yuan, H., Su, Z.: Efficient subcubic alias analysis for c. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems, Languages & Applications, OOPSLA 2014, pp. 829–845. ACM, New York (2014)
35. Zheng, X., Rugina, R.: Demand-driven alias analysis for c. SIGPLAN Not. 43(1), 197–208 (2008)