RC24126 (W0611-203) November 29, 2006
Computer Science

IBM Research Report

CFL-Reachability in Subcubic Time

Swarat Chaudhuri*
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

*Currently at University of Pennsylvania

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

CFL-Reachability in Subcubic Time*

Swarat Chaudhuri
University of Pennsylvania

Abstract

We present an $O(n^3/\log n)$-time algorithm for the all-pairs CFL-reachability problem. The result, obtained via an $O(n^3/\log n)$ algorithm for all-pairs reachability in pushdown automata, breaks a long-standing upper bound and the "cubic bottleneck" for Datalog chain queries and many program analysis problems. Next, using a new, DFS-based, $O(\min\{mn/\log n, n^3/\log^2 n\})$-time algorithm for directed transitive closure, we solve all-pairs reachability in *stack-bounded* pushdown automata in time $O(n^3/\log^2 n)$, improving on the previous cubic bound. Finally, we use fast matrix multiplication to compute reachability in *hierarchical* pushdown automata in $O(n^{2.376})$ time, also breaking the known cubic bound. The results identify a gradation in the complexity of the reachability problem for pushdown automata as recursion is restricted to varying degrees.

*This work was done when the author was an intern at IBM T. J. Watson Research Center, Hawthorne, NY.

1 Introduction

Given an edge-labeled directed graph and a context-free language L over the edge labels, the CFL-reachability problem for a pair of nodes (s, t) is to determine if there is a path from s to t labeled by a word in L (the "all-pairs" problem is defined in the obvious way). Customarily, the size of the instance is the number of nodes in the graph, and the CFL is assumed to have a fixed-size representation. The problem was first phrased by Yannakakis in 1990 [25] in the context of database theory—he showed that Datalog chain query evaluation on a database graph is equivalent to single-source, single-sink CFL-reachability. Subsequently, the problem has found numerous applications, particularly in programming language theory, where classic problems such as interprocedural data-flow analysis, context-sensitive slicing, type-based flow analysis, and important variants of alias and shape analysis are known to reduce to CFL-reachability [19, 18, 16, 17].

An equivalent problem is control-state reachability in nondeterministic pushdown automata (PDAs). Here the goal is to determine, for a pair of control states (q, q') of a PDA, if the automaton can start at state q with an empty stack and reach a state q'. This problem has been of interest to the program verification community in recent times, as PDAs can be used to model recursive programs, and reachability algorithms for PDAs to mechanically check if a program satisfies certain requirements [6, 2]. A special case is the emptiness problem for PDAs, which goes back much farther [12].

CFL-reachability is obviously a generalization of context-free parsing, and an $O(n^3)$ algorithm for its all-pairs variant follows [25] from the Cocke-Younger-Kasami algorithm for CFL-parsing [12]. However, while CFL-parsing may be reduced to boolean matrix multiplication [23] and thus solved in $O(n^{2.376})$-time, no subcubic algorithm has been found even for the restricted problems of PDA-emptiness or single-source, single-sink CFL-reachability (there is also a difference in complexity: CFL-parsing is NL-complete, while CFL-reachability is PTIME-complete). Because of this, researchers in the programming language community have sometimes attributed the "cubic bottleneck" for many program analysis problems to the hardness of CFL-reachability [18, 15]. At the same time, the only non-trivial lower-bound for the problem follows from the fact that CFL-parsing is as hard as boolean matrix multiplication [14].

In this paper, we obtain an $O(n^3/\log n)$ algorithm for all-pairs CFL-reachability via an algorithm of the same complexity for all-pairs reachability in PDAs (a corollary is a subcubic algorithm for emptiness of PDAs). Algorithms achieving such logarithmic speedups are known for a variety of problems, the earliest example being the Four Russians' speedup for boolean matrix multiplication [4]. More recent applications of speed-ups of this sort (some by a factor of $O(\log^2 n)$) include parsing for CFLs [11, 21] and 2-way PDAs [21], all-pairs shortest paths [10, 8, 7], matching and connectivity [10], diameter verification [5], etc. Our algorithm speeds up a dynamic-program computation common to all popular reachability algorithms for PDAs [19, 6, 2], which fills a table indexed by pairs of states (q, q') with information about whether an execution can go from state q and an empty stack to state q' and an empty stack. It has been noted [15] that these algorithms compute a kind of incremental transitive closure. To see why, note that if the automaton can go in the above way from q to q' and also from q' to q'', then it also has an empty-stack to empty-stack path from q to q''. This is a transitive closure operation, and it is "incremental" because new edges between pairs of states are discovered and added as the algorithm progresses (of course, unlike for general dynamic transitive closure,the added edges are not arbitrary). We speed up a variant of this computation by phrasing it as a vector computation involving rows and columns of the table to be filled, and using a fast vector (actually, set) data structure that pre-computes patterns common among vector operations arising in the main loop (similar data structures have been used in other contexts [5, 21, 8, 24]). The algorithm builds on Rytter's algorithm for recognizing 2-way pushdown languages [21], a problem closely related to, but simpler than, reachability in PDAs. Our model of computation is a RAM, but we avoid "suspicious" bit operations to emphasize that the speedup is not merely due to word-level parallelism.

Next we study the reachability problem for *bounded-stack PDAs*, which are PDAs where the stack never grows unboundedly. In addition to being natural, this subclass has a practical motivation: au-

tomata of this type can model structured programs without infinite recursive loops. In spite of this extra structure, however, these automata have not been known to have faster reachability algorithms than general PDAs. We show here that the nature of the reachability problem changes when we limit recursion in this way. Suppose the automaton has a push-transition from state q to state q'. Then q cannot be reachable from q', because if it were, there would be an infinite recursive loop. It turns out that as a result, the edges added while computing transitive closure have a depth-first structure, and we can solve the reachability problem using a transitive closure algorithm for directed graphs that allows the following kind of modifications to the instance: "if an edge (u, v) goes from one strongly connected component to another, then compute all descendants v' of v and add some edges from u based on the answer." Unfortunately, none of the existing subcubic algorithms for transitive closure [1] can handle such modifications. Consequently, we derive a new transitive closure algorithm for directed graphs.

Our transitive closure algorithm speeds up a procedure based on Tarjan's algorithm for strongly connected components (such algorithms have a sizeable literature [13, 9, 22]). The ideas behind the speedup are to use fast sets and cache patterns common to set computations that are repeated many times—the resultant complexity is $O(\min\{mn/\log n, n^3/\log^2 n\})$, where m is the number of edges in the graph. The technique builds on those used by Rytter in the context of 2-PDA recognition [21, 20] (a technique somewhat similar to Rytter's [20] shows up in Williams' recent work on matrix-vector multiplication [24]). Note that the algorithm matches, but does not improve, the best non-algebraic algorithms for transitive closure—Chan [7] mentions a way to compute transitive closure in time $O(mn/\log n)$ by modifying a transitive closure algorithm running on the condensation graph of a graph, and $O(n^3/log^2 n)$-time algorithms for transitive closure are known [21, 5]. However, a combination of these earlier algorithms do not give us an algorithm of the form that we need, as known $O(n^3/\log^2 n)$-time algorithms use a reduction to matrix multiplication and cannot handle modifications of the instance, and a mere $O(mn/\log n)$ complexity would not suffice for our purposes. On the other hand, our new algorithm gives us an $O(n^3/\log^2 n)$ solution to the all-pairs reachability in bounded-stack automata. We also believe that it may have independent interest, especially from a practical point of view.

Finally, we consider a subclass of bounded-stack PDAs known as *hierarchical PDAs* [3], which are PDAs capturing structured programs without recursive calls. Previous reachability algorithms for such automata were cubic; however, we give a simple alternative algorithm that uses a matrix-multiplication routine and runs in time $O(n^{2.376})$. While this is not a very interesting technical result by itself, it shows that there is a gradation in the complexity of reachability algorithms in PDAs as recursion gets constrained to varying degrees. We feel that this is an important, and hitherto unknown, insight.

The structure of the paper is as follows. In Section 2, we present some basic definitions, Section 4 discusses reachability for PDAs, and Section 5 states our result on CFL-reachability. In Section 6, we study reachability for bounded-stack PDAs and derive our transitive closure algorithm. In Section 7, we examine reachability for hierarchical PDAs. We conclude with some discussion in Section 8.

2 Basics

Pushdown automata A *(nondeterministic) pushdown automaton* (PDA) $\mathcal{A}$ has a finite set Q of states, stack and input alphabets Γ and Σ, an initial state q_0, an initial stack symbol γ_0, and a set F of final states. Transitions are picked nondeterministically and are of three forms: (1) $q \xrightarrow{a} q'$—a *local transition* on input symbol a that changes the state from q to q' without touching the stack; (2) $q \xrightarrow{a} (q', push(\gamma))$—here the state changes from q to q', and the stack symbol γ is pushed on the stack; and (3) $q \xrightarrow{a} (q', pop(\gamma))$—$\gamma$ is popped, and state changes from q to q'. We also allow ϵ-transitions corresponding to these transitions, and assume that the initial stack symbol does not appear in a push transition. The size of $\mathcal{A}$ is defined as $n = |Q||\Gamma|$.

A *configuration* of $\mathcal{A}$ is a pair $c = (q, w)$, where $q \in Q$ is a state and $w \in \Gamma^*.\{\gamma_0\}$ is a *stack*. The *initial configuration* is (q_0, γ_0), and the stack is said to be empty whenever it equals γ_0. For $a \in \Sigma$, a

configuration $c' = (q', w')$ is an *a-successor* of $c = (q, \gamma.w)$ if one of the following three conditions holds: (1) $w' = \gamma.w$, and either $q \xrightarrow{a} q'$, or $\gamma.w = \gamma_0$ and $q \xrightarrow{a} (q', pop(\gamma_0))$ (a pop-transition involving the initial stack symbol reads, but does not pop, the stack bottom); (2) $w' = w$, and we have $q \xrightarrow{a} (q', pop(\gamma))$; (3) $w' = \gamma'.\gamma.w$, and for some $\gamma' \in \Gamma$, we have $q \xrightarrow{a} (q', push(\gamma'))$. If c' is an a-successor of c for some a, call c' a *successor* of c. Final-state acceptance and the language $L(\mathcal{A})$ of $\mathcal{A}$ are now defined as usual.

Let $\rightsquigarrow_C$ be the (reflexive) transitive closure of the successor relation. The state q' is *reachable* from the state q (written as $q \rightsquigarrow q'$) if $(q, \gamma_0) \rightsquigarrow_C (q', w.\gamma_0)$ for some $w \in \Gamma^*$. The all-pairs (control-state) reachability problem for a PDA $\mathcal{A}$ is to determine, for each pair of states q, q', whether $q \rightsquigarrow q'$.

We will see that the reachability problem for PDAs is equivalent to CFL-reachability, and our algorithm for the latter goes via the former. However, reachability for PDAs is also of independent interest. For example, PDAs have been used extensively in program analysis as abstractions of structured programs, their pushes and pops modeling procedure calls and returns. Reachability algorithms for PDAs can then be used to check if these abstractions satisfy their specifications [6].

CFL-Reachability Let G be a directed graph with n nodes whose edges are labeled by an alphabet Σ, and let L be a context-free language over Σ given by a fixed-size grammar or automaton. Define paths in G as usual, and lift the labeling on edges to paths in the natural way. The *all-pairs CFL-reachability* problem for G and L is to determine, for every pair of nodes s and t in G, if there is a path from s to t labeled by a word in L (if there is, then we write $s \rightsquigarrow_L t$). Single-source or single-sink variants of the problem are defined in the natural way. The best-known algorithms for all-pairs, single-source, and single-sink CFL-reachability all run in time $O(n^3)$. The single-source, single-sink CFL-reachability problem is PTIME-complete [25]. It also has a trivial reduction from context-free parsing, a problem known to be as hard as boolean matrix multiplication [14].

It is known that CFL-reachability has a linear reduction to reachability for PDAs (there is also a reduction in the other way, but it is unnecessary for our purposes). Consider the CFL-reachability problem for a graph G and the language L of a fixed-size PDA $\mathcal{A}$. Now construct the PDA $\mathcal{A}_G$ that is the product of $\mathcal{A}$ and G. States of $\mathcal{A}_G$ are of the form (q, s), where q is a state of $\mathcal{A}$ and s a node in G, and stack symbols of $\mathcal{A}_G$ are the same as stack symbols of $\mathcal{A}$. A transition $(q, s) \xrightarrow{a} (q', s')$ (likewise, $(q, s) \xrightarrow{a} ((q', s'), push(\gamma))$ and $(q, s) \xrightarrow{a} ((q', s'), pop(\gamma))$) exists in $\mathcal{A}_G$ iff there is an a-labeled edge from s to s' in G, and $\mathcal{A}$ has a transition $q \xrightarrow{a} q'$ (likewise, $q \xrightarrow{a} (q', push(\gamma))$ and $q \xrightarrow{a} (q', pop(\gamma))$). A transition $(q, s) \xrightarrow{\epsilon} (q', s)$ (ϵ-transitions that push or pop are handled similarly) exists in $\mathcal{A}_G$ iff $\mathcal{A}$ has a transition $q \xrightarrow{\epsilon} q'$. It is easily seen that if q_0 and γ_0 are the initial state and stack symbol of $\mathcal{A}$, then $s \rightsquigarrow_L t$ iff $(q_0, s) \rightsquigarrow (q_f, t)$ in $\mathcal{A}_G$ for some final state q_f of $\mathcal{A}$. If G has n nodes, then the size of $\mathcal{A}_G$ is $\theta(n)$, so that for $f(n) = \Omega(n)$, an $O(f(n))$ algorithm for all-pairs reachability in PDAs leads to an $O(f(n))$ algorithm for all-pairs CFL-reachability.

Bounded-stack and hierarchical PDAs We are interested in two special classes of PDAs that arise in applications and, as we shall see, have better algorithms than general PDAs.

The first class, that of *bounded-stack PDAs*, consists of PDAs $\mathcal{A}$ where for any push-transition $q \xrightarrow{a} (q', push(\gamma))$ to exist, q must be unreachable from q'. Intuitively, such a PDA forbids recursive use of push-transitions, so that in any run, the height of the stack stays bounded. To see an application, consider a procedure that accepts a boolean value as a parameter, flips the bit, and, if the result is 1, calls itself recursively. Such a program can be modeled by a bounded-stack PDA.

The second class, that of *hierarchical PDAs*, corresponds to *non-recursive* procedural programs modifying finite data. To define it, we need to add some more structure to PDAs. Consider a PDA $\mathcal{A}$ whose state set Q and stack alphabet Γ are partitioned as $\langle Q_1, Q_2, \ldots, Q_k \rangle$ and $\langle \Gamma_1, \Gamma_2, \ldots, \Gamma_k \rangle$, and let the pair $\langle Q_i, \Gamma_i \rangle$ be the i-th *procedure* of $\mathcal{A}$. We ensure that local transitions are all intra-procedure and push and pop transitions lead from one procedure to another. In fact, pushes and pops need to have some more structure. Let us define a *call map* $\chi : \Gamma \setminus \{\gamma_0\} \to \{1, 2, \ldots, k\}$ that assigns to each push the name of the procedure it "calls". Then it is an invariant that if γ is at the top of the stack at some

point, then the execution is in procedure $Q_{\chi(\gamma)}$. Formally: (1) if $q \xrightarrow{a} q'$ or $q \xrightarrow{\epsilon} q'$, then q and q' belong to Q_i for some i; (2) for every push-transition $q \xrightarrow{a} (q', push(\gamma))$ or $q \xrightarrow{\epsilon} (q', push(\gamma))$, we have $q' \in Q_i$, where $i = \chi(\gamma)$; and (3) for every transition $q \xrightarrow{a} (q', pop(\gamma))$ or $q \xrightarrow{\epsilon} (q', pop(\gamma))$, we either have $\gamma \neq \gamma_0$, $q' \in Q_i$ and $\gamma \in \Gamma_i$ for some i, or $\gamma = \gamma_0 \in \Gamma_i$ and $q, q' \in Q_i$ for some i.

Automata as above are as expressive as pushdown automata [2]. *Hierarchical PDAs* form a proper subclass, requiring that there is a total order $\prec$ on the procedures $1, 2, \ldots, k$ such that for all $\gamma \neq \gamma_0$, if $\chi(\gamma) = j$ and $\gamma \in \Gamma_i$, then $i \prec j$. In other words, pushes (similarly pops) from a component may only lead to a procedure that is higher up (lower down) in this order. It is easily seen that such automata can capture control flow in non-recursive programs.

3 Set operations

In this section, we present a set data structure (called a *fast set* from now on) supporting fast implementations of the following three operations: (1) set difference, (2) insertion of an element into a set, and (3) *assign-union*: given sets X and Y, perform the assignment $X \leftarrow X \cup Y$ and return the new X. Variants of this data structure have been previously used by Rytter [21], Chan [8], and Williams [24], respectively for membership in two-way pushdown languages, all-pairs shortest paths in undirected graphs, and matrix-vector multiplication.

Let U be a universe of n elements, and let $p = \lceil \alpha \log n \rceil$ for some $\alpha > 0$. We partition U in advance into $r = \lfloor n/p \rfloor$ sets $U_1, U_2, \ldots, U_r$, where $|U_i| \leq p$ for all $1 \leq i \leq r$. In a preprocessing phase, for each i and $S \subseteq U_i$, we allocate a record R_S, and build a map $\Pi : U \to \{1, 2, \ldots, r\}$ such that for each $v \in U$, $\Pi(v) = i$ iff $v \in U_i$. We also construct the set $(S \setminus S')$ for each $1 \leq i \leq r$ and $S, S' \subseteq U_i$, representing it as a list, and store it in a table $\mathsf{BlockDiff}$ as an entry $\mathsf{BlockDiff}[R_S, R'_S]$. If $(S \setminus S')$ is a singleton $\{u\}$ (i.e., if S is obtained by inserting $\{u\}$ into S'), we store R_S in a cell $\mathsf{BlockIns}[R_{S'}, u]$ of a table $\mathsf{BlockIns}$. Now note that the bottleneck in preprocessing is filling the table $\mathsf{BlockDiff}$, which has $O(r.2^p.2^p) = O(n^{1+2\alpha}/p)$ cells. As computing each set difference takes $O(p)$ time, preprocessing costs time $O(n^{1+2\alpha})$, which is $O(n^2/\log n)$ for suitable α.

In a fast set representation, a set $X \subseteq U$ is a tuple $\langle R_{X_1}, R_{X_2}, \ldots, R_{X_r} \rangle$, where $X_i = X \cap U_i$ for all i. Now suppose we want to execute the operation $\mathit{Diff}(X, Y)$, which computes a list representation of $(X \setminus Y)$ for fast sets $X = \langle X_1, \ldots, X_r \rangle$ and $Y = \langle Y_1, \ldots, Y_r \rangle$. Letting $\circ$ denote list concatenation, the output is the list $\mathit{Diff}(X, Y) = \mathsf{BlockDiff}[R_{X_1}, R_{Y_1}] \circ \mathsf{BlockDiff}[R_{X_2}, R_{Y_2}] \circ \ldots \circ \mathsf{BlockDiff}[R_{X_r}, R_{Y_r}]$, which can be computed in time $O(r + |\mathit{Diff}(X, Y)|)$.

The operation $\mathit{Ins}(X, u)$, which inserts u into $X = \langle X_1, \ldots, X_r \rangle$, simply replaces the $\Pi(u)$-th coordinate of X by $\mathsf{BlockIns}[X_{\Pi(u)}, u]$. This can be done in $O(1)$-time. The operation $AU(X, Y)$, which takes in fast sets X and Y and assigns $(X \cup Y)$ to X, is implemented as: **for** $x \in \mathit{Diff}(Y, X)$ **do** $\mathit{Ins}(X, x)$. This may be done in time $O(r + |\mathit{Diff}(Y, X)|)$. Then:

Lemma 1 *Any time after an $O(n^2/\log n)$-time preprocessing phase, $\mathit{Diff}(X, Y)$ and $AU(X, Y)$, for sets $X, Y \subseteq U$ represented as fast sets, cost $O(n/\log n + |X \setminus Y|)$ and $O(n/\log n + |Y \setminus X|)$ respectively. The cost of $\mathit{Ins}(X, u)$, where $u \in U$, is $O(1)$.*

While we avoided bit operations in the above to emphasize that we do not "cheat" using bit-level parallelism, fast sets can be implemented more simply if unit-cost arithmetic and logical operations on $O(\log n)$-bit words are allowed. Assume that $U = \{0, 1, \ldots, n-1\}$; in each fast set $X = \langle R_{X_1}, \ldots, R_{X_r} \rangle$, R_{X_i} is now a length-p bit vector. A fast set representation of $Z = X \setminus Y$ is now easily computed in time $O(r)$ using bitwise operations; to list the elements of Z, we repeatedly locate the most significant bit in Z, add its position in X to the output list, and turn it off. This can be done in $O(|Z| + r)$ time [8]. Insertion of $0 \leq x \leq n-1$ involves setting a bit in $R_{\lfloor x/p \rfloor}$, and is obviously $O(1)$.

4 All-pairs reachability in pushdown automata

```
REACHABILITY()
 1  W ← H* ← {(u,u) : u ∈ Q_N} ∪ Locals
 2  while W ≠ ∅
 3  do (u,v) ← remove from W
 4     if u = q, v = q'  for some q, q' ∈ Q
 5        then for γ ∈ Γ
 6                do insert (call_{q,γ}, ret_{q',γ}) into H*, W
 7     for (u',u) ∈ H*  such that  (u',v) ∉ H*
 8     do  insert (u',v) into H* and W
 9     for (v,v') ∈ H*  such that  (u,v') ∉ H*
10     do  insert (u,v') into H* and W
11  for (u,v) s.t. ∃q ∈ Q, γ ∈ Γ with u = call_{q,γ}, v = q
12  do  insert (u,v) into H*
13  H* ←  transitive closure of H*
```

```
FAST-REACHABILITY()
 1  W ← H* ← {(u,u) : u ∈ Q_N} ∪ Locals
 2  while W ≠ ∅
 3  do (u,v) ←  remove from  W
 4     if u = q, v = q'  for some q, q' ∈ Q
 5        then for γ ∈ Γ
 6                do Ins(Col(ret_{q',γ}), call_{q,γ})
 7                   Ins(Row(call_{q,γ}), ret_{q',γ})
 8                   insert (call_{q,γ}, ret_{q',γ}) into W
 9     for u' ∈ Diff(Col(u), Col(v))
10     do Ins(Col(v), u');  Ins(Row(u'), v)
11        insert (u',v) into W
12     for v' ∈ Diff(Row(v), Row(u))
13     do Ins(Col(u), v');  Ins(Row(v'), u)
14        insert (u,v') into W
15  for (u,v) s.t. ∃q ∈ Q, γ ∈ Γ with u = call_{q,γ}, v = q
16  do  insert (u,v) into H*
17  H* ←  transitive closure of H*
```

Figure 1: All-pairs reachability in PDAs

In this section, we give an $O(n^3/\log n)$-time algorithm for reachability in pushdown automata. Our algorithm builds on Rytter's subcubic algorithm [21] to recognize languages of two-way pushdown automata (2-PDAs). This problem may be linearly reduced to single-source, single-sink PDA reachability. Notably, there is also a reduction in the other direction—given a PDA $\mathcal{A}$ where we are to determine reachability, write the transitions of $\mathcal{A}$ out as an input word. Now construct a 2-PDA $\mathcal{M}$ that, in every one of an arbitrary number of rounds, moves its head to an arbitrary transition of $\mathcal{A}$ and tries to simulate it. Using nondeterminism, $\mathcal{M}$ can guess any run of $\mathcal{A}$, and accept the input iff $\mathcal{A}$ has an accepting run. The catch, of course, is that a PDA of size n may have $\theta(n^2)$ transitions, so that this reduction is not linear, and does not solve PDA-reachability in subcubic time.

Let us now proceed to the algorithm. Consider a PDA $\mathcal{A}$ with state set Q and stack alphabet Γ. For simplicity, we assume in this version of the paper that $\mathcal{A}$ does not have a pop-transition of the form $q \xrightarrow{a} (q', pop(\gamma_0))$, where γ_0 is the initial stack symbol—all our results hold even in the absence of this assumption. Now translate $\mathcal{A}$ into a *normal form* $\mathcal{N}(\mathcal{A})$ as follows. Define a set Q' of auxiliary states consisting of a pair of auxiliary states $call_{q,\gamma}$ and $ret_{q,\gamma}$ for each $q \in Q$ and $\gamma \in \Gamma$. The set of states of $\mathcal{N}(\mathcal{A})$ is $Q_N = Q \cup Q'$, and the stack alphabet of $\mathcal{N}(\mathcal{A})$ is Γ. The initial state q_0 and the initial stack symbol γ_0 are the same in $\mathcal{A}$ and $\mathcal{N}(\mathcal{A})$.

Each transition $q \xrightarrow{a} q'$ or $q \xrightarrow{\epsilon} q'$ in $\mathcal{A}$ is also a transition of $\mathcal{N}(\mathcal{A})$. For each transition $q \xrightarrow{a} (q', push(\gamma))$ (similarly $q \xrightarrow{\epsilon} (q', push(\gamma))$) in $\mathcal{A}$, we have two transitions in $\mathcal{N}(\mathcal{A})$: $q \xrightarrow{a} call_{q',\gamma}$ (similarly $q \xrightarrow{\epsilon} call_{q',\gamma}$) and $call_{q',\gamma} \xrightarrow{\epsilon} (q', push(\gamma))$. For each pop-transition $q \xrightarrow{a} (q', pop(\gamma))$ (likewise $q \xrightarrow{\epsilon} (q', pop(\gamma))$) in $\mathcal{A}$, there is in $\mathcal{N}(\mathcal{A})$ a pop-transition $q \xrightarrow{\epsilon} (ret_{q,\gamma}, pop(\gamma))$ followed by a transition $ret_{q,\gamma} \xrightarrow{a} q'$ (likewise $ret_{q,\gamma} \xrightarrow{\epsilon} q'$). It can be shown that for states q, q' and stacks w, w' of $\mathcal{A}$, we have $(q,w) \rightsquigarrow_C (q',w')$ in $\mathcal{A}$ iff $(q,w) \rightsquigarrow_C (q',w')$ in $\mathcal{N}(\mathcal{A})$. Also, if $\mathcal{A}$ is of size n, then $\mathcal{N}(\mathcal{A})$ has $\theta(n)$ states and may be constructed in $O(n)$ time.

Now consider the graph H with node set Q_N and edge relation $\dashrightarrow$, where $\dashrightarrow$ is the least relation such that: (1) for states $u, v \in Q_N$, we have $u \dashrightarrow v$ if $u \xrightarrow{a} v$ in $\mathcal{N}(\mathcal{A})$ (edges of H obtained this way are known as *local edges*); (2) if $q \xrightarrow{a} (q', push(\gamma))$ is a push-transition in $\mathcal{A}$, then $q \dashrightarrow q'$ (these edges are known as *call edges*); (3) if $(q, \gamma_0) \rightsquigarrow_C (q', \gamma_0)$ in $\mathcal{A}$, then for all γ, we have $call_{q,\gamma} \dashrightarrow ret_{q',\gamma}$ (*summary edges*). This graph is called the *summary graph* of $\mathcal{N}(\mathcal{A})$. For example, Fig. 2 depicts the summary graph of the normalized form of a PDA with transitions $q_1 \xrightarrow{a} q_2$, $q_2 \xrightarrow{b} (q_1, push(\gamma))$, and $q_1 \xrightarrow{a} (q_3, pop(\gamma))$ (only the part reachable from states q_1, q_2, q_3 is shown). The one call-edge is dashed,

and the one summary-edge has a hollow head.

Let $\dashrightarrow^*$ be the (reflexive) transitive closure of H. It is known that:

Lemma 2 ([6, 2]) *For states q, q' of $\mathcal{A}$, $q \rightsquigarrow q'$ iff $q \dashrightarrow^* q'$.*

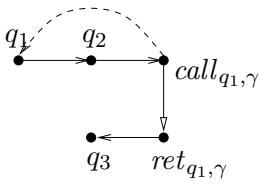

Figure 2: A summary graph

While computing the transitive closure of H would solve our problem, H is not complete initially and its summary edges depend on reachability relations between pairs of states. On the other hand, consider an *incremental* transitive closure algorithm that can answer queries "does $(q, \gamma_0) \rightsquigarrow_C (q', \gamma_0)$ hold?" and support insertion of summary edges. Such an algorithm would solve PDA-reachability (of course, unlike in usual dynamic algorithms, queries and insertions here are not independent). While it is unlikely that there is such an algorithm handling $O(n^2)$ summary edge insertions in $o(n^3)$ total time, this highlights the difference in flavor between graph closure and PDA-reachability.

Let us denote states of $\mathcal{N}(\mathcal{A})$ by $u, v, \ldots$, and let *Locals* be the set of local transitions (of the form $u \xrightarrow{a} v$) of $\mathcal{N}(\mathcal{A})$. Consider the algorithm Reachability (Fig. 1), which solves all-pairs reachability in $\mathcal{N}(\mathcal{A})$ (and hence in $\mathcal{A}$). The goal is to fill up a table H^* indexed by $Q_N \times Q_N$ that is, on termination, an adjacency matrix representation of the transitive closure of H. To do so, we use a worklist W. Line 1 inserts the local edges and trivial reachability facts. The while-loop from line 2–10 computes summary edges and reachability facts that arise via combinations of summary and local edges. By the time control reaches line 11, the uv-th entry $H^*[u, v]$ of H^* (for $u \neq v$) is 1 if $(u, \gamma_0) \rightsquigarrow_C (v, \gamma_0)$ in $\mathcal{N}(\mathcal{A})$, and 0 otherwise. Lines 11 and 12 insert the call edges, and line 13 takes care of reachability facts such as $(q, w) \rightsquigarrow_C (q', w')$, where w is a proper prefix of w'.

Regarding termination, the main loop (line 2) only runs α times, where α is the number of elements inserted into the worklist W. Since we do not try to insert the same pair twice, we have $\alpha \leq n^2$, and:

Lemma 3 Reachability *terminates on any PDA $\mathcal{A}$. On termination, for every pair of states u, v in $\mathcal{N}(\mathcal{A})$, we have $H^*[u, v] = 1$ if $u \rightsquigarrow v$, and $H^*[u, v] = 0$ otherwise.*

Now we speed up the algorithm Reachability using fast sets. Assume without loss of generality that $Q_N = \{0, 1, \ldots, n-1\}$, and represent H^* by row sets $Row(u) = \{v : H^*[u, v] = 1\}$ and column sets $Col(u) = \{v : H^*[v, u] = 1\}$, for $0 \leq u \leq n-1$. On termination, $Row(u)$ contains all v such that $u \rightsquigarrow v$, and $Col(u)$ contains all v such that $v \rightsquigarrow u$. These sets are maintained as fast sets, and the algorithm is rewritten as Fast-Reachability (Fig. 1).

Let us now analyze the complexity of Fast-Reachability. Lines 4–8 cost constant time *per insertion* of an element into H^*. Once an element is inserted, it is not taken out; also, we do not try to insert an element into the same cell twice. Thus, the total cost for lines 4–8 during a run of Fast-Reachability is $O(\alpha)$. Lines 9–11 cost $O(n/\log n + \sigma)$ in time in each iteration of the main while-loop, where σ is the number of entries inserted into H^* in each iteration. Consequently, the total cost for these operations is $O(\alpha.n/\log n + \alpha)$. A similar argument applies to lines 12–14. Lines 15 and 16 cost $O(\alpha)$ time, and the transitive closure in line 17 can be computed in $O(\alpha.n/\log n)$ time using the procedure we give in Section 6.1. Then we have:

Lemma 4 Fast-Reachability *terminates in time $O(\alpha.n/\log n)$, where $\alpha \leq n^2$ is the number of pairs $(u, v) \in Q_N \times Q_N$ such that $u \rightsquigarrow v$. On termination, for every u, v, we have $H^*[u, v] = 1$ if $u \rightsquigarrow v$, and $H^*[u, v] = 0$ otherwise.*

Theorem 1 *All-pairs reachability in a PDA of size n can be determined in time $O(n^3/\log n)$.*

The above implies the first subcubic algorithm for the emptiness problem of PDAs:

Corollary 1 *Given a PDA $\mathcal{A}$ of size n, we can check in $O(n^3/\log n)$ time if $L(\mathcal{A}) = \emptyset$.*

5 CFL-Reachability

By Theorem 1 and the discussion in Section 2, we immediately obtain:

Theorem 2 *The all-pairs CFL-reachability problem for a directed graph with n nodes can be solved in time $O(n^3/\log n)$.*

The above is an improvement of the previous cubic bound for this problem (or, for that matter, single-source, single-sink CFL-reachability). This also implies subcubic solutions to a number of practical problems previously believed to suffer from a "cubic bottleneck". Applications include:

- **Datalog:** *Chain queries* in Datalog are of the form $p(X, Y) \leftarrow q_0(X, Z_1), q_1(Z_1, Z_2), \ldots, q_k(Z_k, Y)$, where the q_i's are binary predicates, and X, Y and the Z_i's are distinct variables. Evaluation of such queries on the graph of a database reduces to the CFL-reachability problem [25].
- **Program analysis:** A large number of classic program analysis problems reduce to CFL-reachability. For example, consider interprocedural dataflow analysis [18, 19], which involves checking if a datum can reach a certain program point— the "context-free" aspect arises because the calls and returns in the program, or equivalently, pushes and pops to/from the program stack, are tracked precisely. Other applications include context-sensitive slicing, type-based flow analysis, and important variants of alias and shape analysis [19, 18, 16, 17].

A lower bound for the problem follows from the fact that boolean matrix multiplication reduces [14] to context-free parsing, which is trivially a special case of CFL-reachability.

6 All-pairs reachability in bounded-stack PDAs

In this section, we give a faster ($O(n^3/\log^2 n)$-time) algorithm for all-pairs reachability in bounded-stack PDAs. Consider a state $call_{q,\gamma}$ in the summary graph H, constructed as in Section 4, of a bounded-stack PDA $\mathcal{A}$ in the normal form. The corresponding node in H is unreachable from the node q, because otherwise the stack of $\mathcal{A}$ grows unboundedly along some execution. Hence, $call_{q,\gamma}$ and q are not in the same strongly connected component (SCC) in H, and a call edge is always between two SCCs.

Now suppose there is no path in H from q to a node with an outgoing call-edge. Given the structure of H, we can compute the set of all nodes reachable from q via exploring H depth-first. However, this lets us add all summary edges of the form $call_{q,\gamma} \dashrightarrow ret_{q',\gamma}$ that can ever possibly be added to H. We can now argue inductively to handle the case when q does reach a node with an outgoing call-edge. What we obtain is that for every call edge $call_{q,\gamma} \dashrightarrow q$ in H, we can: (1) compute the descendants of q in H, depth-first, (2) using the pop-transition relation in $\mathcal{A}$, compute all summary edges from $call_{q,\gamma}$, (3) add these edges to the graph and find all descendants of $call_{q,\gamma}$ reachable through them. In other words, summary edges in H may be added in a "depth-first" order.

Consequently, an algorithm for reachability in bounded-stack PDAs is obtained by modifying a transitive closure algorithm based on Tarjan's algorithm to compute the SCCs of a graph (many such algorithms are known in the literature [13, 9, 22]). These algorithms use the facts that every two nodes in an SCC reach the same set of nodes and that Tarjan's algorithm outputs the SCCs in a bottom-up topological order, so that we can compute the set of nodes in the graph reachable from an SCC while continuing with Tarjan's depth-first algorithm (more on this shortly). We modify this algorithm in the following way: once we have computed the set of nodes reachable from the target of a call-edge (or, more generally, an edge that goes from one SCC to another), we compute the summary edges they induce (this can be done in quadratic time altogether), and treat these new edges as if they were present all along, but are being explored after the call-edge simply because of the DFS order. This algorithm for reachability in a bounded-stack PDA has the same cost as a transitive closure algorithm of this form.

Next we give such a transitive closure algorithm of cost $T(m,n) = O(\min\{mn/\log n, n^3/\log^2 n\})$, where m is the number of edges in the graph. Modifying this algorithm, we obtain a $T(m,n)$-time algorithm for reachability in bounded-stack PDAs, where m and n are the counts of edges and nodes in the summary graph of the PDA. In general, m is $O(n^2)$, so that:

Theorem 3 *All-pairs reachability in a bounded-stack PDA of size n can be computed in time $O(n^3/\log^2 n)$.*

6.1 A DFS-based, $O(\min\{mn/\log n, n^3/\log^2 n\})$-time transitive closure algorithm

```
VISIT(u)
 1  add u to Visited
 2  push(u, L)
 3  u.low ← u.dfsnum ← L.height
 4  T[u] = ∅;   u.rep = ⊥
 5  Out[u] ← ∅
 6  for v ∈ Next(u)
 7  do if v ∉ Visited  then VISIT (v)
 8       if v ∈ Done
 9         then  add v to Out[u]
10         else  u.low ← min(u.low, v.low)
11  if u.low = u.dfsnum
12    then repeat
13                    v ← pop(L)
14                     add v to Done
15                     add v to T[u]
16                    Out[u] ← Out[u] ∪ Out[v]
17                    v.rep ← u
18           until v = u
19         T[u] ← T[u] ∪ ⋃_{v∈Out[u]} T[v.rep]

CLOSURE()
1  Visited ← ∅;   Done ← ∅
2  for  each node u
3  do  if u ∉ Visited  then VISIT (u)
```

Figure 3: Transitive closure of a directed graph

Now we present our algorithm to compute the transitive closure of a directed graph G with n nodes and m edges. We speed up the cubic-time algorithm in Fig. 3. The core lemma used in the base algorithm is that in any DFS tree of G, the nodes in any SCC form a subtree. The node u_0 in an SCC S that is discovered first in a run of the algorithm is marked as the *representative* of S; for each node v in S, the field $v.rep$ stores the representative of S (in this case u_0). A global stack L supporting the usual push and pop operations is maintained; $L.height$ gives the height of the stack at any given time. For every node u, there is a field $u.dfsnum$ containing the height of the stack when it was discovered—and a field $u.low$, which is the minimum *dfsnum*-value of a vertex that a node in the subtree rooted at u points to. The SCCs are discovered using the property that if $u.low = u.dfsnum$, then u is the representative of some SCC.

Now consider the condensation graph $\widehat{G}$ of G: here, the node set is the set of SCCs of G, and an edge (S_1, S_2) exists iff there is an edge (u, v) in G for some $u \in S_1, v \in S_2$. To compute the set $T[u]$ of nodes reachable from any element u of an SCC S, we first compute the set of nodes $S_1, S_2, \ldots, S_k$ of $\widehat{G}$ reachable from S. Then $T[u] = \bigcup_i S_i$. Once we compute this set, we store it in a table indexed by the representatives of the SCCs of G.

Of course, we compute this set as well as generate the SCCs in one depth-first pass of G. To do this, we observe that the SCCs of G are generated in a bottom-up topological order. This is done by lines 12–19 of VISIT. By the time an SCC S is generated, the SCCs reachable from it in $\widehat{G}$ have all been generated (nodes in SCCs already generated are stored in a set *Done*), and the entries of T corresponding to the latters' representatives have been precisely computed.

Then all we need to fill out $T[u_0]$, where u_0 is the representative of S, is to track the edges out of S and take the union of the entries of T corresponding to the children of S in $\widehat{G}$. To do this, we use a table *Out* indexed by nodes of G—for any u in S, $Out[u]$ contains the nodes outside of S to which an edge from u may lead. At the end of the repeat-loop from line 13–18, $Out[u_0]$ contains all nodes outside S with an edge from inside S. Now line 19 takes the union of the entries of T for the SCCs to which these "neighbor nodes" belong.

As for the complexity of this algorithm, note that for each u, VISIT(u) is called at most once. Every line other than 16 and 19 costs time $O(m+n)$ during a run of CLOSURE, and since line 16 tries to add a node to $Out[u]$ once for every edge out of the SCC of u in $\widehat{G}$, its total cost is $O(m)$. Line 19 does a union of two sets of vertices for each edge in $\widehat{G}$, so that its total cost is $O(mn)$. Then:

Lemma 5 CLOSURE *terminates on any graph G with n nodes and m edges in time $O(mn)$. On termination, for every node u of G, $T[u.rep]$ is the set of nodes reachable from u.*

Let us now see how to speed up CLOSURE. Let $p = \lfloor \log n/2 \rfloor$ and $r = \lceil n/p \rceil$. We define a linear order on the set U of nodes of G and partition U into sets $U_1, U_2, \ldots, U_r$, each of size p, such that U_1 contains the first p nodes in the order, U_2 the next p, and so on. As in Sec. 3, we allocate a record R_S for each set $S \subseteq U_i$ and set up a map $\Pi : U \to \{1, \ldots, r\}$ that returns the partition to which a node belongs; also, for any $X \subseteq U$, we define $X_i = X \cap U_i$. Then, given a set $X \subseteq U$ represented as a list, we can use a sort to compute its fast set representation in time $\langle R_{X_1}, \ldots, R_{X_r} \rangle$. Let us now implement CLOSURE such that entries of T are stored as fast sets.

```
MERGE(Tab, u, X)
1  for 1 ≤ i ≤ r
2      do compute X_i
3         if X_i = ∅ continue
4         if Cache[i, R_{X_i}] = ⊥
5            then Cache[i, R_{X_i}] ← ∪_{z∈X_i} Tab[z]
6         Tab[u] ← Tab[u] ∪ Cache[i, R_{X_i}]
```

Figure 4: The speedup routine

Now we show how to speed up line 19 of the procedure VISIT. Consider the procedure MERGE in Fig. 4, which is a way to speed up computation of the recurrence $Tab[u] \leftarrow \bigcup_{v \in X} Tab[v]$, where $X \subseteq V$ (X is represented as a list) and Tab is a table mapping elements of U to fast sets. The routine uses another table $Cache$ (of global scope) such that for each $1 \leq i \leq r$ and for each set $S \subseteq U_i$, we have a table entry $Cache[i, R_S]$ containing either a subset of V, represented as a fast set, or a special value $\perp$. Initially, every entry of $Cache$ equals $\perp$. Note that $Cache$ has at most $r.2^p = O(n^{3/2}/\log n)$ entries.

Let us now use the Assign-Union operation defined in Sec. 3 to implement line 6 of MERGE, and replace line 19 of VISIT by a call to MERGE($T, u, Out[u]$). To see that this leads to a speedup, note that line 5 in MERGE gets executed once for each cell in $Cache$ during a complete run of CLOSURE—i.e., $O(n^{3/2}/\log n)$ times. Each time it is executed, it costs $O(n \log n)$ time (as X_i is of size $O(\log n)$), so that its *total* cost is $O(n^{5/2})$. Thus, the bottleneck is line 6.

Let us now look at the total number of times this line is executed during a run of closure. Since the total size of all the X's during a run of CLOSURE is bounded by m, the emptiness test in line 3 ensures that line 6 is executed $O(m)$ times in total (this is the tighter bound when the graph is sparse). The other obvious bound on this number is $O(r.n)$ (this captures the dense case). Each time it is executed, it costs time $O(n/\log n + \rho)$, where $\rho = |Cache[i, R_{X_i}] \setminus Tab[u]|$ is the number of new elements it inserts into $Tab[u]$. Since the size of $Tab[u]$ is $O(n)$ for all u, it follows that the ρ's add up only to $O(n^2)$ during a complete run of the algorithm. Thus, the total complexity of the modified CLOSURE (let us call this algorithm FAST-CLOSURE) is $O(\min\{mn/\log n, n^3/\log^2 n\})$. Then we have:

Theorem 4 FAST-CLOSURE *computes the transitive closure of a directed graph with n nodes and m edges in time $O(\min\{mn/\log n, n^3/\log^2 n\})$.*

7 All-pairs reachability in hierarchical PDAs

Suppose we are given a hierarchical PDA $\mathcal{A}$, in normal form, with procedures $\langle Q_i, \Gamma_i \rangle$, for $1 \leq i \leq k$, such that pushes (similarly pops) from the i-th procedure can only lead to procedures $j > i$ (similarly $j < i$). Consider the summary graph H, defined as in Section 4, of $\mathcal{A}$. This graph may be partitioned into k subgraphs $H_1, \ldots, H_k$ such that call-edges only run from partitions H_i to partitions H_j, where $j > i$. Also, since there are no push-transitions out of procedure k, there are no summary edges in H_k (note that this property does not hold in general bounded-stack PDAs).

To compute reachability in $\mathcal{A}$, first compute the transitive closure of H_k. Next, for all pairs of states (q, q') in H_k and all appropriate γ, add summary edges $call_{q,\gamma} \dashrightarrow ret_{q',\gamma}$. Now remove the call edges from H_{k-1} and compute its transitive closure and, once this is done, use the newly discovered reachability relations to create new summary edges in subgraphs H_j, where $j < k-1$. Note that we do

not need to touch the graph H_k again. We proceed in this way inductively, processing every H_i only once. Once we have computed the transitive closure of H_1, we add all the call edges from the different H_1's and compute the transitive closure of the entire graph. By Lemma 2, there is an edge from q to q' in the final closure iff $q \rightsquigarrow q'$.

As for complexity, let n be the total number of states in $\mathcal{A}$, and let n_i be the number of states in the subgraph H_i. Let $BM(n) = O(n^{2.376})$ be the time taken to multiply two $n \times n$ boolean matrices. Since transitive closure of a finite relation may be reduced to boolean matrix multiplication, the total cost due to transitive closure computation in the successive phases, as well as the final transitive closure, is $\Sigma_i BM(n_i) + BM(n) = O(BM(n))$. The total cost involved in identifying and inserting the summary and call edges is $O(n^2)$. Assuming $BM(n) = \omega(n^2)$, we have:

Theorem 5 *All-pairs reachability in hierarchical PDAs can be solved in time $O(BM(n))$, where $BM(n) = O(n^{2.376})$ is the time taken to multiply two $n \times n$ boolean matrices.*

8 Conclusion

In this paper, we give the first subcubic algorithms for CFL-reachability, reachability in PDAs, and emptiness of PDAs. We also identify a gradation in the complexity of reachability as recursion in a PDA is restricted. In general PDAs, "summary edges" can arise in arbitrary orders, and all-pairs reachability can be determined in time $O(n^3/\log n)$. For bounded-stack PDAs, summary edges have a "depth-first" structure, and the problem can be solved in $O(n^3/\log^2 n)$ time using a modification of a DFS-based transitive closure algorithm. For hierarchical PDAs, where there is no recursion, the states can be partitioned such that we need to compute the closure of each partition only once.

The natural question is whether this is the best we can do. Is there, for example, an $O(n^3/(\log n)^2)$-time algorithm for CFL-reachability? A harder question is whether CFL-reachability can be reduced to boolean matrix multiplication—this would be very satisfactory as the former is as hard as the latter. Yannakakis has noted [25] that Valiant's lemma [23], used to obtain the fastest known context-free parsing algorithm, can be applied directly to reduce CFL-reachability in *acyclic* graphs to boolean matrix multiplication. However, there seem to be basic difficulties in extending this method to general graphs. In both cases, it may be useful to start with the simpler problem of 2-PDA recognition.

A second set of questions involves our transitive closure algorithm. While our algorithm does not break existing bounds for this problem, it does not use the traditional reduction to matrix multiplication. This raises the question: can it be modified into an $o(mn/\log n)$ algorithm for transitive closure, which would be the best such algorithm for sparse graphs?

Finally, it will be interesting to consider practical appplications inspired by the core ideas behind these (and similar) speedups: optimizing an expensive main loop using preprocessing, and caching common subexpressions in iterated set computations. A heuristic similar to the latter surfaces in symbolic search algorithms that use set data structures such as binary decision diagrams (BDDs). Usually, BDDs are stored in a global pool, and a BDD for an expression that is a subexpression of multiple functions has references from each of them, and is not constructed multiple times. The motivation for this in the real world is to save space, but the parallels with our time-saving technique are intriguing.

Acknowledgements: The author thanks Rajeev Alur, Stephen Fink, Sudipto Guha, and Mihalis Yannakakis for valuable comments.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Series in Computer Science and Information Processing. Addison-Wesley, 1974.

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

[3] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. In *Proceedings of the Sixth ACM Symposium on Foundations of Software Engineering*, pages 175–188. 1998.

[4] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradžev. On economical construction of the transitive closure of an oriented graph. *Soviet Mathematics Doklady*, 11:1209–1210, 1970.

[5] J. Basch, S. Khanna, and R. Motwani. On diameter verification and boolean matrix multiplication, 1995.

[6] A. Boujjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Applications to model checking. In *CONCUR'97: Concurrency Theory, Eighth International Conference*, LNCS 1243, pages 135–150. Springer, 1997.

[7] T. M. Chan. All-pairs shortest paths with real weights in $o(n^3/\log n))$ time. In *WADS*, pages 318–324, 2005.

[8] T. M. Chan. All-pairs shortest paths for unweighted undirected graphs in $o(mn)$ time. In *SODA*, pages 514–523, 2006.

[9] J. Eve and R. Kurki-Suonio. On computing the transitive closure of a relation. *Acta Inf.*, 8:303–314, 1977.

[10] T. Feder and R. Motwani. Clique partitions, graph compression, and speeding-up algorithms. In *STOC*, pages 123–133, 1991.

[11] S. L. Graham, M. A. Harrison, and W. L. Ruzzo. An improved context-free recognizer. *ACM Trans. Program. Lang. Syst.*, 2(3):415–462, 1980.

[12] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[13] P. W. Purdom Jr. A transitive closure algorithm. *BIT*, 10:76–94, 1970.

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

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

[16] J. Rehof and M. Fähndrich. Type-base flow analysis: from polymorphic subtyping to cfl-reachability. In *POPL*, pages 54–66, 2001.

[17] T. Reps. Shape analysis as a generalized path problem. In *PEPM '95: Proceedings of the 1995 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 1–11, New York, NY, USA, 1995. ACM Press.

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

[19] T. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[20] W. Rytter. Time complexity of loop-free two-way pushdown automata. *Inf. Process. Lett.*, 16(3):127–129, 1983.

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

[22] L. Schmitz. An improved transitive closure algorithm. *Computing*, 30:359–371, 1983.

[23] L. G. Valiant. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.*, 10(2):308–315, 1975.

[24] R. Williams. Matrix-vector multiplication in sub-quadratic time (some preprocessing required). In *SODA*, 2007.

[25] M. Yannakakis. Graph-theoretic methods in database theory. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 230–242, 1990.