Subcubic Algorithms for Recursive State Machines

Swarat Chaudhuri
Pennsylvania State University
University Park, PA 16802, USA
swarat@cse.psu.edu

Abstract

We show that the reachability problem for recursive state machines (or equivalently, pushdown systems), believed for long to have cubic worst-case complexity, can be solved in slightly subcubic time. All that is necessary for the new bound is a simple adaptation of a known technique. We also show that a better algorithm exists if the input machine does not have infinite recursive loops.

Categories and Subject Descriptors F.1.1 [*Computation by abstract devices*]: Models of computation—Automata; F.2.2 [*Analysis of algorithms and problem complexity*]: Nonnumerical algorithms and problems—Computations on discrete structures; F.3.2 [*Theory of Computation*]: Semantics of programming languages—Program analysis.

General Terms Algorithms, Theory, Verification

Keywords Recursive state machines, pushdown systems, CFL-reachability, context-free languages, interprocedural analysis, transitive closure, cubic bottleneck.

1. Introduction

Pushdown models of programs have numerous uses in program analysis (Horwitz et al. 1988; Reps et al. 1995, 2003; Alur et al. 2005). *Recursive state machines* (Alur et al. 2005), or finite-state machines that can call other finite-state machines recursively, form a popular class of such models. These machines (called *RSMs* from now on) are equivalent to pushdown systems, or finite-state machines equipped with stacks. They are also natural abstractions of recursive programs: each component finite-state machine models control flow within a procedure, and procedure calls and returns are modeled by calls and returns to/from other machines. Sound analysis of a program then involves algorithmic analysis of an RSM abstracting it.

In this paper, we study the most basic and widely applicable form that such analysis takes: determination of reachability between states. Can an RSM, in some execution, start at a state v and reach the state v'? Because RSMs are pushdown models, any path that the RSM can take respects the nested structure of calls and returns, and reachability analysis of an RSM abstraction of a program gives a *context-sensitive program analysis*. A classic application is interprocedural data-flow analysis— "can a data-flow fact reach a certain program point along a path respecting the nesting of procedure calls?" The problem also shows up in many other program analysis contexts—for example field-sensitive alias analysis (Reps 1998), type-based flow analysis (Rehof and Fähndrich 2001), and shape analysis (Reps 1998).

Reachability for RSMs is equivalent to a well-known graph problem called *context-free language (CFL) reachability*. The question here is: given an edge-labeled directed graph and a context-free grammar over the edge labels, is there a path from node s to node t in the graph that is labeled by a word generated by the grammar? This problem, which may be viewed as a generalization of context-free recognition, was originally phrased in the context of database theory (Yannakakis 1990), where it was shown that Datalog chain query evaluation on the graph representation of a database is equivalent to single-source, single-sink CFL-reachability. It has since been identified as a central problem in program analysis (Reps 1998; Melski and Reps 2000).

All known algorithms for RSM and CFL-reachability follow a dynamic-programming scheme known in the literature as *summarization* (Sharir and Pnueli 1981; Alur et al. 2005; Bouajjani et al. 1997). The idea here is to derive reachability facts of the form (v, v'), which says that the RSM can start at state v with an empty stack and end at state v' with an empty stack. The most well-known algorithms following this scheme (Horwitz et al. 1995; Reps et al. 1995) discover such pairs enumeratively via graph traversal. Unlike context-free recognition, which has a well-known subcubic solution (Valiant 1975), RSM and CFL-reachability have not been known to have subcubic algorithms even in the single-sink, single-source case (for RSM-reachability, the size of an instance is the number of states in it; for CFL-reachability, it is the number of nodes in the input graph). This raises the question: are these problems intrinsically cubic? The question is especially interesting in program analysis as problems like interprocedural data-flow analysis and slicing are not only solvable using RSM-reachability, but also provably as hard. Believing that the answer is "yes", researchers have sometimes attributed the "cubic bottleneck" of these problems to the hardness of RSM or CFL-reachability (Reps 1998; Melski and Reps 2000).

In this paper, we observe that summarization can benefit from a known technique (Rytter 1983, 1985) for speeding up certain kinds of dynamic programming. The idea, developed in the context of language recognition for two-way pushdown automata, is to represent a computation accessing a table as a computation on row and column sets, which are stored using a "fast" set data structure. The latter, a standard data structure in the algorithms literature (Arlazarov et al. 1970; Chan 2007), splits each operation involving a pair of sets into a series of operations on pairs of sets drawn from a small sub-universe. If the sub-universes are sufficiently small, all queries on them may be looked up from a table precomputed exhaustively, allowing us to save work during an expensive main loop. When transferred to the RSM-reachability prob-

POPL'08, January 7–12, 2008, San Francisco, California, USA.

lem with slight modifications, Rytter's method leads to an algorithm that phrases the computation of reachability as a sequence of operations on sets of RSM states, and has an $O(n^3/\log n)$ time complexity. The technique may also be applied to the standard algorithm for CFL-reachability, referenced for example by Melski and Reps (2000), leading to a similar speedup. This implies subcubic solutions for Datalog chain query evaluation as well as the many program analysis applications of RSM-reachability.

Our other contribution is an observation that the reachability problem for RSMs gets easier, so far as worst-case complexity is concerned, as recursion is restricted. We study the reachability problem for *bounded-stack recursive state machines*, which are RSMs where the stack never grows unboundedly in any execution. Machines of this sort have a clear interpretation in program analysis: they capture the flow of control in procedural programs without infinite recursive loops. In spite of this extra structure, they have not been known to have faster reachability algorithms than general RSMs (note that a bounded-stack RSM is in fact a finite-state machine—however, the latter can be exponentially larger than the RSM, so that it is not an option to analyze it instead of applying an RSM-reachability algorithm). We show that it is possible to exploit this structure during reachability analysis. The key observation is that empty-stack-to-empty-stack reachability facts in bounded-stack RSMs can be derived in a *depth-first order*—i.e., if state u has an edge to state v, it is possible to first infer all the states empty-stack-to-empty-stack reachable from v and then use this information to infer the states reachable this way from v (this is not possible for general RSMs). It turns out that, as a result, we can solve the reachability problem using a transitive closure algorithm for directed graphs that allows the following kind of modifications to the instance: "for an edge (u, v) that goes from one strongly connected component to another, 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 can handle such modifications. Consequently, we derive a new transitive closure algorithm for directed graphs that can.

Our transitive closure algorithm speeds up a procedure based on Tarjan's algorithm to determine the strongly connected components of a graph. Such algorithms have a sizable literature (Purdom 1970; Eve and Kurki-Suonio 1977; Schmitz 1983). Their attraction in our setting is that they perform one depth-first traversal of the input graph, computing closure using set operations along the way, so that it is possible to weave the treatment of added edges into the discovery of edges in the original graph. The idea behind the speedup is, once again, to reuse computations on small patterns common to set computations, except this time, it can be taken further and yields a complexity of $O(\min\{mn/\log n, n^3/\log^2 n\})$, where n is the number of nodes in the graph and m the number of edges. This directly leads to an $O(n^3/\log^2 n)$ solution for all-pairs reachability in bounded-stack RSMs.

We finish our study of the interplay of recursion and reachability in RSMs with a note on the reachability problem for *hierarchical state machines* (Alur and Yannakakis 1998). These machines can model control flow in structured programs without recursive calls and form a proper subclass of bounded-stack RSMs. The one published reachability algorithm for such models is cubic (Alur and Yannakakis 1998); here, we give a simple alternative that has the same complexity as boolean matrix multiplication. While this algorithm is almost trivial, taken together with our other results, it indicates a gradation in the complexity of RSM-reachability as recursion is constrained.

The paper is organized as follows. Section 2 defines the three classes of RSMs that interest us, CFL-reachability, and the fast set data structure. Section 3 discusses reachability in general RSMs and CFL-reachability. In Section 4, we study reachability for bounded-stack RSMs, and Section 5 briefly examines reachability in hierarchical state machines. We conclude with some discussion in Section 6.

2. Basics

Recursive state machines (RSMs), introduced by Alur et al. (2005), are finite-state-machines that can call other finite-state-machines recursively. RSMs are equivalent to pushdown systems, and any solution for RSM-reachability can be translated to a solution the same complexity for pushdown systems. In this section, we define three variants of recursive state machines. We also review their connection with the context-free language reachability problem.

Recursive state machines

A *recursive state machine* (RSM) M is a tuple $\langle M_1, M_2, \ldots, M_k \rangle$, where each $M_i = \langle L_i, B_i, Y_i, En_i, Ex_i, \rightarrow_i \rangle$ is a *component* comprising:

- a finite set L_i of *internal states*;
- a finite set B_i of *boxes*;
- a map $Y_i : B_i \rightarrow \{1, 2, \ldots, k\}$ that assigns a component to every box;
- a set $En_i \subseteq L_i$ of *entry states* and a set $Ex_i \subseteq L_i$ of *exit states*;
- an edge relation $\rightarrow_i \subseteq (L_i \cup Retns_i \setminus Ex_i) \times (L_i \cup Calls_i \setminus En_i)$, where $Calls_i = \{(b, en) : b \in B_i, en \in En_{Y_i(b)}\}$ is the set of *calls* and $Retns_i = \{(b, ex) : b \in B_i, ex \in Ex_{Y_i(b)}\}$ the set of *returns* in M_i.

Note that an edge cannot start from a call or an exit state, and cannot end at a return or an entry state. We assume that for every distinct i and j, L_i, B_i, $Calls_i$, $Retns_i$, L_j, B_j, $Calls_j$, and $Retns_j$ are pairwise disjoint. Arbitrary calls, returns and internal states in M are referred to as *states*. The set of all states is given by $V = \bigcup_i (L_i \cup Calls_i \cup Retns_i)$, and the set of states in M_j is denoted by V_j. We also write $B = \bigcup_i B_i$ to denote the collection of all boxes in M. Finally, the extensions of the relations $\rightarrow_i$ and functions Y_i are denoted respectively by $\rightarrow \subseteq V \times V$ and $Y : B \rightarrow \{1, 2, \ldots, k\}$.

For an example of an RSM, see Figure 1-(b). This machine has two components: M_1 and M_2. The component M_1 has an entry state s and an exit state t, boxes b_1 and b_2 satisfying $Y(b_1) = Y(b_2) = 2$, and edges $(s, (b_1, u))$ and $((b_2, v), t)$. The component M_2 has an entry u and an exit v, and an edge (u, v).

The semantics of M is given by an infinite *configuration graph* $\mathcal{C}_M$. Let a *configuration* of M be a pair $c = (v, w) \in V \times B^*$ satisfying the following condition: if $w = b_1 \ldots b_n$ for some $n \geq 1$ (i.e., if w is non-empty), then:

1. $v \in V_{Y(b_n)}$, and
2. for all $i \in \{1, \ldots, n-1\}$, $b_{i+1} \in B_{Y(b_i)}$.

The nodes of $\mathcal{C}_M$ are configurations of M. The graph has an edge from $c = (v, w)$ to $c' = (v', w')$ if and only if one of the following holds:

1. *Local move:* $v \in (L_i \cup Retns_i) \setminus Ex_i$, $(v, v') \in \rightarrow_i$, and $w' = w$;
2. *Call move:* $v = (b, en) \in Calls_i$, $v' = en$, and $w' = w.b$;
3. *Return move:* $v \in Ex_i$, $w = w'.b$, and $v' = (b, v)$.

Intuitively, the string w in a configuration (v, w) is a *stack*, and paths in $\mathcal{C}_M$ define the operational semantics of M. If v is a call (b, en) in the above, then the RSM pushes b on the stack and moves to the entry state en of the component $Y(b)$. Likewise, on reaching

(a)

```
int g;                 void bar()
                       {
void main ()            int y = 0;
{                      }
   int x = 1;
   bar();
   g = 1;
   bar();
L: x = g;
}
```

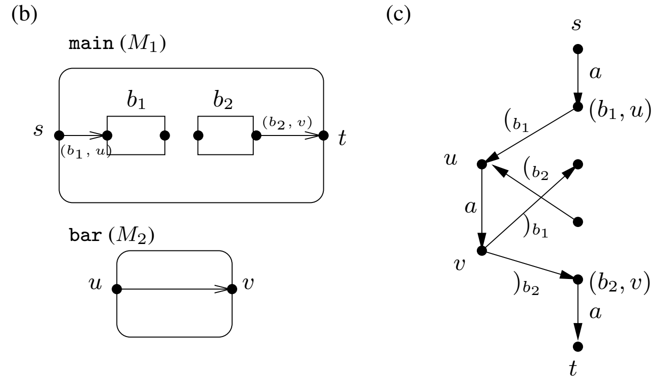

Figure 1. (a) A C example (b) RSM for the uninitialized variable problem (c) CFL-reachability formulation

an exit ex, it pops a frame b off the stack and moves to the return (b, ex). Unsurprisingly, RSMs have linear translations to and from pushdown systems (Alur et al. 2005).

Size The *size* of an RSM is the total number of states in it.

Reachability Reachability in the configuration graph is defined as usual. We call the state v' *reachable* from the state v if a configuration (v', w), for some stack w, is reachable from (v, ϵ) in the configuration graph. Intuitively, the RSM, in this case, has an execution that starts at v with an empty stack and ends at v' with some stack. The state v' is *same-context reachable* from v if (v', ϵ) is reachable from (v, ϵ). In this case the RSM can start at v with an empty stack and reach v' with an empty stack—note that this can happen only if v and v' are in the same component.

The *all-pairs reachability problem* for an RSM is to determine, for each pair of states v, v', whether v' is reachable from v'. The single-source and single-sink variants of the problem are defined in the natural way. We also define the *same-context reachability problem*, where we ask if v' is same-context reachable from v.

All known algorithms for RSM-reachability and pushdown systems, whether all-pairs or single-source/single-sink, same-context or not, rely on a dynamic programming scheme called *summarization* (Sharir and Pnueli 1981; Alur et al. 2005; Bouajjani et al. 1997; Reps et al. 1995), which we will examine in Section 3. The worst-case complexity of all these algorithms is cubic. Tighter bounds are possible if we constrain the number of entry and exit states and/or edges in the input. For example, if each component of the input RSM has one entry and one exit state, then single-source, single-sink reachability can be determined in $O(m + n)$ time, where m is the number of edges in the RSM and n the number of states (the all-pairs problem has the same complexity as graph transitive closure) (Alur et al. 2005). In this paper, in addition to general RSM-reachability, we study reachability algorithms for RSMs constrained in a different way: *by restricting or disallowing recursion*.

To see the use of RSM-reachability in solving a program analysis problem, consider the program in Figure 1-(a). Suppose we want to determine if the variable `g` is uninitialized at the line labeled `L`. This may be done by constructing the RSM in Figure 1-(b). The two components correspond to the procedures `main` and `bar`; states in these components correspond to the control points of the program—e.g., the state s models the entry point of `main`, and (b_2, v) models the point immediately before line `L`. Procedure calls to `bar` are modeled by the boxes b_1 and b_2. For every statement that does not assign to `g`, an edge is added between the states modeling the control points immediately before and after this statement. Then `g` is uninitialized at `L` iff (b_2, v) is reachable from s. More generally, RSM-reachability algorithms can be used to check if a context-sensitive program abstraction satisfies a safety property (Alur et al. 2005). For example, the successful software model checker SLAM (Ball and Rajamani 2001) uses an algorithm for RSM-reachability as a core module.

Bounded-stack RSMs and hierarchical state machines

Now we define two special kinds of RSMs with restricted recursion: *bounded-stack RSMs* and *hierarchical state machines*. We will see later that they have better reachability algorithms than general RSMs.

The class of *bounded-stack RSMs* consists of RSMs M where every call (b, en) is unreachable from the state en. By the semantics of an RSM, the stack of an RSM grows along an edge from a call to the corresponding entry state. Thus, intuitively, a bounded-stack RSM forbids infinite recursive loops, ensuring that in any path in the configuration graph starting with a configuration (v, ϵ), 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. While this program does employ recursion, it never runs into an infinite recursive loop. As a result, it can be modeled by a bounded-stack RSM.

A *hierarchical state machine* (Alur and Yannakakis 1998), on the other hand, forbids recursion altogether. Formally, such a machine is an RSM M where there is a total order $\prec$ on the components $M_1, \dots, M_k$ such that if M_i contains a box b, then $M_{Y(b)} \prec M_i$. Thus, calls from a component may only lead to a component lower down in this order. For example, the RSM in Figure 1-(b) is a hierarchical state machine.

Note that every bounded-stack or hierarchical machine can be translated to an equivalent finite-state machine. However, this causes an exponential increase in size in the worst case, and it is unreasonable to analyze a hierarchical/bounded-stack machine by "flattening" it into a finite-state machine. The question that interests us is: can we determine reachability in a bounded-stack or hierarchical machine in time *polynomial in the input*? The only known way to do this is to use the summarization technique that also works for general RSMs, leading to an algorithm of cubic worst-case complexity.

Context-free language reachability

RSM-reachability is equivalent to a graph problem called *context-free language (CFL) reachability* (Yannakakis 1990; Reps 1998) that has numerous applications in program analysis. Let S be a directed graph whose edges are labeled by an alphabet Σ, and let L be a context-free language over Σ. We say a node t is L-reachable from a node s if there is a path from s to t in S that is labeled by a word in L. The all-pairs CFL-reachability problem for S and L is to determine, for all pairs of nodes s and t, if t is L-reachable from s. The single-source or single-sink variants of the problem are defined in the obvious way. Customarily, the size of the instance is given by the number n of nodes in S, while L is assumed to be given by a fixed-sized grammar G.

Let us now see how, given an instance of RSM-reachability, we can obtain an equivalent CFL-reachability instance. We build a graph whose nodes are states of the input RSM M; for every edge (u, v) in M, S has an edge from u to v labeled by a symbol a. For every call (b, en) in the RSM, S has an edge labeled $(_b$ from (b, en) to en; for every exit ex and return (b, ex) in M, we add a $)_b$-labeled edge in S from ex to (b, ex) (for example, the graph S constructed from the RSM in Figure 1-(b) is shown in Figure 1-(c)). Now, the state v is reachable from the state u in M if and only if v is L-reachable from u in S, where L is given by the grammar $S \to SS \mid (_bS)_b \mid (_bS \mid a$. The translation in the other direction is also easy—we refer the reader to the original paper on RSMs (Alur et al. 2005).

Note that context-free recognition is the special case of CFL-reachability where S is a chain. A cubic algorithm for all-pairs CFL-reachability can be obtained by generalizing the Cocke-Younger-Kasami algorithm (Hopcroft and Ullman 1979) for CFL-recognition—this algorithm again relies on summarization. The problem is known to be equivalent to the problem of evaluating Datalog *chain queries* on a graph representation of a database (Yannakakis 1990). Such queries have the form $p(X, Y) \leftarrow q_0(X, Z_1) \wedge q_1(Z_1, Z_2) \wedge \ldots \wedge q_k(Z_k, Y)$, where the q_i's are binary predicates and X, Y and the Z_i's are distinct variables, and have wide applications. It has also come up often in program analysis—for example, in the context of interprocedural dataflow analysis and slicing, field-sensitive alias analysis, and type-based flow analysis (Horwitz et al. 1988; Reps et al. 1995; Horwitz et al. 1995; Reps 1995, 1998; Rehof and Fähndrich 2001). The "cubic bottleneck" of these analysis problems has sometimes been attributed to the believed cubic hardness of CFL-reachability.

A special case is the problem of *Dyck-CFL-reachability*. The constraint here is that the CFL L is now a language of balanced parentheses. Many program analysis applications of CFL-reachability—e.g., field-sensitive alias analysis of Java programs (Sridharan et al. 2005)—turn out actually to be applications of Dyck-CFL-reachability, though so far as asymptotic bounds go, it is no simpler than the general problem. This problem is equivalent to the problem of same-context reachability in RSMs.

Fast sets

Our algorithms for RSMs use a set data structure that exploits sharing between sets to offer certain set operations at low amortized cost. This data structure—called *fast sets* from now on—is standard technology in the algorithms literature (Chan 2007; Arlazarov et al. 1970) and was used, in particular, in the papers by Rytter (1983, 1985) on two-way pushdown recognition. Its essence is that it splits an operation on a pair of sets into a series of unit-cost operations on small sets. We will now review it.

Let U be a universe of n elements of which all our sets will be subsets. The fast set data structure supports the following operations:

- *Set difference*: Given sets X and Y, return a *list* $\mathit{Diff}(X, Y)$ consisting of the elements of the set $(X \setminus Y)$.
- *Insertion*: Insert a value into a set.
- *Assign-union*: Given sets X and Y, perform the assignment $X \leftarrow X \cup Y$.

Let us assume an architecture with word size $p = \theta(\log n)$. A fast set representation of a set is the bit vector (of length n) for the set, broken into $\lceil n/p \rceil$ words. Then:

- To compute $\mathit{Diff}(X, Y)$, where X and Y are fast sets, we compute the bit vector for $Z = X \setminus Y$ via bitwise operations on the words comprising X and Y. This takes $O(n/p)$ time assuming constant-time logical operations on words. 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. Assuming that it is possible in constant time to check if a word equals 0 and find the most significant bit in a word, this can be done in $O(|Z| + n/p)$ time. Note that the bound is given in terms of the *size of the output*. This is exploited while bounding the amortized cost of a sequence of set differences.
- Insertion of $0 \leq x \leq n-1$ involves setting a bit in the $\lfloor x/p \rfloor$-th word, which can be done in $O(1)$ time.
- The assign-union operation can be implemented by word-by-word logical operations on the components of X and Y, and takes $O(n/p)$ time.

In case the unit-cost operations we need are not available, they can be implemented using table lookup. Let a fast set now be a collection of words of length $p = \lceil \log n/2 \rceil$. In a preprocessing phase, we build tables implementing each of the binary or unary word operations we need by simply storing the result for each of the $O(2^p.2^p) = O(n)$ possible inputs. The time required to build each such table is $O(p.n)$ (assuming linear-time operations on words), and the space requirement is $O(n)$. The costs of our fast set operations are now as before.

3. All-pairs reachability in recursive state machines

Let us now study the reachability problem for recursive state machines. We remind the reader that all known algorithms for this problem are cubic and based on a high-level algorithm called *summarization*. In this section we show that a speedup technique developed by Rytter (1985, 1983) can be directly applied to this algorithm, leading to an $O(n^3/\log n)$-time solution. The modified algorithm computes reachability via a sequence of operations on *sets of states*, each represented as a fast set. In this sense it is a *symbolic* implementation of summarization, rather than an iterative one like the popular algorithm due to Reps et al. (1995). We also show that the standard cubic algorithm for CFL-reachability, referenced for example by Melski and Reps (2000), can be speeded up similarly using Rytter's technique.

3.1 Reachability in RSMs

Let us start by reviewing summarization. We have as input an RSM $M = \langle M_1, \ldots, M_k \rangle$ as in Section 2, with state set V, box set B, edge relation $\to \subseteq V \times V$, and a map $Y : B \to \{1, \ldots, k\}$ assigning components to boxes. The algorithm first determines *same-context reachability* by building a relation $H^s \subseteq V \times V$, defined as the least relation satisfying:

1. if $u = v$ or $u \to v$, then $(u, v) \in H^s$;
2. if $(u, v') \in H^s$ and $(v', v) \in H^s$, then $(u, v) \in H^s$;
3. if $(u, v) \in H^s$ and u is an entry and v is an exit in some component, then for all boxes b such that $(b, u), (b, v) \in V$, we have $((b, u), (b, v)) \in H^s$.

For example, the relation H^s for the RSM in Figure 1-(a) is drawn in Figure 2 (the transitive edges are omitted). While the definition of H^s is recursive, it may be constructed using a least-fixpoint computation. Once it is built, we construct a relation $H \subseteq V \times V$ defined as:

$$\begin{aligned} H \quad = \quad & \to \cup \{((b, en), (b, ex)) \in H^s : b \in B, \text{ and } en \text{ is an entry and } ex \text{ an exit of } Y(b)\} \\ & \cup \{((b, en), en) : en \text{ is an entry in } Y(b)\}, \end{aligned}$$

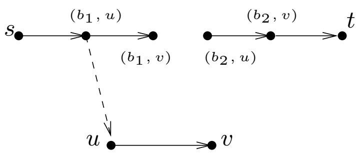

Figure 2. The relation H. H^s is the transitive closure of non-dashed edges, and H^* is the transitive closure of all edges

and compute the (reflexive) transitive closure H^* of the resultant relation (see Figure 2). It is known that:

LEMMA 1 ((Alur et al. 2005; Bouajjani et al. 1997)). *For states v and v' of M, v' is reachable from v iff $(v, v') \in H^*$. Also, v' is same-context reachable from v iff $(v, v') \in H^s$.*

Within the scheme of summarization, there are choices as to how the fixpoint computations for H^s and H^* are carried out. For example, the popular algorithm due to Reps et al. (1995) employs graph search to construct these relations enumeratively. In contrast, the algorithm we now present, obtained by a slight modification of an algorithm by Rytter (1985) for two-way pushdown recognition, phrases the computation as a sequence of operations on *sets of states*. Unlike previous implementations of summarization, our algorithm has a slightly subcubic worst-case complexity.

The algorithm is a modification of the procedure BASELINE-REACHABILITY in Figure 3, which uses a worklist W to compute H^s and H^* in a fairly straightforward way. Line 1 of the baseline routine inserts intra-component edges and trivial reachability facts into H^s and W. The rest of the pairs in H^s are derived by the while-loop from line 2–10, which removes pairs from W one by one and "processes" them. While processing a pair (u, v), we derive all the pairs that it "implies" by rules (2) and (3) in the definition of H^s and *that have not been derived already*, and insert them into H^s and W. At the end of any iteration of the loop, W contains the pairs that have been derived but not yet processed. The loop continues till W is empty. It is easy to see that on its termination, H^s is correctly computed. Lines 11-14 now compute H^*.

Note that a pair is inserted into W only when it is also inserted into H^s, so that the loop has one iteration per insertion into H^s. At the same time, a pair is never taken out of H^s once it is inserted, and no pair is inserted into it twice. Let n be the size of the RSM, and let $\alpha \leq n^2$ be an upper bound on the number of pairs (u, v) such that v is reachable from u. Then the loop has $O(\alpha)$ iterations.

Let us now determine the cost of each iteration. Assuming we can insert an element in H^s and W in constant time, lines 4–6 cost constant time per insertion of an element into H^s. Thus, the total cost for lines 4–6 during a run of BASELINE-REACHABILITY is $O(\alpha)$. The for-loops at line 7 and line 9 need to identify all states u' and v' satisfying their conditions for insertion. Done enumeratively, this costs $O(n)$ time per iteration, causing the total cost of the loop to be $O(\alpha n)$. As for the rest of the algorithm, line 14 may be viewed as computing the (reflexive) transitive closure of a graph with n states and $O(\alpha)$ edges. This may clearly be done in $O(\alpha n)$ time. Then:

LEMMA 2. BASELINE-REACHABILITY *terminates on any RSM M in time $O(\alpha.n)$, where $\alpha \leq n^2$ is the number of pairs $(u, v) \in V \times V$ such that v is reachable from u. On termination, for every pair of states u and v, v is reachable from u iff $(u, v) \in H^*$, and v is same-context reachable from u iff $(u, v) \in H^s$.*

BASELINE-REACHABILITY()
1 $W \leftarrow H^s \leftarrow \{(u, u) : u \in V\} \cup \rightarrow$
2 **while** $W \neq \emptyset$
3 **do** $(u, v) \leftarrow$ remove from W
4   **if** u is an entry state and v an exit state in a component M_i
5   **then for** b such that $Y(b) = i$
6   **do** insert $((b, u), (b, v))$ into H^s, W
7   **for** $(u', u) \in H^s$ such that $(u', v) \notin H^s$
8   **do** insert (u', v) into H^s and W
9   **for** $(v, v') \in H^s$ such that $(u, v') \notin H^s$
10   **do** insert (u, v') into H^s and W
11 $H^* \leftarrow H^s$
12 **for** calls $(b, en) \in V$
13 **do** insert $((b, en), en)$ into H^*
14 $H^* \leftarrow$ transitive closure of H^*

Figure 3. Baseline procedure for RSM-reachability

To convert the baseline procedure into a set-based algorithm, interpret the relation H^s as an $n \times n$ table, and denote the u-th row and column as sets (respectively denoted by $Row(u)$ and $Col(u)$). Then we have $Row(u) = \{v : (u, v) \in H^s\}$ and $Col(u) = \{v : (v, u) \in H^s\}$. Now observe that the for-loops at lines 7 and 9 can be captured by *set difference operations*. The for-loop in line 7–8 may be rewritten as:

for $u' \in (Col(u) \setminus Col(v))$ **do** insert (u', v) into H^s and W,

and the for-loop in line 9–10 may be rewritten as:

for $v' \in (Row(v) \setminus Row(u))$ **do** insert (u, v') into H^s and W.

Our set-based algorithm for RSM-reachability —called REACHABILITY from now on— is obtained by applying these rewrites to BASELINE-REACHABILITY. Clearly, REACHABILITY terminates after performing $O(\alpha)$ set difference and insertion operations, and when it does, the tables H^* and H^s respectively capture reachability and same-context reachability.

We may, of course, use any set data structure offering efficient difference and insertion in our algorithm. If the cost of set difference is linear, then the algorithm is cubic in the worst-case. The complexity, however, becomes $O(n\alpha/\log n) = O(n^3/\log n)$ if we use the fast set data structure of Section 2. To see why, assume that the rows and columns of H^s are represented as fast sets and that set difference and insertion are performed using the operations *Diff* and *Ins* described earlier. In each iteration of the main loop, the inner loops first compute the difference of two sets of size n, then, for every element in the answer, inserts a pair into H^s (this involves inserting an element into a row and a column) and W. If the i-th iteration of the main loop inserts σ_i pairs into H^s, the time spent on the operation *Diff* in this iteration is $O(n/\log n + \sigma_i)$. Since the result is returned as a list, the cost of iteratively inserting pairs in it into H^* and W is also $O(\sigma_i)$. The cost of these operations summed over the entire run of REACHABILITY is $O(\alpha.n/\log n + \sum_i^\alpha \sigma_i) = O(\alpha n/\log n + \alpha) = O(\alpha n/\log n)$. The only remaining bottleneck is the transitive closure in line 14 of the baseline procedure. This may be computed in $O(\alpha.n/\log n)$ time using the procedure we give in Section 4.1. The total time complexity then becomes $O(\alpha n/\log n)$— i.e., $O(n^3/\log n)$.

As for the space requirement of the algorithm, $\Theta(n^2)$ space is needed just to store the tables H^s and H^*. The space required by tables implementing word operations, if unit-cost word operations are not available, is subsumed by this factor. Thus we have:

THEOREM 1. *The algorithm* REACHABILITY *solves the all-pairs reachability and same-context-reachability problems for an RSM with n states in $O(n^3/\log n)$ time and $O(n^2)$ space.*

Readers familiar with Rytter's $O(n^3/\log n)$-time algorithm (Rytter 1985) for recognition of two-way pushdown languages will note that our subcubic algorithm is very similar to it. Recall that a two-way pushdown automaton (2-PDA) is a pushdown automaton which, on reading a symbol, can move its "reading head" one step forward and back on the input word, while changing its control state and pushing/popping a symbol on/off its stack. The language recognition problem for 2-PDAs is: "given a word w of length n and a 2-PDA $\mathcal{A}$ of constant size, is w accepted by $\mathcal{A}$?" This problem may be linearly reduced to the reachability problem for RSMs. Notably, there is also a reduction in the other direction. Given an RSM M where we are to determine reachability, write out the states and transitions of M as an input word. Now construct a 2-PDA $\mathcal{A}$ that, in every one of an arbitrary number of rounds, moves its head to an arbitrary transition of M and tries to simulate the execution. Using nondeterminism, $\mathcal{A}$ can guess any run of M, and accept the input if and only if M has an execution from a state u to a state v. This may suggest that a subcubic algorithm for RSM-reachability already exists. The catch, however, is that an RSM of size n may have $\Omega(n^2)$ transitions, so that this reduction outputs an instance of quadratic size. Clearly, it cannot be combined with Rytter's algorithm to solve reachability in RSMs in cubic (let alone subcubic) time.

On the other hand, what Rytter's algorithm actually does is to speed up a slightly restricted form of summarization. Recall the routine BASELINE-REACHABILITY, and let $u, v, \ldots$ be positions in a word rather than states of an RSM. Just like us, Rytter derives pairs (u, v) such that the automaton has an empty-stack to empty-stack execution from u to v. One of the rules he uses is:

> Suppose (u, v) is already derived. If $\mathcal{A}$ can go from u' to u by pushing γ, and from v to v' by popping γ, then derive (u', v').

This rule is analogous to Rule (3) in our definition of summarization:

> Suppose (u, v) is already derived. If u is an entry and v is an exit in some component and b is a box such that $(b, u), (b, v) \in V$, then derive $((b, u), (b, v))$.

The two rules differ in the number of new pairs they derive. Because the size of $\mathcal{A}$ is fixed, Rytter's rule can generate at most a constant number of new pairs for a fixed pair (u, v). On the contrary, our rule can derive a linear number of new pairs for given (u, v). Other than the fact that Rytter deals with pairs of positions and we deal with RSM states, this is the only point of difference between the baseline algorithms used in the two cases. At first glance, this difference may seem to make the algorithm cubic, as the above derivation happens inside a loop with a quadratic number of iterations. Our observation is that a tighter analysis is possible: our rule above only does a constant amount of work *per insertion* of a pair into H^s. Thus, over a complete run of the algorithm, its cost is quadratic and subsumed by the cost of the other lines, even after the speedup is applied. For the rest of the algorithm, Rytter's complexity arguments carry over.

3.2 CFL-reachability

As RSM-reachability and CFL-reachability are equivalent problems, the algorithm REACHABILITY can be translated into a set-based, subcubic algorithm for CFL-reachability. However, Rytter's technique can also be directly applied to the standard algorithm for CFL-reachability, described for example by Melski and Reps (2000). Now we show how. Let us have an instance (S, G) of CFL-reachability, where S is an edge-labeled graph with n nodes and G is a constant-sized context-free grammar. Without loss of generality, it is assumed that the right-hand side of each rule in G has at most two symbols. The algorithm in Melski and Reps' paper—called BASELINE-CFL-REACHABILITY and shown in Figure 4—computes tuples (u, A, v), where u, v are nodes of S and A is a terminal or non-terminal, such that there is a path from u to v labeled by a word w that G can derive from A. A worklist W is used to process the tuples one by one; derived tuples are stored in a table H^s. It is easily shown, by arguments similar to those for RSM-reachability, that the algorithm is cubic and requires quadratic space. On termination, a tuple (u, I, v), where u, v are nodes and I the initial symbol of G, is in H^s iff v is CFL-reachable from u.

BASELINE-CFL-REACHABILITY()

1 $W \leftarrow H^s \leftarrow \{(u, A, v) : u \xrightarrow{a} v$ in S, and $A \rightarrow a$ in $G\}$
2 $\quad\quad \cup\{(u, A, u) :\ A \rightarrow \epsilon$ in $G\}$
3 **while** $W \neq \emptyset$
4 **do** $(u, B, v) \leftarrow$ remove from W
5 $\quad$ **for** each production $A \rightarrow B$
6 $\quad$ **do if** $(u, A, v) \notin H^s$
7 $\quad\quad$ **then** insert (u, A, v) into H^s, W
8 $\quad$ **for** each production $A \rightarrow CB$
9 $\quad$ **do for** each edge (u', C, u) such that $(u', A, v) \notin H^s$
10 $\quad\quad$ **do** insert (u', A, v) into H^s and W
11 $\quad$ **for** each production $A \rightarrow BC$
12 $\quad$ **do for** each edge (v, C, v') such that $(v, A, v') \notin H^s$
13 $\quad\quad$ **do** insert (v, A, v') into H^s and W

Figure 4. Baseline algorithm for CFL-reachability

As in case of RSM-reachability, now we store the rows and columns of H^s as fast sets of $O(n)$ size. For a node u and a non-terminal A, the row $Row(u, A)$ (similarly the column $Col(u, A)$), stores the set of nodes u' such that (u, A, u') (similarly (u', A, u)) is in H^s. Now, the bottlenecks of the algorithm are the two nested loops (lines 8–10 and 11–13). We speed them up by implementing them using set difference operations— for example, the loop from line 8–10 is replaced by:

for each production $A \rightarrow CB$
do for $u' \in (Col(u, C) \setminus Col(v, A))$
$\quad$ **do** insert (u', A, v) into H^s and W.

Assuming a fast set implementation, the cost for this loop is in a given iteration of the main loop is $O(n/\log n + \sigma)$, where σ is the number of new tuples inserted into H^s. Since the number of insertions into H^s is $O(n^2)$, its total cost during a complete run of the algorithm is $O(n^3/\log n)$. The same argument holds for the other loop. Let us call the modified algorithm CFL-REACHABILITY. By the discussion above:

THEOREM 2. *The algorithm* CFL-REACHABILITY *solves the all-pairs CFL-reachability problem for a fixed-sized grammar and a graph with n nodes in $O(n^3/\log n)$ time and $O(n^2)$ space.*

Theorem 2 improves the previous cubic bound for all-pairs—or, for that matter, single-source, single-sink— CFL-reachability. By our discussion in Section 2, this implies subcubic, set-based algorithms for Datalog chain query evaluation as well as the many program analysis applications of CFL-reachability.

4. All-pairs reachability in bounded-stack RSMs

Is a better algorithm for RSM-reachability possible if the input RSM is bounded-stack? In this section, we show that this is indeed the case. As we mentioned earlier, the only previously known way to solve reachability in bounded-stack machines is to use summarization, which gives a cubic algorithm; speeding it up using the technique we presented earlier leads to a factor-log n speedup. Now we show that the bounded-stack property gives us a second logarithmic speedup. Our algorithm combines graph search with a speedup technique used by Rytter (1983, 1985) to recognize languages of loop-free 2-way PDAs[1]. Unlike the algorithm for general RSMs, it is not just an application of existing techniques, and we consider it the main new algorithm of this paper.

We start by reviewing search-based algorithms for reachability in (general) RSMs. Let M be an RSM as in Section 2, and recall the relation H defined in Section 3—henceforth, we view it as a graph and call it the *summary graph* of M. The edges of H are classified as follows:

- Edges $((b, en), en)$, where b is a box and en is an entry state in $Y(b)$, are known as *call edges*;
- Edges $((b, en), (b, ex))$, where b is a box, and en is an entry and ex an exit in $Y(b)$, are called *summary edges*;
- Edges that are also edges of M are called *local edges*.

Note that a state v is same-context reachable from a state u iff there is a path in H from u to v made only of local and summary edges. Let the set of states same-context reachable from u be denoted by $H^s(u)$. While the call and local edges of H are specified directly by M, we need to determine reachability between entries and exits in order to identify the summary edges. The search-based formulation of summarization (Reps et al. 1995; Horwitz et al. 1995) views reachability computation for M (or, in other words, computation of the transitive closure H^* of H) as a restricted form of *incremental* transitive closure. A search algorithm is employed to compute reachability in H; when an exit ex is found to be same-context-reachable from en, the summary edge $((b, en), (b, ex))$ is *added to the graph*. The algorithm must now explore these added edges along with the edges in the original graph.

Let us now assume that M is bounded-stack. Consider any call (b, en) in the summary graph H. Because M is bounded-stack, this state is unreachable from the state en. Hence, (b, en) and en are not in the same strongly connected component (SCC) in H, and a call edge is always between two SCCs. The situation is sketched in Figure 5. The nodes are states of M (en is an entry and ex is an exit in the same component, while b is a box), and the large circles denote SCCs. We do not draw edges within the same SCC—the dotted line from en to ex indicates that ex is same-context reachable from en.

We will argue that all summary edges in H may be discovered using a variant of depth-first graph search (DFS). To start with, let us assume that the summary graph H is acyclic, and consider a call (b, en) in it. First we handle the case when no path in H from en contains a call. As a summary-edge always starts from a call, this means that no such path contains a summary-edge either, and the part of H reachable from en is not modified due to summary edge discovery. Thus, the set $H^s(en)$ of states v same-context reachable (i.e., reachable via summary and local edges) from en can be computed by exploring H depth-first from en. Further, because the graph is acyclic, the same search can label each such v with the set $H^s(v)$. This is done as follows:

- if v has no children, then $H^s(v) = \{v\}$;
- if v has children $u_1, u_2, \ldots, u_m$, then

$$H^s(v) = \bigcup_i H^s(u_i).$$

[1] A loop-free 2-PDA is one that has no infinite execution on any word. The recognition problem for loop-free 2-PDAs reduces to reachability in *acyclic* RSMs—i.e., RSMs whose configuration graphs are cycle-free. Obviously, these are less general than bounded-stack RSMs.

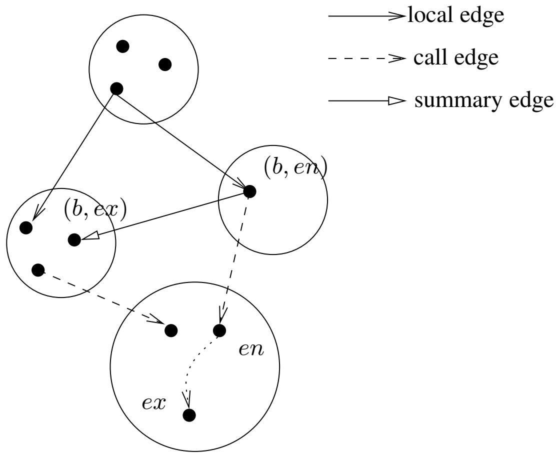

Figure 5. All-pairs reachability in bounded-stack RSMs

Once we have computed the set $H^s(en)$ of such v-s that are same-context reachable from en, we can, consulting the transition relation of M, determine all summary edges $((b, en), (b, ex))$. Note that these are the only summary edges from (b, en) that can ever be added to H. However, these summary edges may now be explored via *the same depth-first traversal*—we may view them simply as edges explored after the call-edge to en due to the DFS order. The same search can compute the set $H^s(u)$ for each new state u found to be reachable from the return (b, ex). Note that descendants of (b, ex) may also be descendants of en—for example, a descendant x of en may be reachable from a different entry point en' of $Y(b)$, which may be "called" by a call reachable from (b, ex). In other words, the search from (b, ex) may encounter some *cross-edges*, thus needing to use some of the H^s-sets computed during the search from en. Once the H^s-sets for en and all summary-children (b, ex) are computed, we can compute the set $H^s((b, en))$. Since we are only interested in reachability via summary and local edges and a call has no local out-edges, this set is the union of the H^s-sets for the summary children.

Now suppose there are at most $p \geq 1$ call states in a path in H from en. Let the state (b', en') be the first call reached from en in a depth-first exploration— because of the bounded-stack property, no descendant of en' can reach en in H. Now, there can be at most $(p - 1)$ calls in a path from en', so that can inductively determine the summary edges from (b', en'), explore these edges, and label every state v in the resultant tree by the set $H^s(v)$. It is easy to see that this DFS can be "weaved" into the DFS from en.

The above algorithm, however, will not work when H has cycles. This is because in a graph with cycles, a simple DFS cannot construct the sets $H^s(v)$ for all states v. This difficulty, however, may be resolved if we use, instead of a plain DFS, a transitive closure algorithm based on Tarjan's algorithm to compute the SCCs of a graph (Aho et al. 1974). Many such algorithms are known in the literature (Purdom 1970; Eve and Kurki-Suonio 1977; Schmitz 1983). Let $Reach(v)$ denote the set of nodes reachable from a node v in a graph. The first observation that these algorithms use is that for any two nodes v_1 and v_2 in the same SCC of a graph, we have $Reach(v_1) = Reach(v_2)$. Thus, it is sufficient to compute the set $Reach$ for a single representative node per SCC. The second main idea is based on a property of Tarjan's algorithm. To understand it, we will have to define the *condensation graph* $\widehat{G}$ of a graph G:

- the nodes of $\widehat{G}$ are the SCCs of G;

- the edge set is the least set constructed by: "if, for nodes S_1 and S_2 of $\widehat{G}$, G has nodes $u \in S_1, v \in S_2$ such that there is an edge from u to v, then $\widehat{G}$ has an edge from S_1 to S_2."

Now, Tarjan's algorithm, when running on a graph G, "piggy-backs" a depth-first search of the graph and outputs the nodes of $\widehat{G}$ in a bottom-up topological order. This is possible because the condensation graph of any graph is acyclic. For example, running on the graph in Figure 5 (let us assume that all the edges are known), the algorithm will first output the SCC containing en, then the one containing (b, ex), then the one containing (b, en), etc. We can, in fact, view the algorithm as performing a DFS on the condensation graph of G. In the same way as when our input graph was acyclic, we can now compute, for every node S in the condensation graph, the set of nodes $Reach(S)$ reachable from that SCC, defined as:

$$Reach(S) = \bigcup_{u \in S} Reach(u).$$

For each S, this set is known by the time the algorithm returns from the first node in S to have been visited in the depth-first search.

Assuming that we have a transitive closure algorithm of the above form, let us focus on bounded-stack RSMs again. Let us also suppose that we are only interested in same-context reachability. We apply the transitive closure algorithm to the graph H after modifying it in the two following ways. First, we ensure that the sets $Reach(u)$, for a state u, only contain descendants of u reachable via local and summary edges— this requires a trivial modification of the algorithm. To understand the second modification, consider once again a call (b, en) in a summary graph H; note that the call edge $((b, en), en)$ is an edge in the condensation graph $\widehat{H}$. Thus, the set $Reach(S_{en})$, where S_{en} is the SCC of en, is known by the time the transitive closure algorithm is done exploring this edge. Now we can construct all summary edges from (b, en) and *add them as outgoing edges* from (b, en), viewing them, as in the acyclic case, as normal edges appearing after the call-edge in the order of exploration. The set $Reach(S_{(b,en)})$ can now be computed.

By the time the above algorithm terminates, $Reach(S_u) = H^s(u)$ for each state u— i.e., we have determined all-pairs same-context reachability in the RSM. To determine all-pairs reachability, we simply insert the call edges into the summary graph, and compute its transitive closure. In fact, we can do better: with some extra book-keeping, it is possible to compute reachability in the same depth-first search used to compute same-context reachability (i.e., summary edges).

Next we present an algorithm for graph transitive closure that, in addition to being based on Tarjan's algorithm, also uses fast sets to achieve a subcubic complexity. Using the technique outlined above, we modify it into an algorithm for bounded-stack RSM-reachability of $O(n^3/\log^2 n)$ complexity.

4.1 Speeding up search-based transitive closure

The algorithm that we now present combines a Tarjan's-algorithm-based transitive closure algorithm (studied, for example, by Schmitz (1983) or Purdom (1970)) with a fast-set-based speedup technique used by Rytter (1983, 1985) to solve the recognition problem for a subclass of 2-PDAs. While subcubic algorithms for graph transitive closure have been known for a long time, this is, so far as we know, the first algorithm that is based on graph traversal and yet runs in $O(n^3/\log^2 n)$ time. Both these features are necessary for an $O(n^3/\log^2 n)$-time algorithm on bounded-stack RSMs.

As in our previous algorithms, we start with a baseline cubic-time algorithm and speed it up using fast sets. This algorithm, called BASELINE-CLOSURE and shown in Figure 6, is simply a DFS-based transitive closure algorithm. Let us first see how it detects strongly connected components in a graph G. The main

VISIT(u)
1 add u to $Visited$
2 $push(u, L)$
3 $low(u) \leftarrow dfsnum(u) \leftarrow height(L)$
4 $Reach(u) \leftarrow \emptyset$; $rep(u) \leftarrow \perp$
5 $Out(u) \leftarrow \emptyset$; $Next(u) = \{$ children of u $\}$
6 **for** $v \in Next(u)$
7 **do if** $v \notin Visited$ **then** VISIT (v)
8 **if** $v \in Done$
9 **then** add v to $Out(u)$
10 **else** $low(u) \leftarrow \min(low(u), low(v))$
11 **if** $low(u) = dfsnum(u)$
12 **then repeat**
13 $v \leftarrow pop(L)$
14 add v to $Done$
15 add v to $Reach(u)$
16 $Out(u) \leftarrow Out(u) \cup Out(v)$
17 $rep(v) \leftarrow u$
18 **until** $v = u$
19 $Reach(u) \leftarrow Reach(u) \cup \bigcup_{v \in Out(u)} Reach(rep(v))$

BASELINE-CLOSURE()
1 $Visited \leftarrow \emptyset$; $Done \leftarrow \emptyset$
2 **for** each node u
3 **do if** $u \notin Visited$ **then** VISIT (u)

Figure 6. Transitive closure of a directed graph

idea is that in any DFS tree of G, the nodes belonging to a particular 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, $rep(v)$ denotes the representative of S (in this case u_0). A global stack L supporting the usual push and pop operations is maintained; $height(L)$ gives the height of the stack at any given time. As soon as we discover a node, we push it on this stack—note that for any SCC, the representative is the first node to be on this stack. For every node u, $dfsnum(u)$ is the height of the stack when it was discovered, and $low(u)$ equals, once the search from u has returned, the minimum $dfsnum$-value of a node that a descendant of u in the DFS tree has an edge to. Now observe that if $low(u) = dfsnum(u)$ at the point when the search is about to return from a node u, then u is the representative of some SCC. We maintain the invariant that all the elements above and inclusive of u in the stack belong to the SCC of u. Before returning from u, we pop all these nodes and output them as an SCC. Nodes in SCCs already generated are stored in a set $Done$.

Now we shall see how to generate the set of nodes reachable from a node of G. Let S be an SCC of G; we want to compute the set $Reach(S)$ of nodes reachable from S. Consider the condensation graph $\widehat{G}$ of G, where S is a node. If S has no children in the graph, then $Reach(S) = S$; if it has children $S_1, S_2, \ldots, S_k$, then $Reach(S) = \bigcup_i Reach(S_i)$. Once this set is computed, we store it in a table $Reach$ 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. Recall that the SCCs of G are generated in a bottom-up topological order (the outputting of SCCs is done by lines 12–19 of VISIT, the recursive depth-first traversal routine of our algorithm). By the time S is generated, the SCCs reachable from it in $\widehat{G}$ have all been generated, and the entries of $Reach$ corresponding to the representatives of these reachable SCCs have been precisely computed. Then all we need to fill out $Reach(u_0)$, where u_0 is the representative of S, is to track the edges out of S and take the union of S and the entries of $Reach$ corresponding to the children of S in $\widehat{G}$. Note that these outgoing edges could either be edges in the DFS tree or DFS "cross edges." They are tracked using 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 computes the set of nodes reachable from u_0.

As for the time 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 BASELINE-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 nodes for each edge in $\widehat{G}$, so that its total cost is $O(mn)$. As for space complexity, the sets $Reach(u)$ can be stored using $O(n^2)$ space, a cost that subsumes the space requirements of the other data structures. Then we have:

LEMMA 3. BASELINE-CLOSURE *terminates on any graph G with n nodes and m edges in time $O(mn)$. On termination, for every node u of G, $Reach(rep(u))$ is the set of nodes reachable from u. The algorithm requires $O(n^2)$ space.*

We will now show a way to speed up the procedure BASELINE-CLOSURE using a slight modification of Rytter's (1983, 1985) speedup for loop-free 2-PDAs. Let V be the set of all nodes of G (we have $|V| = n$), $p = \lceil \log n/2 \rceil$, and $r = \lceil n/p \rceil$. We use fast set representations of sets of nodes $X \subseteq V$—each such set is represented as a sequence r words, each of length p. We will need to convert a list representation of X into a fast set representation as above. It is easy to see that this can be done using a sort in $O(n \log n)$ time.

/* speeds up the operation
$Reach(u) \leftarrow \bigcup_{v \in Out(u)} Reach(rep(v))$ */
let $x_1, \ldots, x_r$ be the words in the fast set for $Out(u)$ in
SPEEDUP()
1 compute $\langle x_1, \ldots, x_r \rangle$
2 **for** $1 \leq i \leq r$
3 **do if** $x_i = \mathbf{0}$ **continue**
4   **if** $Cache(i, x_i) = \perp$
5   **then** $Cache(i, x_i) \leftarrow \cup_{v \in Set(i,x_i)} Reach(rep(v))$
6    $Reach(u) \leftarrow Reach(u) \cup Cache(i, x_i)$

Figure 7. The speedup routine

Now recall that the bottleneck of the baseline algorithm is line 19 of the routine VISIT, which costs $O(mn)$ over an entire run of the algorithm. Now we show how to speed up this line. First, let us implement BASELINE-CLOSURE such that entries of the table $Reach$ are stored as fast sets, and the sets $Out(u)$ are represented as lists. Now consider the procedure SPEEDUP in Fig. 7, which is a way to speed up computation of the recurrence $Reach(u) \leftarrow \bigcup_{v \in Out(u)} Reach(rep(v))$. The idea is cache the value $(\cup_{v \in X} Reach(rep(v)))$ exhaustively for all non-empty sets X that are sufficiently small, and use this cache to compute the value for larger sets $Out(u)$. This is done using a table $Cache$ (of global scope) such that for each $1 \leq i \leq r$ and for each word $w \neq \mathbf{0}$ of length p, we have a table entry $Cache(i, w)$ containing either a subset of V, represented as a fast set, or a special "null" value $\perp$ (note that the pair (i, w) uniquely identifies a subset of V of size at most p—this set is denoted by $Set(i, w)$). Initially, every entry of $Cache$ equals $\perp$.

Let us now use the Assign-Union operation for fast sets (see Section 2) to implement line 6 of SPEEDUP, and replace line 19 of VISIT by a call to SPEEDUP. To see that this leads to a speedup, note that $Cache$ has at most $r.2^p = O(n^{3/2}/\log n)$ entries. Now, line 5 in SPEEDUP gets executed at most once for each cell in $Cache$ during a complete run of CLOSURE—i.e., $O(r.2^p) = O(n^{3/2}/\log n)$ times. Each time it is executed, it costs $O(n)$ time (as $Set(i, x_i)$ is of size $O(\log n)$ and as union of two entries of $Reach$ costs $O(n/\log n)$ time), so that its *total* cost is $O(n^{5/2}/\log n)$. Thus, the bottleneck is line 6. Let us compute the total number of times this line is executed during a run of closure. Since the total size of all the $Out(u)$'s during a run of BASELINE-CLOSURE is bounded by m, the emptiness test in line 3 ensures that line 6 is executed $O(m)$ times in total during a run of the closure algorithm (this is the tighter bound when the graph is sparse). The other obvious bound on the number of executions of this line is $O(r.n)$ (this captures the dense case). Each time it is executed, it costs time $O(r)$. Thus, the total complexity of the modified algorithm (let us call this algorithm CLOSURE) is $O(\min\{m.r, r.n.r\})$—i.e., $O(\min\{mn/\log n, n^3/\log^2 n\})$.

As for the space requirement of the algorithm, each fast set stored in a cell of the table $Cache$ costs space $O(n)$. As $Cache$ has $O(n^{3/2}/\log n)$ cells, the total cost of maintaining this table is $O(n^{5/2}/\log n)$. The space costs of the other data structures, including the table needed for fast sets operations if unit-cost word operations are not available, is subsumed by this cost. Hence we have:

THEOREM 3. CLOSURE *computes the transitive closure of a directed graph with n nodes and m edges in*

$$O(\min\{mn/\log n, n^3/\log^2 n\})$$

time and $O(n^{5/2}/\log n)$ space.

4.2 Bounded-stack RSMs

Using the ideas discussed earlier in this section, the algorithm CLOSURE can now be massaged into a reachability algorithm for bounded-stack RSMs. Figure 8 shows pseudocode for a baseline algorithm for same-context reachability in bounded-stack RSMs obtained by modifying BASELINE-CLOSURE. The sets $H^s(u)$ in the new algorithm correspond to the sets $Reach(u)$ in the transitive closure algorithm. The main difference lies in lines 14–17, which insert the summary edges into the graph. Also, as it is same-context reachability that we are computing, a child is added to the set $Out(u)$ only if it is reached along a local or summary edge (the "else" condition in line 17). A correctness argument may be given following the discussion earlier in this section.

Adding an extra transitive closure step at the end of this algorithm gives us an algorithm for reachability. With some extra bookkeeping, it is possible to evade this last step and compute reachability and same-context reachability in the same search—we omit the details. The speedups discussed earlier in this section may now be applied. Let us call the resultant algorithm STACK-BOUNDED-REACHABILITY. It is easy to see that its complexity is the same as that of CLOSURE. The only extra overhead is that of inserting the summary edges, and it is subsumed by the costs of the rest of the algorithm. Thus, the algorithm STACK-BOUNDED-REACHABILITY has time complexity $O(\min\{mn/\log n, n^3/\log^2 n\})$, where m and n are the number of edges and nodes in the summary graph of the RSM. The space complexity is as for CLOSURE. In general, m is $O(n^2)$, so that:

THEOREM 4. *The algorithm* STACK-BOUNDED-REACHABILITY *computes all-pairs reachability in a bounded-stack RSM of size n in $O(n^3/\log^2 n)$ time and $O(n^{5/2}/\log n)$ space.*

We note that an algorithm as above cannot be obtained from any of the existing subcubic algorithms for graph transitive closure. All previously known $O(n^3/\log^2 n)$-time algorithms for graph

```
VISIT(u)
  add u to Visited
  push(u, L)
  low(u) ← dfsnum(u) ← height(L)
  H^s(u) ← ∅;   rep(u) ←⊥
  Out(u) ← ∅
  if u is an internal state
    then Next(u) ← {v : u → v}
    else if u is a call (b, en)
           then Next(u) ← {en}
           else Next(u) ← ∅
  for v ∈ Next(u)
  do if v ∉ Visited then VISIT (v)
     if v ∈ Done
       then if u = (b, en) is a call and v = en
              then for exit states ex ∈ H^s(en)
                     do add (b, ex) to Next(u)
              else add v to Out(u)
       else low(u) ← min(low(u), low(v))
  if low(u) = dfsnum(u)
    then repeat
              v ← pop(L)
              add v to Done
              add v to H^s(u)
              Out(u) ← Out(u) ∪ Out(v)
              rep(v) ← u
         until v = u
         H^s(u) ← H^s(u) ∪ ⋃_{v∈Out(u)} H^s(rep(v))

BASELINE-SAME-CONTEXT-STACK-BOUNDED-REACHABILITY()
  Visited ← ∅;   Done ← ∅
  for each state u
  do if u ∉ Visited then VISIT (u)
```

Figure 8. Same-context reachability in bounded-stack RSMs

transitive closure use reductions to boolean matrix multiplication and do not permit online edge addition even if, as is the case for bounded-stack RSMs, these edges arise in a special way. While Chan (2005) has observed that DFS-based transitive closure may be computed in time $O(mn/\log n)$ using fast sets, this complexity does not suffice for our purposes.

5. Reachability in hierarchical state machines

As we saw, the reason why reachability in bounded-stack RSMs is easier than general RSM-reachability is that summary edges in the former case have a "depth-first" structure. For hierarchical state machines, the structure of summary edges is restricted enough to permit an algorithm with the same complexity as boolean matrix multiplication.

Let us have as input a hierarchical state machine M with components $M_1, \ldots, M_k$, such that a call from the component M_i can only lead to a component M_j for $j > i$. The summary graph H of M 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 $j > i$. As the component M_k does not call any other component, there are no summary edges in H_k.

To compute reachability in M, first compute the transitive closure of H_k. Next, for all entries en and exits ex of M_k and all boxes b with $Y(b) = k$, add summary edges $((b, en), (b, ex))$. 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 process the graph H_k again. We proceed inductively, processing every H_i only once. Once the transitive closure of H_1 is computed, we add all the call edges from the different H_1's and compute the transitive closure of the entire graph. By Lemma 1, there is an edge from v to v' in the final closure iff v' is reachable from v.

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 state machines 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.*

Of course, the above procedure is far from compelling—the cubic, summarization-based reachability algorithm published in the original reference on the analysis of these machines (Alur and Yannakakis 1998) is going to outperform it in any reasonable application. However, taken together with our other results, it highlights a gradation in the structure of the summary graph and the complexity of RSM-reachability as recursion in the input RSM is constrained.

6. Conclusion

In this paper, we have adapted a simple existing technique into the first subcubic algorithms for RSM-reachability and CFL-reachability, and identified a way to exploit constraints on recursion during reachability analysis of RSMs. In summarization-based analysis of general RSMs, summary edges can arise in arbitrary orders, and all-pairs reachability can be determined in time $O(n^3/\log n)$. For bounded-stack RSMs, 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 state machines, the problem is essentially that of computing transitive closure of the components.

Given that RSM-reachability is a central algorithmic problem in program analysis, the natural next step is to evaluate the practical benefits of these contributions. Such an effort should remember that real implementations of RSM-reachability-based program analyses apply heuristics such as cycle elimination and node clustering, and are often fine-tuned to the specific problem at hand. Thus, instead of implementing our algorithms literally, the goal should be to explore combinations of techniques known to work in practice with the high-level ideas used in this paper. As for algorithmic directions, a natural question is whether this is the best we can do. A hard open question is whether all-pairs CFL-reachability can be reduced to boolean matrix multiplication. This would be especially satisfactory as the former can be trivially seen to be as hard as the latter. Yannakakis (1990) has noted that Valiant's reduction of context-free recognition to boolean matrix multiplication (Valiant 1975) 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.

Another set of questions involves stack-bounded RSMs and our transitive closure. Given a program without infinite recursion, can we automatically generate a stack-bounded abstraction that can be analyzed faster than a general RSM abstraction? Can our transitive closure algorithm have applications in other areas—for example, databases? Recall that, being a search-based algorithm, it does not require the input graph to be explicitly represented, and is suitable for computing partial closure—i.e., computing the sets of nodes reachable from some, rather than all, nodes. Algorithms with such features have been studied with theoretical as well as practical motivations— a new engineering question would be to see how well the techniques of this paper combine with them.

Acknowledgements: The author thanks Rajeev Alur, Byron Cook, Stephen Fink and Mihalis Yannakakis for valuable comments. An anonymous referee pointed out that Rytter's speedup could be applied directly to the classical CFL-reachability algorithm; we thank him or her for this.

References

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.

R. Alur and M. Yannakakis. Model checking of hierarchical state machines. In *6th ACM Symposium on Foundations of Software Engineering*, pages 175–188, 1998.

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.

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. ISSN 0197–6788.

T. Ball and S. Rajamani. The SLAM toolkit. In *13th International Conference on Computer Aided Verification*, pages 260–264, 2001.

A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Applications to model checking. In *8th International Conference on Concurrency Theory*, LNCS 1243, pages 135–150, 1997.

T. M. Chan. All-pairs shortest paths with real weights in $o(n^3/\log n))$ time. In *9th Workshop on Algorithms and Data Structures*, pages 318–324, 2005.

T. M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. In *39th ACM Symposium on Theory of Computing*, pages 590–598, 2007.

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

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

S. Horwitz, T. W. Reps, and D. Binkley. Interprocedural slicing using dependence graphs (with retrospective). In *Best of Programming Language Design and Implementation*, pages 229–243, 1988.

S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *3rd ACM Symposium on Foundations of Software Engineering*, pages 104–115, 1995.

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

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

J. Rehof and M. Fähndrich. Type-base flow analysis: from polymorphic subtyping to CFL-reachability. In *28th ACM Symposium on Principles of Programming Languages*, pages 54–66, 2001.

T. Reps. Shape analysis as a generalized path problem. In *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 1–11, 1995.

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

T. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *22nd ACM Symposium on Principles of Programming Languages*, pages 49–61, 1995.

T. W. Reps, S. Schwoon, and S. Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *10th Static Analysis Symposium*, pages 189–213, 2003.

W. Rytter. Time complexity of loop-free two-way pushdown automata. *Information Processing Letters*, 16(3):127–129, 1983.

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

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

M. Sharir and A. Pnueli. Two approaches to interprocedural dataflow analysis. *Program Flow Analysis: Theory and Applications*, pages 189–234, 1981.

M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. In *20th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 59–76, 2005.

L. G. Valiant. General context-free recognition in less than cubic time. *Journal of Computer and System Sciences*, 10(2):308–315, 1975.

M. Yannakakis. Graph-theoretic methods in database theory. In *9th ACM Symposium on Principles of Database Systems*, pages 230–242, 1990.