Implicit Enumeration of Strongly Connected Components *

Aiguo Xie and Peter A. Beerel
Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-2562

Abstract

This paper presents a BDD-based implicit algorithm to compute all maximal strongly connected components of directed graphs. The algorithm iteratively applies reachability analysis and sequentially identifies SCCs. Experiments suggest that the algorithm dramatically outperforms the only existing implicit method which must compute the transitive closure of the adjacency-matrix of the graphs.

1 Introduction

Decomposing a directed graph (digraphs) into its (maximal) strongly connected components (SCCs) is a fundamental graph problem [1] and has many important applications in CAD. Generally speaking, SCC decomposition often divides a digraph problem into subproblems, one for each SCC. The solution to the original problem can be constructed by combining the solutions to the subproblems, sometimes with the aid of the component graph (i.e., the structure of connections among SCCs). Using an explicit data structure such as an *adjacency-list* or an *adjacency-matrix* [1], the decomposition of a digraph $G(V, E)$ (with V being the set of its nodes and E the set of its edges) can be solved in linear time (i.e., $O(V + E)$) using an *explicit* depth-first search [2, 1]. However, in many real application, the size of the digraph can be very large (e.g. with more than 10^{20} nodes), which prohibits the use of explicit methods.

On the other hand, implicit algorithms using BDDs [3] have been found which solve orders of magnitude larger problems (e.g. computing reachable states of a finite state machine in formal verification) than explicit methods. In particular, several BDD-based algorithms have been designed to identify *terminal* SCCs (those from which no other SCC can be reached) in the reachable state space of finite state machines, by iteratively applying reachability analysis [4, 5, 6], and by computing the transitive closure of the transition relation of the machine [7, 8]. An important difference between the two methods is that transitive-closure-based method computes all terminal SCCs simultaneously whereas the reachability-analysis-based computes terminal SCCs sequentially. Since computing the transition closure of the transition relation often requires excessive CPU time and memory, and finite state machines often have limited number of SCCs, the reachability-analysis-based methods are often dramatically faster. However, the transitive-closure-based method can

*This work is funded in part by a NSF CAREER Award MIP-9502386.

be easily adapted to compute *all* SCCs, and is the only known implicit method for this purpose.

Motivated by [4], our proposed algorithm sequentially identifies all SCCs using iterative application of reachability analysis. Experiments show that our algoritm is much faster and solves much larger problems than the transitive-closure-based method adapted from [7], especially when the graph has a small number of SCCs. The remainder of this paper is organized as follows. Section 2 reviews implicit representation of digraphs using BDDs, and summaries the transitive-closure-based method (TC-based method below) for computing all SCCs. Section 3 details our reachability-analysis-based method (RA-based method below). The performance of the two methods are compared in Section 4. We conclude the paper in Section 5, discussing a potential application of our algorithm to formal verification.

2 Preliminaries and the TC-based method

Let $G(V, E)$ denote a digraph where V is the set of its nodes, and for any two nodes $u, v \in V$, $(u, v) \in E$ iff there is an edge from u to v. A vector $(v_1, v_2, \cdots, v_{n+1})$ of nodes is a *path* of length $n+1$ (from v_1 to v_{n+1}) iff $(v_i, v_{i+1}) \in E$, for all $i = 1, \cdots, n$. Node v is *reachable* from u (denoted by $u \rightsquigarrow v$) if there is a path from u to v. A (maximal) *strongly connected component* (SCC) is a (maximal) subset of nodes where every pair of nodes are reachable from each other. Below, when we say SCCs, we refer to the maximal ones. We denote by $\mathcal{A}(G)$ the set of SCCs in G. For convenience, we call a node a non-SCC node if it does not belong to any SCC, an SCC node otherwise.

Let the nodes be labeled with distinct numbers from $\{1, 2, \cdots, |V|\}$. The digraph can then be represented by an adjacency matrix $M_{|V|\times|V|}$ [1] whose element $M(u, v)$ is 1 if $(u, v) \in E$, and 0 otherwise. The digraph may also be represented by a BDD $N(X, Y)$ by encoding the rows and columns of its adjacency matrix M with two sets of BBD variables X and Y. The BDD $N(X(u), Y(v))$ evaluates to 1 if $M(u, v) = 1$ where $X(u)$ and $Y(v)$ are the vectors encoding the labels of u and v, respectively. Details of BDDs and their common operations can be found in [3]. For convenience, we sometimes use node u to refer to the BDD encoding of its labeling.

The TC-based symbolic method first computes the transitive closure of N, denoted by N^*. By definition, $N^*(u, v) = 1$ iff $u \rightsquigarrow v$. The transpose of N^* denoted by N^{*t} has the property that $N^{*t}(u, v) = 1$ iff $v \rightsquigarrow u$. Thus, u and v belong to a same SCC iff $N^*(u, v) \wedge N^{*t}(u, v) = 1$, where $\wedge$ denotes the BDD AND [3]. Consequently, the union of all SCC states (denoted by H) can be computed as $H = \exists_Y N^* \wedge N^{*t}(X, Y)$, where $\exists$ denotes the BDD *existential quantification* [3]. Finally, the SCC containing a particular state v in H can be computed as $\exists_Y X(v) \wedge N^* \wedge N^{*t}(X, Y)$.

3 The RA-based method

Computing the transitive closure of the adjacency-matrix of digraph as required by the TC-based method is equivalent to computing the reachability set of every node (the set of nodes reachable from the given node). Our method which is based on reachability analysis (the RA-based method below) needs to compute the reachability sets of potentially very few nodes. To better describe our method, we need to introduce a few notations related to the reachability sets and several properties of such sets.

3.1 Forward and backward sets

The forward set $F(v)$ of a node $v \in V$ is the set of nodes reachable from v. That is, $F(v) = \{u \in V \mid v \leadsto u\}$. Similarly, its backward set $B(v)$ is the set of nodes that can reach v. That is, $B(v) = \{u \in V \mid u \leadsto v\}$. One of the properties of $F(v)$ and $B(v)$ is given in Lemma 3.1, which states that the intersection of the two sets (if not empty) is an SCC. Additionally, v is a non-SCC node if the intersection is empty.

Lemma 3.1 *[4, 6, 5] If v is a node of $G(V, E)$, then $F(v) \cap B(v) \in \mathcal{A}(G)$.*

A subset $U \subseteq V$ is *SCC-closed* if no SCC intersects with both U and its complement $V \backslash U$. That is, any SCC must be either completely contained in such a set or they do not overlap.

Lemma 3.2 *Both backward and forward sets of any node are SCC-closed.*

Proof: (Omitted due to the space limitation.) □

Theorem 3.1 *The difference of the backward sets of any two nodes is SCC-closed.*

Proof: Let u and v be any two nodes. If $B(u) \cap B(v)$ is empty, the theorem is trivial. If $B(u) \cap B(v) \neq \emptyset$, then either $B(u) \subseteq B(v)$ or $B(v) \subseteq B(u)$. If $B(u) = B(v)$, the theorem is again trivial. Now, let us assume $B(u) \subset B(v)$. Suppose the difference D of the two sets, i.e., $D = B(v) \backslash B(u)$, is not SCC-closed, then there must be an SCC, say A such that $A \cap D \neq \emptyset$ and $A \backslash D \neq \emptyset$. However, since $B(v)$ is SCC-closed by Lemma 3.2, we have $A \subseteq B(v)$, and thus $A \backslash D \subseteq B(v)$. But since $B(v) = D \cup B(u)$, then $A \backslash D \subseteq B(u)$. This implies $A \cap B(u) \neq \emptyset$, suggesting $B(u)$ is not SCC-closed. A contradiction to Lemma 3.2. □

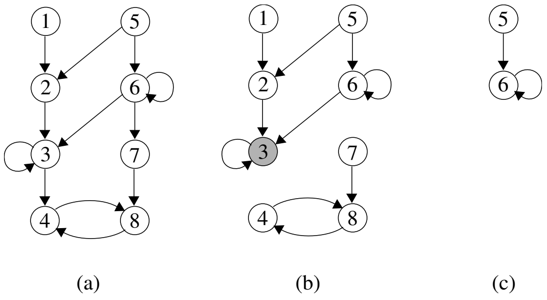

Figure 1: A digraph and its decomposition.

Remark: By similar arguments, one can show that the difference of any two of $B(u)$, $B(v)$, $F(u)$ and $F(v)$ is SCC-closed, and more generally, that the difference of any two SCC-closed sets is SCC-closed.

Example: In the digraph shown in Figure 1(a), $B(3) = \{1, 2, 3, 5, 6\}$, $F(3) = \{3, 4, 8\}$, and $B(4)$ =$\{1,2,3,4, 5,6,7, 8\}$. From Lemma 3.1, $B(3) \cap F(3) = \{3\}$ is an SCC. From Lemma 3.2, $B(3)$ is SCC-closed. Moreover, the difference $B(4) \backslash B(3) = \{4, 7, 8\}$ is also SCC-closed from Theorem 3.1.

3.2 The algorithm

Lemma 3.2 and Theorem 3.1 suggest that a digraph may be first partitioned into backward sets (or differences of backward sets) of some of its nodes, and then the computing of SCCs of the graph can be restricted to each of the partitions. This leads to a divide-and-conquer strategy. For instance, the digraph in Figure 1(a) can be partitioned into $B(3)$ and $B(4) \backslash B(3)$, which reduces the problem to finding SCCs within the two subsets independently, as illustrated in Figure 1(b).

Because a digraph G and its transpose G^t (the graph obtained by reversing all the edges of G) has the same set of SCCs [1], the above divide-and-conquer strategy may also be carried out in term of forward sets. In fact, the

```
SCC_DECOMP(N, V)
    V' ← V;
    while(V' ≠ 0) {
        v ← random_take(V');
        B(v) ← backward_set(v, V', N);
        SCC_DECOMP_RECUR(v, B(v), N);
        V' ← V' ∧ ¬(v ∨ B(v));
    }
```

Figure 2: The top-level algorithm.

```
FMD_PRED(W, U, N)
    pred ← 0;
    front ← W;
    bound ← U;
    while(front ≠ 0) {
        x ← ∃_Y front(Y) ∧ N(X, Y) ∧ bound(X);
        y ← ∃_Y bound(Y) ∧ N(X, Y) ∧ bound(X);
        front ← x ∧ ¬y;
        pred ← pred ∨ front;
        bound ← bound ∧ ¬front;
    }
    return pred;
```

Figure 3: Computing the predecessors with finite maximum distance.

partitioning is also possible to be carried out by interleaving the computation of backward sets and forward sets. However, our experience shows that none of these three different partitioning schemes leads to a significant advantage over the others. For this reason, we focus on explaining the algorithm based on the backward-set partitioning scheme.

At the top level, the algorithm first picks a node from the set of remaining nodes (the entire set V at the beginning), computes its backward set restricted to the remaining nodes, and then decomposes the backward set into SCCs and a set of non-SCC nodes. This process repeats until the remaining set of nodes is empty.

The procedure is formally described in Figure 2. In the algorithm, function `random_take`(V') randomly picks a node from the set V'. Function `backward_set`(v, V', N) returns the backward set of v *restricted to* set V', which can be implemented efficiently using the fixed-point computation [9]. The remainder of this subsection explains the core procedure `SCC_DECOMP_RECUR`$(v, B(v), N)$ which recursively decomposes $B(v)$ into SCCs and a set of non-SCC nodes.

To describe the procedure `SCC_DECOMP_RECUR`, we introduce a concept of predecessors with finite maximum distance, also referred to as *FMD predecessors*. A node u is an FMD predecessor of node v if any path from u to v has finite length. That is to say, any path from u to v must not pass any SCC. A necessary condition for this is that u must be a non-SCC node. The set of FMD predecessors of a set W of nodes is defined to be the set of all FMD predecessors of nodes in W, denoted by $FMD\_PRED(W)$. As an example, in Figure 1(b), $FMD\_PRED(\{3\}) = \{1, 2\}$. Node 5 is not a FMD predecessors of node 3 because it may pass the SCC $\{6\}$ to reach node 3. Figure 3 gives an implicit algorithm that iteratively computes the set $FMD\_PRED(W)$ restricted to a set U.

The recursive procedure `SCC_DECOMP_RECUR`$(v, B(v), N)$ works as follows. It first computes the forward set $F(v)$ of v restricted to $B(v)$. If $F(v)$ is not empty, then it is an SCC due to Lemma 3.1. Otherwise, node v is a non-SCC node. In either case, $F(v) \vee v$ can be removed from $B(v)$ from further consideration. Next, the FMD predecessors of $F(v) \vee v$ is computed, and are subsequently removed from further consideration. Then, from the remaining set of nodes, the procedure computes the set of immediate predecessors (IP) of these already removed nodes.

To decompose the remaining set of nodes, a node u is randomly picked from the IP set and its backward set $B(u)$ is computed. The procedure then calls itself to decompose $B(u)$. After it returns, u and $B(u)$ are removed from the remaining set of nodes and the IP set gets updated. If the remaining set of nodes is not empty, another node w is picked up from the updated IP set, and its backward set $B(w)$ is recursively decomposed. This process repeats until the remaining set of nodes is empty. A formal description of the procedure is given in Figure 4.

Example: In figure 1(b), a call to `SCC_DECOMP_RECUR`(3, B(3), N) first computes $F(3, B(3), N) = \{3\}$, and reports $F(3) \cap B(3) = \{3\}$ to be an SCC. Next, it computes $FMD\_PRED(\{3\}) = \{1, 2\}$ which is reported to be a set of non-SCC nodes, and the IP set is computed to be $\{5, 6\}$. At this point, the procedure has reduced the problem to decomposing the digraph in Figure 1(c). Suppose node 6 is picked from the IP set, the procedure calls `SCC_DECOMP_RECUR`(6,$B(6)$,N) which, in this case, leaves no more nodes to be decomposed when it returns.

```
SCC_DECOMP_RECUR(v, B(v), N)
    F(v)←forward_set(v, B(v), N);
    if(F(v) ≠ 0)
        report F(v) an SCC;
    else
        report v non-SCC;
    x←F(v) ∨ v;
    R←B(v) ∧ x̄;
    y←FMD_PRED(x, R, N);
    report y non-SCC;
    R←R ∧ ȳ;
    IP←backward_set(y ∨ x, R, N);
    while(R ≠ 0) {
        v←random_take(IP);
        B(v)←backward_set(v, R, N));
        SCC_DECOMP_RECUR(v, B(v));
        R←R ∧ ¬(R ∨ B(v));
        IP←IP ∧ ¬(IP ∨ B(v));
    }
```

Figure 4: Recursive decomposition of a backward set.

3.3 The complexity

As shown in Figure 4, procedure `SCC_DECOMP_RECUR` performs a constant number of reachability analyses before it calls itself. Each reachability analysis must have completed in $O(\mathcal{D})$ BDD operations where $\mathcal{D}$ is the diameter of the graph. Since a call to `SCC_DECOMP_RECUR` removes at least one node from further consideration, `SCC_DECOMP_RECUR`(v,$B(v)$,N) cannot call itself for more than $|B(v)|$ times. Thus, a trivial upper bound on the complexity of our algorithm is $O(|V| \times \mathcal{D})$ in BDD operations.

From our experiments, we observe that a call to `SCC_DECOMP_RECUR` typically removes an SCC before the procedure calls itself. In practice, the algorithm terminates in $O(|\mathcal{A}(G)| \times \mathcal{D})$ time (in BDD operations). Note further that the reachability analysis performed by our algorithm is always restricted to the subset of the nodes that is currently under consideration. In other words, the analysis needs a potentially much smaller number of BDDs operations than the diameter of the graph. In this respect, our complexity analysis above is pessimistic.

4 Experiments

We have implemented both TC-based and RA-based methods with `vis-1.3` [10]. In particular, the transitive closure of adjacency-matrix of the graph is computed by iteratively squaring the matrix. This section compares their performance using the examples distributed with the `vis-1.3` package each of

Example	State space			CPU secs	
	$\|V\|$	$\|V'\|$	SCCs	TC	RA
abp	484	444	11	5.41	0.40
arbit	5,568	5,568	1	797	0.52
bakery	2,886	2,604	31	197	3.24
coherence	94,748	94,688	2	`mo`	41.0
cdnew	186,876	139,520	3,469	`to`	202
eisenberg	1,611	1,609	1	`to`	0.49
emodel	34,133	376	12	to	96.0
scheduler	2.4e+6	2.4e+6	1	`mo`	1.19
minMax30	2.1e+26	2.1e+26	1	`mo`	6.68
tcp	3.9e+22	-	-	`mo`	`to`

Table 1: The experimental results where $|V|$ is the number of reachable states, $|V'|$ denotes the number of states belonging to SCCs, `mo` denotes memory out and `to` denotes time out after 1 hour of CPU time.

which specifies a finite state machine. The methods are used to compute the SCCs in the reachable state space of the finite state machines. The package is run on a Sun Ultra 10 with 640 Mbytes of memory. Table 1 gives the experimental results of a representative set of the examples.

Since the TC_method simultaneously computes all SCCs, we report its run time as soon as it computes the transition closure of the adjacency-matrix and ANDs the transitive closure with its transpose. That is, the time to list all the SCCs is not included for in the reported run times of the TC_method.

As shown in the Table, the TC-method solves only a few small examples whereas our RA-based method solves most of them. Even for those solved by the TC-based method, the RA-based method is faster by orders of magnitude. The RA-based method runs out of time in the last example. The reason is that the example contains excessive SCCs and the RA-based method cannot finish reporting all of the them in time, which highlights the limitation of our method.

5 Conclusion

Computing strongly connected components (SCCs) of a directed graph is a generic graph problem. For this purpose, we have proposed an implicit algorithm using BDDs. The method iteratively applies reachability analysis and computes SCCs sequentially. It has been used to compute the SCCs in the reachable state space of a set of finite state machines. Compared with an existing symbolic method which requires the transitive closure of the adjacency-matrix of the graph, our method is shown to be much faster, requires much less memory, and consequently solves much larger problems. Nevertheless, when there are excessive SCCs, our approach may not complete in a reasonable amount of time. Consequently, it might be interesting to explore a hybrid approach between our RA-based method and the TC-based method, which may be able to find multiple SCCs of relatively small size simultaneously but without computing the transitive closure of the entire graph.

We are currently studying a potential application of our method to the *bad-cycle-detection* problem in formal verification [11, 12]. We expect that when the reachable state space has a small number of SCCs, our method may outperform the Emerson-Lei algorithm [11] and its refinements (e.g., [12]).

References

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

[2] R. E. Tarjan. Depth first search and linear graph algorithms. *ACM Journal on Computing*, 1(2), 1972.

[3] R. E. Bryant. Graph-based algorithm for boolean function munipulation. *IEEE Transactions on Computers*, 35, August 1986.

[4] A. Xie and P. A. Beerel. Efficient state classification of finite state Markov chains. *IEEE Transactions on Computer-Aided Design*, 17(12):1334–1338, December 1998.

[5] V. Singhal. *Design Replacements for Sequential Circuits.* PhD thesis, University of California at Berkeley, 1996.

[6] S. Qadeer, R. K. Brayton, V. Singhal, and C. Pixley. Latch redundancy removal without global reset. In *Proc. International Conf. Computer Design (ICCD)*, Oct. 1996.

[7] G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Markovian analysis of large finite state machines. *IEEE Transactions on Computer-Aided Design*, 15(12):1479–1493, December 1996.

[8] G. Haster, A. Mathur, and P. Banerjee. An implicit algorithm for finding steady states and its applications to FSM verifcation. In *Proc. ACM/IEEE Design Automation Conference*, pages 611–614, 1998.

[9] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design*, 13(4):401–424, April 1994.

[10] Brayton et al. VIS: A system for verfication and synthesis. In *Proc. International Conference on Computer Aided Verification*, pages 428–432, 1996.

[11] E. A. Emerson and C. L. Lei. Efficient model checking in fragments of the propositional modal mu-calculus. In *Proceedings of LICS 1986*, pages 267–278, 1986.

[12] R. H. Hardin, R. P. Kurshan, S. K. Shukla, and M. Y. Vardi. A new heuristic for bad cycle detection using bdds. In *Proc. International Workshop on Computer Aided Verification*, pages 268–278, 1997.