Comp. & Maths. with Appls. Vol. 7, pp. 67-72
Pergamon Press Ltd., 1981. Printed in Great Britain

A STRONG-CONNECTIVITY ALGORITHM AND ITS APPLICATIONS IN DATA FLOW ANALYSIS†

M. SHARIR
Courant Institute, New York University, New York, U.S.A.

Communicated by J. T. Schwartz

(*Received January* 1980)

Abstract—We present a new linear algorithm for constructing all strongly connected components of a directed graph, and show how to apply it in iterative solution of data-flow analysis problems, to obtain a simple algorithm which improves the Hecht–Ullman algorithm.

1. INTRODUCTION

We present here a new algorithm for constructing all strongly-connected components of a directed graph. Like Tarjan's well known algorithm ([1], Chap. 5.7) it uses a depth-first spanning tree (forest) T, and is linear in the number of nodes and edges of the graph. However, our algorithm differs from Tarjan's in that it produces these components in reverse postorder of their roots (relative to T), and also orders the nodes within each component in reverse postorder. Together, these orderings induce a modified reverse postorder of the graph nodes, which facilitates certain iterative algorithms related to data-flow analysis. We will describe and analyze such a data-flow algorithm which improves the algorithm of Hecht and Ullman[2]. Even though the ordering we are concerned with can also be obtained using a slightly modified variant of Tarjan's algorithm, we present our algorithm as a simpler (though not necessarily more efficient) alternative.

The strong-connectivity algorithm is presented and analyzed in Section 2. Section 3 discusses its applications to the solution of data-flow analysis problems.

2. A STRONG-CONNECTIVITY ALGORITHM

Let us assume that we are given a directed graph G rooted at a unique 'entry' node r. Let N be the set of nodes of G and E the set of its edges. For each $n \in N$ denote by SCC(n) the *strongly-connected component* of G containing n, i.e. the maximal set of nodes containing n such that G restricted to that set is strongly connected. Let T be a depth-first spanning tree for G.

The following algorithm will compute the following objects: a list SCCS of roots of strongly-connected components in their reverse postorder (relative to T) and a map SCCNODES mapping each (root of a) strongly connected component to a list of its nodes in the same reverse post-order. The algorithm proceeds in the following steps:

Algorithm SCOMPS

(1) Initialize SCCS to the null list and SCCNODES to the null map. Also initialize an auxiliary map SCCROOT, which will map each node in N to the root of its strongly-connected component, to the null map.

(2) Iterate through N in reverse postorder (relative to T). Let $h \in N$ be the node currently visited.

(3) If SCCROOT(h) is still undefined, then h is the root of a new strongly connected component. In this case we compute the set

$S(h) = \{h\} \cup \{w$: w is a T-descendant of h which can reach h along a path consisting solely of T-descendants of $h\}$

†This research has been supported by NSF Grant MCS076-00116 and U.S. DOE Office of Energy Research Contract EY-76-C-02-3077.

and extend the SCCROOT map, by mapping all $w \in S(h)$ to h. The set $S(h)$ is computed as follows (square brackets denote ordered tuples).

```
$S(h) := [h]$;
NEW: $= [w: (w, h) \in E$ and $w$ is a $T$-descendant of $h]$;
(while NEW is not empty)
    remove an element $w$ from NEW;
    SCCROOT($w$): $= h$;
    add $w$ to $S(h)$;
    add to NEW all nodes $v$ where $(v, w) \in E$, $v$ is a $T$-descendant of $h$ and SCCROOT($v$)
            is undefined;
end while;
```

Note that we can test the condition 'v is a T-descendant of h' rapidly using the formula

$$v \text{ is a } T\text{-descendant of } h \text{ iff } \mathrm{pre}(h) < \mathrm{pre}(v) \leq \mathrm{pre}(h) + \#\text{descendants of } h$$

(where pre(x) denotes the preorder index of a node x). However, using the special properties of depth-first spanning trees, we can replace the above test by the following simpler test.

In the above construction of $S(h)$, v is a T-descendant of h iff $\mathrm{post}(v) < \mathrm{post}(h)$ (where post(x) is the postorder index of a node x).

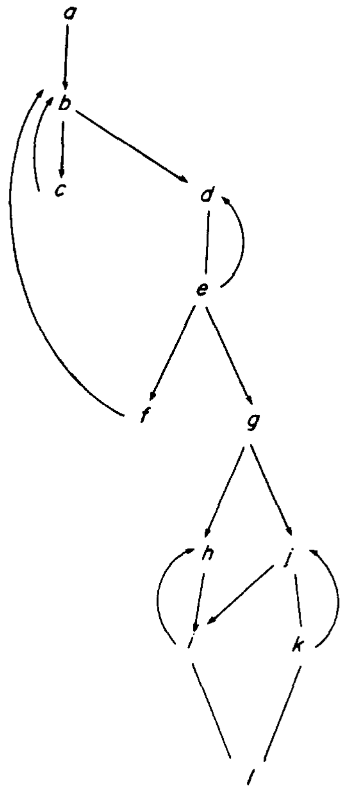
Indeed, to explain the nonobvious implication, assume that $\mathrm{post}(v) < \mathrm{post}(h)$. Then either v is a T-descendant of h or else v is to the left of h, but only the first case is possible, because in the second case we would have the impossible left-to-right cross edge (v, w). (We are grateful to Gerald Fisher for this observation.)

Having computed $S(h)$, we add h to SCCS, and set SCCNODES(h): $= [h]$.

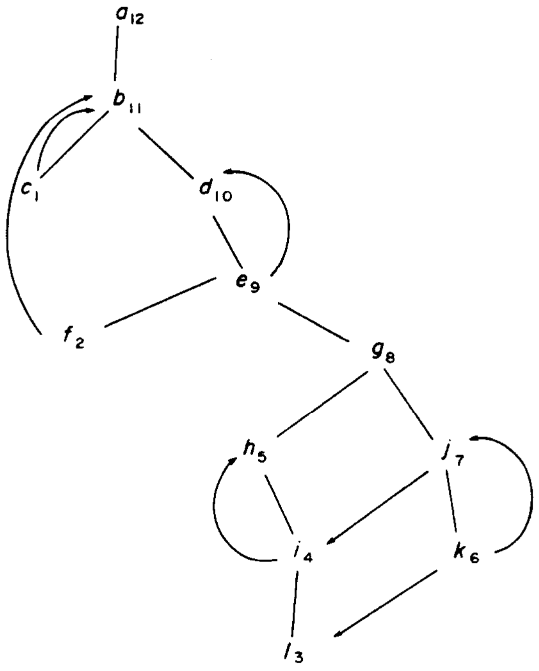
(4) If SCCROOT(h) $= u$ is already defined, we add h to the end of SCCNODES(u).

Before proving the correctness of this algorithm, we give an example illustrating it.

Example. Consider the following flow graph:



which has the following DFST (we label nodes with their postorder indices):



The reverse postorder is then

$$a\quad b\quad d\quad e\quad g\quad j\quad k\quad h\quad i\quad l\quad f\quad c.$$

We now list the actions taken by our algorithm as it iterates through that sequence:

Node visited	*Action*
a	SCCROOT(a): = a; SCCS: = [a]; SCCNODES(a): = [a]
b	SCCROOT(b, c, d, e, f): = b; SCCS: = [a, b]; SCCNODES(b): = [b]
d	SCCNODES(b): = [b, d]
e	SCCNODES(b): = [b, d, e]
g	SCCROOT(g): = g; SCCS: = [a, b, g]; SCCNODES(g): = [g]
j	SCCROOT(j, k): = j; SCCS: = [a, b, g, j] SCCNODES(j): = [j];
k	SCCNODES(j): = [j, k];
h	SCCROOT(h, i): = h; SCCS: = [a, b, g, j, h] SCCNODES(h): = [h]
i	SCCNODES(h): = [h, i]
l	SCCROOT(l): = l; SCCS: = [a, b, g, j, h, l] SCCNODES(l): = [l]
f	SCCNODES(b): = [b, d, e, f]
c	SCCNODES(b): = [b, d, e, f, c]

Next we prove the correctness and linearity of algorithm SCOMPS.

LEMMA 1

For each h processed by step (3) of the algorithm, $S(h)$ is a subtree of T rooted at h.

Proof. If $v \in S(h)$ and u is a T-ancestor of v and a T-descendant of h, then obviously u will also be added to $S(h)$.

Q.E.D.

Lemma 2

For any two nodes h_1, h_2 processed during step (3) of the algorithm, $S(h_1) \cap s(h_2) = \emptyset$.

Proof. Suppose that h_1 has a higher postorder number than h_2 (i.e. h_1 is visited before h_2). Suppose that there exists $w \in S(h_1) \cap S(h_2)$. Then w is a T-descendant of both h_1 and h_2, so that h_1 must be a T-ancestor of h_2, and Lemma 1 implies then that $h_2 \in S(h_1)$. But then SCCROOT(h_2) will be set to h_1 when h_1 is processed, so that h_2 cannot be processed by step (3). This contradiction proves our assertion.

Q.E.D.

Lemma 3

For each h processed by step (3) we have $S(h) = \mathrm{SCC}(h)$.

Proof. Obviously, $S(h)$ is strongly-connected and contains h. To show maximality, we proceed by induction on the nodes in reverse postorder. Let $h = r$, the first node in reverse postorder, and suppose that there exists some $w \in \mathrm{SCC}(h) - S(h)$. Then w is obviously a T-descendant of h from which a path can reach h, so that $w \in S(h)$, which contradicts our assumption. Next suppose the assertion to be true for all nodes preceding some node h processed during step (3) of the algorithm, but that $\mathrm{SCC}(h) - S(h)$ is not empty and contains a node w. If w has a higher postorder index than h, then so has $v = \mathrm{SCCROOT}(w)$, and by the induction hypothesis, $\mathrm{SCC}(v) = S(v)$. However, this implies that $h \in S(v)$, and hence h cannot be processed by step (3), since SCCROOT(h) will already have been set to v by the time h is visited. Therefore h must have a higher postorder index than w. But then if w and all other nodes lying on a path from w to h (which exists since $w \in \mathrm{SCC}(h)$) are T-descendants of h, then w would have been placed in $S(h)$ by definition. Therefore, at least one node w_1 along such a path must lie to the left of h or above h in the tree T. It is easy to see then that this path must pass through some T-ancestor u of h and w_1 (note that $u \neq h$). Thus u, h and w_1 all belong to the same strongly-connected component which then also contains $v = \mathrm{SCCROOT}(u)$. But v has a higher postorder index than h, so that, by the induction hypothesis, $\mathrm{SCC}(v) = S(v)$. Hence $h \in S(v)$, which implies, as before, that h cannot be processed by step (3) of the algorithm. Hence $\mathrm{SCC}(h) = S(h)$ and this completes the proof.

Q.E.D.

Theorem 1

Algorithm SCOMPS correctly computes all strongly-connected components of G in the required external and internal orders, and takes $O(\max(\#N, \#E))$ time.

Proof. Lemma 3 asserts that each set $S(h)$ constructed in step (3) is a strongly-connected component, and Lemma 2 implies that no such component is computed more than once. Since the union of all these sets is N (easily seen by noting that at the end of the iteration step (2) SCCROOT must be everywhere defined), it follows that the algorithm computes all strongly connected components of the graph. Our claims concerning the external ordering of these components in SCCS, as well as the internal orderings of their nodes as reflected in SCCNODES, then follow immediately.

To analyze the time complexity of the algorithm, we note that the only section of the algorithm whose linearity in max $(\#N, \#E)$ is nontrivial is the computation of the sets $S(h)$. Nevertheless, we claim that in the construction of the NEW sets, no edge is considered more than once. This is because an edge is considered only when its target is being added to some set $S(h)$, and by Lemma 2 this can happen only once during execution of the algorithm. For similar reasons, no node is added to any NEW set more than once. Hence this part of the algorithm is also linear in max $(\#N, \#E)$.

Q.E.D.

Next we note some useful properties of our ordering of strongly connected components (similar properties are discussed in [3]).

Lemma 4

Let h_1, h_2 be roots of different strongly-connected components, and let $u \in \mathrm{SCC}(h_1)$, $v \in \mathrm{SCC}(h_2)$ be such that $(u, v) \in E$. Then h_1 has a higher postorder number than h_2.

Proof. If not, then h_2 is either a T-ancestor of h_1 or else h_2 is to the right of h_1. In the first case h_2 is also a T-ancestor of u and we can reach h_2 from u via the edge (u, v) which enters

SCC(h_2). Hence $u \in$ SCC(h_2), a contradiction. In the second case (u, v) is an impossible left-to-right cross edge. These contradictions prove the lemma.

Q.E.D.

Corollary 5

If each strongly-connected component of G is reduced to a single-node, then the reduced graph is acyclic, and is topologically sorted by the reverse postorder of the head nodes of strongly-connected components.

Corollary 6

Every path p in G can be decomposed as $p_{h_1} \| p_{h_2} \| \ldots \| p_{h_k}$, where $h_1, \ldots, h_k$ are roots of strongly-connected components in reverse postorder, and where the subpath p_{h_i} passes only through nodes of SCC(h_i).

3. APPLICATION TO DATA-FLOW ANALYSIS

The above observations yield a simple but useful improvement of the iterative data-flow algorithm of Hecht and Ullman[2]. This algorithm solves the following set of equations for the data-values x_n attached to a given flow-graph (i.e. a rooted directed graph) G with root r and set of nodes N, where each x_n belongs to a given semilattice L, the functions $f_{(m,n)}$ belong to a class F of isotone maps acting on L, and $x_0 \in L$ is an initial value.

$$(*) \qquad \begin{aligned} & x_r \le x_0 \\ & x_n = \wedge\{f_{(m,n)}(x_m)\colon (m, n) \in G\}, \quad n \in N. \end{aligned}$$

The algorithm in [2] arranges nodes of N in reverse postorder (with respect to a depth-first spanning tree), and then iterates throgh this sequence repeatedly (starting with some "largest" initial value of the x_n's), applying (*) to obtain successive approximations of the solution, till these approximations converge to the maximal fixpoint of (*).

As already noted in [3], this approach suffers from some obvious inefficiencies. For example, let $n_1, \ldots, n_k$ be the nodes of N in reverse postorder, and assume that G contains only one loop, consisting of the nodes $n_i, \ldots, n_j$. Then, obviously, $n_1, \ldots, n_{i-1}$ have to be processed only once, as there is no information "feedback" to these nodes from succeeding nodes. Also, once information has stabilized along that loop, $n_{j+1}, \ldots, n_k$ also need be processed only once. However, each iteration of the Hecht–Ullman algorithm would process the whole sequence $n_1, \ldots, n_k$.

We therefore propose to revise the Hecht–Ullman algorithm as follows:

(1) Apply the strong-connectivity algorithm of the previous section to the flow-graph G, to obtain SCCS and SCCNODES.

(2) Initialize the solution map x so that $x_r = x_0$, and for all $n \in N - \{r\}$ put $x_r = \Omega$, where Ω is a special largest element of L denoting undefined data-value.

(3) Iterate once through all the nodes of SCCS (in their reverse postorder).

(4) Let h be the currently visited node. Iterate through the nodes in SCCNODES(h) (i.e. in their relative reverse postorder) repeatedly, applying (*) to obtain successive approximations to the solution as in the Hecht–Ullman algorithm, till information stabilizes for all these nodes.

Theorem 2

The above algorithm converges if the semilattice L is well-founded, and yields the maximal fixpoint of (*).

Proof. Convergence is proved in a standard manner by noting that the successive values of the solution map x form a nonincreasing sequence in L^N with only finitely-many repetitions. Since we compute x by successive approximations, starting with an initial value which is larger than any solution of (*), it is sufficient to show that the final value of x is a solution of (*), to deduce that it is the maximal fixpoint solution. To see this, let $n \in N$ be a node for which $x_n \ne \wedge\{f_{(m,n)}(x_m)\colon (m, n) \in E\}$ when the algorithm terminates. Let h be the root of the strongly-connected component containing n. By Lemma 4, each predecessor m of n either also belongs to SCC(h), or else to SCC(h') for some h' preceding h in SCCS. This implies that the final values of x at n and its predecessors are the same as these values at the end of processing SCC(h) in step (4). But these values must satisfy (*), which contradicts our assumption.

Q.E.D.

Remarks. (1) Step (4) can impose an *a priori* upper bound on the number of iterations required for each strongly-connected component S, provided that the data-flow analysis is of the "bitvectoring" type (such as available expressions analysis), or, more generally, has a framework of the kind described in [4]. As in [4], $d_S + 1$ iterations suffice, where d_S is the maximal number of back edges along any acyclic path in S. For practical purposes, this bound can be estimated by another bound $d'_S + 1$, where d'_S is the number of back-edge target nodes in S. These quantities can easily be computed during the execution of the strong-connectivity algorithm SCOMPS, and can then be used to limit the number of iterations required in step (4) of the algorithm.

(2) It follows from the last remark that if $S_1, \ldots, S_k$ are the strongly-connected components of G in the reverse postorder of their roots, then the sequence

$$S = \sum_{i=1}^{k} (d_{S_i} + 1) * S_i$$

(where summation corresponds to tuple concatenation and multiplication by an integer to tuple replication) is a (strong) *node-listing* for G in the terminology of [3]. This node listing represents a good compromise between the coarse Hecht–Ullman approach and more sophisticated, but more intricate node-listing methods. Indeed, in the light of the comments in [3], it is clear that our algorithm can be regarded as essentially using a crude node-listing which corresponds to that described by the standard Hecht–Ullman iteration for each strongly-connected component of the graph, and as combining these listings in the way suggested in [3].

(3) Applications to interprocedural data-flow analysis of the attribute-analysis algorithm described above and of the strong-connectivity algorithm are given in [5].

REFERENCES

1. A. U. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms.* Addison–Wesley, Reading, Mass. (1974).
2. M. S. Hecht and J. D. Ullman, A simple algorithm for global data flow analysis problems. *SIAM J. Computing* **4**, 519–532 (1975).
3. K. W. Kennedy, Node listings applied to data-flow analysis. *Proc. 2nd Symp. on the Principles of Programming Languages,* pp. 10–21. Palo-Alto, California (1975).
4. J. B. Kam and J. D. Ullman, Global data flow analysis and iterative algorithms. *JACM* **23**, 158–171 (1976).
5. J. T. Schwartz and M. Sharir, *A Design for Optimizations of the Bitvectoring Class.* Courant Comp. Sci. Rept. No. 17, Courant Institute, New York (1979).