On Finding the Strong Components in a Directed Graph

Esko Nuutila
Eljas Soisalon-Soininen

Laboratory of Information Processing Science
Helsinki University of Technology
Otakaari 1, SF-02150 Espoo, Finland

enu@cs.hut.fi, ess@cs.hut.fi

Abstract

We present two improved versions of Tarjan's algorithm for finding the strongly connected (or strong, for short) components in a directed graph. The original algorithm of Tarjan can be considered to contain two (interleaved) traversals of the graph. First, a depth-first search traverses all edges of the graph and constructs a depth-first spanning forest. Second, once a root of a strong component is found, all its descendants that are not elements of previously found strong components are marked as elements of this component. This second traversal is implemented by using a pushdown stack, where all nodes are pushed when entered by the depth-first search. When a root of a component is exited, then all nodes down to the root are popped from the stack and they form the component in question. When the graph is a tree or a dag the second traversal, and thus the use of the pushdown stack, is completely unnecessary. Our versions of the algorithm minimize the use of the pushdown stack. The first version stores only those nodes that are not roots of strong components. The second version stores only possible roots of nontrivial (containing at least one cycle) components. For graphs that are almost trees or dags these improvements almost completely eliminate the second traversal.

TKO-B 94/93
ISBN 951-22-1403-2
ISNN 0785-6601

TKK OFFSET 1993

1 Introduction

A strong component of a directed graph consists of nodes that are all *path equivalent* to each other, i.e., for every pair of nodes a and b in a strong component there is a path from a to b and from b to a. Finding the strong components in a directed graph has a variety of applications [1, 2, 4, 6, 7, 12, 14, 15, 16]. Tarjan [15] published an elegant linear time[1] algorithm for the problem in 1972, and since then his algorithm has appeared in numerous textbooks, e.g. in [1, 7, 12, 14]. The algorithm utilizes *depth-first search* in a clever way.

The basic idea in Tarjan's algorithm is easiest to understand by studying where the nodes of the strong components are located in the *depth-first spanning forest* of a graph. Figure 1 shows a graph with 7 strong components and one possible depth-first spanning forest of the graph. The strong components are encircled. As we can see, the nodes of every strong component form a tree in the spanning forest. For determining the strong components it is sufficient to recognize the root of each component, i.e., the node in the component having the lowest number in the *depth-first order* implied by the spanning forest. When the roots have been recognized, the nodes in a component C are obtained as those descendants of its root r_C that are not descendants of any other component root $r_{C'}$ that is a descendant of r_C. For example, in the graph of Figure 1 the component roots are v_1, v_3, v_6, v_7, v_{11}, v_{13}, and v_{14}. Besides v_{11}, the component whose root v_{11} is, contains v_{12} and v_{15} because they are the only descendants of v_{11} that do not belong to other strong components.

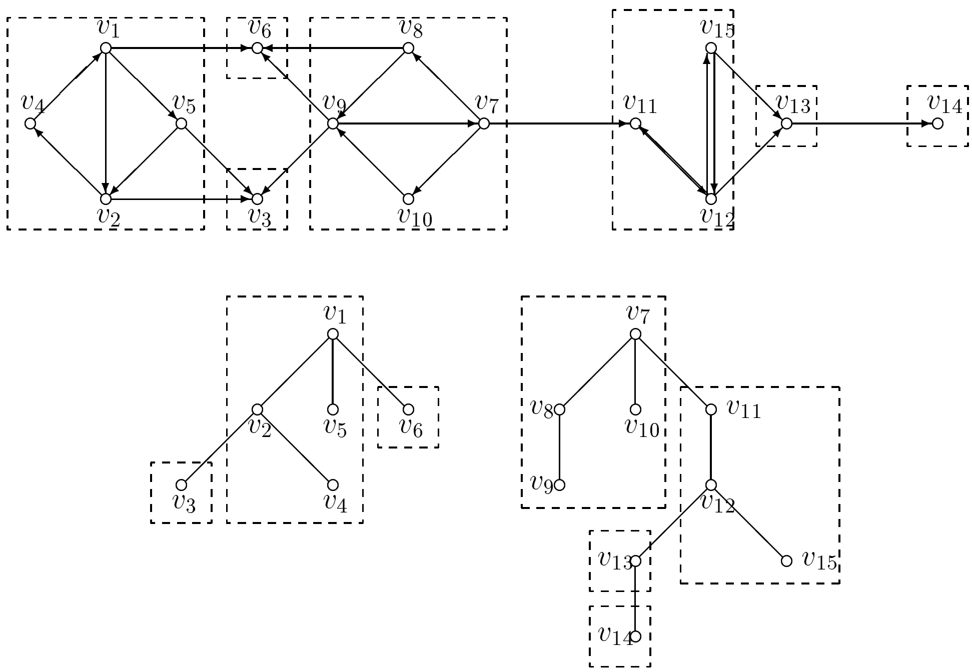

Figure 1: A graph, its strong components, and its depth-first spanning forest.

[1] $O(n + e)$ where n is the number of nodes and e is the number of edges in the input graph

For recognizing the roots a variable called *root* is defined for each node. When a node a is entered in the depth-first search root$[a]$ is set as the depth-first number of a. As we process the edges leaving a node a we change the value of root$[a]$ whenever we detect a path that leads back to some node b that is a predecessor of a in the depth-first spanning forest. Such a path implies a cycle in the graph and we know that the root of all nodes in the cycle is the root of node b. Finally, when exiting a node we know that this node is the root of a whole strong component if and only if its root value is equal to its depth-first number.

The recognition of the roots of the strong components is easy, if we always delete the found component from the graph, i.e., while the search of a component is performed, we never find edges pointing to already found components. Then the situation is always as searching for the first component, and whenever we find a node with a lower root value than the current node we can set this lower value as the root of the current node.

In practice, for efficiency reasons, we should not delete the newly found strong components, but simply mark those nodes that are known to be included in some recognized component. As noted earlier, all strong components form a tree in the depth-first spanning forest. Thus this marking is an easy traversal of all descendants of the found component root that are not already marked. Clearly, this traversal can be implemented by pushing all nodes of a depth-first spanning tree in the order of construction onto a pushdown stack, and whenever a component root is found, by popping (and marking) all nodes up to the root in question. In this way, whenever we find a cross edge, we know that it has to be taken into account if and only if it points to a node not yet marked.

In the present paper we suggest improvements upon the above strategy of Tarjan [15] for determining the strong components. We try to minimize the use of pushdown stack such that the closer the graph is to a dag the less the pushdown stack is needed. We give two algorithms. The first one stores only nodes that are not component roots into the pushdown stack, and the second one stores only nodes that may be roots of nontrivial components (components that contain at least one cycle) into the pushdown stack. In both cases, when the graph is a dag, the pushdown stack is not needed at all.

Purdom [11] and Munro [9] describe other strong component algorithms that are based on the depth-first search. The idea of these algorithms is that when a cycle is detected during the depth-first traversal it is replaced by a single node. The extra work required by this replacement makes these algorithms inferior to Tarjan's algorithm. The worst case running times of Purdom's and Munro's algorithms are $O(n^2)$ and $O(e + n \log n)$, respectively. Purdom and Munro apply their strong component algorithms to transitive closure computation.

Besides Tarjan's algorithm, there is another linear time algorithm that has been presented in many textbooks [2, 4, 6, 16], e.g. However, this algorithm, attributed in [2] to R.Kosaraju and published in [13], requires one depth-first traversal of the given graph and another traversal of the reversed graph, i.e., the graph obtained by reversing the edges of the original graph. Reversing a graph can be done in linear time, but is quite time consuming for large graphs.

2 Tarjan's Algorithm

Let $G = (A, R)$ be a directed graph, where A is the set of nodes and R is the set of edges. By the *strongly connected component* of a node a in A, denoted $SCC(a)$, we mean the equivalence class of a under the equivalence relation $R^* \cap (R^{-1})^*$, where R^* denotes the reflexive transitive closure of R and R^{-1} denotes the inverse of R. Thus $SCC(a) = SCC(b)$ if and only if aR^*b and bR^*a.

In finding the strongly components of (A, R) we make use of the algorithm of Figure 2.

```
procedure TRAVERSE(node a);
begin
   mark a visited;
   for each edge (a,b) in R do
      if b is not visited then
         TRAVERSE(b)
end.

begin /* Main program */
   unmark all nodes in A;
   for each node a in A do
     if a is not visited then
       TRAVERSE(a)
end.
```

Figure 2: Depth-first traversal of a graph (A, R).

During the execution of the algorithm, procedure TRAVERSE enters and exits every node once. The order in which the nodes are entered and exited can be represented as a permutation of the sequence of the $2|A|$ symbols $\mathbf{enter}(a_1)$, $\mathbf{exit}(a_1)$, $\mathbf{enter}(a_2)$, $\mathbf{exit}(a_2)$, $\ldots$, $\mathbf{enter}(a_n)$, $\mathbf{exit}(a_n)$, where $a_i \in A$. We call such a permutation the *depth-first traversal* of (A, R) *induced by* the algorithm. Note that there are in general many depth-first traversals of a graph: the procedure TRAVERSE can fix arbitrarily the order in which it handles the edges leaving a node.

Let (A, R) be a graph and π its depth-first traversal. The *depth-first traversal of node* $a \in A$, denoted $\pi(a)$, is a subsequence of π that begins with $\mathbf{enter}(a)$ and ends with $\mathbf{exit}(a)$. The traversal of a *contains* the traversal of b, denoted $a \prec\!\!\prec_\pi b$, if $\pi(b)$ is a proper subsequence of $\pi(a)$. Note that the traversals are always properly nested, i.e., if $a \neq b$ then either $a \prec\!\!\prec_\pi b$, or $b \prec\!\!\prec_\pi a$, or $\pi(a)$ and $\pi(b)$ do not overlap. The traversal of a *immediately contains* the traversal of b, denoted $a \prec_\pi b$, if $a \prec\!\!\prec_\pi b$ and there is no c such that $c \prec\!\!\prec_\pi b$ and $a \prec\!\!\prec_\pi c$. The *depth-first spanning forest with respect to* π is the graph $(A, \prec_\pi)$. By the corresponding *depth-first order of* (A, R) *with respect to* π we mean the total order $\leq_\pi$ on A in which $a <_\pi b$ if and only if **enter**(a) appears in π before **enter**(b), or equivalently if and only if a is marked visited before b.

We first present a simple algorithm for recognizing the minimum elements, i.e., the *roots* of the strong components with respect to a depth-first order $\leq_\pi$. The algorithm is based on the idea that we always delete a component when it has completely been detected. Thus the situation is always the same as when searching for the first component. A variable called *root* is defined for each node, and when a node a is entered in the depth-first search we set $\text{root}[a] := a$. Once a back edge (a, b), i.e., an edge that points to an already entered but not yet exited node b, is encountered, we conclude that there is a cycle and all nodes in the cycle have the same root as b. Moreover, assuming that the search of the first root is under way, an encountered cross edge (a, b) implies that all entered but not yet exited nodes have root $\leq_\pi \text{root}[b]$. (A cross edge (a, b) points to an exited node b smaller than a). Thus it is clear that the algorithm given in Figure 3 correctly determines the roots of all strong components, because we always delete the already found components from the graph.

```
procedure SCC(node a);
begin
   mark a visited;
   root[a] := a;
   for each edge (a,b) in R do begin
     if b is not visited then
       SCC(b);
     root[a] := min{root[a],root[b]}
   end;
   if root[a] = a then
      remove all nodes b that are descendants of a
      (including a) and all edges leaving or entering
      these nodes and create a new strongly connected
      component containing these nodes
end.

begin /* Main Program */
   unmark all nodes in A;
   initialize STACK to empty;
   for each node a in A do
     if a is not visited then
       SCC(a)
end.
```

Figure 3: Procedure SCC.

Notice that the deletion of nodes and edges in the algorithm of Figure 3 has been done correctly because the nodes in each component form a tree in the spanning forest $(A, \prec_\pi)$. As the result, a node a is a strong component root if and only if root$[a] = a$, and $SCC(a)$ is the set of nodes that are descendants of a in $(A, \prec_\pi)$ but not descendants of any other component root a' that is a descendant of a.

It is clear that we need not explicitly delete the recognized strong components from the graph but we may simply mark them as already found. Moreover, because of the tree structure of the strong components in the spanning forest, we may push nodes onto a stack when entering them in the search and pop the whole newly found strong component whenever a root is found. This implementation of procedure SCC [15], called TSCC, is given in Figure 4. See the appendix for example runs of TSCC and the other algorithms.

```
(1)     procedure TSCC(node a);
(2)     begin
(3)        mark a visited;
(4)        root[a] := a;
(5)        push a onto STACK;
(6)        for each edge (a,b) in R do begin
(7)          if b is not visited then begin
(8)            TSCC(b);
(9)            root[a] := min{root[a],root[b]}
(10)         end else if b is not an element of a found component then
(11)           root[a] := min{root[a],root[b]}
(12)       end;
(13)       if root[a] = a then begin
(14)         start a new strongly connected component S;
(15)         while (top(STACK) >= a) begin
(16)           b := pop STACK;
(17)           mark b as an element of a found component;
(18)           insert b into S
(19)         end
(20)       end
(21)    end.
```

Figure 4: Procedure TSCC recognizes the strong components by using a stack of nodes.

Note that the depth-first order $\leq_\pi$ (that is used at lines (9) and (11) of procedure TSCC for computing the minimum root value) can be implemented by defining a variable *dfsvalue* for every node and using a global counter *dfscounter*. Dfsvalue can also be used for marking the nodes visited and as elements of a found component. In the main program we should then assign a unique value, e.g., 0, to dfscounter and dfsvalue$[a]$ of every node a. Node a is marked visited by incrementing dfscounter and assigning its value to dfsvalue$[a]$. Node is marked as an element of a found component by assigning another unique value, e.g., $n + 1$, where n is the number of nodes in the graph, to dfsvalue$[a]$. This kind of implementation is presented, for example, in [12].

Example 1. Consider the graph in Figure 1. In Figure 5 we present the trace of one possible depth-first traversal of Tarjan's algorithm on that graph. In this case, the main program processes the nodes in order v_1, v_2, ... , v_{15}. Procedure TSCC processes the edges leaving the node in the order indicated by the adjacency lists represented in Figure 5. The picture of the graph contains the final root values for every node (separated by a colon from the name of the node) and the root values that are propagated through the edges during the traversal (next to the edges). The solid line edges are in $\prec_\pi$. They represent recursive calls of procedure TSCC. The dotted line edges point to nodes the processing of which has not completed yet. Procedure TSCC uses these to get new, possibly smaller, root values. The crossed edges point to nodes that are in already detected components; TSCC ignores these edges. $\Box$

Given a graph $G = (V, E)$ and the set of its strong components C, consider the graph $G_{scc} = (C, E')$ where $E' = \{(c_1, c_2) | c_1 \neq c_2 \wedge \exists x \in c_1 \wedge \exists y \in c_2 \wedge (x, y) \in E\}$. G_{scc} is a dag. A *topological order* $<$ of dag $D = (A, R)$ is any total order on A where $(a, b) \in R$ implies $a < b$. A *reverse topological order* of dag $D = (A, R)$ is any total order on A where $(a, b) \in R$ implies $b < a$. The following fact is useful in some applications of strong component detection.

Fact 1 *Tarjan's algorithm given in Figure 4 detects the strong components of a graph G in a reverse topological order of graph G_{scc}.*

3 The Improved Algorithms

Assume that the graph (A, R) is a tree or a dag. Then every strong component consists of one node only, and there is no need for the second traversal that identifies the whole newly found component. Thus the use of a pushdown stack is unnecessary and we would like to avoid it. Also cyclic graphs may contain such trivial components the processing of which does not require a pushdown stack.

Remember that TSCC pushes nodes onto the stack so that when the whole component has been identified the nodes can be marked as elements of that component. Even if we do not explicitely need to construct the components the nodes have to be marked so that TSCC can distinguish between intercomponent and intracomponent (cross-)edges.

We have developed two new algorithms that do not use the pushdown stack in the case of a tree or a dag. Our first algorithm NEWSCC1 is based on the following simple observation: A new strong component is detected when processing its root node. Thus, pushing the root node onto the stack is unnecessary. NEWSCC1 pushes a node onto the stack if and only if it is not a root of a strong component. When a graph is a tree or a dag all its nodes are root nodes and therefore the stack is not used. Only small modifications are needed to the original algorithm. First, node a is not pushed onto the stack in the beginning of NEWSCC1. It is pushed first if $root[a] <_\pi a$ when all edges (a, b) have been processed. Second, the processing of a

Node	Adjacency list	Node	Adjacency list
v_1	(v_2, v_5, v_6)	v_9	(v_6, v_7, v_3)
v_2	(v_3, v_4)	v_{10}	(v_9)
v_3	$()$	v_{11}	(v_{12})
v_4	(v_1)	v_{12}	(v_{13}, v_{15}, v_{11})
v_5	(v_3, v_2)	v_{13}	(v_{14})
v_6	$()$	v_{14}	$()$
v_7	(v_8, v_{10}, v_{11})	v_{15}	(v_{12}, v_{13})
v_8	(v_9, v_6)		

Traversal	Stack	Traversal	Stack
enter(v_1)	$()$	**enter**(v_7)	$()$
enter(v_2)	(1)	**enter**(v_8)	(7)
enter(v_3)	$(2, 1)$	**enter**(v_9)	$(8, 7)$
exit(v_3)	$(2, 1)$	**exit**(v_9)	$(9, 8, 7)$
enter(v_4)	$(2, 1)$	**exit**(v_8)	$(9, 8, 7)$
exit(v_4)	$(4, 2, 1)$	**enter**(v_{10})	$(9, 8, 7)$
exit(v_2)	$(4, 2, 1)$	**exit**(v_{10})	$(10, 9, 8, 7)$
enter(v_5)	$(4, 2, 1)$	**enter**(v_{11})	$(10, 9, 8, 7)$
exit(v_5)	$(5, 4, 2, 1)$	**enter**(v_{12})	$(11, 10, 9, 8, 7)$
enter(v_6)	$(5, 4, 2, 1)$	**enter**(v_{13})	$(12, 11, 10, 9, 8, 7)$
exit(v_6)	$(5, 4, 2, 1)$	**enter**(v_{14})	$(13, 12, 11, 10, 9, 8, 7)$
exit(v_1)	$()$	**exit**(v_{14})	$(13, 12, 11, 10, 9, 8, 7)$
		exit(v_{13})	$(12, 11, 10, 9, 8, 7)$
		enter(v_{15})	$(12, 11, 10, 9, 8, 7)$
		exit(v_{15})	$(15, 12, 11, 10, 9, 8, 7)$
		exit(v_{12})	$(15, 12, 11, 10, 9, 8, 7)$
		exit(v_{11})	$(10, 9, 8, 7)$
		exit(v_7)	$()$

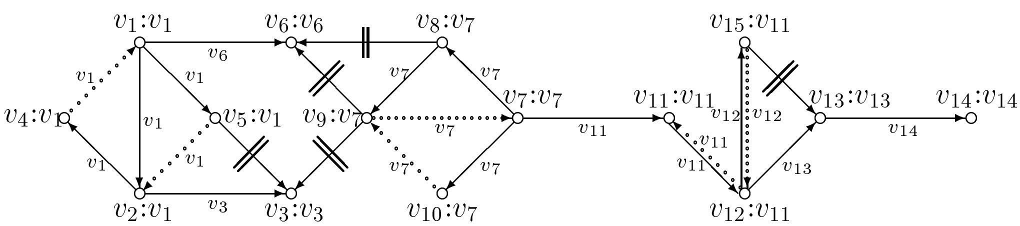

Figure 5: Procedure TSCC applied to the graph of Figure 1

newly found strong component is slightly different, because the root node is not on the stack. The algorithm is given in Figure 6.

```
procedure NEWSCC1(node a);
begin
   mark a visited;
   root[a] := a;
   for each edge (a,b) in R do begin
     if b is not visited then begin
       NEWSCC1(b);
       root[a] := min{root[a],root[b]}
     end else if b is not an element of a found component then
       root[a] := min{root[a],root[b]}
   end;
   if root[a] = a then begin
     start a new strongly connected component S;
     insert a into S;
     while (top(STACK) > a) do begin
       b := pop STACK;
       mark b as an element of a found component;
       insert b into S
     end
   end else
     push a onto STACK
end.
```

Figure 6: Procedure NEWSCC1 pushes a node onto the stack only if it is not a root node.

Assuming that TSCC correctly computes the strong components, it is easy to prove the correctness of NEWSCC1. The computation of the root values and the detection of the root node of a component is similar in both algorithms. If TSCC detects that a is the root of a new strong component then all nodes on the stack with root value $\geq_\pi a$ (including a itself) are popped off the stack and included in the new component. If NEWSCC1 detects that a is the root of a new strong component then a is included in the new component and all nodes on stack with root value $>_\pi a$ are popped off the stack and included in the new component. Thus, the only thing we have to show is that NEWSCC1 pushes onto the stack all the same nodes as TSCC, except the root nodes. But this is trivial, since the root value of every non-root node n must be less than n when all edges starting from n have been processed. Thus, every non-root node is pushed onto the stack in the end of NEWSCC1.

Example 2. Consider again the graph in Figure 1. Our new algorithm NEWSCC1 visits the nodes in the same order as Tarjan's algorithm TSCC and assigns the same root values. However, TSCC pushed all fifteen nodes onto the stack. NEWSCC1 pushes only eight nodes onto the stack. Nodes v_1, v_3, v_6, v_7, v_{11}, v_{13}, and v_{14} are not pushed because they are component roots. The trace and the contents of the stack are shown in Figure 7. □

Traversal	Stack	Traversal	Stack
enter(v_1)	$()$	**enter**(v_7)	$()$
enter(v_2)	$()$	**enter**(v_8)	$()$
enter(v_3)	$()$	**enter**(v_9)	$()$
exit(v_3)	$()$	**exit**(v_9)	(9)
enter(v_4)	$()$	**exit**(v_8)	$(8,9)$
exit(v_4)	(4)	**enter**(v_{10})	$(8,9)$
exit(v_2)	$(2,4)$	**exit**(v_{10})	$(10,8,9)$
enter(v_5)	$(2,4)$	**enter**(v_{11})	$(10,8,9)$
exit(v_5)	$(5,2,4)$	**enter**(v_{12})	$(10,8,9)$
enter(v_6)	$(5,2,4)$	**enter**(v_{13})	$(10,8,9)$
exit(v_6)	$(5,2,4)$	**enter**(v_{14})	$(10,8,9)$
exit(v_1)	$()$	**exit**(v_{14})	$(10,8,9)$
		exit(v_{13})	$(10,8,9)$
		enter(v_{15})	$(10,8,9)$
		exit(v_{15})	$(15,10,8,9)$
		exit(v_{12})	$(12,15,10,8,9)$
		exit(v_{11})	$(10,8,9)$
		exit(v_7)	$()$

Figure 7: The trace of procedure NEWSCC1 applied to the graph of Figure 1.

Our second algorithm called NEWSCC2 uses a complementary approach. The algorithm is presented in Figure 8. Here we push onto the stack only so called *candidate root nodes*. They are nodes that may be roots of non-trivial components, i.e., those nodes that are entered by back edges. Whenever a cross edge (a, b) is encountered we test (at line (9) of NEWSCC2) whether or not root$[b]$ is on the stack. If it is, we know that a and b are in the same component; otherwise b is in some previously found component.

Note the difference between the test at line (10) of TSCC and the test at line (9) of NEWSCC2. In TSCC we test whether or not b itself is on the stack. Both these tests always yield the same result, but the test in NEWSCC2 has the additional benefit that we only have to push candidate root nodes onto the stack and later mark them as elements of found components.

Whenever a real root is found, i.e., root$[a] = a$ for a node a, we pop nodes from the stack until the top node is smaller than the root a and mark these nodes as elements of a found component.

Trees and dags have only trivial components, thus there are no nodes to be pushed onto the stack. The complete algorithm is presented in Figure 8.

The algorithm given in Figure 8 does not explicitly output the nodes in the components. Only the nodes that are popped off the stack (as well as the root node a) are marked as elements of a found component. This is sufficient, because the

```
 procedure NEWSCC2(node a);
 begin
    mark a visited;
    root[a] := a;
    for each edge (a,b) in R do begin
      if b is not visited then begin
        NEWSCC2(b)
        root[a] := min{root[a], root[b]}
      end else if root[b] is not an element of a found component then
        root[a] := min{root[a], root[b]}
    end;
    if root[a] = a then
      if top(STACK) >= a then
        repeat
          mark top(STACK) as an element of a found component;
          pop STACK
        until top(STACK) < a
      else
        mark a as an element of a found component
   else if root[a] is not on STACK then
      push root[a] onto STACK
end.

begin /* Main program */
   unmark all nodes in A;
   initialize STACK to contain a node value < any graph node;
   for each node a in A do
     if a is not visited then
       NEWSCC2(a)
end.
```

Figure 8: The procedure NEWSCC2.

algorithm always tests if root[b] (instead of b) is in an already found strong component. However, the nodes in the component can directly be obtained from $(A, \prec_\pi)$ as descendants of roots. In addition, we may easily create a list of nodes for each candidate root appearing in the stack such that we obtain the nodes in the component when popping root values from the stack.

Example 3. When applied to the graph in Figure 1 NEWSCC2 pushes only four nodes v_1, v_7, v_{11}, and v_{12} onto the stack. The trace and the contents of the stack are shown in Figure 9. □

Traversal	Stack	Traversal	Stack
enter(v_1)	()	**enter**(v_7)	()
enter(v_2)	()	**enter**(v_8)	()
enter(v_3)	()	**enter**(v_9)	()
exit(v_3)	()	**exit**(v_9)	(7)
enter(v_4)	()	**exit**(v_8)	(7)
exit(v_4)	(1)	**enter**(v_{10})	(7)
exit(v_2)	(1)	**exit**(v_{10})	(7)
enter(v_5)	(1)	**enter**(v_{11})	(7)
exit(v_5)	(1)	**enter**(v_{12})	(7)
enter(v_6)	(1)	**enter**(v_{13})	(7)
exit(v_6)	(1)	**enter**(v_{14})	(7)
exit(v_1)	()	**exit**(v_{14})	(7)
		exit(v_{13})	(7)
		enter(v_{15})	(7)
		exit(v_{15})	$(12, 7)$
		exit(v_{12})	$(11, 12, 7)$
		exit(v_{11})	(7)
		exit(v_7)	()

Figure 9: The trace of procedure NEWSCC2 applied to the graph of Figure 1.

4 Analysis

The time complexity of both the original Tarjan's algorithm and the modified algorithms is of the same order of magnitude: $O(n+e)$, where n is the number of nodes and e is the number of edges in the input graph. This is easy to see: all algorithms visit each node once and scan every edge leaving the node once. The loop that pop the stack (the `while` loop at line (15) in TSCC and in NEWSCC1 and the `repeat` loop at line (14) in NEWSCC2) is executed at most n times. Thus, the run times of the algorithms differ from each other only by constant factors.

The only difference between algorithms TSCC and NEWSCC1 is in the number of push and pop operations that the algorithms do. Let n and e be, respectively, the number of nodes and edges in the input graph and let $s = |C|$ be the number of strong components in the input graph. The run time of TSCC is

$$T_{\text{TSCC}}(n, e, s) = \alpha n + \beta e + \gamma s$$

where α, β, and γ are constants that depend on the actual implementation of the algorithm. Algorithm NEWSCC1 pushes a node onto the stack unless it is the root of a strong component. The number P_{NEWSCC1} of nodes pushed by NEWSCC1 is therefore

$$P_{\text{NEWSCC1}} = n - s$$

If $s = n$, i.e. the input graph is a dag, then NEWSCC1 does no stack operations.

Thus, the run time of NEWSCC1 is

$$T_{\text{NEWSCC1}}(n, e, s) = T_{\text{TSCC}}(n, e, s) - \delta s$$

where δ is a constant that corresponds the stack operations (one pop and one push operation plus a test) required by one node.

The run time of algorithm NEWSCC2 differs from the run times of TSCC and NEWSCC1 in two ways. First, NEWSCC2 tests if root$[b]$ is in an already found component, whereas TSCC and NEWSCC1 test if b is in an already found component. This additional indirection may slightly increase the run time of NEWSCC2. Second, NEWSCC2 pushes only some candidate root nodes onto the stack, whereas TSCC pushes all nodes and NEWSCC1 pushes all nodes that are not component roots. The number p_c of nodes pushed (and popped) by NEWSCC2 when processing a strong component c is

$$\begin{array}{ll} p_c = 0, & |c| = 1 \\ 0 < p_c < |c|, & |c| > 1 \end{array}$$

The total number P_{NEWSCC2} of nodes pushed (and popped) by NEWSCC2 is

$$P_{\text{NEWSCC2}} = \sum_{c \in C} p_c$$

Using the inequality for p_c we get

$$0 \leq P_{\text{NEWSCC2}} \leq n - s = P_{\text{NEWSCC1}}$$

where s is $|C|$ the number of strong components. Thus, NEWSCC2 always pushes at most as many nodes as NEWSCC1.

5 Experiments

To get a more realistic picture of the run times of the algorithms and the number of nodes pushed and popped by NEWSCC2 we implemented the algorithms and run several tests using randomly generated directed graphs as input. Two kinds of graphs were used:

1. General directed random graphs $G(n, p)$ where n is the number of nodes in the graph and $0 \leq p \leq 1$ is the probability, for each edge to exist. If $p = 0$ there are no edges and if $p = 1$ there are n^2 edges. The number of edges leaving a node is binomially distributed with parameters n and p because each edge exists with probability p and there are n possible target nodes. Therefore the graph can be generated by computing for each node i the number e_i of edges leaving node i by using binomial distribution and then randomly choosing e_i distinct target nodes for those edges (see [5]; a fast algorithm for computing binomially distributed random numbers is presented in [10]).

2. Random dags $D(n, p)$. Parameters n and p are similar to those of the general directed random graphs. The generation of a random dag is similar to the generation of a general random graph, but the nodes are ordered and an edge (n_i, n_j) is possible only if $i < j$. Thus, if $p = 0$ there are no edges and if $p = 1$ there are $n(n-1)/2$ edges.

The test program took four parameters: the number of nodes n, the edge probability p, the kind of the graph (general or dag), and a seed for the random number generator. In every test run, the program first generated the random graph using the parameters and then computed the desired output measures by applying all the three algorithms on the same random graph. This so called *common random numbers* technique reduces variance in the different output measures (see [8]) and speeds up the testing. For every experiment, several test runs were made with different seed values for the random number generator in order to get more accurate measures.

To aid the selection of the test parameters we first examined s, the number of strong components in general random graphs and the size of the largest strong component. The results are presented in Figure 10. As we can see, the values in z-axis are nearly independent of n, but highly dependent on np. When np, the expected number of edges leaving a node, is close to 1, s is close to n, i.e., every node is in its own, single node strong component. When np grows, s decreases rapidly, and when np is over 5 there is typically only one strong component. Similarly, the size of the largest component grows rapidly when np grows. Usually, the graph seems to consist of one “giant” component and a set of trivial components[2]. In this model of directed random graphs, the probability of generating a graph that contains two or more big components is very low. Based on these results, we decided to study more closely the input parameter pairs (n, p) for which $0 \leq np \leq 10$.

In the first series of experiments we examined the stack behavior of the different algorithms as a function of the size of the input graph. We only used general (cyclic)

[2]We do not know whether these results have been shown analytically, but similar results has been shown about the number of accessible nodes when searching an undirected or a directed random graph, and about the number and sizes of components in undirected random graphs, see [3, 5]

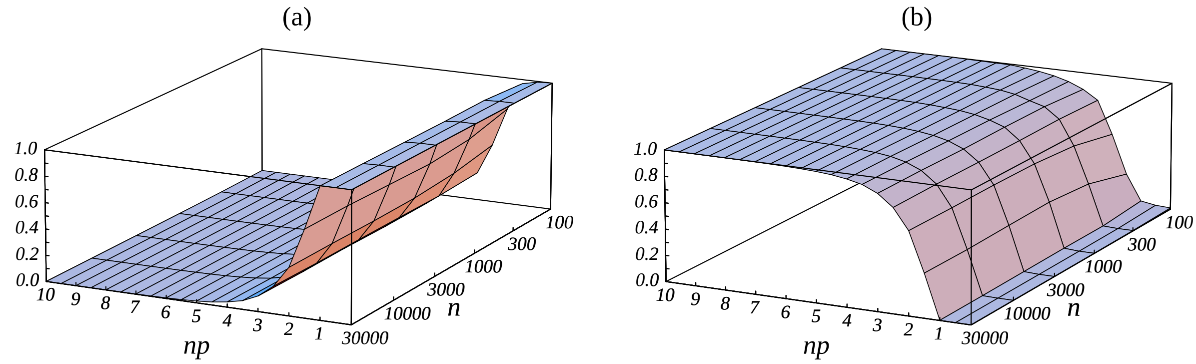

Figure 10: The average number of strong components (a) and the size of the largest component (b) divided by the number of nodes.

random graphs as inputs because neither NEWSCC1 nor NEWSCC2 uses the stack when the input graph is acyclic. For every pair (n, p) of parameters we measured the average number of nodes pushed (which equals the average number of nodes popped) by the different algorithms using ten different seed values. The average pushdown counts divided by n, the number of nodes, are presented in Figure 11. Also here, the values in z-axis depend more on np than on n. NEWSCC2 seems to use much less stack operations and stack space as TSCC and NEWSCC1. NEWSCC1 and TSCC behave similarly when $np > 5$. Remember, however, that when the graph is acyclic NEWSCC1 (and NEWSCC2) does not use the stack at all.

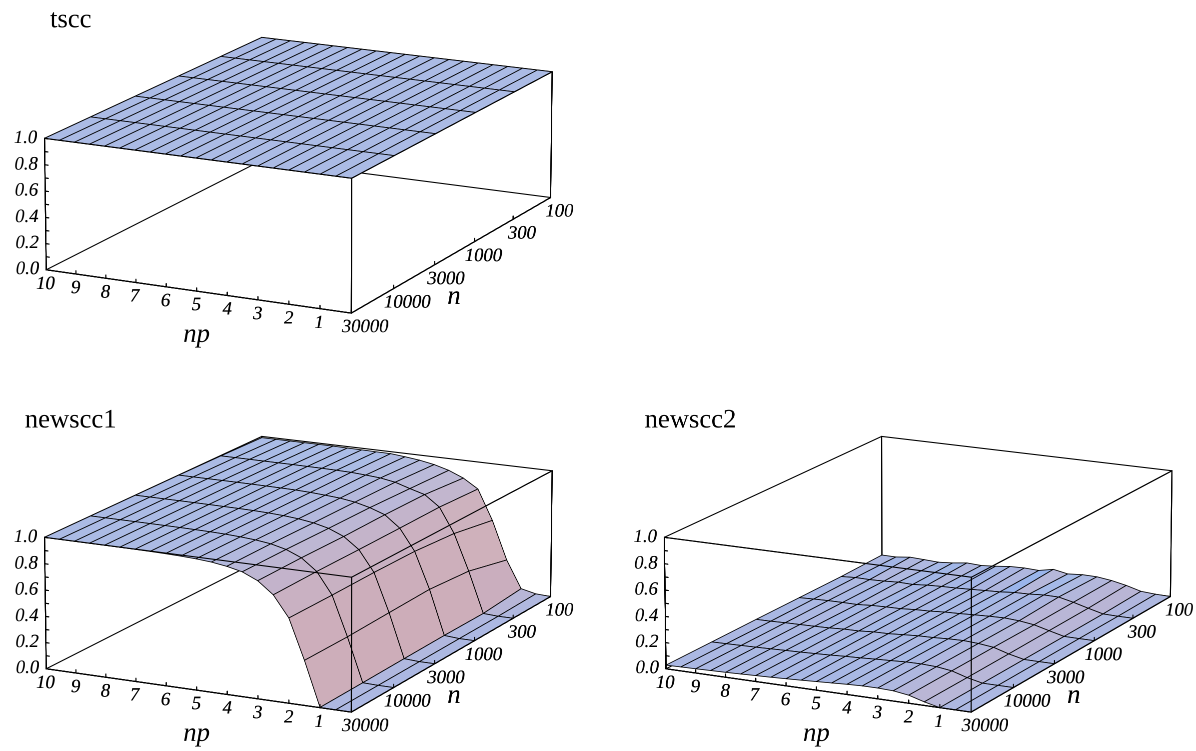

Figure 11: The average pushdown count divided by the number of nodes.

The stack behavior of NEWSCC2 can be explained in the following way: when the graph is very sparse ($np < 1$) most components are trivial. Thus, very few nodes are pushed onto the stack. When np approaches 5 the number of components decreases. In addition to small components, there is usually one giant component, but it does not have very many edges. This increases the probability of pushing false candidate roots, but it seems that less than 10% of nodes are pushed on the average. When np grows over 5 and the graph becomes denser the probability of pushing a false candidate root decreases. When NEWSCC2 processes a dense graph that contains only one strong component it usually pushes only the actual root of the graph.

In the second series of experiments we tested the run times of the different algorithms using general random graphs as inputs. The experiments were executed on a SUN/Sparcstation 2 workstation. To eliminate the effect of other processes running at the same time, every algorithm was applied fifty times to the same random graph and the minimum running time was selected as the result. The run times of NEWSCC1 and NEWSCC2 divided by the run time of TSCC are presented in Figure 12.

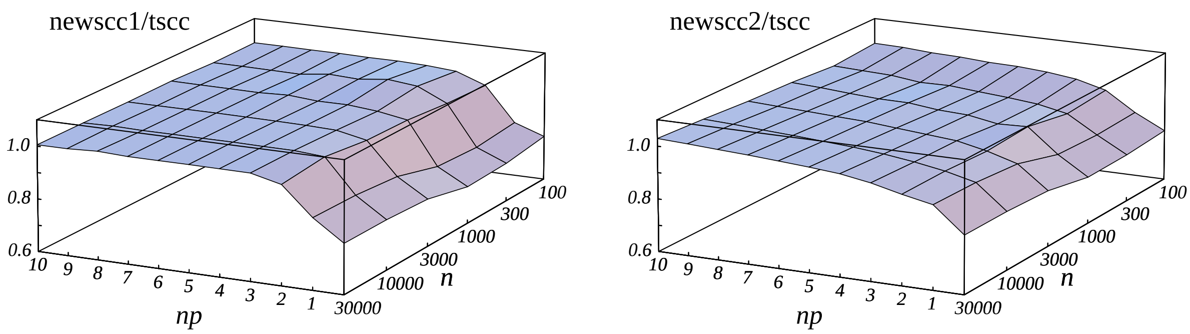

Figure 12: The run time of NEWSCC1 and NEWSCC2 divided by the run time of TSCC. General graphs are used as input.

Also these measures depend much more on np than on n. When $p = 0$ the run times of NEWSCC1 and NEWSCC2 were up to 26% and 23% smaller, respectively, than the run time of TSCC. When np was over 3 NEWSCC1 and TSCC ran equally fast. When np was about 10 NEWSCC2 was slightly slower (2%) than TSCC and NEWSCC1. This is caused by the extra indirection in line (9) of NEWSCC2 that we could not optimize away. When we used a complete graph of 1000 nodes as input NEWSCC2 was 11% slower than TSCC and NEWSCC1.

The third series of experiments was like the second, but random dags were used as inputs. The results are presented in Figure 13.

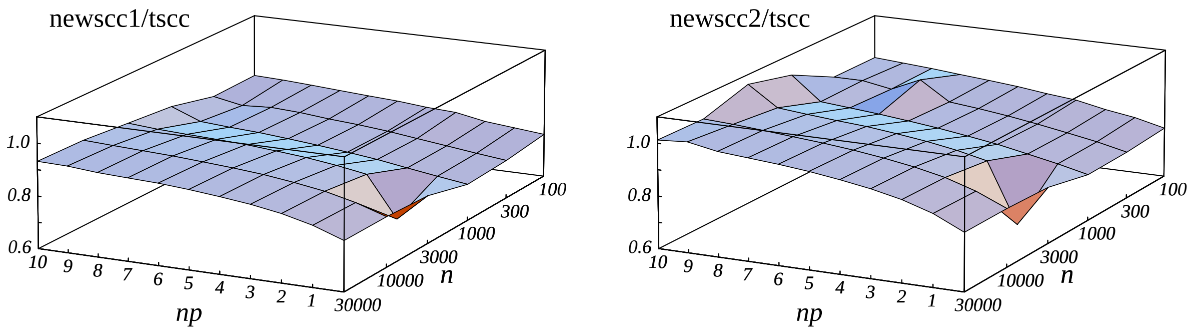

Figure 13: The minimum run time of NEWSCC1 and NEWSCC2 divided by the mimimum run time of TSCC. Dags are used as input.

Also in this case, when $p = 0$ the run times of NEWSCC1 and NEWSCC2 were up to 26% and 23% smaller, respectively, than the run time of TSCC. Unlike with cyclic graphs, NEWSCC1 was about 10% faster than TSCC even for $np = 10$. Also in this case NEWSCC2 was slightly slower than TSCC and NEWSCC1 when $np = 10$.

6 Summary

We have presented two improved versions of Tarjan's algorithm for finding the strong components in a directed graph. The new algorithms use the pushdown stack more economically than the original algorithm. The first new algorithm, called NEWSCC1 differs from the original algorithm by pushing only non-root nodes onto the stack. The second new algorithm, called NEWSCC2, pushes only candidate root nodes onto the stack. Candidate root nodes are nodes that are entered by back-edges. Neither of these new algorithms pushes any nodes onto the stack when the input graph is acyclic. The analysis shows that NEWSCC2 pushes onto the stack at most as many nodes as NEWSCC1 and NEWSCC1 pushes $n - s$ nodes onto the stack where n is the number of nodes in the graph and s is the number of strong components in the graph. The original algorithm pushes all the n nodes of the input graph onto the stack. Our experiments showed that on the average, NEWSCC2 uses much less stack space than NEWSCC1 and the original algorithm. In our test runs NEWSCC1 was up to 26% and NEWSCC2 up to 23% faster than the original algorithm. NEWSCC1 is never slower than the original algorithm, but NEWSCC2 was 11% slower than NEWSCC1 and the original algorithm with a complete graph of 1000 nodes.

References

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

[2] A.V.Aho, J.E.Hopcroft, and J.D.Ullman, *Data Structures and Algorithms*, Addison-Wesley, Reading, Mass., 1983.

[3] B.Bollobás, *Random Graphs*, Academic Press, 1985.

[4] G.Brassard and P.Bratley, *Algorithmics: Theory and Practice*, Prentice-Hall International, Inc., New Jersey, 1988.

[5] S.Kapidakis, Average-Case Analysis of Graphs-Searching Algorithms, PhD Thesis, Report CS-TR-286-90, Princeton University, Department of Computer Science, October 1990.

[6] J.H.Kingston, *Algorithms and Data Structures: Design, Correctness, Analysis*, Addison-Wesley, Reading, Mass., 1990.

[7] U.Manber, *Introduction to Algorithms: A Creative Approach*, Addison-Wesley, Reading, Mass., 1989.

[8] C.McGeoch, Analyzing Algorithms by Simulation, *Computing Surveys* **24:2**, June 1992, 195–212.

[9] I. Munro, Efficient Determination of the Transitive Closure of a Directed Graph. *Information Processing Letters* **1** (1971), 56–58.

[10] W.H.Press, B.P.Flannery, S.A.Teukolsky, and W.T.Vetterling, *Numerical Recipes in C*, Cambridge University Press, Cambridge, Mass., 1988.

[11] P. Purdom Jr., A Transitive Closure Algorithm. *BIT* **10** (1970), 76–94.

[12] R.Sedgewick, *Algorithms, 2nd Edition*, Addison-Wesley, Reading, Mass., 1988.

[13] M.Sharir, A strong-connectivity algorithm and its application in data flow analysis. *Computers and Mathematics with Applications* **7** (1981), 67–72

[14] S.Sippu and E.Soisalon-Soininen, *Parsing Theory. Vol. I: Languages and Parsing*, Springer-Verlag, Berlin Heidelberg, 1988.

[15] R.E.Tarjan, Depth-first search and linear graph algorithms. *SIAM J. Comput.* **1** (1972), 146–160.

[16] M.A.Weiss, *Data Structures and Algorithm Analysis*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, Calif., 1992.