On Finding the Strongly Connected 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 the detection of strongly connected components in a directed graph. Our new algorithms handle sparse graphs and graphs with many trivial components (containing only one node) more economically than Tarjan's original algorithm. As an application we present an efficient transitive closure algorithm.

Keywords: Design of algorithms; strongly connected components; Tarjan's algorithm; transitive closure

1 Introduction

Consider a directed graph $G = (V, E)$, where V is the set of nodes and $E \subseteq V \times V$ is the set of edges. A *path* from node v_0 to node v_k in G is an alternating sequence $(v_0, (v_0, v_1), v_1, (v_1, v_2), \ldots, v_{k-1}, (v_{k-1}, v_k), v_k)$ of nodes and edges that belong to V and E, respectively. Two nodes v and w in G are *path equivalent* if there is a path from v to w and a path from w to v. Path equivalence partitions V into maximal disjoint sets of path equivalent nodes. These sets are called the *strongly connected components* of the graph.

Tarjan [6] has presented an elegant algorithm that finds the strongly connected components in $O(n + e)$ time, where n is the number of nodes and e is the number of edges in the input graph. Although Tarjan's algorithm is asymptotically optimal it does some unnecessary work. In this paper we show how this can be avoided.

Tarjan's algorithm can be considered to contain two interleaved traversals of the graph. First, a depth-first search traverses all edges and constructs a depth-first spanning forest. Second, once a so called *root* of a strongly connected component is found, all its descendants that are not elements of previously found components are marked as elements of this component. This second traversal is implemented by using a stack, where each node is stored when entered by the depth-first search. When a root of a component is exited, all nodes down to the root are removed from the stack and they form the component in question.

We present two improved versions of Tarjan's algorithm. The new algorithms reduce the second traversal without slowing down the first traversal. This is achieved by not storing all nodes on the stack. The first algorithm does not store the component root nodes on the stack. The second algorithm stores only possible root nodes of nontrivial (containing at least one cycle) components on the stack. As an application of the second new algorithm we present a simple and efficient transitive closure algorithm.

For acyclic graphs, our new algorithms completely eliminate the second traversal, and for cyclic graphs the number of nodes to be traversed is reduced. The new algorithms handle sparse graphs and graphs with many single node components more economically than Tarjan's original algorithm.

Besides Tarjan's algorithm, another linear time algorithm is presented in many textbooks. This algorithm is attributed in [1] to R.Kosaraju and published in [5]. It requires one depth-first traversal of the input graph and another traversal of the graph obtained by reversing the edges of the input graph. Reversing a graph can be done in linear time, but is quite time consuming for large graphs.

Recently, Jiang [3] has presented a new linear time strongly connected component algorithm. The algorithm is based on a traversal strategy that is a combination of depth-first and breadth-first traversal. Jiang's algorithm is aimed at reducing disk operations in a situation where the whole graph does not fit into the main memory.

2 Tarjan's algorithm

We review here the basic ideas of Tarjan's algorithm. We use a notation that differs from the original presentation [6], but that simplifies the description.

Tarjan's algorithm is presented in Figure 1. It consists of a recursive procedure *VISIT* and a main program that applies procedure *VISIT* to each node that has not already been visited. Procedure *VISIT* enters the nodes of the graph in depth-first order. For each strongly connected component C, the first node of C that procedure *VISIT* enters is called the *root* of component C. The main goal of the algorithm is to find the component roots. For this purpose, we define a variable $root[v]$ for each node v. When procedure *VISIT* is processing node v, $root[v]$ contains a candidate node for the root of the component containing v.

Initially (at line 3), node v itself is the root candidate. When procedure *VISIT* processes the edges leaving node v (at lines 5-8), new root candidates are obtained from children nodes that belong to the same component as v. The *MIN* operation (at line 7) compares the nodes with respect to the order in which procedure *VISIT* has entered them, i.e., $MIN(x, y) = x$ if procedure *VISIT* entered node x before it entered node y, otherwise $MIN(x, y) = y$. A simple way to implement this is to use an array and a counter to assign a unique depth-first number to each node. When procedure *VISIT* has processed all edges leaving v, $root[v] = v$ if and only if v is the root of the component containing v (line 9). Note however, that if v is not a component root we do not know if $root[v]$ is the right root of the component containing v.

To distinguish between nodes belonging to the same component as node v and nodes belonging to other components, a Boolean variable $InComponent[w]$ is defined for each node w. Its initial value is **False**. When a component C is fully detected procedure *VISIT* sets $InComponent[w] =$ **True** for each node w that belongs to C (lines 10-13). A stack is used for this purpose. Each node is stored on the stack in the beginning of procedure *VISIT*. When the component is fully detected the nodes belonging to it are on top of the stack. Procedure *VISIT* removes them from the stack and sets the *InComponent* values to **True**. The nodes belonging to the newly found component could be output, but this is omitted in Figure 1.

Note that a clever implementation of the algorithm can use a single computer word for each node to represent the depth-first number of the node and its *InComponent* status and to record whether the node is already entered. No explicit Boolean vector is needed to represent the *InComponent* variables.

3 Improved algorithm 1

If the input graph $G = (V, E)$ is acyclic then each strongly connected component consists of a single node. Thus, there is no need for the second traversal that identifies the newly found component and the use of a stack is unnecessary. Cyclic graphs also may contain such trivial components.

Our first new algorithm, presented in Figure 2, does not use the stack when processing trivial components. The algorithm is based on the following property of Tarjan's algorithm: a new strongly connected component is detected when processing its root node. During the second traversal that marks the nodes of the component, we would have access to the root node even if it were not stored on the stack. Therefore, procedure *VISIT1* stores a node v on the stack only after it has processed all edges leaving v and knows that v is not a component root (at line 14). Further, the processing of a newly found component is slightly different, because the root node is not on the stack. Each nonroot node of a component was entered after the component root. Therefore (at lines 10-13), we remove nodes from the stack as long as the topmost node is greater than the root node (with respect to the order in which procedure *VISIT1* entered the nodes) and set the *InComponent* variables **True**. The *InComponent* variable of the root node is set **True** at line 9.

4 Improved algorithm 2

We examine now the possibility to further reduce the second traversal in Tarjan's algorithm. Obviously, if we have to output the components we need an access to each node of the component. But if we only want to detect the component roots, for instance, to compute the number of strong components, we can do better than in Algorithm 1.

Examine line 7 in Figure 1. This is the only place where we test if the child node w belongs to the same component as node v. Note that w belongs to the same component as v if and only if $root[w]$ belongs to the same component as v. Thus, tests "$InComponent[w]$" and "$InComponent[root[w]]$" always yield the same result.

Our second algorithm, presented in Figure 3, is based on this idea. Node x is a *final candidate root* if $x = root[y]$ for some node y when all edges leaving y have been processed. Procedure *VISIT2* stores each final candidate root of a nontrivial component on the stack. This is done at line 15 in Figure 3. Note that a clever implementation can use the same computer word that holds the depth-first number and other status information to record if the node is on the stack. When a nontrivial component is detected its final candidate root nodes are on top of the stack. The algorithm removes nodes from the stack until the topmost node is smaller than the actual root node (in the depth-first ordering) and sets the *InComponent* variables **True**. If the component is trivial the algorithm only sets $InComponent[v] =$ **True** (at line 14). To prevent stack underflow, the stack is initialized to contain a sentinel value smaller than any node of the input graph.

We conjecture that the second traversal cannot be completely removed, at least without changing the first traversal. If we remove the second traversal we can access only the component root node when we detect a new component. Thus, only the *InComponent* variable of the root node can be set **True**. After a nonroot node w has been processed we do not necessarily have a fixed length access path from w to the root of the component containing w. Thus, testing if w belongs to an already detected component cannot be done in constant time, which slows the first traversal.

5 Analysis

The time complexity of Tarjan's algorithm and our new algorithms is $O(n+e)$, where n is the number of nodes and e is the number of edges in the input graph. The first traversal in all algorithms takes $O(n + e)$ time and the second traversal takes $O(n)$ time. Since our new algorithms change only the second traversal, no improvements should be expected when the input graph is dense, i.e., when the number of edges e of the graph is of order n^2. When the graph is sparse, i.e., the number of edges is of order n, our algorithms run faster than the original algorithm. The difference between the actual run times depends on the implementation.

The main difference between the three algorithms is the number of nodes stored on the stack. Let P_T, P_1, and P_2 be the number of nodes stored on the stack in Tarjan's algorithm and in our algorithms 1 and 2, respectively. Tarjan's algorithm stores all n nodes on the stack. Thus, $P_T = n$. Algorithm 1 stores a node on the stack unless it is a component root. Thus, $P_1 = n - s$, where s is the number of strongly connected components in the input graph. Let p_C be the number of nodes stored on the stack by Algorithm 2 when processing a component C. If C is trivial then no nodes are stored on the stack. If C is nontrivial then at least one node is not a final candidate root node and thus not stored on the stack. Thus, $0 \leq p_C \leq |C| - 1$, where $|C|$ is the number of nodes in component C. $P_2 = \sum_{C \in SCC} p_C$, where SCC is the set of all strongly connected components in the input graph. Using the inequality for p_C we get $0 \leq P_2 \leq n - s = P_1 < P_T$. Thus, Tarjan's algorithm always stores more nodes on the stack than Algorithm 1, and Algorithm 2 stores at most as many nodes on the stack as Algorithm 1. According to our experiences, P_2 is almost always much smaller than P_T and P_1.

Note that if the graph does not completely fit into the main memory, the space savings in Algorithms 1 and 2 may reduce the number of disk operations significantly.

6 Transitive closure computation

Algorithm 2 can be used as a basis for a simple and efficient transitive closure algorithm. The main idea of the algorithm is to compute the successor sets (the nodes reachable via a non-null path) only for the final candidate roots.

The algorithm is presented in Figure 4. Consider a strongly connected component C. Each node of C has the same successor set $Succ[C]$. $Succ[C]$ consists of two parts: First, if C is a nontrivial component $Succ[C]$ contains each node of C. The nodes of C are included into $Succ[C]$ by inserting each node into the successor set of its final candidate root (lines 12 and 24). Second, $Succ[C]$ contains the nodes in the components adjacent from C and their successor sets. These are collected during the processing of each node v of component C. When the edges leaving node v are scanned we insert a final candidate root of each component adjacent from v into a local set $Roots[v]$ (line 7). After this, the nodes in $Roots[v]$ and their successor sets are added into the successor set of the final candidate root of node v (line 9).

When component C is fully detected we combine the successor sets of its final candidate roots to get $Succ[C]$ (line 17). $Succ[C]$ is propagated to each final candidate root at line 18. Thus, after the processing of component C the successor set of each node v of component C can be found in $Succ[root[v]]$.

This algorithm has several benefits compared to previous transitive closure algorithms that use strongly connected components. First, the edges leaving a node are scanned only during the depth-first search. They are not needed later, since the *Roots* set contains sufficient information for the successor set computation. Compared to the algorithm by Schmitz [4] this saves much CPU-time, and, if the graph does not fully fit into the main memory, also much disk I/O. Note that the *Roots* set of a node is local: it is not needed after the algorithm exits a node. Second, a (partial) successor set is computed only for each final candidate root, not for all nodes as in [2]. According to our experiences, the set of final candidate root nodes usually contains only a small number of nodes that are not actual component roots. Thus, the union operation at line 9 of procedure *TC* usually inserts the nodes directly to the successor set of the component root.

Acknowledgements

We thank Otto Nurmi and the anonymous referees for many useful comments on the manuscript.

References

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

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

[3] B. Jiang. I/O- and CPU-optimal recognition of strongly connected components. *Information Processing Letters*, 45(3):111–115, March 1993.

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

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

[6] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, June 1972.

```
procedure VISIT(v);
begin
    root[v] := v; InComponent[v] := False;
    PUSH(v, stack);
    for each node w such that (v, w) ∈ E do begin
        if w is not already visited then VISIT(w);
        if not InComponent[w] then root[v] := MIN(root[v], root[w])
    end;
    if root[v] = v then
        repeat
            w := POP(stack);
            InComponent[w] := True;
        until w = v
end;
begin/* Main program */
    stack := ∅;
    for each node v ∈ V do
        if v is not already visited then VISIT(v)
end.
```

Figure 1: Tarjan's algorithm detects the strongly connected components of graph $G = (V, E)$.

```
procedure VISIT1(v);
begin
    root[v] := v; InComponent[v] := False;
    for each node w such that (v, w) ∈ E do begin
        if w is not already visited then VISIT1(w);
        if not InComponent[w] then root[v] := MIN(root[v], root[w])
    end;
    if root[v] = v then begin
        InComponent[v] := True;
        while TOP(stack) > v do begin
            w := POP(stack);
            InComponent[w] := True;
        end
    end else PUSH(v, stack);
end;
begin/* Main program */
    stack := ∅;
    for each node v ∈ V do
        if v is not already visited then VISIT1(v)
end.
```

Figure 2: Algorithm 1 stores only nonroot nodes on the stack.

```
procedure VISIT2(v);
begin
    root[v] := v; InComponent[v] := False;
    for each node w such that (v, w) ∈ E do begin
        if w is not already visited then VISIT2(w);
        if not InComponent[root[w]] then root[v] := MIN(root[v], root[w])
    end;
    if root[v] = v then
        if TOP(stack) ≥ v then
            repeat
                w := POP(stack);
                InComponent[w] := True;
            until TOP(stack) < v;
        else InComponent[v] := True;
    else if root[v] is not on stack then PUSH(root[v], stack);
end;
begin/* Main program */
    Initialize stack to contain a value < any node in V;
    for each node v ∈ V do
        if v is not already visited then VISIT2(v)
end.
```

Figure 3: Algorithm 2 stores only final candidate root nodes on the stack.

```
procedure TC(v);
begin
    root[v] := v; InComponent[v] := False;
    for each node w such that (v, w) ∈ E do begin
        if w is not already visited then TC(w);
        if not InComponent[root[w]] then root[v] := MIN(root[v], root[w])
        else Roots[v] :=Roots[v] ∪ {root[w]}
    end;
    Succ[root[v]] :=Succ[root[v]] ∪ ⋃_{r∈Roots[v]} ({r} ∪ Succ[r]);
    if root[v] = v then
        if TOP(stack) ≥ v then begin
            Succ[v] :=Succ[v] ∪ {v};
            repeat
                w := POP(stack);
                InComponent[w] := True;
                if v ≠ w then begin
                    Succ[v] :=Succ[v] ∪ Succ[w];
                    Succ[w] :=Succ[v]    /* Pointer assignment, not a copy */
                end
            until TOP(stack) < v;
        end else InComponent[v] := True;
    else begin
        if root[v] is not on stack then PUSH(root[v], stack);
        Succ[root[v]] :=Succ[root[v]] ∪ {v}
    end
end;
begin/* Main program */
    Initialize stack to contain a value < any node in V;
    for each node v ∈ V do
        if v is not already visited then TC(v)
end.
```

Figure 4: A transitive closure algorithm based on Algorithm 2.