Depth First Search and Strong Components

CMU 15-451 Spring 2003 D. Sleator

1. Introduction

Depth first search is a very useful technique for analyzing graphs. For example, it can be used to:

- Determine the connected components of a graph.
- Find cycles in a directed or undirected graph.
- Find the biconnected components of an undirected graph.
- Topologically sort a directed graph.
- Determine if a graph is planar, and finding an embedding of it if it is.
- Find the strong components of a directed graph.

If the graph has n vertices and m edges then depth first search can be used to solve all of these problems in time $O(n+m)$, that is, linear in the size of the graph.

2. Depth First Search in Directed Graphs

We assume that the graph is represented as an adjacency structure, that is, for every vertex v there is a set $adj(v)$ which is the set of vertices reachable by following one edge out of v. Let V be the set of vertices in the graph, and let E be the set of edges. To do a depth first search we keep two pieces of information associated with each vertex v. One is a the depth first search numbering, $num(v)$, and the other is $mark(v)$, which indicates that v is currently on the recursion stack.

Here is the depth first search procedure:

```
$i \leftarrow 0$
for all $x \in V$ do $num(v) \leftarrow 0$
for all $x \in V$ do $mark(v) \leftarrow 0$
for all $x \in V$ do
    if $num(x) = 0$ then DFS($x$)

DFS($v$)
    $i \leftarrow i + 1$
    $num(v) \leftarrow i$
    $mark(v) \leftarrow 1$
    for all $w \in adj(v)$ do
        if $num(w) = 0$ then DFS($w$)          [$(v, w)$ is a tree edge ]
        else if $num(w) > num(v)$ then         [$(v, w)$ is a forward edge ]
        else if $mark(w) = 0$ then             [$(v, w)$ is a cross edge]
        else                                   [$(v, w)$ is a back edge]
    $mark(v) \leftarrow 0$
end DFS
```

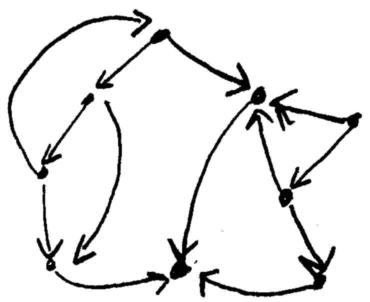
This process examines all edges and vertices. The call DFS(v) is made exactly once for each vertex of the graph. Each edge is placed into exactly one of four classes by the algorithm: tree edges, forward edges, cross edges, and back edges.

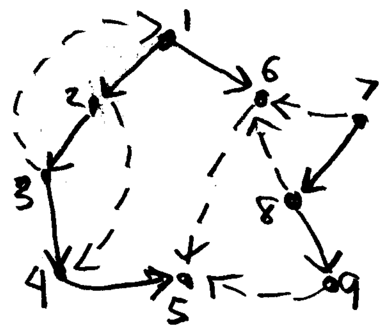
This classification of the edges is not a property of the graph alone. It also depends on the ordering of the vertices in $adj(v)$ and on the ordering of the vertices in the loop that calls the DFS procedure. The *num* and *mark* fields are not actually necessary to accomplish a complete search of the graph. All that is needed to do that is a single bit for each vertex that indicates whether or not that vertex has already been searched. (This bit is zero for vertex v if and only if $num(v) = 0$). The depth first labeling (the *num* field) has some very useful properties that we shall make use of.

The tree edges have the property that either zero or one of them points to a given vertex. Therefore, they define a collection of trees, called the *depth-first spanning forest* of the graph. The root of each tree is the lowest numbered vertex in it (the one that was searched first). These rooted trees allow us to define the ancestor and descendant relations among vertices. The four types of edges are related to the spanning forest as follows:

- The forward edges are edges from a vertex to a descendant of it that are not tree edges. This is because the test $num(w) > num(v)$ indicates that w was explored after the call to DFS(v). Since the call to DFS(v) is not yet complete w must be a descendant of v.
- The cross edges are edges from a vertex v to a vertex w such that the subtrees rooted at v and w are disjoint. This follows because $mark(w) = 0$ so the exploration of w is complete, and was complete before the call to DFS(v). Therefore v is not in a subtree rooted at w. Vertex w is not in a subtree rooted at v because $num(w) < num(v)$.
- The back edges are edges from a vertex to an ancestor of it. The fact that $mark(w) = 1$ indicates that the w is on the recursion stack and is thus an ancestor of v.

Below is an example of a graph and a corresponding depth first spanning forest.





We shall repeatedly make use of one very general property of the depth-first numbering of the graph. This is embodied in the following lemma.

Lemma 1 *Let T_v be the subtree of the spanning forest rooted at v, and define T_w similarly. Suppose that T_v and T_w are disjoint. Then all the depth-first numbers in T_v are greater than all of those in T_w or all the depth-first numbers in T_v are less than all of those in T_w. Furthermore, if there is an edge from a vertex in T_v to a vertex in T_w then all of the depth first search numbers in T_w are less than all of them in T_v.*

Proof. The first part follows as an immediate consequence of the fact that the depth first search algorithm always finishes searching a subtree of a vertex before going on to any other disjoint subtree. This also proves the second part, because the only way there could be an edge from T_v to T_w, and for these two subtrees to be disjoint is if T_w had been completely searched before vertex v was searched, so T_w has lower numbered vertices than T_v. $\square$

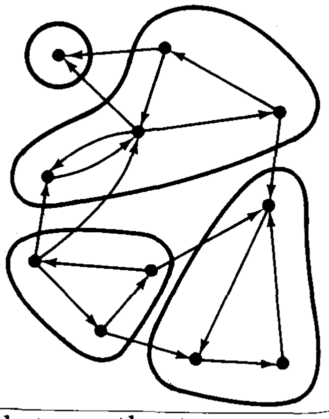
3. Strong Components

Two vertices v and w are *equivalent*, denoted $v \equiv w$ if there exists a path from v to w and one from w to v. The relation "$\equiv$" so defined is an equivalence relation because it satisfies the following three properties:

Reflexivity:	$w \equiv w$. This is because a path of length zero goes from w to w and vice versa.
Symmetry:	If $v \equiv w$ then $w \equiv v$. This follows immediately from the definition.
Transitivity:	If $v \equiv w$ and $w \equiv x$ then $v \equiv x$. If there is a path from v to w, and one from w to x, then there is a path from v to x. The same reasoning shows that there is a path from x to v.

This equivalence relation induces a partitioning of the vertices of the graph into components in which each pair of vertices in a component are equivalent. These are called the ***strongly connected components*** or *strong components* of the graph. Our goal is to devise an algorithm which will compute the strong components of a graph.

Here is an example of a directed graph partitioned into its strong components.



There is a very close relationship between the strong components of a graph and the depth first spanning forest of the graph. We shall present an algorithm that uses depth first search to find the strong components of the graph. The algorithm augments slightly the depth first search described above. These changes do not effect the depth-first spanning forest or the classification of edges described above.

To simplify the proof of correctness of the algorithm, we assume that there is a vertex R in the graph from which there are edges to each other vertex. (If there is no such vertex then we can add one. The other strong components remain unchanged by the addition of R and its edges.) This vertex is in its own strong component, since there is no path to it from any other vertex. Let the depth first search begin from R. Now there is only one tree in the depth first spanning forest. (The algorithm we show later does not assume the existence of R. In fact the algorithm as stated works correctly without this assumption.)

Definition*A vertex is called a base of a strong component if it has the lowest depth-first search number (num field) of any vertex in the strong component.*

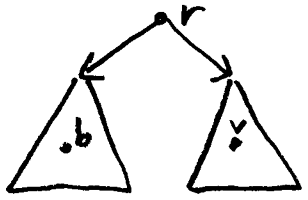
Lemma 2 *Let b be the base of a strong component X. Then for all $v \in X$, v is a descendant of b, and all vertices on the path from b to v are in X.*

Proof. First we prove the first part. Let v be any vertex besides b in X. We know that either (1) v descends from b, or (2) b descends from v, or (3) neither of the above. (2) is impossible, because if b descended from v, then its depth-first number would be greater than v's, which contradicts the assumption that b is a base.

Suppose (3). There must be a path from b to v, because they are in the same strong component. Consider one such path, p, and let r be the least common ancestor of all the vertices on the path. (In other words, r is the vertex such that all the vertices on the path descend from it, but this property does not hold for any of its descendants. Since there is only one tree in the spanning forest there must be such a vertex.)

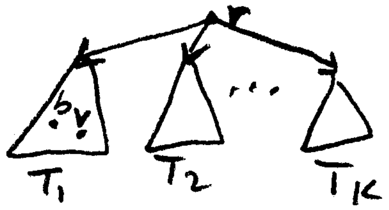
We claim that r must be on the path p. To prove this seems to require two cases.

Case 1: b and v descend from different children of r.



Let T_b be the subtree containing b and let T_v be the subtree containing v. Since $num(b) < num(v)$ and T_b and T_v are disjoint, there cannot be any edge from a vertex in T_b to one in T_v. This follows from Lemma 1. Therefore the only way path p can get from b to v is by going through r.

Case 2: b and v descend from the same child of r.



Suppose that the path p does not contain r. Then the path must touch the subtrees of at least two distinct children of r. (If not, then a child of r would be a lower common ancestor than r of the path p.) Call the subtrees rooted at the children of r $T_1, T_2, \ldots, T_k$. Assume without loss of generality that T_1 contains b and v. Path p starts in T_1 goes through a sequence of subtrees of r, then returns to T_1. Each time the path changes from one subtree to another, all of the numbers in the new subtree must be less than all those of the old subtree, by Lemma 1. So it is impossible for such a path to return to T_1. The conclusion is that the path must go through r.

Ok, so the path goes through r, so what? Well, since r is an ancestor of b, its depth-first search number is less than that of b. It is also in the same strong component as b because there is a path from b to r and a path (along tree edges) from r to b. But b was supposed to be the lowest numbered vertex in the strong component. This shows that (3) is impossible.

Of the three original alternatives, only (1) is left. Therefore v is a descendant of b.

The second part of the lemma is now trivial. Let x be a vertex on the path from v to b. There is a path from b to x (via tree edges). There is also a path from x to b by first going from x to v then, from v to b. Therefore x is in the same strong component as b. $\square$

Lemma 3 *Let b be a base vertex. Let $b_1, b_2, \ldots, b_k$ be all of the base vertices that descend from b. Then b's strong component is the set of vertices descending from b but not descending from any of $b_1, b_2, \ldots, b_k$.*

Proof. Assume the contrary, *i.e.* that there is a vertex, v, in the same strong component as b and which descends from b and b_i. There must be a path from v to b. There also a path from b to b_i to v (following tree edges). Therefore b and b_i are in the same strong component, which contradicts the assumption that b and b_i are bases of distinct strong components. $\square$

Definition: *Let $lowlink(v)$ be the minimum numbered vertex in the same strong component as v that can be reached from v by following in zero more tree edges followed by at most one back or cross edge.*

Lemma 4 *A vertex v is a base if and only if $num(v) = lowlink(v)$.*

Proof. We first assume that $num(v) > lowlink(v)$ and prove that v is not a base. Be definition of *lowlink*, there is a vertex w in the same strong component of v such that $lowlink(v) = num(w)$. Therefore $num(v) > num(w)$, and v cannot be a base.

To prove the converse, assume that v is not a base. Let b be the base of the strong component containing v. Then there must be a path p from v to b. By Lemma 2 b must be an ancestor of v. Let y be the first vertex along the path p that is not in the subtree rooted at v, and let x be the vertex before y on the path. Because the subtree rooted at y is disjoint from that rooted at v, and there is an edge from a descendant of v (namely x) to y, we can apply Lemma 1 and conclude that $num(y) < num(v)$. This shows that $lowlink(v) \leq num(y)$, since y is in the same strong component as v and is reachable by following tree edges, then one back or cross edge. Combining this with the fact that $num(y) < num(v)$ gives $lowlink(v) < num(v)$. $\square$

4. The Strong Components Algorithm

We can now present the algorithm for computing strong components. A stack S containing a particular set of vertices is maintained by the algorithm.

```
$S \leftarrow$ empty stack
$i \leftarrow 0$
for all $x \in V$ do $num(v) \leftarrow 0$
for all $x \in V$ do
        if $num(x) = 0$ then STRONG($x$)

STRONG($v$)
        $i \leftarrow i + 1$
        $num(v) \leftarrow i$
        $lowlink(v) \leftarrow i$
        push $v$ onto the stack $S$
        for all $w \in adj(v)$ do
                if $num(w) = 0$ then                              [$(v, w)$ is a tree edge ]
                        STRONG($w$)
                        $lowlink(v) \leftarrow \min(lowlink(v), lowlink(w))$
                else if $num(w) < num(v)$ then                  [$(v, w)$ is a back or cross edge ]
                        if $w \in S$                               [$w$ is in the same strong component as $v$]
                           then $lowlink(v) \leftarrow \min(lowlink(v), num(w))$
        if $lowlink(v) = num(v)$ then                            [$v$ is a base vertex]
                pop vertices off the stack while $num$(stack top)$\geq num(v)$
                all those vertices popped off the stack are output as a strong component
end STRONG
```

To begin to prove that this algorithm is correct, we first show something about the state of the stack.

Lemma 5 *It is always the case that if $x, y \in S$ and $num(x) < num(y)$ then there exists a path from x to y.*

Proof. We use induction. We need concern ourselves only with the case when new vertices are added to S. Suppose that within the call to STRONG(v) a call is made to STRONG(w) that adds w to S. Assume that the lemma held before this.

Since $num(w)$ is larger than any other we need only consider the vertices x such that $num(x) < num(w)$. If $x = v$ than the lemma still holds since there is an edge from v to w. If not, then since v was in S before, by the induction hypothesis there is a path from x to v. Combining this with the edge from v to w gives a path from x to w, finishing the proof. □

Theorem: *After the call to* STRONG(v) *is complete it is the case that:*

(1) $lowlink(v)$ has been correctly computed.

(2) All the strongly connected components contained in the subtree rooted at v have been output.

Proof. Assume that all the prior completed recursive calls to STRONG satisfy (1) and (2). We'll show that this call does also.

First we show (1) is true in the current call (assuming (1) and (2) are satisfied for all prior completed calls) then we use this (and all prior (1)'s and (2)'s) to show (2).

Clearly *lowlink*(v) is computed correctly assuming one thing: the test $w \in S$ is satisfied exactly when w is in the same strong component as v. If $w \in S$ then by Lemma 5 there exists a path from w to v. This, combined with edge (v, w) ensures that v and w are in the same strong component. If $w \notin S$ then since $num(w) < num(v)$ the recursive call to w must have been completed and the strong component containing w has been output. This strong component has been correctly computed by induction hypothesis (2).

We now show (2). By the proof just given, when the output phase is reached, *lowlink*(v) has been correctly computed. By Lemma 4 v is a base vertex.

The ensuing loop pops everything from the end of the stack (including v) into a strong component. These vertices are exactly those that are descendants of v but not in another strong component (here we're using induction hypothesis (2)). By Lemma 3, this is the strong component with v as a base. □

We would like to withdraw our assumptions that the graph has a vertex R connected to all other vertices, and R is searched first by the algorithm. Compare the following two alternatives: (1) Run the above algorithm on a graph G, or (2) add a new vertex R to G with an edge to all other vertices, and run the search above by calling STRONG(R). By syntatic analysis of the program above, it is easy to see that the these two processes will differ only in that (2) will output one extra strong component at the end (the one consisting of R). This is because while running (2), inside the call to STRONG(R), the loop through the vertices adjacant vertices of R will behave exactly like the loop on the outside running in (1).

It is tempting to consider what happens to the algorithm if the line:

"$lowlink(v) \leftarrow \min(lowlink(v), num(w))$"

Is replaced by:

"$lowlink(v) \leftarrow \min(lowlink(v), lowlink(w))$"

This new *lowlink* function is not the same as the old one, yet the algorithm will still work.

Lemma 6 *The modified strong components algorithm works.*

Proof. Let $ll(w)$ be the new *lowlink* function. (That is, change the line indicated above, then replace *lowlink* by *ll* everywhere in the program.) Since $lowlink(x) \leq num(x)$ this change can only decrease the function, that is $ll(x) \leq lowlink(x)$. Our goal is to show that the test $lowlink(v) = num(v)$ is true in the running of the original algorithm exactly when $ll(v) = num(v)$ in the running of the new algorithm.

One way is easy. If the test $ll(v) = num(v)$ is satisfied, then it must be the case that in the running of the original algorithm the test $lowlink(v) = num(v)$ is also satisfied, because *lowlink*(v) is trapped between $ll(v)$ and $num(v)$.

Conversely suppose that $lowlink(v) = num(v)$, that is, that v is a base vertex. Our goal is to show that $ll(v) = lowlink(v)$. The reason is that even though $ll(x)$ can be smaller than *lowlink*(x), its value is still the depth first number (*num* field) of of another vertex in the same strong component as x. If v is a base vertex there is no other vertex in the same strong component as v with a smaller *num* field. Therefore in this case $ll(v) = lowlink(v)$, which completes the proof. □