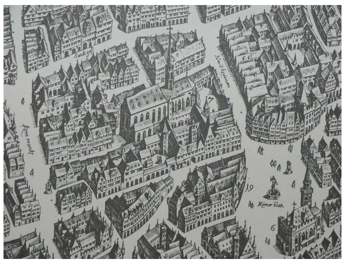
Graph Traversal



Suppose you are working in the traffic planning department of a town with a nice medieval center[1]*. An unholy coalition of shop owners, who want more street-side parking, and the Green Party, which wants to discourage car traffic altogether, has decided to turn most streets into one-way streets. You want to avoid the worst by checking whether the current plan maintains the minimal requirement that one can still drive from every point in town to every other point.*

In the language of graphs (see Sect. 2.9), the question is whether the directed graph formed by the streets is strongly connected. The same problem comes up in other applications. For example, in the case of a communication network with unidirectional channels (e.g., radio transmitters), we want to know who can communicate with whom. Bidirectional communication is possible within the strongly connected components of the graph.

We shall present a simple, efficient algorithm for computing strongly connected components (SCCs) in Sect. 9.2.2. Computing SCCs and many other fundamental problems on graphs can be reduced to systematic graph exploration, inspecting each edge exactly once. We shall present the two most important exploration strategies: *breadth-first search* , in Sect. 9.1, and *depth-first search*, in Sect. 9.2. Both strategies construct forests and partition the edges into four classes: *tree* edges comprising the forest, *forward* edges running parallel to paths of tree edges, *backward* edges running antiparallel to paths of tree edges, and *cross* edges that connect two different branches of a tree in the forest. Figure 9.1 illustrates the classification of edges.

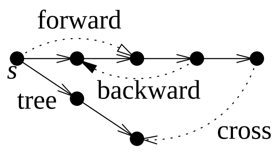


Fig. 9.1. Graph edges classified as tree edges, forward edges, backward edges, and cross edges

[1] The copper engraving above shows a part of Frankfurt around 1628 (M. Merian).

9.1 Breadth-First Search

A simple way to explore all nodes reachable from some node s is *breadth-first search* (BFS). BFS explores the graph *layer by layer*. The starting node s forms layer 0. The direct neighbors of s form layer 1. In general, all nodes that are neighbors of a node in layer i but not neighbors of nodes in layers 0 to $i-1$ form layer $i+1$.

The algorithm in Fig. 9.2 takes a node s and constructs the BFS tree rooted at s. For each node v in the tree, the algorithm records its distance $d(v)$ from s, and the parent node *parent*(v) from which v was first reached. The algorithm returns the pair $(d, parent)$. Initially, s has been reached and all other nodes store some special value $\perp$ to indicate that they have not been reached yet. Also, the depth of s is zero. The main loop of the algorithm builds the BFS tree layer by layer. We maintain two sets Q and Q'; Q contains the nodes in the current layer, and we construct the next layer in Q'. The inner loops inspect all edges (u,v) leaving nodes u in the current layer Q. Whenever v has no parent pointer yet, we put it into the next layer Q' and set its parent pointer and distance appropriately. Figure 9.3 gives an example of a BFS tree and the resulting backward and cross edges.

BFS has the useful feature that its tree edges define paths from s that have a minimum number of edges. For example, you could use such paths to find railway connections that minimize the number of times you have to change trains or to find paths in communication networks with a minimal number of hops. An actual path from s to a node v can be found by following the parent references from v backwards.

Exercise 9.1. Show that BFS will never classify an edge as forward, i.e., there are no edges (u,v) with $d(v) > d(u)+1$.

```
Function bfs(s : NodeId) : (NodeArray of NodeId) × (NodeArray of 0..n)
    d = ⟨∞,...,∞⟩ : NodeArray of NodeId                          // distance from root
    parent = ⟨⊥,...,⊥⟩ : NodeArray of NodeId
    d[s] := 0
    parent[s] := s                                                // self-loop signals root
    Q = ⟨s⟩ : Set of NodeId                                       // current layer of BFS tree
    Q' = ⟨⟩ : Set of NodeId                                       // next layer of BFS tree
    for ℓ := 0 to ∞ while Q ≠ ⟨⟩ do                               // explore layer by layer
        invariant Q contains all nodes with distance ℓ from s
        foreach u ∈ Q do
            foreach (u,v) ∈ E do                                  // scan edges out of u
                if parent(v) = ⊥ then                             // found an unexplored node
                    Q' := Q' ∪ {v}                                // remember for next layer
                    d[v] := ℓ + 1
                    parent(v) := u                                // update BFS tree
        (Q, Q') := (Q', ⟨⟩)                                       // switch to next layer
    return (d, parent)                // the BFS tree is now {(v,w) : w ∈ V, v = parent(w)}
```

Fig. 9.2. Breadth-first search starting at a node s

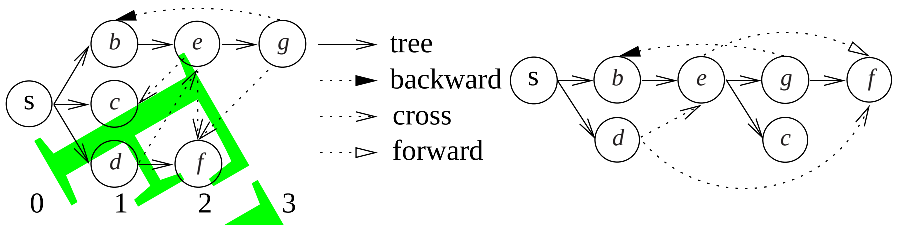


Fig. 9.3. An example of how BFS (*left*) and DFS (*right*) classify edges into tree edges, backward edges, cross edges, and forward edges. BFS visits the nodes in the order s, b, c, d, e, f, g. DFS visits the nodes in the order s, b, e, g, f, c, d

Exercise 9.2. What can go wrong with our implementation of BFS if $parent[s]$ is initialized to $\perp$ rather than s? Give an example of an erroneous computation.

Exercise 9.3. BFS trees are not necessarily unique. In particular, we have not specified in which order nodes are removed from the current layer. Give the BFS tree that is produced when d is removed before b when one performs a BFS from node s in the graph in Fig. 9.3.

Exercise 9.4 (FIFO BFS). Explain how to implement BFS using a single FIFO queue of nodes whose outgoing edges still have to be scanned. Prove that the resulting algorithm and our two-queue algorithm compute exactly the same tree if the two-queue algorithm traverses the queues in an appropriate order. Compare the FIFO version of BFS with Dijkstra's algorithm described in Sect. 10.3, and the Jarník–Prim algorithm described in Sect. 11.2. What do they have in common? What are the main differences?

Exercise 9.5 (graph representation for BFS). Give a more detailed description of BFS. In particular, make explicit how to implement it using the adjacency array representation described in Sect. 8.2. Your algorithm should run in time $O(n+m)$.

Exercise 9.6 (connected components). Explain how to modify BFS so that it computes a spanning forest of an undirected graph in time $O(m+n)$. In addition, your algorithm should select a *representative* node r for each connected component of the graph and assign it to $component[v]$ for each node v in the same component as r. Hint: scan all nodes $s \in V$ and start BFS from any node s that it still unreached when it is scanned. Do not reset the parent array between different runs of BFS. Note that isolated nodes are simply connected components of size one.

Exercise 9.7 (transitive closure). The *transitive closure* $G^+ = (V, E^+)$ of a graph $G = (V, E)$ has an edge $(u, v) \in E^+$ whenever there is a path from u to v in E. Design an algorithm for computing transitive closures. Hint: run $bfs(v)$ for each node v to find all nodes reachable from v. Try to avoid a full reinitialization of the arrays d and *parent* at the beginning of each call. What is the running time of your algorithm?

9.2 Depth-First Search

You may view breadth-first search as a careful, conservative strategy for systematic exploration that looks at known things before venturing into unexplored territory. In this respect, *depth-first search* (DFS) is the exact opposite: whenever it finds a new node, it immediately continues to explore from it. It goes back to previously explored nodes only if it runs out of options. Although DFS leads to unbalanced, strange-looking exploration trees compared with the orderly layers generated by BFS, the combination of eager exploration with the perfect memory of a computer makes DFS very useful. Figure 9.4 gives an algorithm template for DFS. We can derive specific algorithms from it by specifying the subroutines *init*, *root*, *traverseTreeEdge*, *traverseNonTreeEdge*, and *backtrack*.

DFS marks a node when it first discovers it; initially, all nodes are unmarked. The main loop of DFS looks for unmarked nodes s and calls $DFS(s,s)$ to grow a tree rooted at s. The recursive call $DFS(u,v)$ explores all edges (v,w) out of v. The argument (u,v) indicates that v was reached via the edge (u,v) into v. For root nodes s, we use the “dummy” argument (s,s). We write $DFS(*,v)$ if the specific nature of the incoming edge is irrelevant to the discussion at hand. Assume now that we are exploring edge (v,w) within the call $DFS(*,v)$.

If w has been seen before, w is already a node of the DFS forest. So (v,w) is not a tree edge, and hence we call $traverseNonTreeEdge(v,w)$ and make no recursive call of *DFS*.

If w has not been seen before, (v,w) becomes a tree edge. We therefore call $traverseTreeEdge(v,w)$, mark w, and make the recursive call $DFS(v,w)$. When we return from this call, we explore the next edge out of v. Once all edges out of v have been explored, we call *backtrack* on the incoming edge (u,v) to perform any summarizing or cleanup operations needed and return.

At any point in time during the execution of *DFS*, there are a number of active calls. More precisely, there are nodes v_1, v_2, $\ldots v_k$ such that we are currently exploring edges out of v_k, and the active calls are $DFS(v_1,v_1)$, $DFS(v_1,v_2)$, $\ldots$, $DFS(v_{k-1},v_k)$. In this situation, we say that the nodes v_1, v_2, $\ldots$, v_k are *active* and form the DFS recursion stack. The node v_k is called the *current node*. We say that a node v has been reached when $DFS(*,v)$ is called, and is finished when the call $DFS(*,v)$ terminates.

Exercise 9.8. Give a nonrecursive formulation of DFS. You need to maintain a stack of active nodes and, for each active node, the set of unexplored edges.

9.2.1 DFS Numbering, Finishing Times, and Topological Sorting

DFS has numerous applications. In this section, we use it to number the nodes in two ways. As a by-product, we see how to detect cycles. We number the nodes in the order in which they are reached (array *dfsNum*) and in the order in which they are finished (array *finishTime*). We have two counters *dfsPos* and *finishingTime*, both initialized to one. When we encounter a new root or traverse a tree edge, we set the

```
**Depth-first search of a directed graph** $G = (V, E)$
unmark all nodes
init
foreach $s \in V$ do
    if $s$ is not marked then
        mark $s$                                   // make $s$ a root and grow
        $root(s)$                                  // a new DFS tree rooted at it.
        $DFS(s, s)$

Procedure $DFS(u, v : NodeId)$                     // Explore $v$ coming from $u$.
    foreach $(v, w) \in E$ do
        if $w$ is marked then $traverseNonTreeEdge(v, w)$     // $w$ was reached before
        else    $traverseTreeEdge(v, w)$                      // $w$ was not reached before
                mark $w$
                $DFS(v, w)$
    $backtrack(u, v)$                              // return from $v$ along the incoming edge
```

Fig. 9.4. A template for depth-first search of a graph $G = (V, E)$. We say that a call $DFS(*, v)$ explores v. The exploration is complete when we return from this call

dfsNum of the newly encountered node and increment *dfsPos*. When we backtrack from a node, we set its *finishTime* and increment *finishingTime*. We use the following subroutines:

init:	$dfsPos = 1 \;:\; 1..n$; $\quad finishingTime = 1 \;:\; 1..n$
$root(s)$:	$dfsNum[s] := dfsPos{+}{+}$
$traverseTreeEdge(v, w)$:	$dfsNum[w] := dfsPos{+}{+}$
$backtrack(u, v)$:	$finishTime[v] := finishingTime{+}{+}$

The ordering by *dfsNum* is so useful that we introduce a special notation '$\prec$' for it. For any two nodes u and v, we define

$$u \prec v \Leftrightarrow dfsNum[u] < dfsNum[v] \;.$$

The numberings *dfsNum* and *finishTime* encode important information about the execution of *DFS*, as we shall show next. We shall first show that the DFS numbers increase along any path of the DFS tree, and then show that the numberings together classify the edges according to their type.

Lemma 9.1. *The nodes on the DFS recursion stack are sorted with respect to* $\prec$.

Proof. *dfsPos* is incremented after every assignment to *dfsNum*. Thus, when a node v becomes active by a call $DFS(u, v)$, it has just been assigned the largest *dfsNum* so far. □

*dfsNum*s and *finishTime*s classify edges according to their type, as shown in Table 9.1. The argument is as follows. Two calls of DFS are either nested within each other, i.e., when the second call starts, the first is still active, or disjoint, i.e., when the

Table 9.1. The classification of an edge (v,w). Tree and forward edges are also easily distinguished. Tree edges lead to recursive calls, and forward edges do not

Type	$dfsNum[v] < dfsNum[w]$	$finishTime[w] < FinishTime[v]$
Tree	Yes	Yes
Forward	Yes	Yes
Backward	No	No
Cross	No	Yes

second starts, the first is already completed. If $DFS(*,w)$ is nested in $DFS(*,v)$, the former call starts after the latter and finishes before it, i.e., $dfsNum[v] < dfsNum[w]$ and $finishTime[w] < finishTime[v]$. If $DFS(*,w)$ and $DFS(*,v)$ are disjoint and the former call starts before the latter, it also ends before the latter, i.e., $dfsNum[w] < dfsNum[v]$ and $finishTime[w] < finishTime[v]$. The tree edges record the nesting structure of recursive calls. When a tree edge (v,w) is explored within $DFS(*,v)$, the call $DFS(v,w)$ is made and hence is nested within $DFS(*,v)$. Thus w has a larger DFS number and a smaller finishing time than v. A forward edge (v,w) runs parallel to a path of tree edges and hence w has a larger DFS number and a smaller finishing time than v. A backward edge (v,w) runs antiparallel to a path of tree edges, and hence w has a smaller DFS number and a larger finishing time than v. Let us look, finally, at a cross edge (v,w). Since (v,w) is not a tree, forward, or backward edge, the calls $DFS(*,v)$ and $DFS(*,w)$ cannot be nested within each other. Thus they are disjoint. So w is marked either before $DFS(*,v)$ starts or after it ends. The latter case is impossible, since, in this case, w would be unmarked when the edge (v,w) was explored, and the edge would become a tree edge. So w is marked before $DFS(*,v)$ starts and hence $DFS(*,w)$ starts and ends before $DFS(*,v)$. Thus $dfsNum[w] < dfsNum[v]$ and $finishTime[w] < finishTime[v]$. The following Lemma summarizes the discussion.

Lemma 9.2. *Table 9.1 shows the characterization of edge types in terms of dfsNum and finishTime.*

Exercise 9.9. Modify DFS such that it labels the edges with their type. What is the type of an edge (v,w) when w is on the recursion stack when the edge is explored?

Finishing times have an interesting property for directed acyclic graphs.

Lemma 9.3. *The following properties are equivalent: (i) G is an acyclic directed graph (DAG); (ii) DFS on G produces no backward edges; (iii) all edges of G go from larger to smaller finishing times.*

Proof. Backward edges run antiparallel to paths of tree edges and hence create cycles. Thus DFS on an acyclic graph cannot create any backward edges. All other types of edge run from larger to smaller finishing times according to Table 9.1. Assume now that all edges run from larger to smaller finishing times. In this case the graph is clearly acyclic. □

An order of the nodes of a DAG in which all edges go from left to right is called a *topological sorting*. By Lemma 9.3, the ordering by decreasing finishing time is a topological ordering. Many problems on DAGs can be solved efficiently by iterating over the nodes in a topological order. For example, in Sect. 10.2 we shall see a fast, simple algorithm for computing shortest paths in acyclic graphs.

Exercise 9.10 (topological sorting). Design a DFS-based algorithm that outputs the nodes in topological order if G is a DAG. Otherwise, it should output a cycle.

Exercise 9.11. Design a BFS-based algorithm for topological sorting.

Exercise 9.12. Show that DFS on an undirected graph does not produce any cross edges.

9.2.2 Strongly Connected Components

We now come back to the problem posed at the beginning of this chapter. Recall that two nodes belong to the same strongly connected component (SCC) of a graph iff they are reachable from each other. In undirected graphs, the relation "being reachable" is symmetric, and hence strongly connected components are the same as connected components. Exercise 9.6 outlines how to compute connected components using BFS, and adapting this idea to DFS is equally simple. For directed graphs, the situation is more interesting; see Fig. 9.5 for an example. We shall show that an extension of DFS computes the strongly connected components of a digraph G in linear time $\mathrm{O}(n+m)$. More precisely, the algorithm will output an array *component* indexed by nodes such that $component[v] = component[w]$ iff v and w belong to the same SCC. Alternatively, it could output the node set of each SCC.

Exercise 9.13. Show that the node sets of distinct SCCs are disjoint. Hint: assume that SCCs C and D have a common node v. Show that any node in C can reach any node in D and vice versa.

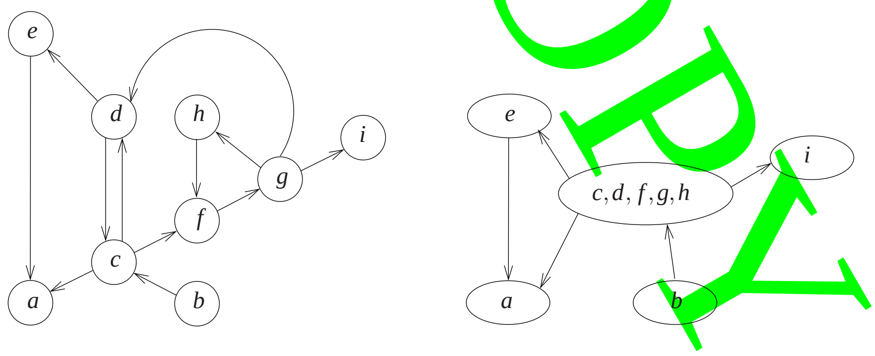


Fig. 9.5. A digraph G and the corresponding shrunken graph G_s. The SCCs of G have node sets $\{a\}$, $\{b\}$, $\{c,d,f,g,h\}$, $\{e\}$, and $\{i\}$

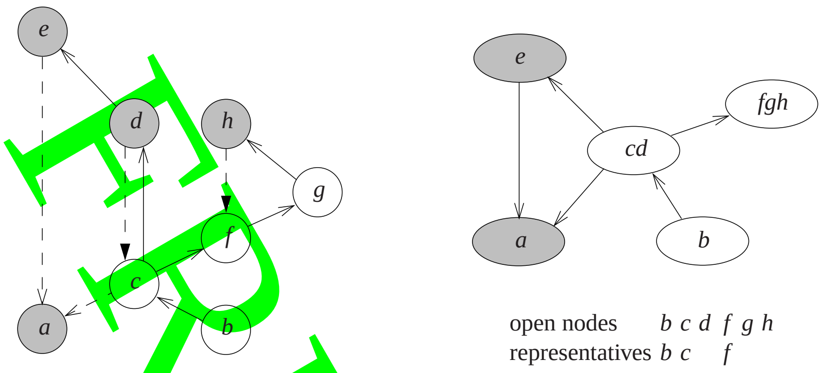


Fig. 9.6. A snapshot of DFS on the graph of Fig. 9.5 and the corresponding shrunken graph. The first DFS was started at node a and a second DFS was started at node b, the current node is g, and the recursion stack contains b, c, f, g. The edges (g,i) and (g,d) have not been explored yet. Edges (h,f) and (d,c) are back edges, (e,a) is a cross edge, and all other edges are tree edges. Finished nodes and closed components are shaded. There are closed components $\{a\}$ and $\{e\}$ and open components $\{b\}$, $\{c,d\}$, and $\{f,g,h\}$. The open components form a path in the shrunken graph with the current node g belonging to the last component. The representatives of the open components are the nodes b, c, and f, respectively. DFS has reached the open nodes in the order b, c, d, f, g, h. The representatives partition the sequence of open nodes into the SCCs of G_c

The idea underlying the algorithm is simple. We imagine that the edges of G are added one by one to an initially edgeless graph. We use $G_c = (V, E_c)$ to denote the current graph, and keep track of how the SCCs of G_c evolve as edges are added. Initially, there are no edges and each node forms an SCC of its own. For the addition step, it is helpful to introduce the notion of a *shrunken graph*. We use G_c^s to denote the shrunken graph corresponding to G_c. The nodes of G_c^s are the SCCs of G_c. If C and D are distinct SCCs of G_c, we have an edge (C,D) in G_c^s iff there are nodes $u \in C$ and $v \in D$ where (u,v) is an edge of G_c. Figure 9.5 shows an example.

Lemma 9.4. *The shrunken graph G_c^s is acyclic.*

Proof. Assume otherwise, and let $C_1, C_2, \ldots, C_{k-1}, C_k$ with $C_k = C_1$ be a cycle in G_c^s. Recall that the C_i are SCCs of G_c. By the definition of G_c^s, G_c contains an edge (v_i, w_{i+1}) with $v_i \in C_i$ and $w_{i+1} \in C_{i+1}$ for $0 \leq i < k$. Define $v_k = v_1$. Since C_i is strongly connected, G_c contains a path from w_{i+1} to v_{i+1}, $0 \leq i < k$.Thus all the v_i's belong to the same SCC, a contradiction. $\square$

How do the SCCs of G_c and G_c^s change when we add an edge e to G_c? There are three cases to consider. (1) Both endpoints of e belong to the same SCC of G_c. The shrunken graph and the SCCs do not change. (2) e connects nodes in different SCCs but does not close a cycle. The SCCs do not change, and an edge is added to the shrunken graph. (3) e connects nodes in different SCCs and closes one or more cycles. In this case, all SCCs lying on one of the newly formed cycles are merged into a single SCC, and the shrunken graph changes accordingly.

In order to arrive at an efficient algorithm, we need to describe how we maintain the SCCs as the graph evolves. If the edges are added in arbitrary order, no efficient simple method is known. However, if we use DFS to explore the graph, an efficient solution is fairly simple to obtain. Consider a depth-first search on G and let E_c be the set of edges already explored by DFS. A subset V_c of the nodes is already marked. We distinguish between three kinds of SCC of G_c: unreached, open, and closed. Unmarked nodes have indegree and outdegree zero in G_c and hence form SCCs consisting of a single node. This node is isolated in the shrunken graph. We call these SCCs *unreached*. The other SCCs consist of marked nodes only. We call an SCC consisting of marked nodes *open* if it contains an active node, and *closed* if it contains only finished nodes. We call a marked node "open" if it belongs to an open component and "closed" if it belongs to a closed component. Observe that a closed node is always finished and that an open node may be either active or finished. For every SCC, we call the node with the smallest DFS number in the SCC the *representative* of the SCC. Figure 9.6 illustrates these concepts. We state next some important invariant properties of G_c; see also Fig. 9.7:

(1) All edges in G (not just G_c) out of closed nodes lead to closed nodes. In our example, the nodes a and e are closed.
(2) The tree path to the current node contains the representatives of all open components. Let S_1 to S_k be the open components as they are traversed by the tree path to the current node. There is then a tree edge from a node in S_{i-1} to the representative of S_i, and this is the only edge into S_i, $2 \leq i \leq k$. Also, there is no edge from an S_j to an S_i with $i < j$. Finally, all nodes in S_j are reachable from the representative r_i of S_i for $1 \leq i \leq j \leq k$. In short, the open components form a path in the shrunken graph. In our example, the current node is g. The tree path $\langle b,c,f,g\rangle$ to the current node contains the open representatives b, c, and f.
(3) Consider the nodes in open components ordered by their DFS numbers. The representatives partition the sequence into the open components. In our example, the sequence of open nodes is $\langle b,c,d,f,g,h\rangle$ and the representatives partition this sequence into the open components $\{b\}$, $\{c,d\}$, and $\{f,g,h\}$.

We shall show below that all three properties hold true generally, and not only for our example. The three properties will be invariants of the algorithm to be developed. The first invariant implies that the closed SCCs of G_c are actually SCCs of G, i.e., it is justified to call them closed. This observation is so important that it deserves to be stated as a lemma.

Lemma 9.5. *A closed SCC of G_c is an SCC of G.*

Proof. Let v be a closed vertex, let S be the SCC of G containing v, and let S_c be the SCC of G_c containing v. We need to show that $S = S_c$. Since G_c is a subgraph of G, we have $S_c \subseteq S$. So, it suffices to show that $S \subseteq S_c$. Let w be any vertex in S. There is then a cycle C in G passing through v and w. The first invariant implies that all vertices of C are closed. Since closed vertices are finished, all edges out of them have been explored. Thus C is contained in G_c, and hence $w \in S_c$. □

The Invariants (2) and (3) suggest a simple method to represent the open SCCs of G_c. We simply keep a sequence *oNodes* of all open nodes in increasing order of DFS numbers, and the subsequence *oReps* of open representatives. In our example, we have $oNodes = \langle b, c, d, f, g, h \rangle$ and $oReps = \langle b, c, f \rangle$. We shall later see that the type *Stack* **of** *NodeId* is appropriate for both sequences.

Let us next see how the SCCs of G_c develop during DFS. We shall discuss the various actions of DFS one by one and show that the invariants are maintained. We shall also discuss how to update our representation of the open components.

When DFS starts, the invariants clearly hold: no node is marked, no edge has been traversed, G_c is empty, and hence there are neither open nor closed components yet. Our sequences *oNodes* and *oReps* are empty.

Just before a new root will be marked, all marked nodes are finished and hence there cannot be any open component. Therefore, both of the sequences *oNodes* and *oReps* are empty, and marking a new root s produces the open component $\{s\}$. The invariants are clearly maintained. We obtain the correct representation by adding s to both sequences.

If a tree edge $e = (v, w)$ is traversed and hence w becomes marked, $\{w\}$ becomes an open component on its own. All other open components are unchanged. The first invariant is clearly maintained, since v is active and hence open. The old current node is v and the new current node is w. The sequence of open components is extended by $\{w\}$. The open representatives are the old open representatives plus the node w. Thus the second invariant is maintained. Also, w becomes the open node with the largest DFS number and hence *oNodes* and *oReps* are both extended by w. Thus the third invariant is maintained.

Now suppose that a nontree edge $e = (v, w)$ out of the current node v is explored. If w is closed, the SCCs of G_c do not change when e is added to G_c since, by Lemma 9.5, the SCC of G_c containing w is already an SCC of G *before* e is traversed. So, assume that w is open. Then w lies in some open SCC S_i of G_c. We claim

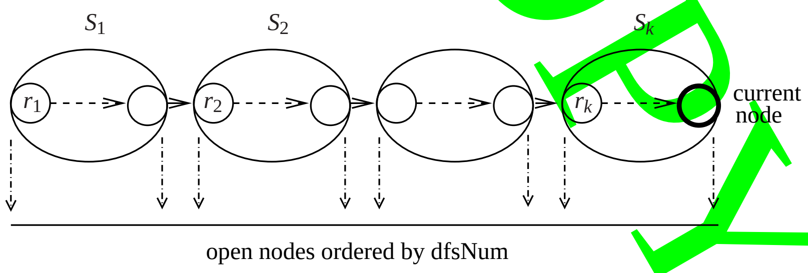


Fig. 9.7. The open SCCs are shown as ovals, and the current node is shown as a **bold** circle. The tree path to the current node is indicated. It enters each component at its representative. The horizontal line below represents the open nodes, ordered by *dfsNum*. Each open SCC forms a contiguous subsequence, with its representative as its leftmost element

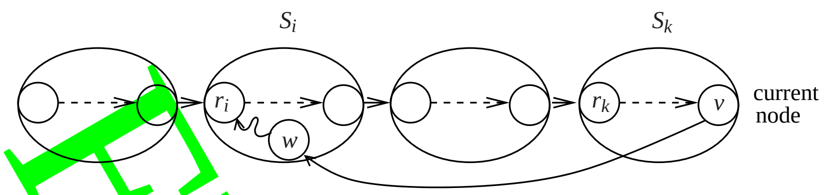


Fig. 9.8. The open SCCs are shown as ovals and their representatives as circles to the left of an oval. All representatives lie on the tree path to the current node v. The nontree edge $e = (v, w)$ ends in an open SCC S_i with representative r_i. There is a path from w to r_i since w belongs to the SCC with representative r_i. Thus the edge (v, w) merges S_i to S_k into a single SCC

that the SCCs S_i to S_k are merged into a single component and all other components are unchanged; see Fig. 9.8. Let r_i be the representative of S_i. We can then go from r_i to v along a tree path by invariant (2), then follow the edge (v, w), and finally return to r_i. The path from w to r_i exists, since w and r_i lie in the same SCC of G_c. We conclude that any node in an S_j with $i \leq j \leq k$ can be reached from r_i and can reach r_i. Thus the SCCs S_i to S_k become one SCC, and r_i is their representative. The S_j with $j < i$ are unaffected by the addition of the edge.

The third invariant tells us how to find r_i, the representative of the component containing w. The sequence *oNodes* is ordered by *dfsNum*, and the representative of an SCC has the smallest *dfsNum* of any node in that component. Thus $dfsNum[r_i] \leq dfsNum[w]$ and $dfsNum[w] < dfsNum[r_j]$ for all $j > i$. It is therefore easy to update our representation. We simply delete all representatives r with $dfsNum[r] > dfsNum[w]$ from *oReps*.

Finally, we need to consider finishing a node v. When will this close an SCC? By invariant (2), all nodes in a component are tree descendants of the representative of the component, and hence the representative of a component is the last node to be finished in the component. In other words, we close a component iff we finish a representative. Since *oReps* is ordered by *dfsNum*, we close a component iff the last node of *oReps* finishes. So, assume that we finish a representative v. Then, by invariant (3), the component S_k with representative $v = r_k$ consists of v and all nodes in *oNodes* following v. Finishing v closes S_k. By invariant (2), there is no edge out of S_k into an open component. Thus invariant (1) holds after S_k is closed. The new current node is the parent of v. By invariant (2), the parent of v lies in S_{k-1}. Thus invariant (2) holds after S_k is closed. Invariant (3) holds after v is removed from *oReps*, and v and all nodes following it are removed from *oNodes*.

It is now easy to instantiate the DFS template. Fig. 9.10 shows the pseudocode, and Fig. 9.9 illustrates a complete run. We use an array *component* indexed by nodes to record the result, and two stacks *oReps* and *oNodes*. When a new root is marked or a tree edge is explored, a new open component consisting of a single node is created by pushing this node onto both stacks. When a cycle of open components is created, these components are merged by popping representatives from *oReps* as long as the top representative is not to the left of the node w closing the cycle. An SCC S is closed when its representative v finishes. At that point, all nodes of S are stored

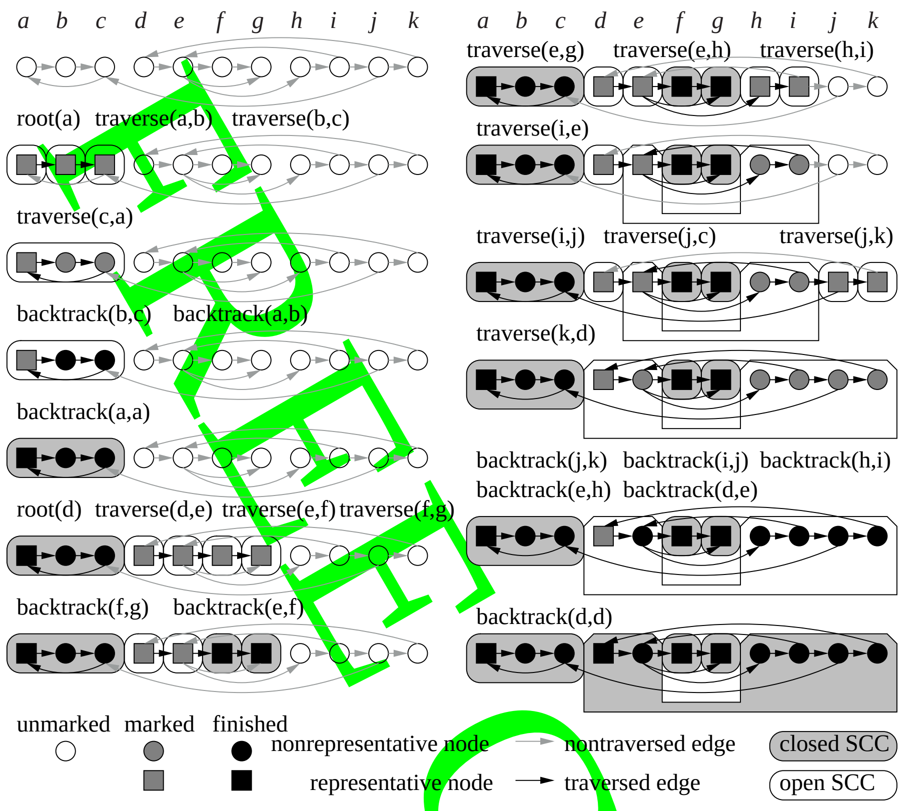

a b c d e f g h i j k
root(a) traverse(a,b) traverse(b,c)
traverse(c,a)
backtrack(b,c) backtrack(a,b)
backtrack(a,a)
root(d) traverse(d,e) traverse(e,f) traverse(f,g)
backtrack(f,g) backtrack(e,f)
traverse(e,g) traverse(e,h) traverse(h,i)
traverse(i,e)
traverse(i,j) traverse(j,c) traverse(j,k)
traverse(k,d)
backtrack(j,k) backtrack(i,j) backtrack(h,i)
backtrack(e,h) backtrack(d,e)
backtrack(d,d)
unmarked marked finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC

Fig. 9.9. An example of the development of open and closed SCCs during DFS. Unmarked nodes are shown as empty circles, marked nodes are shown in gray, and finished nodes are shown in black. Nontraversed edges are shown in gray, and traversed edges are shown in black. Open SCCs are shown as empty ovals, and closed SCCs are shown as gray ovals. We start in the situation shown at the upper left. We make a a root and traverse the edges (a,b) and (b,c). This creates three open SSCs. The traversal of edge (c,a) merges these components into one. Next, we backtrack to b, then to a, and finally from a. At this point, the component becomes closed. Please complete the description

above v in *oNodes*. The operation *backtrack* therefore closes S by popping v from *oReps*, and by popping the nodes $w \in S$ from *oNodes* and setting their *component* to the representative v.

Note that the test $w \in$ *oNodes* in *traverseNonTreeEdge* can be done in constant time by storing information with each node that indicates whether the node is open or not. This indicator is set when a node v is first marked, and reset when the component of v is closed. We give implementation details in Sect. 9.3. Furthermore, the while loop and the repeat loop can make at most n iterations during the entire execution of the algorithm, since each node is pushed onto the stacks exactly once. Hence, the execution time of the algorithm is $O(m+n)$. We have the following theorem.

```
init:
    component : NodeArray of NodeId                          // SCC representatives
    oReps = ⟨⟩ : Stack of NodeId                       // representatives of open SCCs
    oNodes = ⟨⟩ : Stack of NodeId                             // all nodes in open SCCs
root(w) or traverseTreeEdge(v,w):
    oReps.push(w)                                                         // new open
    oNodes.push(w)                                                       // component
traverseNonTreeEdge(v,w):
    if $w \in oNodes$ then
        while $w \prec oReps.top$ do oReps.pop            // collapse components on cycle
backtrack(u,v):
    if $v = oReps.top$ then
        oReps.pop                                                            // close
        repeat                                                           // component
            $w := oNodes.pop$
            $component[w] := v$
        until $w = v$
```

Fig. 9.10. An instantiation of the DFS template that computes strongly connected components of a graph $G = (V, E)$

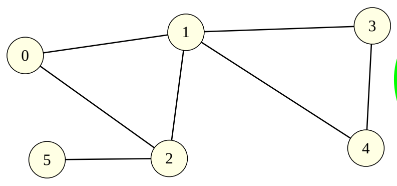


Fig. 9.11. The graph has two 2-edge connected components, namely $\{0,1,2,3,4\}$ and $\{5\}$. The graph has three biconnected components, namely the subgraphs spanned by the sets $\{0,1,2\}$, $\{1,3,4\}$ and $\{2,5\}$. The vertices 1 and 2 are articulation points

Theorem 9.6. *The algorithm in Fig. 9.10 computes strongly connected components in time* $O(m+n)$.

Exercise 9.14 (certificates). Let G be a strongly connected graph and let s be a node of G. Show how to construct two trees rooted at s. The first tree proves that all nodes can be reached from s, and the second tree proves than s can be reached from all nodes.

Exercise 9.15 (2-edge-connected components). An undirected graph is 2-edge-connected if its edges can be oriented so that the graph becomes strongly connected. The 2-edge-connected components are the maximal 2-edge-connected subgraphs; see Fig. 9.11. Modify the SCC algorithm shown in Fig. 9.10 so that it computes 2-edge-connected components. Hint: show first that DFS of an undirected graph never produces any cross edges.

Exercise 9.16 (biconnected components). Two nodes of an *undirected* graph belong to the same *biconnected component* (BCC) iff they are connected by an edge or there are two edge-disjoint paths connecting them; see Fig. 9.11. A node is an *articulation point* if it belongs to more than one BCC. Design an algorithm that computes biconnected components using a single pass of DFS. Hint: adapt the strongly-connected-components algorithm. Define the representative of a BCC as the node with the second smallest *dfsNum* in the BCC. Prove that a BCC consists of the parent of the representative and all tree descendants of the representative that can be reached without passing through another representative. Modify *backtrack*. When you return from a representative v, output v, all nodes above v in *oNodes*, and the parent of v.

9.3 Implementation Notes

BFS is usually implemented by keeping unexplored nodes (with depths d and $d+1$) in a FIFO queue. We chose a formulation using two separate sets for nodes at depth d and nodes at depth $d+1$ mainly because it allows a simple loop invariant that makes correctness immediately evident. However, our formulation might also turn out to be somewhat more efficient. If Q and Q' are organized as stacks, we will have fewer cache faults than with a queue, in particular if the nodes of a layer do not quite fit into the cache. Memory management becomes very simple and efficient when just a single array a of n nodes is allocated for both of the stacks Q and Q'. One stack grows from $a[1]$ to the right and the other grows from $a[n]$ to the left. When the algorithm switches to the next layer, the two memory areas switch their roles.

Our SCC algorithm needs to store four kinds of information for each node v: an indication of whether v is marked, an indication of whether v is open, something like a DFS number in order to implement "$\prec$", and, for closed nodes, the *NodeId* of the representative of its component. The array *component* suffices to keep this information. For example, if *NodeId*s are integers in $1..n$, $component[v] = 0$ could indicate an unmarked node. Negative numbers can indicate negated DFS numbers, so that $u \prec v$ iff $component[u] > component[v]$. This works because "$\prec$" is never applied to closed nodes. Finally, the test $w \in oNodes$ simply becomes $component[v] < 0$. With these simplifications in place, additional tuning is possible. We make *oReps* store *component* numbers of representatives rather than their IDs, and save an access to $component[oReps.top]$. Finally, the array *component* should be stored with the node data as a single array of records. The effect of these optimization on the performance of our SCC algorithm is discussed in [132].

9.3.1 C++

LEDA [118] has implementations for topological sorting, reachability from a node (*DFS*), DFS numbering, BFS, strongly connected components, biconnected components, and transitive closure. BFS, DFS, topological sorting, and strongly connected components are also available in a very flexible implementation that separates representation and implementation, supports incremental execution, and allows various other adaptations.

The Boost graph library [27] uses the *visitor concept* to support graph traversal. A visitor class has user-definable methods that are called at *event points* during the execution of a graph traversal algorithm. For example, the DFS visitor defines event points similar to the operations *init*, *root*, *traverse*∗, and *backtrack* used in our DFS template; there are more event points in Boost.

9.3.2 Java

The JDSL [78] library [78] supports DFS in a very flexible way, not very much different from the visitor concept described for Boost. There are also more specialized algorithms for topological sorting and finding cycles.

9.4 Historical Notes and Further Findings

BFS and DFS were known before the age of computers. Tarjan [185] discovered the power of DFS and provided linear-time algorithms for many basic problems related to graphs, in particular biconnected and strongly connected components. Our SCC algorithm was invented by Cheriyan and Mehlhorn [39] and later rediscovered by Gabow [70]. Yet another linear-time SCC algorithm is that due to Kosaraju and Sharir [178]. It is very simple, but needs two passes of DFS. DFS can be used to solve many other graph problems in linear time, for example ear decomposition, planarity testing, planar embeddings, and triconnected components.

It may seem that problems solvable by graph traversal are so simple that little further research is needed on them. However, the bad news is that graph traversal itself is very difficult on advanced models of computation. In particular, DFS is a nightmare for both parallel processing [161] and memory hierarchies [141, 128]. Therefore alternative ways to solve seemingly simple problems are an interesting area of research. For example, in Sect. 11.8 we describe an approach to constructing minimum spanning trees using *edge contraction* that also works for finding connected components. Furthermore, the problem of finding biconnected components can be reduced to finding connected components [189]. The DFS-based algorithms for biconnected components and strongly connected components are almost identical. But this analogy completely disappears for advanced models of computation, so that algorithms for strongly connected components remain an area of intensive (and sometimes frustrating) research. More generally, it seems that problems for undirected graphs (such as finding biconnected components) are often easier to solve than analogous problems for directed graphs (such as finding strongly connected components).