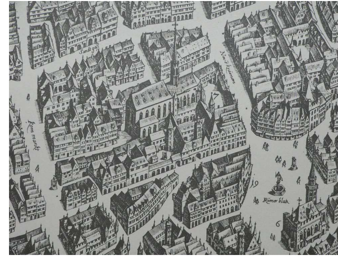
9

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 minimum requirement that one can still drive from every point in town to every other point.*

In the language of graphs (see Sect. 2.12), 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. 9.3.2. 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* (BFS) in Sect. 9.1 and *depth-first search* (DFS) in Sect. 9.3. 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. Cross edges are all remaining edges; they connect two different subtrees in the forest. Figure 9.1 illustrates the classification of edges.

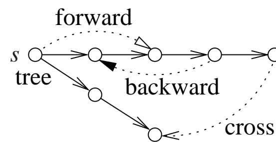


Fig. 9.1. Graph edges classified as tree edges, forward edges, backward edges, and cross edges.

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

BFS can be parallelized to some extent and is discussed in Sect. 9.2. In contrast, DFS seems inherently difficult to parallelize. In Sect. 9.4, we therefore discuss only the traversal of directed acyclic graphs (DAGs) – a problem where DFS is used as the standard sequential solution. We also discuss several problems on undirected graphs, where a conversion to a DAG is an important step towards a parallel algorithm.

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 (or, digraph) *layer by layer*. The starting node s forms layer 0. The direct neighbors (or, respectively, successors) of s form layer 1. In general, all nodes that are neighbors (or, successors) of a node in layer i but not neighbors (or, successors) of nodes in layers 0 to $i-1$ form layer $i+1$. Instead of saying that node v belongs to layer i, we also say that v has *depth* i or *distance* i from s.

The algorithm in Fig. 9.3 takes a node s and constructs the BFS tree rooted at s. This tree comprises exactly the nodes that are reachable from 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 0. 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 9.2 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 smallest number of hops. An actual path from s to a node v can be found by following the parent references from v backwards.

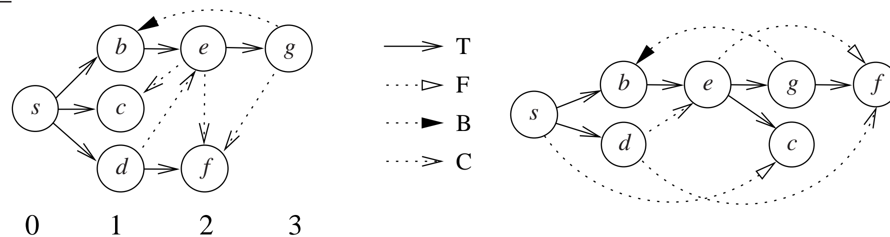


Fig. 9.2. BFS (*left*) and DFS (*right*) classify edges into tree (T), backward (B), cross(C), and forward edges (F). BFS visits the nodes in the order s, b, c, d, e, f, g and partitions them into layers $\{s\}$, $\{b,c,d\}$, $\{e,f\}$, and $\{g\}$. DFS visits the nodes in the order s, b, e, g, f, c, d.

```
Function bfs(s : NodeId) : (NodeArray of 0..n) × (NodeArray of NodeId)
    d = ⟨∞,...,∞⟩ : NodeArray of 0..n                              // 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
                    parent(v) := u                                  // update BFS tree
                    d[v] := ℓ + 1
                    Q′ := Q′ ∪ {v}                                  // remember for next layer
        (Q,Q′) := (Q′,⟨⟩)                                           // switch to next layer
    return (d,parent)                     // the BFS tree is now {(v,w) : w ∈ V, v = parent(w)}
```

Fig. 9.3. Breadth-first search starting at a node s

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

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 the order in which 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. 9.2.

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 $\mathrm{O}(n+m)$.

Exercise 9.6 (BFS in undirected graphs). Assume the bidirected representation of undirected graphs. Show that edges are traversed in at most one direction, i.e., only the scanning of one of the directed versions (u,v) or (v,u) of an undirected edge $\{u,v\}$ can add a node to Q'. When does neither directed version add a node to Q'?

Exercise 9.7 (connected components). Explain how to modify BFS so that it computes a spanning forest of an undirected graph in time $\mathrm{O}(m+n)$. 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 $s \in V$ in an outer loop 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 1.

Exercise 9.8 (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 of length 1 or more 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 Parallel Breadth-First Search

Here, we parallelize each iteration of the main loop of the BFS algorithm shown in Fig. 9.3. We first describe the parallelization for the shared-memory model and assume that it is sufficient to process the nodes in the current layer Q in parallel. Then Sect. 9.2.3 outlines what has to be changed for the distributed-memory model. Section 9.2.4 explains how to handle nodes with very high degree by also parallelizing the loop over the edges out of a single node u.

9.2.1 Shared-Memory BFS

Suppose the current layer Q is represented as a global array. We run the loop over Q in parallel. Several of the load-balancing algorithms presented in Chap. 14 can be used. We describe the variant using prefix sums given in Sect. 14.2. Since the work for a node is roughly proportional to its outdegree, we compute the prefix sum over the outdegrees of the nodes in Q, i.e., for node $Q[j]$, we compute $\sigma[j] := \sum_{k \le j} \mathrm{outdegree}(Q[k])$. Let $m_\ell := \sigma[|Q|]$ be the total outdegree of the layer. Node $Q[j]$ is then assigned to PE $\lceil \sigma[j]p/m_\ell \rceil$. PE j finds the last node it has to process using binary search in σ, searching for jm_ℓ/p.

To parallelize the loop over Q, we avoid contention on Q' by splitting it into local pieces – each PE works on a local array Q' storing the nodes it enqueues. After the loop finishes, these local pieces are copied to the global array Q for the next iteration. Each PE copies its own piece. The starting addresses in Q can be computed as a prefix sum over the piece sizes.

A further dependence between computations on different nodes is caused by multiple edges between nodes in the current layer and a node v in the next layer. If these are processed at the same time by different PEs, there might be write contention for $d[v]$ and *parent*(v). In that case, v could also be inserted into Q' multiple times.

Let us first discuss a solution for CRCW-PRAMs with *arbitrary* semantics for concurrent writes; see Sect. 2.4.1. First, note that in iteration ℓ of the main loop, all competing threads will write the same value $\ell+1$ to $d[v]$. Moreover, we do not care which node u will successfully set $parent(v):=u$. Finally, we can avoid duplicate entries in Q' by only inserting v into Q' if $parent(v)=u$. The lockstep synchronization of PRAMs will ensure that we read the right value here.

This PRAM algorithm yields the following bound.

Theorem 9.1. *On an arbitrary-CRCW-PRAM, BFS from s can be implemented to run in time*

$$\mathrm{O}\left(\frac{m+n}{p}+D\cdot(\Delta+\log p)\right),$$

where D is the largest BFS distance from s and Δ is the maximum degree of a node.

Proof. Consider iteration ℓ of the main loop, working on n_ℓ nodes in Q with m_ℓ outgoing edges. The prefix sum for load balancing takes time $\mathrm{O}(n_\ell/p+\log p)$. Each PE will be assigned at most $\lceil m_\ell/p\rceil+\Delta$ edges. Each edge can be processed in constant time. The prefix sum for finding positions in Q takes time $\mathrm{O}(\log p)$. Copying Q' to Q takes time $\mathrm{O}(m_\ell/p+\Delta)$. Summing over all iterations, we get the time bound

$$\sum_{\ell=0}^{D}\mathrm{O}\left(\frac{m_\ell}{p}+\frac{n_\ell}{p}+\Delta+\log p\right)=\mathrm{O}\left(\frac{m+n}{p}+D(\Delta+\log p)\right). \qquad \square$$

On an asynchronous shared-memory machine, a conservative solution is to use a CAS instruction to set parent pointers. $CAS(parent(v),\perp,u)$ could be used to make sure that exactly one predecessor u of v succeeds in establishing itself as the parent of v. Only when this CAS succeeds, are Q' and $d[v]$ subsequently updated. However, this is one of the rare cases where we can avoid a CAS instruction despite write conflicts. As long as the accesses to $d[v]$ and $parent(v)$ are atomic, the same discussion as for the PRAM algorithm will ensure that we get consistent values. The only complication that we cannot rule out is occasional duplicate entries in Q'. Figure 9.4 gives an example. This might incur additional work but does not render the computation incorrect. The only explicit synchronization we need is a barrier synchronization (see Sect. 13.4.2) after incrementing $\ell+1$ in the main loop.

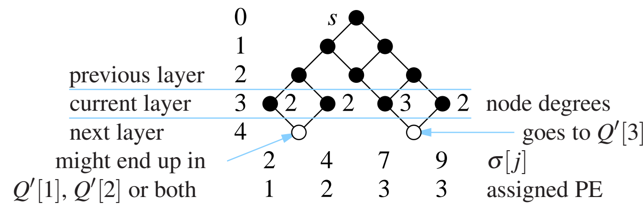


Fig. 9.4. Computations for layer 3 during a shared-memory BFS. Nodes 1 to 4 of the current layer are assigned to PEs $\lceil 2\cdot 3/9\rceil=1$, $\lceil 4\cdot 3/9\rceil=2$, $\lceil 7\cdot 4/9\rceil=3$, and $\lceil 9\cdot 3/9\rceil=3$ respectively. Since nodes 3 and 4 are handled by the same PE, the second node on the next layer goes to $Q'[3]$. Since nodes 1 and 2 are handled by distinct PEs, the first node on the next layer may end up in $Q'[1]$ or in $Q'[2]$ or in both.

9.2.2 Shared-Memory BFS Implementation

Here, we implement the algorithm presented in Sect. 9.2.1 in C++11. Listing 9.1 introduces a static graph data structure implemented as an adjacency array and a generic parallel prefix sum function. Listing 9.2 contains the shared-memory BFS implementation.

Listing 9.1. Utility functions for C++ BFS implementation

```
class Graph {                                                              1
  int n;                                                                   2
  size_t m;                                                                3
  std::vector<size_t> begin;                                               4
  std::vector<int> adj; // adjacency array representation                  5
public:                                                                    6
  vector<int>::const_iterator beginNeighbor(const int v) const {           7
    return adj.cbegin() + begin[v];                                        8
  }                                                                        9
  vector<int>::const_iterator endNeighbor(const int v) const {             10
    return adj.cbegin() + begin[v + 1];                                    11
  }                                                                        12
  int getN() const { return n; } // number of nodes                        13
  size_t getM() const { return m; } // number of edges                     14
};                                                                         15
template <class Iterator, class F, class B>                                16
void prefixSum(Iterator outBegin, Iterator outEnd, int iPE, int p,         17
    Iterator tmp, F f, B & barrier) {                                      18
  const size_t begin = (outEnd−outBegin)*iPE/p;                            19
  const size_t end = (outEnd−outBegin)*(iPE+1)/p;                          20
  size_t sum = 0, i = begin;                                               21
  for (; i != end ; ++i) *(outBegin + i) = (sum += f(i));                  22
  *(tmp + iPE) = sum;                                                      23
  barrier.wait(iPE, p);                                                    24
  size_t a = 0;                                                            25
  for(i=0; i< iPE; ++i) a += *(tmp + i);                                   26
  for(i=begin; i!=end; ++i) *(outBegin + i) += a;                          27
}//SPDX−License−Identifier: BSD−3−Clause; Copyright(c) 2018 Intel Corporation   28
```

Since we are targeting a moderate number of threads, we do not use the prefix sum algorithm presented in Sect.13.3, whose asymptotic execution time is logarithmic in p, but a simpler code with linear execution time $O(p+n/p)$ but favorable constant factors. The *beginNeighbor* function returns a pointer to the first edge out of a node. The *endNeighbor* function returns a pointer to the first edge after the last edge out of a node. For a node with no outgoing edge, both functions return the same value. We do not provide graph initialization functions in this listing, as they are trivial. The *prefixSum* function takes the output range specified by pointer iterators *outBegin* and *outEnd*, the identifier of the PE (*iPE*), the total number of PEs p, the pointer to a temporary array *tmp*, and a function object f. The use of a function object allows user code to compute values on demand without requiring them to be stored

in intermediate arrays. In lines 27–30 of Listing 9.2, we call *prefixSum* with a lambda function (line 29) that computes the outdegree of a node.

We come to the details of *prefixSum*. In lines 19 and 20, the array boundaries for *iPE*'s subrange are computed. Then the prefix sum within the local range is computed in line 22. The total sum of the range is saved in the *tmp* array (line 23). A barrier synchronization (Sect. 13.4.3) is executed in line 24 to ensure that the array *tmp* is completely filled before the PE accumulates the total sums of the PEs with smaller identifiers (lines 25 and 26) and adds the total aggregate to the local output items (line 27).

Listing 9.2 shows the bulk of the implementation. The parallel BFS routine *pBFS* accepts the total number of PEs p, the input graph g, the root of the BFS tree s, the array of output distances d, and the *parent* array. In lines 6–13, we initialize the arrays d, *parent*, Q, *Qp*, and *sigma* and/or preallocate space for them. These arrays correspond to d, *parent*, Q, Q', and σ in the abstract algorithm in Sect. 9.2.1. *Qp* is an array of arrays storing the local *Qp* for each PE. The entries of *Qp* are *padded* to have the size of a full cache line (line 2) in order to avoid false sharing; see Sect. B.3. The atomic flag *done* and the barrier *barrier* are created and initialized in lines 14–15. Then the C++ threads are created and started. Recall that the constructor of a thread takes two arguments, a worker function passed as a function argument and a second argument that is passed to the worker function by the thread; see App C. The call of the constructor extends through lines 18–65. The worker function is defined in-place (lines 19–65). The second parameter is the loop index i (line 65) so that *iPE* is set to i in the i-th thread. The loop in line 66 waits until all *worker* threads are completed.

The body of the BFS worker function is given in lines 19–64. Before entering the main loop over BFS levels l, each PE initializes its part of the output arrays d and *parent* (lines 19–23). The PE which has the root s in its range initializes the distance d and *parent* of s in line 24. The main loop extends over lines 26–63. It first computes the prefix sum σ over outdegrees of the nodes in Q (lines 27–30). After the thread barrier on the next line, the local array *Qp* is cleared. Then some memory is preallocated for it to reduce future reallocations and copying on capacity overflows (lines 32–34).

Each PE determines in lines 36–38 the range of nodes (from *curQ* to $endQ-1$) assigned to it using the *upper_bound* binary search function from the C++ standard library. Edges (u,v) out of nodes belonging to the PE are traversed with two nested *for*-loops in lines 39–51. If the PE does not see a valid *parent* identifier for node v, then it updates its *parent* with u, the distance d with l, and adds u to the local *Qp* array (lines 45–48).

After a barrier, the PE computes the position (*outPos*) in Q, where it subsequently moves its local part of *Qp* (lines 52–54). To this end, it sums the sizes of the parts of *Qp* with smaller identifiers. The last PE resizes Q and *sigma* to accommodate $|Q'|$ elements. If there is nothing left to be done, this is signaled using the *done* flag (lines 55–59). After a further barrier, the local array *Qp* is copied to the global array Q (line 61). A final barrier ensures that Q is in a consistent state before the next iteration of the outermost loop is started.

Exercise 9.9. For each of the barrier synchronizations in Listing 9.2, give an example of what could go wrong if this barrier were omitted.

Listing 9.2. Shared-memory BFS implementation in C++

```
// pad to avoid false sharing
typedef pair<vector<int>, char [64 - sizeof(vector<int>)] > PaddedVector;

void pBFS(unsigned p, const Graph & g, const int s,
                vector<int> & d, vector<int> & parent)
{
  d.resize(g.getN());
  parent.resize(g.getN());
  vector<int> Q;
  Q.reserve(g.getN());
  Q.push_back(s);
  vector<PaddedVector> Qp(p);
  vector<size_t> sigma(1), tmp(p);
  sigma.reserve(g.getN());
  atomic<bool> done(false);
  Barrier barrier(p);
  vector<thread> threads(p);
  for (unsigned i = 0; i < p; ++i)// go parallel
    threads[i] = thread([&](const unsigned iPE) { // worker function
      const size_t beginI = iPE*g.getN()/p;
      const size_t endI = (iPE + 1)*g.getN()/p;
      fill(d.begin() + beginI, d.begin() + endI,
        (numeric_limits<int>::max)());
      fill(parent.begin()+beginI, parent.begin()+endI,INVALID_NODE_ID);
      if(s >= beginI && s < endI) { d[s] = 0; parent[s] = s; }
      int l = 1;
      for(; !done; ++l) {
        prefixSum(sigma.begin(), sigma.end(), iPE, p, tmp.begin(),
            /* a lambda function */ [&] (int j)
            { return g.endNeighbor(Q[j]) - g.beginNeighbor(Q[j]);},
            barrier);
        barrier.wait(iPE, p);
        Qp[iPE].first.clear();
        size_t ml = sigma.back();
        Qp[iPE].first.reserve(2*ml/p); // preallocate memory
        size_t curQ = upper_bound(sigma.cbegin(), sigma.cend(),
                                  iPE*ml/p) - sigma.cbegin();
        const size_t endQ = upper_bound(sigma.cbegin(), sigma.cend(),
              (iPE+1)*ml/p) - sigma.cbegin();
        for (; curQ != endQ; ++curQ) { // loop over nodes
          const int u = Q[curQ];
          auto vIter = g.beginNeighbor(u);
          auto vEnd = g.endNeighbor(u);
          for (; vIter != vEnd; ++vIter) { // loop over edges
```

```
          const int v = *vIter; // target of current edge                    44
          if(parent[v] == INVALID_NODE_ID) { // not visited yet              45
            parent[v] = u;                                                   46
            d[v] = l;                                                        47
            Qp[iPE].first.push_back(v); // queue for next layer              48
          }                                                                  49
        }                                                                    50
      }                                                                      51
      barrier.wait(iPE, p);                                                  52
      size_t outPos = 0;                                                     53
      for(int j=0; j < iPE; ++j) outPos += Qp[j].first.size();               54
      if(iPE == p-1) {                                                       55
        Q.resize(outPos + Qp[iPE].first.size());                             56
        sigma.resize(Q.size());                                              57
        if (Q.empty()) done = true;                                          58
      }                                                                      59
      barrier.wait(iPE, p);                                                  60
      copy(Qp[iPE].first.cbegin(), Qp[iPE].first.cend(),                     61
        Q.begin() + outPos);                                                 62
      barrier.wait(iPE, p);                                                  63
    }                                                                        64
  }, i);                                                                     65
  for (auto & t : threads) t.join();                                         66
}//SPDX-License-Identifier: BSD-3-Clause; Copyright(c) 2018 Intel Corporation 67
```

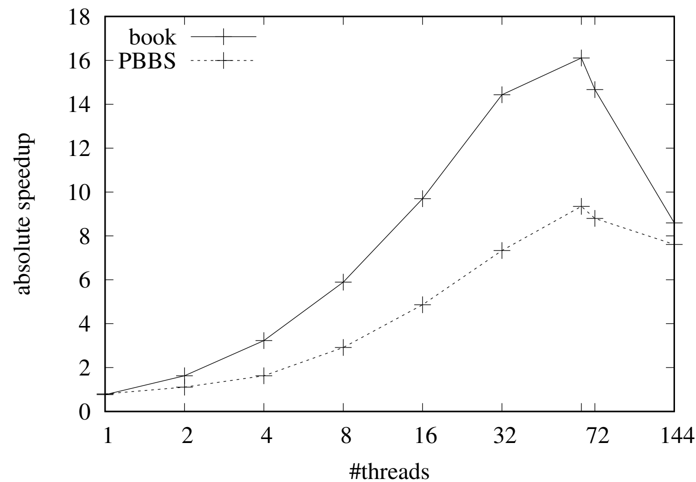


Fig. 9.5. Speedup of parallel BFS implementations over sequential BFS for the grid graph.

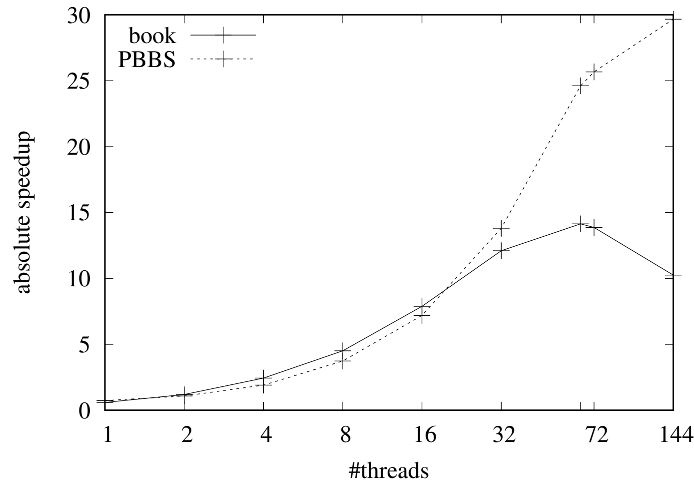


Fig. 9.6. Speedup of parallel BFS implementations over sequential BFS for the R-MAT graph

We measured the speedup of this implementation on the four-socket machine described in Appendix B and compared it with the implementation from the Problem Based Benchmark Suite (PBBS) [296]. Figures 9.5 and 9.6 show the speedup as a function of the number of threads for two graphs with $n = 10^7$ nodes – a three-dimensional grid and the R-MAT recursive matrix graph [63] with parameters $(.3,.1,.1,.5)$ (about $5n$ edges). Our implementation outperforms PBBS for the grid instance. For the R-MAT instance[2], our code is slightly better than PBBS for up to 16 threads, but does not scale to the full number of 72 available cores and gets even worse for 144 threads (using hyper-threading). The main difference between the two implementations is that we use static load balancing based on prefix sums, whereas PBBS uses the dynamic load balancing integrated into Cilk. This implementation difference may explain the performance difference. Static load balancing has less overhead and hence works well on instances with small average queue size such as the grid instance. Its layers are of size $O(\sqrt{n})$. The R-MAT graph has a much smaller diameter and hence a larger average queue size.

9.2.3 Distributed-Memory BFS

Our design for BFS on distributed-memory machines is based on the "owner computes" principle. We assign each node u of the graph to a PE $u.p$, which does the work related to node u. If u is in the set Q of nodes to be scanned, PE $u.p$ is responsible for doing this. However, when scanning edge (u,v), PE $i = u.p$ delegates this to PE $j = v.p$ since node v is the object actually affected. Hence, PE i sends the edge (u,v) to PE j. Figure 9.7 gives pseudocode. Of course, an efficient algorithm will

[2] The instance "local random" from [296] exhibited similar behavior.

handle local edges (u,v) with $u.p = v.p$ without explicit communication and it will deliver the messages in bulk fashion – gathering all local messages destined for the same PE in a message buffer and delivering this buffer as a whole or in large chunks. For example, a library based on the BSP model can be used here.

```
Function dBFS(s : NodeId) : (NodeArray of 0..n) × (NodeArray of NodeId)
  d = ⟨∞,...,∞⟩ : NodeArray of 0 .. n                              // distance from root
  parent = ⟨⊥,...,⊥⟩ : NodeArray of NodeId
  Q = ⟨⟩ : Set of NodeId                                        // current layer of BFS tree
  if s.p = i_proc then d[s] := 0; parent[s] := s; Q := ⟨s⟩
  for ℓ := 0 to ∞ while ∃i : Q@i ≠ ⟨⟩ do                          // explore layer by layer
    invariant Q contains all local nodes with distance ℓ from s
    foreach u ∈ Q do
      foreach (u,v) ∈ E do                                          // scan edges out of u
          post message (u,v) to PE v.p
    deliver all messages
    Q := {}
    foreach received message (u,v) do
      if parent(v) = ⊥ then                                     // found an unexplored node
        parent(v) := u                                                  // update BFS tree
        d[v] := ℓ + 1
        Q := Q ∪ {v}                                              // remember for next layer
  return (d, parent)                  // the BFS tree is now {(v,w) : w ∈ V, v = parent(w)}
```

Fig. 9.7. Distributed-memory BFS starting at a node s

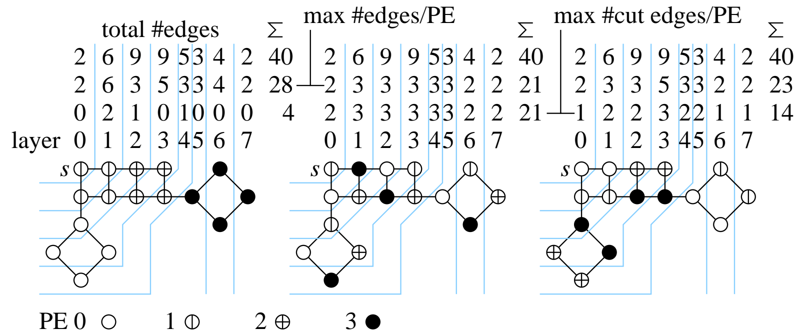


Fig. 9.8. Distributed-memory BFS with three different partitions. *Left*: four compact blocks – bad load balance but low communication. *Middle*: scattered nodes – good load balance but high communication. *Right*: eight scattered blocks of size two – a compromise. The light lines indicate the partition of the graph into layers. The top row of numbers shows the number of edges incident to the nodes of the current layer, the second row shows the maximum number of edges that have to be handled by a single PE, and the third row shows the maximum number of cut edges that have to be handled by a single PE. The column Σ contains the row sums.

Note that we are facing a trade-off between communication overhead and load balance here. Using a graph partition with a small overall cut as proposed in Sect. 8.6 will result in a small overall communication volume. However, in graphs with high locality, the BFS will initially run just on PE $s.p$, spreading only slowly over further PEs. On the other hand, assigning nodes to random PEs will destroy locality, but parallelism will spread very fast and the work will be well balanced. Perhaps compromises are in order here – working with a partition with $k \gg p$ pieces spread randomly over the PEs. Figure 9.8 illustrates this trade-off.

9.2.4 Handling High-Degree Nodes

We apply the general approach described on page 266. We use the same prefix sum σ over outdegrees for load balancing as in Sect. 9.2.1. However, we use it to actually distribute edges. The kth edge of node $Q[i]$ gets the number $\sigma[j-1]+k$. Edge i will be processed by PE $\lceil ip/m_\ell \rceil$. This means that PE j performs binary search for $k_j := jm_l/p$ in σ. So suppose $\sigma[i] \le k_j < \sigma[i+1]$. Then PE j starts scanning edges at edge $k_j - \sigma[i]$ of node i.

Exercise 9.10. Show that with the above modification of the algorithm, you can sharpen Theorem 9.1 to obtain running time

$$\mathrm{O}\left(\frac{m+n}{p} + D\log p\right).$$

9.3 Depth-First Search

You may view breadth-first search as a careful, conservative strategy for systematic exploration that completely inspects 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 to go forward. Although DFS leads to unbalanced 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 9.9 gives an algorithm template for DFS. We can derive specific algorithms from it by specifying the subroutines *init*, *root*, *traverseTreeEdge*, *traverseNonTreeEdge*, and *backtrack*.

DFS uses node marks. Initially, all nodes are unmarked. Nodes are marked *active* when they are discovered and their exploration begins. Once the exploration of a node is completed, the mark is changed to *completed* and keeps this value until the end of the execution. The main loop of DFS looks for unmarked nodes s and calls $\mathit{DFS}(s,s)$ to grow a tree rooted at s. The recursive call $\mathit{DFS}(u,v)$ organizes the exploration 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 $\mathit{DFS}(*,v)$ if the specific nature of the incoming edge is irrelevant to the discussion at hand.

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

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

Fig. 9.9. 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.

The call $DFS(*, v)$ first marks v as active and then inspects all edges (v, w) out of v. Assume now that we are exploring edge (v, w) with end node w. If w has been seen before (w is marked as either active or completed), 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 (w is unmarked), (v, w) becomes a tree edge. We therefore call *traverseTreeEdge*(v, w) and make the recursive call of $DFS(v, w)$. When we return from this call, we explore the next edge out of v. Once all edges out of v have been explored, the procedure *backtrack*(u, v) is called, where (u, v) is the edge in the call *DFS* for v; it performs summarizing and cleanup work. We then change the mark of v to "completed" 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, precisely the nodes v_1 to v_k are marked *active* and the recursion stack contains the sequence $\langle (v_1, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k) \rangle$. More compactly, we say that the recursion stack contains $\langle v_1, \ldots, v_k \rangle$. A node is called active (or completed) if it is marked active (or completed, respectively). The node v_k is called the *current node*. We say that a node v has been *reached* when $DFS(*, v)$ has already been called. So the reached nodes are the active and the completed nodes.

Exercise 9.11. Give a nonrecursive formulation of DFS. There are two natural realizations. One maintains the stack of active nodes and, for each active node, the set of unexplored edges (it suffices to keep a pointer into the list or array of outgoing edges of the active node). The other maintains a stack of all unexplored edges emanating from active nodes. When a node is activated, all its outgoing edges are pushed onto this stack.

9.3.1 DFS Numbering, Completion 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 completed (array *compNum*). We have two counters, *dfsPos* and *compPos*, both initialized to 1. When we encounter a new root or traverse a tree edge, we set the *dfsNum* of the newly encountered node and increment *dfsPos*. When we backtrack from a node, we set its *compNum* and increment *compPos*. We use the following subroutines:

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

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

$$u \prec v \Leftrightarrow \mathit{dfsNum}[u] < \mathit{dfsNum}[v].$$

The numberings *dfsNum* and *compNum* 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. They can also be used to encode the node marks during the execution of DFS as follows. We use *init* to initialize *dfsNum* and *compNum* with the all-zero vector. Then a node is unmarked if and only if its *dfsNum* is equal to 0. It is active if and only if its *dfsnum* is positive and *compNum* is 0. The node is completed if and only if its *compNum* is positive.

Lemma 9.2. *The nodes on the DFS recursion stack are ordered with respect to* $\prec$.

Proof. *dfsPos* is incremented after every assignment to *dfsNum*. Thus, when a node v is made active by a call $\mathit{DFS}(u,v)$ and is put on the top of the recursion stack, it has just been assigned the largest *dfsNum* so far. □

*dfsNum*s and *compNum*s classify edges according to their type, as shown in Table 9.1. The argument is as follows. We first observe that 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 second starts, the first is already completed. If $\mathit{DFS}(*,w)$ is nested in $\mathit{DFS}(*,v)$, the former call starts after the latter and finishes before it, i.e., $\mathit{dfsNum}[v] < \mathit{dfsNum}[w]$ and $\mathit{compNum}[w] < \mathit{compNum}[v]$. If $\mathit{DFS}(*,w)$ and $\mathit{DFS}(*,v)$ are disjoint and the former call starts before the latter, it also ends before the latter, i.e., $\mathit{dfsNum}[w] < \mathit{dfsNum}[v]$ and $\mathit{compNum}[w] < \mathit{compNum}[v]$.

Next we observe that the tree edges record the nesting structure of recursive calls. When a tree edge (v,w) is explored within $\mathit{DFS}(*,v)$, the call $\mathit{DFS}(v,w)$ is made and hence is nested within $\mathit{DFS}(*,v)$. Thus w has a larger DFS number and a smaller completion number than v. A forward edge (v,w) runs parallel to a path of tree edges

Table 9.1. The classification of an edge (v,w). The last column indicates the mark of w at the time when the edge (v,w) is explored.

type	$dfsNum[v] < dfsNum[w]$	$compNum[w] < compNum[v]$	Mark of w
tree	Yes	Yes	unmarked
forward	Yes	Yes	completed
backward	No	No	active
cross	No	Yes	completed

and hence w has a larger DFS number and a smaller completion number than v. We can distinguish tree and forward edges by the mark of w at the time when the edge (v,w) is inspected. If w is unmarked, the edge is a tree edge. If w is already marked, the edge is a forward edge. In the case of a forward edge, w is marked as completed; if w were active, it would be part of the recursion stack. Since v is the topmost node of the recursion stack, this would imply $dfsNum(w) < dfsNum(v)$, a contradiction.

A backward edge (v,w) runs antiparallel to a path of tree edges, and hence w has a smaller DFS number and a larger completion number than v. Furthermore, when the edge (v,w) is inspected, the call $DFS(*,v)$ is active and hence, by the nesting structure, so is the call $DFS(*,w)$. Thus w is active when the edge (v,w) is inspected.

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 completed 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 completed before $DFS(*,v)$ starts and hence $DFS(*,w)$ starts and ends before $DFS(*,v)$. Thus $dfsNum[w] < dfsNum[v]$ and $compNum[w] < compNum[v]$. The following lemma summarizes this discussion.

Lemma 9.3. *Table 9.1 shows the characterization of edge types in terms of dfsNum and compNum and the mark of the endpoint at the time of the inspection of the edge.*

Completion numbers have an interesting property for directed acyclic graphs.

Lemma 9.4. *The following properties are equivalent:*

(a) G is a DAG;
(b) DFS on G produces no backward edges;
(c) all edges of G go from larger to smaller completion numbers.

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. This shows that (a) implies (b). All edges except backward edges run from larger to smaller completion numbers, according to Table 9.1. This shows that (b) implies (c). Finally, assume that all edges run from larger to smaller completion numbers. In this case the graph is clearly acyclic. This shows that (b) implies (a). □

An order of the nodes of a DAG in which all edges go from earlier to later nodes is called a *topological sorting*. By Lemma 9.4, the ordering by decreasing completion

number 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 9.12. Modify DFS such that it labels the edges with their type.

Exercise 9.13 (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 9.14. Design a BFS-based algorithm for topological sorting.

Exercise 9.15. In a DFS on an undirected graph, it is convenient to explore edges in only one direction. When an undirected edge $\{v,w\}$ is inspected for the first time, say in the direction from v to w, it is "turned off" in the adjacency list of w and not explored in the opposite direction. Show that DFS (with this modification) on an undirected graph does not produce any cross edges or forward edges.

9.3.2 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 if and only if 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 9.7 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. Figure 9.10 shows a graph and its strongly connected components. It also illustrates the concept of the *shrunken graph* G^s corresponding to a directed graph G, which will turn out to be extremely useful for this section. The nodes of G^s are the SCCs of G. If C and D are distinct SCCs of G, we have an edge (C,D) in G^s if and only if there is an edge (u,v) in G with $u \in C$ and $v \in D$.

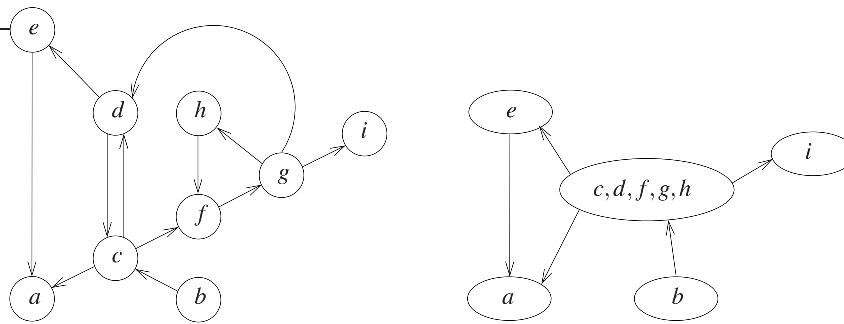


Fig. 9.10. A digraph G and the corresponding shrunken graph G^s. The SCCs of G have node sets $\{a\}$, $\{b\}$, $\{c,d,f,g,h\}$, $\{e\}$, and $\{i\}$.

Exercise 9.16. 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.

Lemma 9.5. *The shrunken graph G^s with respect to a digraph G 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^s. Recall that the C_i's are SCCs of G. By the definition of G^s, G contains an edge (v_i, w_{i+1}) with $v_i \in C_i$ and $w_{i+1} \in C_{i+1}$ for $0 \le i < k$. Define $v_k = v_1$. Since C_i is strongly connected, G contains a path from w_{i+1} to v_{i+1}, $0 \le i < k$. Thus all the v_i's belong to the same SCC, a contradiction. □

We shall show that the strongly connected components of a digraph G can be computed using DFS in linear time $\mathrm{O}(n+m)$. More precisely, the algorithm outputs an array *component* indexed by nodes such that $\mathit{component}[v] = \mathit{component}[w]$ if and only if v and w belong to the same SCC. Alternatively, it could output the node set of each SCC.

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. We use G_c^s to denote the shrunken graph of G_c.

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. Then 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 easy to obtain. Consider a depth-first search on G, let E_c be the set of edges already explored by DFS, and let $G_c = (V, E_c)$ be the current graph. Recall that a node is either unmarked, active, or completed. We distinguish between three kinds of SCCs of G_c: unreached, open, and closed. Unmarked nodes have indegree and outdegree 0 in G_c and hence form SCCs consisting of a single node. The corresponding node in the shrunken graph is isolated. 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 completed 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 completed and that an open node may be either active or completed. For every SCC, we call the node with the smallest DFS number in the SCC the *representative* of the SCC. Figure 9.11 illustrates these concepts. We next state some important invariant properties of G_c; see also Fig. 9.12:

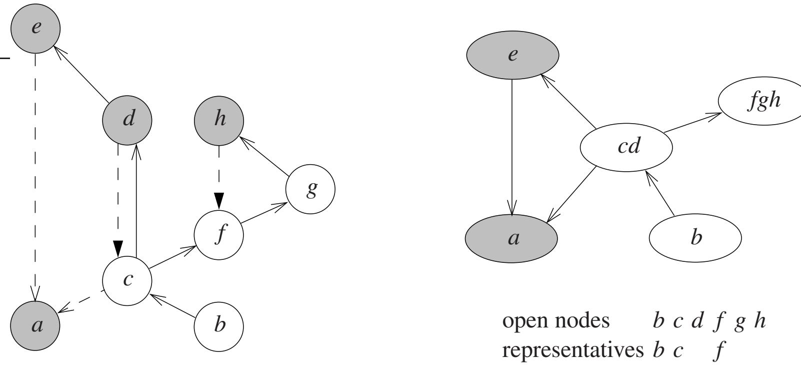


Fig. 9.11. A snapshot of DFS on the graph in Fig. 9.10 and the corresponding shrunken graph. The first DFS was started at node a and a second DFS was started at node b, the current node is g, and the recursion stack contains b, c, f, g. The edges (g,i) and (g,d) have not been explored yet. Edges (h,f) 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.

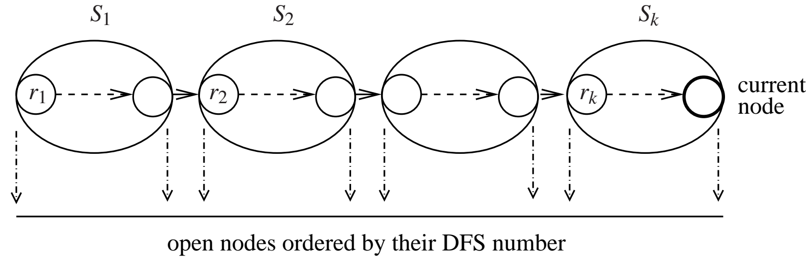


Fig. 9.12. The open SCCs are shown as ovals, and the current node is shown as a **bold** circle. The tree path to the current node is indicated. It enters each component at its representative. The horizontal line below represents the open nodes, ordered by *dfsNum*. Each open SCC forms a contiguous subsequence, with its representative as its leftmost element.

(a) 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.

(b) 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 in G_c 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 $1 \le i \le j \le k$. 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.

(c) Consider the nodes in the 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 9.6. *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 invariant (a) implies that all vertices of C are closed. Since closed vertices are completed, all edges out of them have been explored. Thus C is contained in G_c, and hence $w \in S_c$. □

The invariants (b) and (c) 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 number, 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 see later that both sequences are best kept as a stack (type *Stack* **of** *NodeId*).

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. The sequences *oNodes* and *oReps* are empty.

Just before a new root is to be marked, i.e., the construction of a new DFS tree is started, all marked nodes are completed 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 is marked as active, $\{w\}$ becomes an open component on its own. All other open components are unchanged. The invariant (a) 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 invariant (b) is maintained. Also, w becomes the open node with

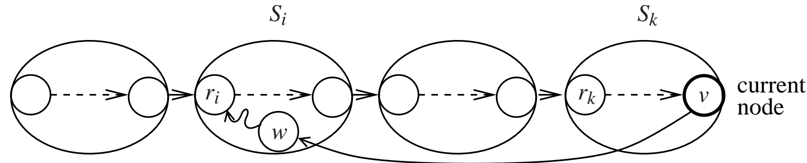


Fig. 9.13. The open SCCs are shown as ovals and their representatives as circles on the left side of the oval. All representatives lie on the tree path to the current node v. The nontree edge $e = (v, w)$ ends in an open SCC S_i with representative r_i. There is a path from w to r_i since w belongs to the SCC with representative r_i. Thus edge (v, w) merges S_i to S_k into a single SCC.

the largest DFS number and hence *oNodes* and *oReps* are both extended by w. Thus the invariant (c) 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 9.6, 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 that the SCCs S_i to S_k are merged into a single component and all other components are unchanged; see Fig. 9.13. Let r_i be the representative of S_i. We can then go from r_i to v along a tree path by invariant (b), 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 invariant (c) 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 completing a node v. When will this close an SCC? Completion of v will close the the SCC containing it if and only if v was the only remaining active node in the SCC. By invariant (b), 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 completed in the component. In other words, we close a component if and only if we complete its representative. Since *oReps* is ordered by *dfsNum*, we close a component if and only if the last node of *oReps* completes. So, assume that we complete a representative v. Then, by invariant (c), the component S_k with representative $v = r_k$ consists of v and all nodes in *oNodes* following v. Completing v closes S_k. By invariant (a), there is no edge out of S_k into an open component. Thus invariant (a) holds after S_k is closed. If $k = 1$, the exploration of the entire DFS tree is completed and invariants (b) and (c) clearly hold. If $k \geq 2$, the new current node is the parent of v. By invariant (b), the parent of

v lies in S_{k-1}. Thus invariant (b) holds after S_k is closed. Invariant (c) holds after v is removed from *oReps*, and v and all nodes following it are removed from *oNodes*.

```
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 ∈ oNodes then
        while w ≺ 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. 9.14. An instantiation of the DFS template that computes strongly connected components of a graph $G = (V,E)$

It is now easy to instantiate the DFS template. Figure 9.14 shows the pseudocode, and Fig. 9.15 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 all representatives from *oReps* having a larger DFS number than w. An SCC S is closed when its representative v finishes. At that point, all nodes of S are stored 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. 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. 9.5. 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.

Theorem 9.7. *The algorithm in Fig. 9.14 computes strongly connected components in time* $O(m+n)$.

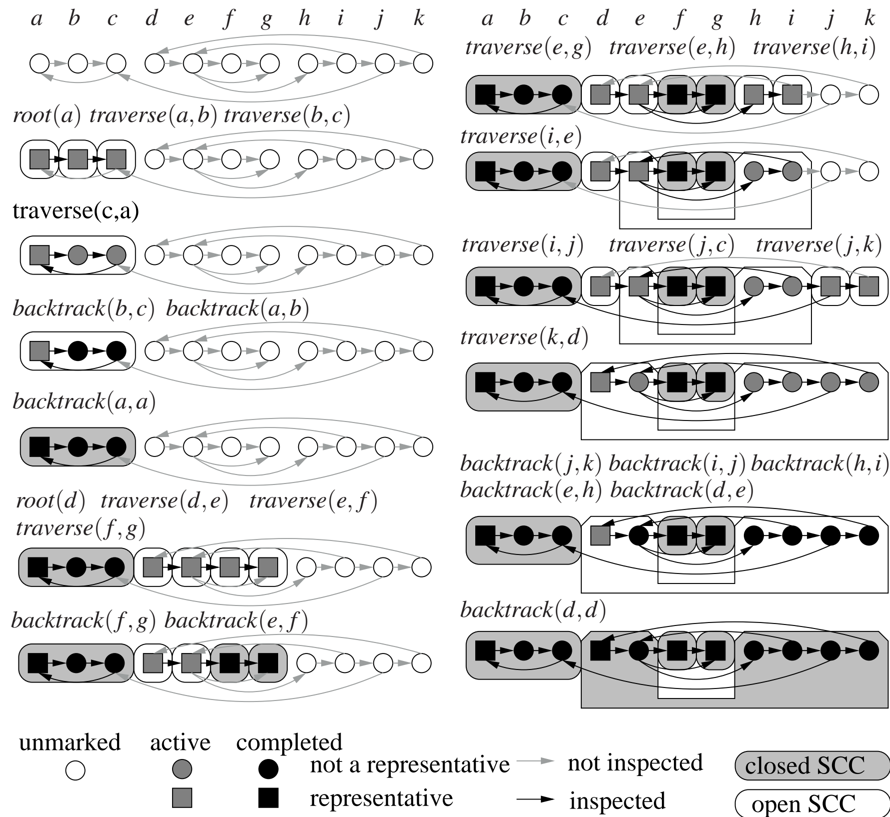


Fig. 9.15. An example of the development of open and closed SCCs during DFS. Unmarked nodes are shown as empty circles, active nodes are shown in gray, and completed nodes are shown in black. Nontraversed edges are shown in gray, and traversed edges are shown in black. Open SCCs are shown as unfilled closed curves, and closed SCCs are shaded gray. Representatives are shown as squares and nonrepresentatives are shown as circles. 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 and then to a, and finally complete a. At this point, the component becomes closed. Exercise: Please complete the description.

Exercise 9.17 (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 that s can be reached from all nodes. Can you modify the SCC algorithm so that it constructs both trees?

Exercise 9.18 (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. 9.16. Modify the SCC algorithm shown in Fig. 9.14 so that it computes 2-

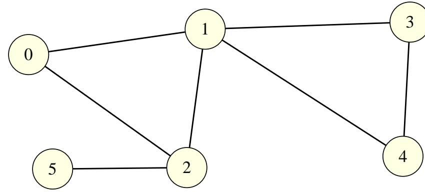


Fig. 9.16. The graph has two 2-edge-connected components, namely $\{0,1,2,3,4\}$ and $\{5\}$. The graph has three biconnected components, namely the subgraphs spanned by the sets $\{0,1,2\}$, $\{1,3,4\}$, and $\{2,5\}$. The vertices 1 and 2 are articulation points.

edge-connected components. Hint: Use the fact that DFS of an undirected graph does not produce either forward or cross edges (Exercise 9.15).

Exercise 9.19 (biconnected components). An *articulation point* in an undirected graph is a node whose removal disconnects the graph. An undirected graph without an articulation point is called *biconnected*. Two trivial cases are a single node and two nodes connected by an edge. Show that a graph with more than two nodes is biconnected if and only if every pair of distinct nodes is connected by two node-disjoint paths; see Fig. 9.16. A *biconnected component* (BCC) of an undirected graph is a maximal biconnected subgraph. Biconnected components are pairwise edge-disjoint. They may share nodes. The nodes that belong to more than one BCC are precisely the articulation points. Design an algorithm that computes the biconnected components of an undirected graph 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.

***Exercise 9.20 (open ear decomposition of biconnected graphs).** An open ear decomposition of a graph is a sequence of paths $P_0, \ldots, P_k$ with the following properties: P_0 is a simple cycle and each P_i is a path whose endpoints lie on one of the preceding paths, whose endpoints are distinct, and which is internally disjoint from the preceding paths.

(a) Show: If a graph has an open ear decomposition, it is biconnected.
(b) Show: Every biconnected graph has an open ear decomposition. Hint: Consider a DFS on a biconnected graph. Decompose G into a set of paths $P_0, P_1, \ldots$ as follows. First consider any back edge (u,s) ending in the root s of the DFS tree. The first path P_0 consists of the back edge (u,s) plus the tree path from s to u. To construct P_i, choose a back edge (u,v) where v lies on $V_{i-1} := \cup_{j<i} P_i$ and then trace back tree edges from u until a node in V_{i-1} is encountered. In what order should one consider the back edges so that an open ear decomposition results?

9.4 Parallel Traversal of DAGs

In this section we describe a simple parallel algorithm for traversing the nodes of directed acyclic graphs (DAGs) in topological order. This immediately yields an algorithm for topological sorting but also serves as an algorithm template for other graph problems, including maximal independent sets and graph coloring. Indeed, the template can be viewed as a way to parallelize a class of greedy graph algorithms.

We have already seen a sequential algorithm for topological sorting in Sect. 2.12 and Exercise 8.3. We maintain the current indegree of each node and initialize a set with the nodes of indegree 0. Repeatedly, we remove all nodes in the set and their outgoing edges from the graph and add the nodes whose indegree becomes 0 to the set. This algorithm performs $D+1$ iterations, where D is the length of the longest path in the network. Each iteration can be parallelized. We describe the distributed-memory version of the algorithm. Figure 9.17 gives pseudocode. As in parallel BFS (Sect. 9.2.3), we maintain a local array Q of local nodes that are ready to be processed. For DAG traversal, this means that they have indegree 0. Each PE iterates through the nodes u in its part of Q and sends messages to the PEs responsible for handling the nodes v reached by the edges out of u. Using prefix sums over the size of Q on each PE, we can assign unique numbers to the nodes which overall form a topological sorting[3] of the nodes of V. After all messages have been delivered, the incoming messages are processed. For topological sorting, the only thing that needs to be done for a message (u,v) is to decrement the indegree counter $\delta^-[v]$, and, if it has dropped to 0, to put v into Q.

Our DAG traversal algorithm can be generalized into an algorithm template whose main abstraction is sending messages along edges. More concretely, we get basic subroutines for initialization, sending messages, receiving a message, and processing the last message to a node. This is quite elegant because it completely abstracts from the parallel machine architecture used. Indeed, the basic approach was originally invented for external-memory graph algorithms, where it is known as time forward processing [72]. Using external-memory priority queues (see also Sect. 6.3) for message delivery, time forward processing yields algorithms with sorting complexity. The parallel complexity of the algorithm is a more complicated issue. To keep matters simple, we just consider a simple case in an exercise.

Exercise 9.21. Let Δ denote the maximum degree and D the length of the longest path in a DAG. Explain how to do topological sorting in time

$$\mathrm{O}\left(\frac{m+n}{p}+D(\Delta+\log n)\right)$$

on a CRCW-PRAM model that allows fetch-and-decrement in constant time.

[3] Note that in each iteration, we process nodes that have the same length of the longest path from a source node of the graph. Thus there can be no edges between them. Hence, the relative numbering of these nodes is arbitrary.

```
Function traverseDAG                                   // let V denote the set of local nodes.
  δ⁻ = ⟨indegree(v) : v ∈ V⟩ : NodeArray of ℕ₀
  topOrder : NodeArray of ℕ
  Q = ⟨v ∈ V : δ⁻[v] = 0⟩ : Set of NodeId                 // and further initializations
  for (pos:=0;   ∑_i |Q|@i > 0;)                            // explore layer by layer
    offset := pos + ∑_{i<i_proc} |Q|@i      // offset for local PE numbers using prefix sums
    foreach u ∈ Q do
      topOrder[u] := ++offset
      foreach (u,v) ∈ E do post message (u,v) to PE v.p
  pos+=∑_i |Q|@i                                            // advance to next layer
  deliver all messages
  Q := {}
  foreach received message (u,v) do                         // process message
    if −−δ⁻[v] = 0 then                                     // process last message to v
      Q := Q ∪ {v}                                          // remember for next layer
return topOrder
```

Fig. 9.17. Topological sorting using SPMD distributed-memory traversal of DAGs

We now give instantiations of the parallel DAG traversal template for two additional basic problems on *undirected graphs* – maximal independent sets and graph coloring. Another example (shortest paths) can be found in Sect. 10.2. The basic trick is to convert the undirected graph to a DAG by choosing an ordering of the nodes and then to convert an undirected edge $\{u,v\}$ to the directed edge $(\min(u,v), \max(u,v))$. In some cases it is a good idea to choose a random ordering of the nodes, since this keeps the longest path length D short, at least for graphs with small node degrees.

Lemma 9.8. *The DAG G' resulting from an undirected graph G with maximum degree Δ and a random ordering of the nodes has expected maximum path length $D = \mathrm{O}(\Delta + \log n)$.*

Proof. There are fewer than $n\Delta^\ell$ simple paths of length ℓ in G; there are n choices for the first node of the path and at most Δ choices for each subsequent node. The probability that any particular one of these paths becomes a path in the DAG is $1/\ell!$. Hence, the probability that there is any path of length ℓ in G' is at most

$$p_\ell = n\frac{\Delta^\ell}{\ell!} \le n\left(\frac{\Delta e}{\ell}\right)^\ell.$$

This estimate uses (A.9). We show that this probability is small for $\ell = \Omega(\Delta + \log n)$. First note that $(\Delta e/\ell)^\ell$ decreases with growing ℓ for $\ell \ge \Delta$. This holds since

$$\frac{d}{d\ell}\ln\left(\frac{\Delta e}{\ell}\right)^\ell = \frac{d}{d\ell}(\ell(\ln(\Delta e) - \ln \ell)) = \ln(\Delta e) - \ln \ell - 1 = \ln \Delta - \ln \ell.$$

For $\ell \ge \ell_0 = 5\max(\Delta, \ln n)$, we obtain

$$\left(\frac{\ell}{\Delta e}\right)^{\ell} \geq \left(\frac{\ell_0}{\Delta e}\right)^{\ell_0} \geq \left(\frac{5\Delta}{\Delta e}\right)^{5\ln n} \geq \left(\frac{5}{e}\right)^{5\ln n} = e^{\ln n \ln\left(\frac{5}{e}\right)^5} = n^{3.047\ldots}.$$

Hence, for $\ell \geq \ell_0$, $p_\ell \leq n \cdot 1/n^{3.047\ldots} = n^{-2.047\ldots}$. We can now bound the expectation of D, namely,

$$\mathrm{E}[D] \leq \ell_0 - 1 + \sum_{\ell_0 \leq \ell \leq n} p_\ell \ell \leq \ell_0 - 1 + n \cdot n^{-2.047\ldots} \leq \ell_0 - 1 + 1 = \ell_0,$$

where the last inequality holds for sufficiently large n. □

Figure 9.18 gives an example. Note that the undirected graph on the left has a diameter of 8; the longest path is from node 4 to node b and has eight edges. In constrast, the longest path in the DAG obtained by randomly numbering the nodes is fairly short – only three edges.

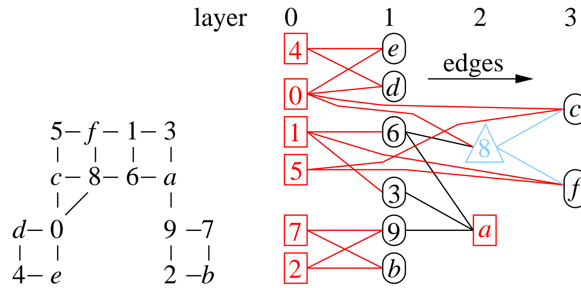


Fig. 9.18. An undirected graph with 16 nodes with randomly assigned node IDs. The DAG resulting from this node ordering ($0 < 1 < \ldots < 9 < a < \ldots < f$) has longest path length $D = 3$. The shapes of the boxes around the nodes on the right hand side indicate their color when the DAG is used to color the nodes; see Sect. 9.4.2.

9.4.1 Maximal Independent Sets

An independent set $I \subseteq V$ of an undirected graph $G = (V, E)$ is a set of nodes with no edges between them, i.e., $E \cap \binom{I}{2} = \emptyset$. Independent sets are an important concept in graph theory and are immediately relevant to parallel processing since several PEs can concurrently update the vertices of an independent set while being sure that the neighbors of the nodes in I will not change.

The following instantiation of the DAG traversal template yields an independent set I. We represent I by an array I of Boolean values and initialize it to 1 everywhere. At the end, $I[v]$ is true if and only if v is in the independent set. The only information we need to pass along an edge (u, v) is whether u is in I. A node v remains in I only if there is no edge (u, v) with $u \in I$. The following pseudocode summarizes this instantiation:

init: $I = \langle 1, \ldots, 1 \rangle$: *NodeArray* **of** $\{0,1\}$
message sent for edge (u,v)*:* $I[u]$
on receiving message x to v: $I[v] := I[v] \wedge x$

In the example graph in Fig. 9.18, nodes 0, 1, 2, 4, 5, and 7 are processed first and are all put into the independent set. Nodes 3, 6, 9, *b*, *d*, and *e* in layer 1, and nodes 8, *c*, and *f* in layers 2 and 3 receive messages from these nodes and hence stay out of the independent set. However, node *a* in layer 2 becomes part of the independent set since it receives no message – all its neighbors stay out of the independent set.

9.4.2 Graph Coloring

Recall, that graph coloring asks for colors to be assigned to nodes such that no two neighboring nodes have the same color; see also Sect. 12.5.2.1. A greedy heuristic for this task is only slightly more complicated than the one for computing a maximal independent set. We encode colors as positive integers. Nodes send their color along outgoing edges. Each node chooses the smallest color that is not already taken by one of its predecessors in the DAG. The following pseudocode shows how to implement the heuristic efficiently. Each node v pushes received messages onto a stack $S[v]$. Note that this is possible in constant time per incoming edge. When all messages have been received, we find the smallest color not in the stack $S[v]$ in time $O(|S[v]|)$. We do this using an auxiliary array *used*, recording the colors in $S[v]$. Since $S[v]$ can contain very large colors but we are guaranteed to find a free color in $1..|S[v]|+1$, it suffices to record colors less than or equal to $|S[v]|+1$:

init: c : *NodeArray* **of** $\mathbb{N}$*;* S : *NodeArray* **of** *Stack* **of** $\mathbb{N}$
  foreach $v \in Q$ **do** $c[v] := 1$
message sent for edge (u,v)*:* $c[v]$
on receiving message x to v: $S[v].push(x)$
postprocess messages to v: $used = \langle 0, \ldots, 0 \rangle$: $Array[1..|S[v]|+1]$ **of** $\{0,1\}$
  while $S[v] \neq \emptyset$ **do** $x := S.pop;$ **if** $x \leq |used|$ **then** $used[x] := 1$
  $c[v] := \min\{i \in 1..|used| : \neg used[i]\}$

Figure 9.18 gives an example. Nodes 0, 1, 2, 4, 5, and 7 have indegree 0 in the DAG and thus set their color to 1. The nodes in the subsequent layer, 3, 6, 9, *b*, *d*, and *e* receive only messages that color 1 is taken. Hence, color 2 is the first free color for all of them. Node *a* receives messages only from nodes 3, 6, and 9, which all have color 2. Hence, color 1 is the first free color for node *a*. In contrast, node 8 receives color 1 from node 0 and color 2 from node 6. Hence, its first free color is 3. Finally, nodes *c* and *f* receive colors 1 and 3, so that their first free color is 2.

Exercise 9.22. Show that this heuristic ensures that $\Delta + 1$ colors suffice where Δ is the maximum degree of the graph.

9.5 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 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 shall 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 $u \prec v$ if and only if $component[u] > component[v]$. This works because "$\prec$" is never applied to closed nodes. Finally, the test $w \in oNodes$ 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 optimizations on the performance of our SCC algorithm is discussed in [218].

9.5.1 C++

LEDA [194] 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 [50] and the LEMON graph library [200] use 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 and LEMON.

9.5.2 Java

The JGraphT [166] library supports DFS in a very flexible way, not very much different from the visitor concept described for Boost and LEMON. There are also more specialized algorithms, for example for biconnected components.

9.6 Historical Notes and Further Findings

BFS and DFS were known before the age of computers. Tarjan [305] 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 [70] and later rediscovered by Gabow [118]. Yet another linear-time SCC algorithm is that due to Kosaraju and Sharir [292]. 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 [263] and memory hierarchies [214, 228]. Therefore alternative ways to solve seemingly simple problems are an interesting area of research. For example, in Sect. 11.9 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 [309]. The DFS-based algorithms for biconnected components and strongly connected components are almost identical. But this analogy completely disappears for advanced models of computation. Thus, parallel algorithms for strongly connected components remain an area of intensive (and sometimes frustrating) research (e.g., [104, 154]). More generally, it seems that problems for undirected graphs (such as finding biconnected components) are easier to solve than analogous problems for directed graphs (such as finding strongly connected components).

Parallel BFS has become a very popular benchmark for graph processing; see `graph500.org/`. Amazing performance values have been achieved by exploiting rather special properties of the benchmark graphs. In particular, most of the work is done in a very small number of layers.