Lecture 14: Elementary Graph Algorithms

MCS/CS 401 Computer Algorithms I — July 20, 2022
Instructor: Danko Adrovic

Lecture Outline

- Elementary Graph Algorithms
  - Breadth-first search
  - Time complexity of BFS algorithm
  - Shortest paths
  - Depth-first search
  - Time complexity of DFS algorithm
  - Topological sort
  - Time complexity of TOPOLOGICAL-SORT
  - Strongly connected components

1 Elementary Graph Algorithms

Recall that we represent the graphs in such a way that is easy for computers to analyze them. For us, this representations of graphs will consist of *adjacency-lists* and *adjacency-matrices* representations. Specifically, we assume that all our algorithms take as input graphs that are given in form of *adjacency-lists* representations. This assumption will be reflected in the way we will measure the run time complexity of the algorithms, which will often involve both the set of vertices V and set of edges E.

Today we will take another closer look at the two simple graph-searching algorithms: breadth-first search algorithm and the depth-first search algorithm. In addition, we will explore a new kind of sorting method, called topological sorting of a directed acyclic graph, which is closely related to the depth-first search algorithm. Lastly, we will explore the notion and computation of *strongly connected components in directed graph* using an algorithm that is based on the depth-first search algorithm.

1.1 Breadth-first search

Breadth-first search is one of the simplest algorithms for searching a graph and and it serves as a basis for many important graph algorithms, such as Dijkstra's (single-source shortest-paths) and Prim's (minimum-spanning-tree algorithm) algorithms.

Given a graph $G = (V, E)$ and a "starting" source vertex s, breadth-first search systematically explores the edges of G to "discover" every vertex that is reachable from s. It computes the distance (smallest number of edges) from s to each reachable vertex. It also produces a "breadth-first tree" with root s that contains all reachable vertices. For any vertex v reachable from s, the simple path in the breadth-first tree from s to v corresponds to a shortest path from s to v in G, that is, a path containing the smallest number of edges. The algorithm is applicable for both directed and undirected graphs, where with the directed graphs we must respect the arrow orientation.

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

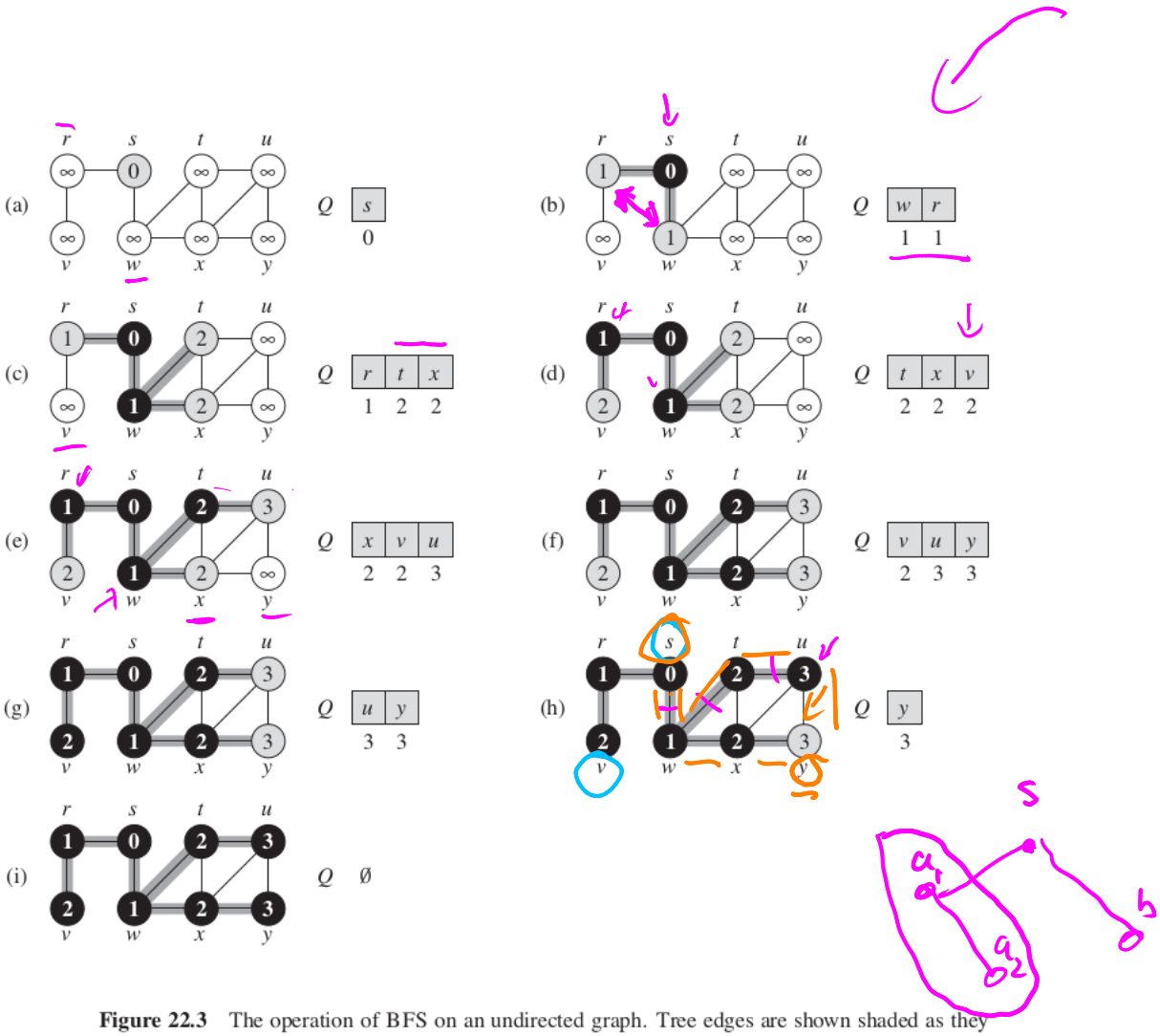


Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. The value of $u.d$ appears within each vertex u. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances appear below vertices in the queue.

The "steps" the BFS algorithm performs are illustrated below. Keep in mind that in a queue Q, elements are added to the back of the queue and removed from the front of the queue using ENQUEUE and DEQUEUE operations. In addition, elements are added to the queue *only* if they are not already in the queue.

```
remove s, visit s: Q = [w,r]   (add w then r to Q).
remove w, visit w: Q = [r,t,x] (add t then x to Q)
remove r, visit r: Q = [t,x,v] (add v to Q)
remove t, visit t: Q = [x,v,u] (add u to Q)
remove x, visit x: Q = [v,u,y] (add y to Q)
remove v, visit v: Q = [u,y]   (nothing to be added)
remove u, visit u: Q = [y]     (nothing to be added)
remove y, visit y: Q = []      (algorithm stops when Q is empty)
```

Note that the image (i) denotes a tree (all vertices and gray shaded edges), which we cal a *BFS-Tree*. Furthermore, note that the vertices in the final image (i) denote the *distances* of each vertex from the source vertex s. We measure the distance between two vertices in the number of edges that separate the two vertices.

1.2 Time complexity of BFS algorithm

For a input graph $G = (V, E)$, which was given in form of *adjacency-list* representation, the time complexity is as follows. The operations of enqueue and dequeue take constant time $O(1)$, hence the total time needed to queue operations is $O(V)$. Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once. Since the sum of the lengths of all the adjacency lists is $\Theta(E)$, the total time spent in scanning adjacency lists is $O(E)$. The overhead for initialization is $O(V)$ and, therefore, the total running time of the BFS procedure is $O(V + E)$. That is, it is linear in the size of the adjacency-list representation of G.

1.3 Shortest paths

In the previous lecture we remarked that the BFS finds the distance to each reachable vertex in a graph $G = (V, E)$ from a given source vertex we labeled $s \in V$. The ability to determine the shortest paths between vertices is an important task for many algorithms.

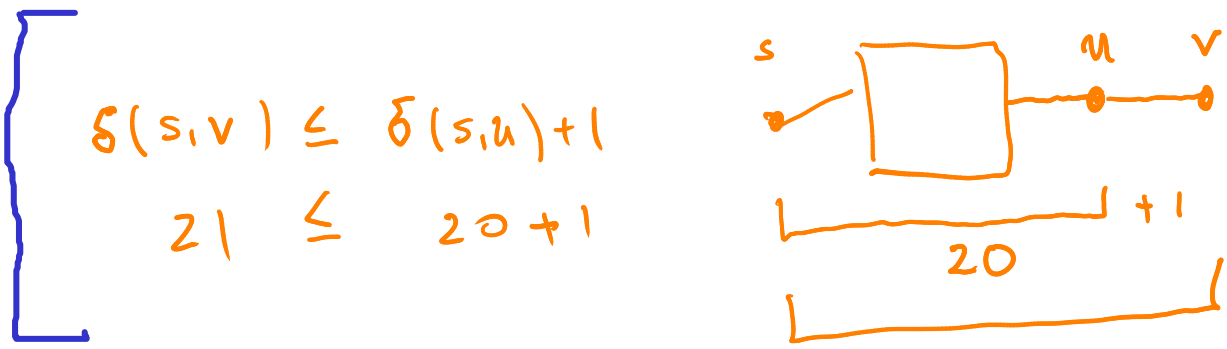
Definition 1 (Shortest-path distance)**.** *The shortest-path distance $\delta(s, v)$ from s to v is the minimum number of edges in any path from vertex s to vertex v. If there is no path from s to v, then $\delta(s, v) = \infty$. We call the path of length $\delta(s, v)$ from s to v a shortest path from s to v.*

Lemma 1. *Let $G = (V, E)$ be a directed or undirected graph, and let $s \in V$ be an arbitrary vertex. Then, for any edge $(u, v) \in E$, $\delta(s, v) \leq \delta(s, u) + 1$.*

Proof. If u can be reached from s, then so can v, as there is an edge between u and v. In that case the shortest path from s to v cannot be longer than the shortest path from s to u followed by the edge (u, v), and therefore the inequality holds. If u is not reachable from s, then $\delta(s, u) = \infty$, and the inequality holds. □

1.4 Depth-first search

The strategy applied by the depth-first search is to search "deeper" in the graph whenever possible. Depth-first search explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it. After all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source. The algorithm repeats this entire process until it has discovered every vertex.



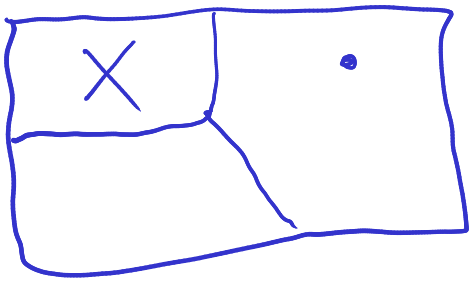
```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
1  time = time + 1          // white vertex u has just been discovered
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]    // explore edge (u, v)
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK          // blacken u; it is finished
9  time = time + 1
10 u.f = time
```

The depth-first search colors vertices during the search to indicate their state. Each vertex is initially white, is grayed when it is discovered in the search, and is blackened when it is finished, that is, when its adjacency list has been examined completely. Furthermore, depth-first search also timestamps each vertex. Each vertex v has two timestamps: the first timestamp $v.d$ records when v is first discovered (and grayed), and the second timestamp $v.f$ records when the search finishes examining v's adjacency list (and blackens v). These are the "fractional values" inside each vertex.

In contrast to breadth-first search algorithm, whose subgraph forms a tree, the subgraph produced by a depth-first search may be composed of several trees, because the search may repeat from multiple sources. Such trees are separated by an edge labeled "C" for "cross".

The following illustrates a DFS algorithm on a directed graph with cycles. It may help to recall the "pre-order" traversals (or in-order, or post-order) for cases when the graph is a tree during the example below. Any of the "pre/in/post-order" traversals are DFS traversals, except they are running on trees, rather than general graphs.



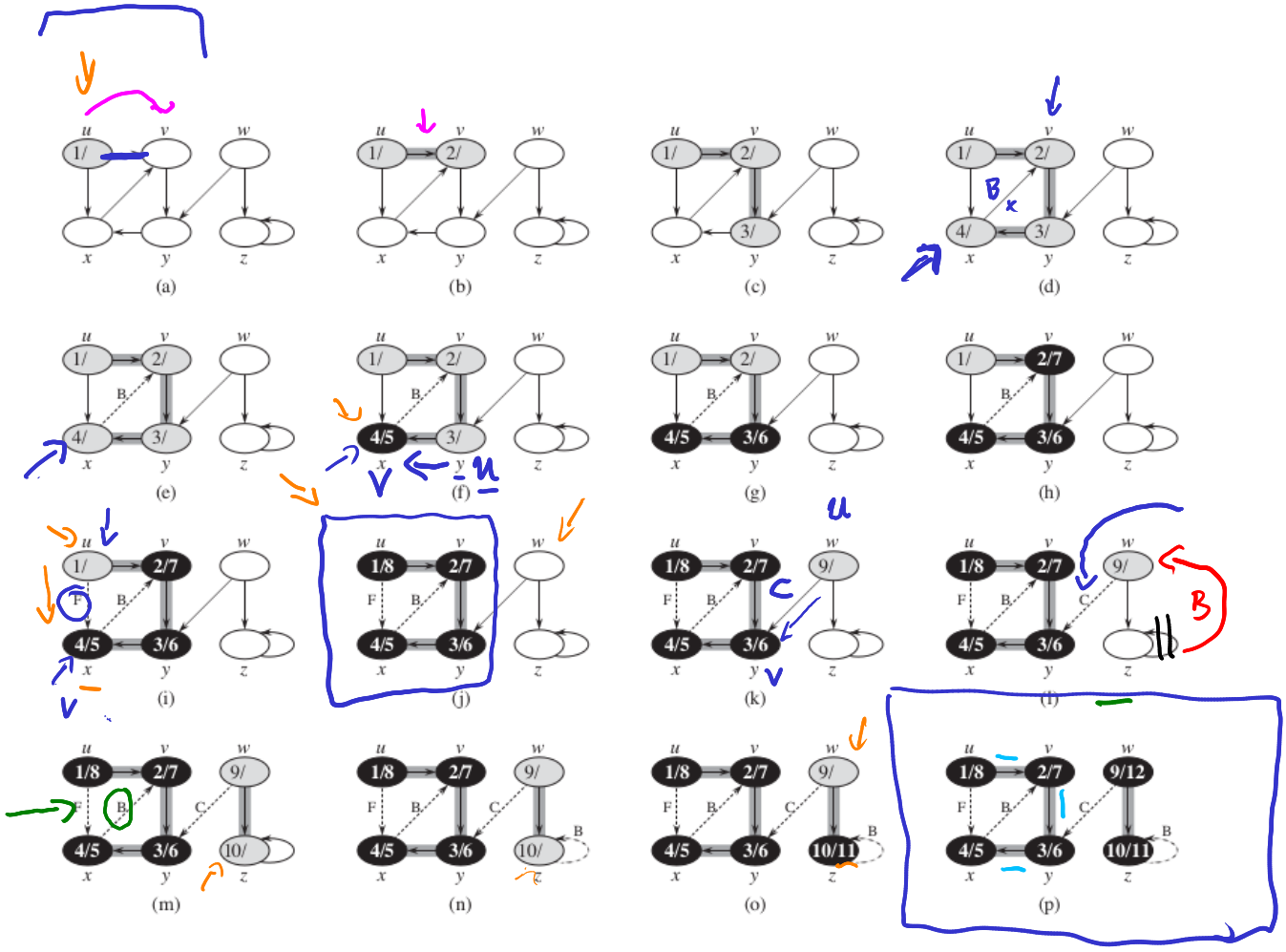


Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Timestamps within vertices indicate discovery time/finishing times.

Classification of edges

The depth-first search algorithm's interactions with a graph G may result in a depth-first forest, which we denote as G_π. A depth-first forest is a collection of depth-first trees. In such a and depth-first forest G_π, we can encounter four types of edges.

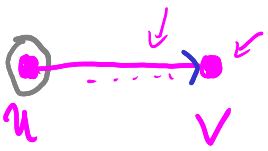
Tree edges are edges in the depth-first forest G_π. An edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).

Back edges are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. We consider self-loops, which may occur in directed graphs, to be back edges.

Forward edges are those non-tree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.

Cross edges are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.

The DFS algorithm has enough information to classify some edges as it encounters them. The key idea is that when we first explore an edge (u, v), the color of vertex v tells us something about the edge.



1. WHITE indicates a tree edge
2. GRAY indicates a back edge
3. BLACK indicates a forward or cross edge

The first case is immediate from the specification of the algorithm. For the second case, observe that the gray vertices always form a linear chain of descendants corresponding to the stack of active DFS-VISIT invocations; the number of gray vertices is one more than the depth in the depth-first forest of the vertex most recently discovered. Exploration always proceeds from the deepest gray vertex, so an edge that reaches another gray vertex has reached an ancestor. The third case handles the remaining possibility.

An undirected graph may entail some ambiguity in how we classify edges, since (u, v) and (v, u) are really the same edge. In such a case, we classify the edge as the first type in the classification list that applies. Equivalently, we classify the edge according to whichever of (u, v) or (v, u) the search encounters first.

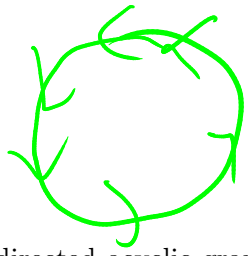
We now show that forward and cross edges never occur in a depth-first search of an undirected graph.

Theorem 1. *In a depth-first search of an undirected graph G, every edge of G is either a tree edge or a back edge.*

Proof. Let (u, v) be an arbitrary edge of G, and suppose without loss of generality that $u.d < v.d$, where the $.d$ operation denotes the discovery time. Then the search must discover and finish v before it finishes u (while u is gray), since v is on u's adjacency list. If the first time that the search explores edge (u, v), it is in the direction from u to v, then v is undiscovered (white) until that time, for otherwise the search would have explored this edge already in the direction from v to u. Therefore, (u, v) becomes a tree edge. If the search explores (u, v) first in the direction from v to u, then (u, v) is a back edge, since u is still gray at the time the edge is first explored. □

1.5 Time complexity of DFS algorithm

The loops on lines $1-3$ and $5-7$ of DFS take time $\Theta(V)$, exclusive of the time to execute the calls to DFS-VISIT. The procedure DFS-VISIT is called exactly once for each vertex $v \in V$, since vertex u on which DFS-VISIT is invoked must be white and the first act of DFS-VISIT is to paint vertex u gray. During an execution of DFS-VISIT(G, v), the loop on lines $4-7$ executes $|Adj[v]|$ times. Since $\sum_{v \in V} |Adj[v]| = \Theta(E)$, the total cost of executing lines $4-7$ of DFS-VISIT is $\Theta(E)$. Therefore, the running time of DFS is $\Theta(V + E)$.



1.6 Topological sort

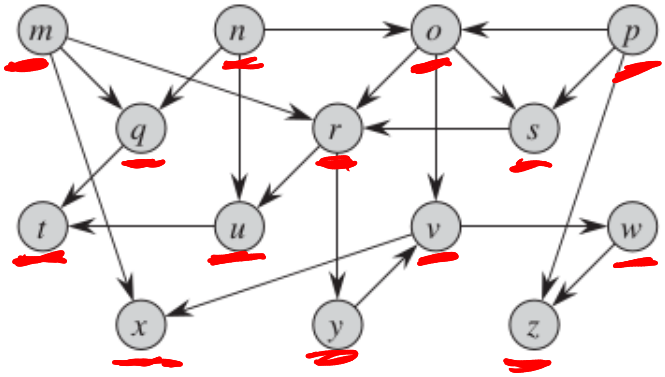
We can use depth-first search to perform a *topological sort* of a directed acyclic graph (or a "dag"). A topological sort of a dag $G = (V, E)$ is a linear ordering of all its vertices such that if graph G contains an edge (u, v), then u appears before v in the ordering. In particular, we can view a topological sort of a graph as an ordering of its vertices along a horizontal line so that all directed edges go from left to right. Topological sorting is different from the usual kind of "sorting" we applied to integers, for example, in previous lessons. In addition, if the graph contains a cycle, then no linear ordering is possible, hence the condition that graph G be an acyclic graph.

```
TOPOLOGICAL-SORT(G)
1. call DFS(G) to compute finishing times v.f for each vertex v
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices
```

The topological sorting is used when we want to establish a priority of elements, not necessarily based on their "size" as it is the case with usual sorting, for example, that of integers. As an illustration, consider the delivery of building material to a construction site where homes a being built. In such a case, any material related to the basement construction must be delivered before any material related to the roof construction is delivered. In particular, one expects the material delivery to happen in order of basement, then material for the walls, and then the material for the roof. This order represent a result of a topological sorting.

Let us look at an example, using a graph, whose elements we want to sort topologically, using the *TOPOLOGICAL-SORT(G)* algorithm above. In addition, *DFS(G)* will select elements to be processed in *alphabetically*. This is merely a convention but it helps during the illustration of the steps. In principle, any order can be used or elements can be selected at random.

Example 1 (TOPOLOGICAL-SORT on graph G).



Vertex	discovery time	finish time
m	1	20
q	2	5
t	3	4
r	6	19
u	7	8
y	9	18
v	10	17
w	11	14
z	12	13
x	15	16
n	21	26
o	22	25
s	23	24
p	27	28

The result of the topological sort on this example is a linked list of the form:
$p, n, o, s, m, r, y, v, x, w, z, u, q, t$, where all elements are listed according to their finish time.

1.7 Time complexity of TOPOLOGICAL-SORT

Because *DFS* algorithms takes time $\Theta(V + E)$ and it takes $O(1)$ time to insert each of the $|V|$ vertices into the front of the linked list, the *TOPOLOGICAL-SORT* algorithms also takes time $\Theta(V + E)$.

Lemma 2. *A directed graph G is acyclic if and only if a depth-first search of G yields no back edges.*

Theorem 2. *TOPOLOGICAL-SORT produces a topological sort of the directed acyclic graph provided as its input.*

Proof. Suppose that DFS is run on a given dag $G = (V, E)$ to determine finishing times for its vertices. It suffices to show that for any pair of distinct vertices $u, v \in V$, if G contains an edge from u to v, then $v.f < u.f$. Consider any edge (u, v) explored by $DFS(G)$. When this edge is explored, v cannot be gray, since then v would be an ancestor of u and (u, v)

would be a back edge, contradicting the Lemma above. Therefore, v must be either white or black. If v is white, it becomes a descendant of u, and so $v.f < u.f$. If v is black, it has already been finished, so that $v.f$ has already been set. Because we are still exploring from u, we have yet to assign a timestamp to $u.f$, and so once we do, we will have $v.f < u.f$, proving the theorem. □

1.8 Strongly connected components

One of the classic application of depth-first search is a decomposition of a directed graph into its *strongly connected components*. This operations is, in many ways, fundamental such that many algorithms that work with directed graphs begin with such a decomposition. After decomposing the graph into strongly connected components, such algorithms run separately on each one and then combine the solutions according to the structure of connections among components.

Definition 2 (Strongly Connected Component)**.** *A strongly connected component of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$, such that for every pair of vertices u and v in C, we have both $u \rightsquigarrow v$ and $v \rightsquigarrow u$.*

The symbol $\rightsquigarrow$ means "leads to" and the definition above, in other words, says that vertices u and v are reachable from each other.

Let the graph $G = (V, E)$ be a directed graph, where the set V denotes the set of vertices of G and E denote the set of edges in G. Consider the set $E^T = \{(u, v) : (v, u)\}$, consisting of all edges in E with directions *reversed*. Let $G^T = (V, E^T)$ be a new graph whose edge directions have been reversed. The following algorithm, with a run time complexity $O(V + E)$ computes the **strongly connected components** of G.

STRONGLY-CONNECTED-COMPONENTS(G)

1. Call DFS(G) to compute finishing times $u.f$ for each vertex u
2. Compute G^T
3. Call DFS(G^T), but in the main loop of DFS, consider the vertices in order of *decreasing* $u.f$ (as computed in line 1)
4. Output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

An example of the algorithm and its output are given in (a) and (b). In (c) we see a **contracted** version of the output, where $G^{SCC} = (V^{SCC}, E^{SCC})$ denotes the **component graph**, in which each component has been compressed into a single vertex. The image (c) now illustrates the importance of the classification of edges and their role in the construction of all the algorithms we have seen in this lecture!

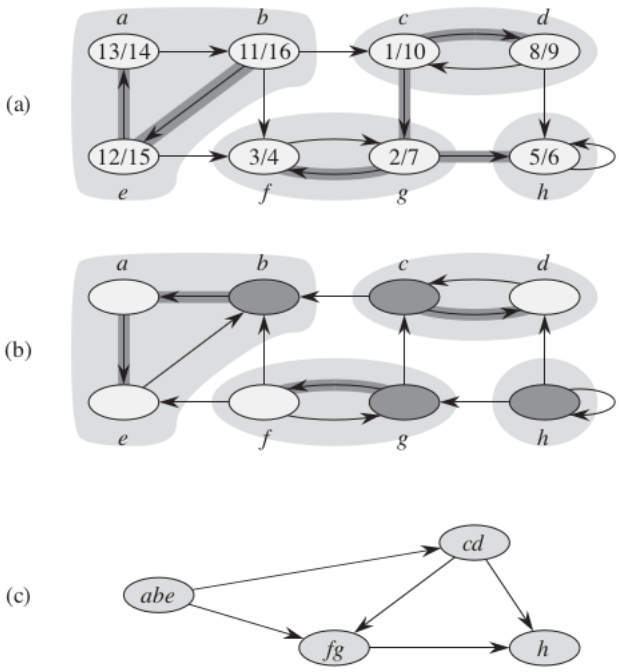


Figure 22.9 **(a)** A directed graph G. Each shaded region is a strongly connected component of G. Each vertex is labeled with its discovery and finishing times in a depth-first search, and tree edges are shaded. **(b)** The graph G^T, the transpose of G, with the depth-first forest computed in line 3 of STRONGLY-CONNECTED-COMPONENTS shown and tree edges shaded. Each strongly connected component corresponds to one depth-first tree. Vertices b, c, g, and h, which are heavily shaded, are the roots of the depth-first trees produced by the depth-first search of G^T. **(c)** The acyclic component graph G^{SCC} obtained by contracting all edges within each strongly connected component of G so that only a single vertex remains in each component.