AG44

UTBM – A2018

abderrahim.chariete@utbm.fr, gaber@utbm.fr

Chapter 3: Graph searching methods

1. Introduction

This chapter presents methods for searching a graph. Searching a graph means systematically following the edges of the graph to visit the vertices of the graph. A graph-searching algorithm can discover much about the structure of a graph. Many algorithms begin by searching their input graph to obtain this structural information. Several other graph algorithms elaborate on basic graph searching. Techniques for searching a graph lie at the heart of the field of graph algorithms.

Firstly, we present a simple graph-searching algorithm called breadth-first search and shows how to create a breadth-first tree. Then, we present depth-first search and proves some standard results about the order in which depth-first search visits vertices. Therefore, we provide a real's applications of depth-first search:

- Topologically sorting a directed acyclic graph.
- Finding the strongly connected components of a directed graph.

2. Breadth-first search:

Breadth-first search is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms. Prim's minimum-spanning-tree algorithm and Dijkstra's single-source shortest-paths algorithm use ideas like those in breadth-first search.

Given a graph $G = \{V, E\}$ and a distinguished 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 u reachable from s, the simple path in the breadth-first tree from s to u corresponds to a "shortest path" from s to u in G, that is, a path containing the smallest number of edges. The algorithm works on both directed and undirected graphs.

Breadth-first search is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. That is, the algorithm discovers all vertices at distance k from s before discovering any vertices at distance $k + 1$.

To keep track of progress, breadth-first search colors each vertex white, gray, or black. All vertices start out white and may later become gray and then black. A vertex is discovered the first time it is encountered during the search, at which time it becomes nonwhite. Gray and black vertices, therefore, have been discovered, but breadth-first search distinguishes between them to ensure that the search proceeds in a breadth-first manner. If $(u, v) \in E$ and vertex u is black, then vertex v is either gray or black; that is, all vertices adjacent to black vertices have been discovered. Gray vertices may have some adjacent white vertices; they represent the frontier between discovered and undiscovered vertices.

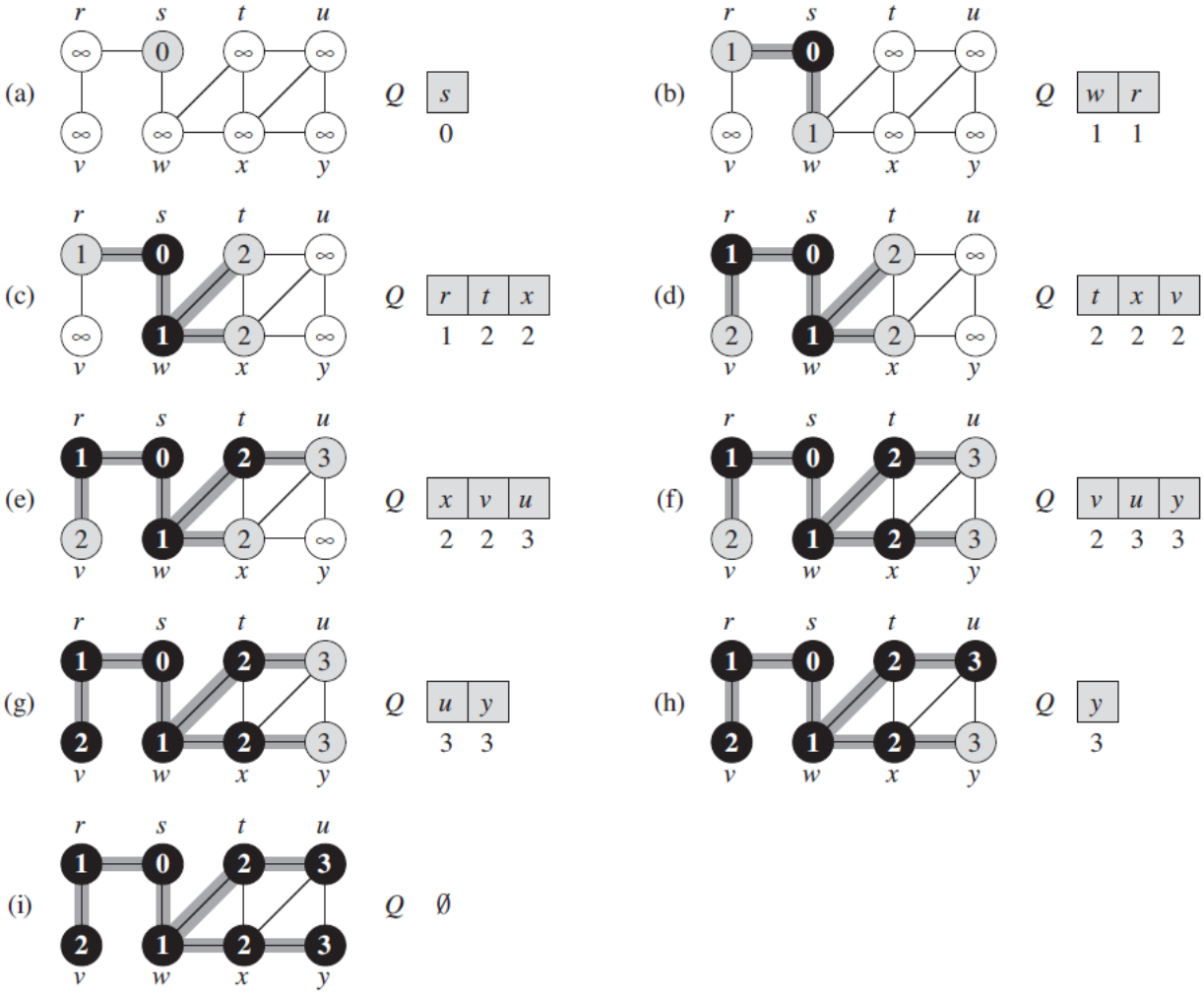
```
BFS (G, s)
 1.  for each vertex u ∈ G.V – {s}
 2.        u.color = WHITE
 3.        u.dist = 1
 4.        u.pred = NIL
 5.  s.color = GRAY
 6.  s.dist = 0
 7.  s.pred = NIL
 8.  Q = ∅;
 9.  ENQUEUE (Q, s)
10.  while Q ∉ ∅ ;
11.        u = DEQUEUE (Q)
12.        for each v ∈ G.Adj[u]
13.         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

Breadth-first search constructs a breadth-first tree, initially containing only its root, which is the source vertex s. Whenever the search discovers a white vertex v while scanning the adjacency list of an already discovered vertex u, the vertex v and the edge (u, v) are added to the tree. We say that u is the predecessor or parent of v in the breadth-first tree. Since a vertex is discovered at most once, it has at most one parent. Ancestor and descendant relationships in the breadth-first tree are defined relative to the root s as usual: if u is on the simple path in the tree from the root s to vertex v, then u is an ancestor of v and v is a descendant of u.

The breadth-first-search procedure BFS below assumes that the input graph $G = (V, E)$ is represented using adjacency lists. It attaches several additional attributes to each vertex in the graph. We store the color of each vertex $u \in V$ in the attribute $u.color$ and the predecessor of u in the attribute $u.pred$. If u has no predecessor (for example, if $u = s$ or u has not been discovered), then $u.pred$ = NIL.

The attribute $u.dist$ holds the distance from the source s to vertex u computed by the algorithm. The algorithm also uses a first-in, first-out queue Q to manage the set of gray vertices.

The following figure illustrates the progress of BFS on a sample graph.



The procedure BFS works as follows. With the exception of the source vertex s, lines 1–4 paint every vertex white, set $u.dist$ to be infinity for each vertex u, and set the parent of every vertex to be NIL. Line 5 paints s gray, since we consider it to be discovered as the procedure begins. Line 6 initializes $s.dist$ to 0, and line 7 sets the predecessor of the source to be NIL. Lines 8–9 initialize Q to the queue containing just the vertex s.

The **while** loop of lines 10–18 iterates if there remain gray vertices, which are discovered vertices that have not yet had their adjacency lists fully examined. This **while** loop maintains the following invariant:

At the test in line 10, the queue Q consists of the set of gray vertices.

Although we won't use this loop invariant to prove correctness, it is easy to see that it holds prior to the first iteration and that each iteration of the loop maintains the invariant. Prior to the first iteration, the only gray vertex, and the only vertex in Q, is the source vertex s. Line 11 determines the gray vertex u at the head of the queue Q and removes it from Q. The **for** loop of lines 12–17 considers each vertex v in the adjacency list of u. If v is white, then it has not yet been discovered, and the procedure discovers it by executing lines 14–17. The procedure paints vertex v gray, sets its distance $v.dist$ to $u.dist$ +1, records u as its parent $v.pred$, and places it at the tail of the queue Q. Once the procedure has examined all the vertices on u's adjacency list, it blackens u in line 18. The loop invariant is maintained because whenever a vertex is painted gray (in line 14) it is also enqueued (in line 17), and whenever a vertex is dequeued (in line 11) it is also painted black (in line 18). The results of breadth-first search may depend upon the order in which the neighbors of a given vertex are visited in line 12: the breadth-first tree may vary, but the distances $dist$ computed by the algorithm will not.

A. Breadth-first trees

The procedure BFS builds a breadth-first tree as it searches the graph. The tree corresponds to the *pred* attributes. More formally, for a graph $G = (V, E)$ with source s, we define the ***predecessor subgraph*** of G as $G_{pred} = (V_{pred}, E_{pred})$, where

$V_{pred} = \{v \in V: v.pred \neq \text{NIL}\} \cup \{s\}$.

and

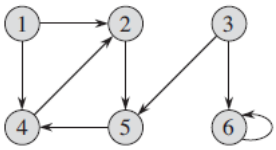
$E_{pred} = \{(v.pred, v): v \in V_{pred} - \{s\}\}$.

The predecessor subgraph G_{pred} is a ***breadth-first tree*** if V_{pred} consists of the vertices reachable from s and, for all $v \in V_{pred}$, the subgraph G_{pred} contains a unique simple path from s to v that is also a shortest path from s to v in G. A breadth-first tree is in fact a tree, since it is connected and $|E_{pred}| = |V_{pred}| - 1$. We call the edges in E_{pred} ***tree edges***.
The following lemma shows that the predecessor subgraph produced by the BFS procedure is a breadth-first tree.

B. Exercises

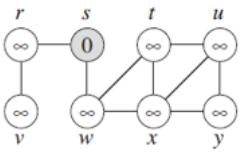
- **Exo1.**

  Show the *dist* and *pred* values that result from running breadth-first search on the directed graph of the following figure, using vertex 3 as the source.



- **Exo2.**

  Show the *dist* and *pred* values that result from running breadth-first search on the undirected graph of the following figure, using vertex u as the source.



- **Exo3.**

  What is the running time of BFS if we represent its input graph by an adjacency matrix and modify the algorithm to handle this form of input?

3. Depth-first search:

The strategy followed by depth-first search is, as its name implies, to search "deeper" in the graph whenever possible. Depth-first search explores edges out of the most recently discovered vertex _ that still has unexplored edges leaving it. Once 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.

As in breadth-first search, whenever depth-first search discovers a vertex _ during a scan of the adjacency list of an already discovered vertex u, it records this event by setting _'s predecessor attribute *v.pred* to u. Unlike breadth-first search, whose predecessor subgraph forms a tree, the predecessor subgraph produced by a depth-first search may be composed of several trees, because the search may repeat from multiple sources. Therefore, we define the ***predecessor subgraph*** of a depth-first search slightly differently from that of a breadth-first search: we let $G_{pred} = (V, E_{pred})$, where

$E_{pred} = \{(v.pred, v): v \in V \text{ and } v.pred \neq \text{NIL}\}$.

The predecessor subgraph of a depth-first search forms a depth-first forest comprising several depth-first trees. The edges in E_{pred} are tree edges.

```
DFS (G, s)
 1.  for each vertex u ∈ G.V
 2.        u.color = WHITE
 3.        u.pred = 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   // while 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.pred = u
 7.              DFS-VISIT (G, v)
 8.  u.color = BLACK    // blacken u; it is finished
 9.  time = time + 1
10.  u.f = time
```

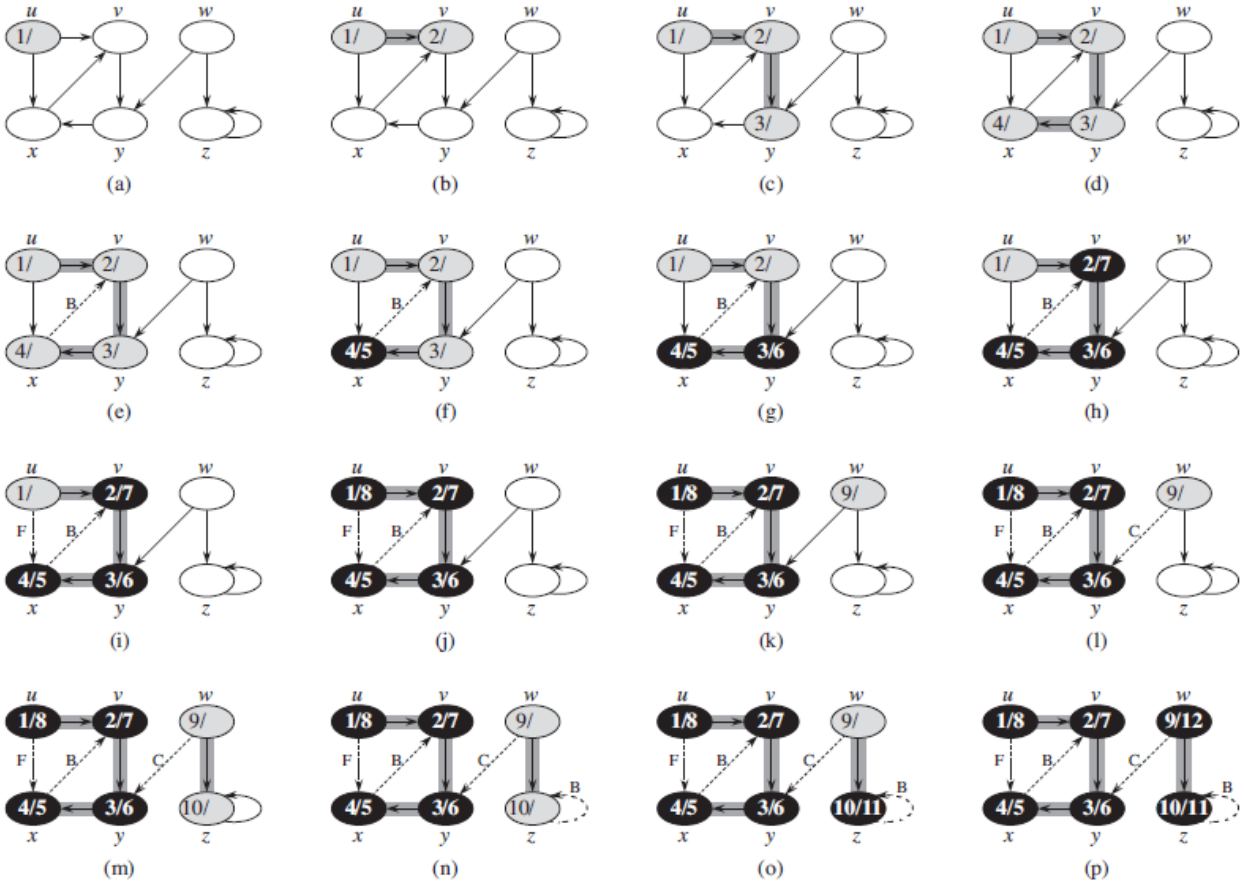
As in breadth-first search, depth-first search colors vertices during the search to indicate their state. Each vertex is initially white, is grayed when it is discovered in the search, and is blackened when it is finished, that is, when its adjacency list has been examined completely. This technique guarantees that each vertex ends up in exactly one depth-first tree, so that these trees are disjoint.

Besides creating a depth-first forest, 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 timestamps provide important information about the structure of the graph and are generally helpful in reasoning about the behavior of depth-first search.

The procedure DFS below records when it discovers vertex u in the attribute $u.d$ and when it finishes vertex u in the attribute $u.f$. These timestamps are integers between 1 and 2 $|V|$, since there is one discovery event and one finishing event for each of the $|V|$ vertices. For every vertex u, $u.d < u.f$: Vertex u is WHITE before time $u.d$, GRAY between time $u.d$ and time $u.f$, and BLACK thereafter.

The pseudocode above is the basic depth-first-search algorithm. The input graph G may be undirected or directed. The variable time is a global variable that we use for timestamping.

The following figure illustrates the progress of DFS:

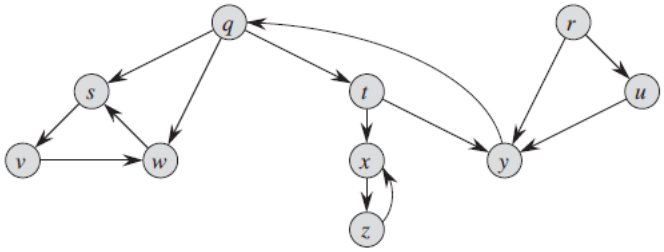


Exercises:

- **Exo1:**
  What is the running time of DFS?

- **Resp1:**
  The loops on lines 1–3 and lines 5–7 of DFS take time, $\Theta(V)$, exclusive of the time to execute the calls to DFS-VISIT. As we did for breadth-first search, we use aggregate analysis. The procedure DFS-VISIT is called exactly once for each vertex $v \in V$, since the vertex u on which DFS-VISIT is invoked must be white and the first thing DFS-VISIT does is paint vertex u gray. During an execution of DFS-VISIT (G, v), the loop on lines 4–7 executes $|Adj[v]|$ times. Since:
  $\sum_{v \in V} |Adj[v]| = \Theta(V + E)$
  the total cost of executing lines 4–7 of DFS-VISIT is , $\Theta(E)$. The running time of DFS is therefore , $\Theta(V + E)$.

- **Exo2:**
  Show how depth-first search works on the following graph. Assume that the for loop of lines 5–7 of the DFS procedure considers the vertices in alphabetical order and assume that each adjacency list is ordered alphabetically. Show the discovery and finishing times for each vertex.



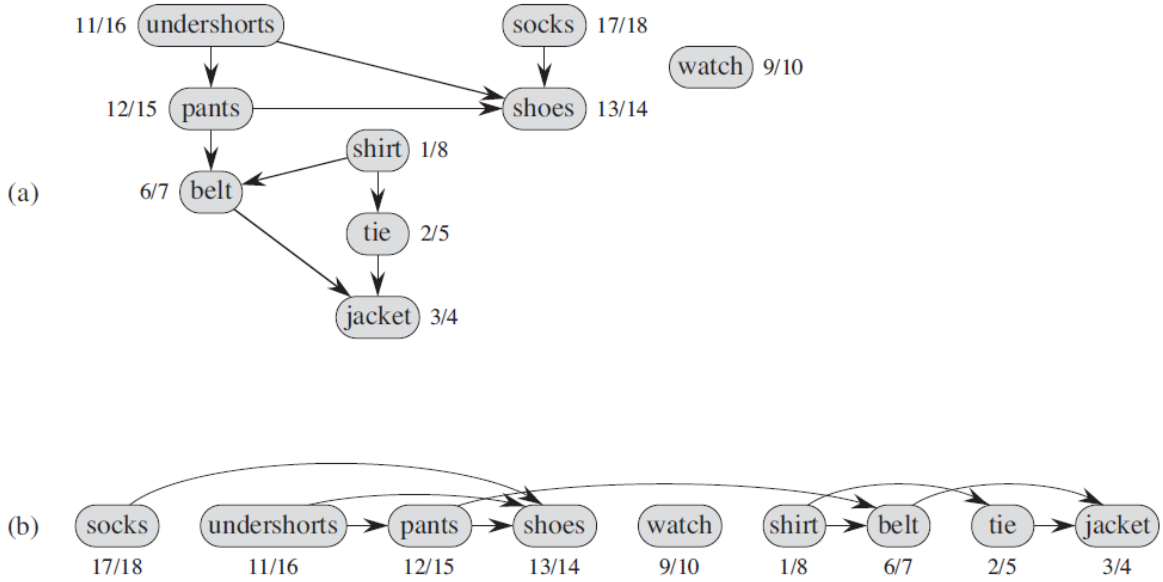
4. Topological sort

This section shows how we can use depth-first-search to perform a topological sort of a directed acyclic graph, or a "DAG" as it is sometimes called. A topological sort of a DAG $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering. (If the graph contains a cycle, then no linear ordering is possible.) 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.

The following simple algorithm topologically sorts a DAG:

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

Many applications use directed acyclic graphs to indicate precedence's among events. The following figure gives an example that arises when Professor CHARIETE gets dressed in the morning. The professor must don certain garments before others (e.g., socks before shoes). Other items may be put on in any order (e.g., socks and pants). A directed edge (u, v) in the DAG of the figure (a) indicates that garment u must be donned before garment v. A topological sort of this DAG therefore gives an order for getting dressed. The figure (b) shows the topologically sorted DAG as an ordering of vertices along a horizontal line such that all directed edges go from left to right.



The figure (b) shows how the topologically sorted vertices appear in reverse order of their finishing times.

A. Topological sort algorithm complexity

We can perform a topological sort in time $\Theta(V + E)$, since depth-first search takes $\Theta(V + E)$ time and it takes $O(1)$ time to insert each of the $|V|$ vertices onto the front of the linked list.

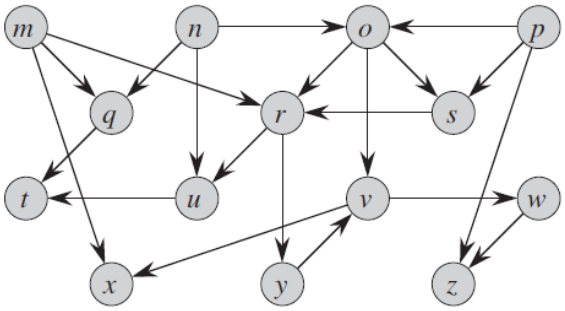
B. Topological sort applications

- Planning and scheduling.
- The algorithm can also be modified to detect cycles

C. Exercises

- ***Exo-1:***

Show the ordering of vertices produced by TOPOLOGICAL-SORT when it is run on the DAG of the following figure. Assume that the **for** loop of lines 5–7 of the DFS procedure considers the vertices in alphabetical order and assume that each adjacency list is ordered alphabetically.



- ***Exo-2:***

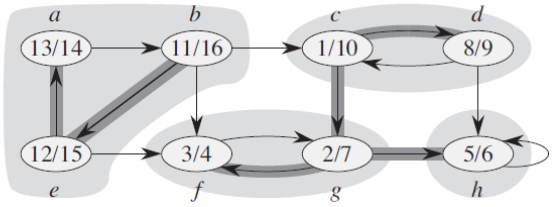
Give a linear-time algorithm that takes as input a directed acyclic graph $G = (V, E)$ and two vertices s and t, and returns the number of simple paths from s to t in G. For example, the directed acyclic graph of the previous figure contains exactly four simple paths from vertex p to vertex v: *pov*, *poryv*, *posryv*, and *psryv*. (Your algorithm needs only to count the simple paths, not list them.).

5. Strongly connected components

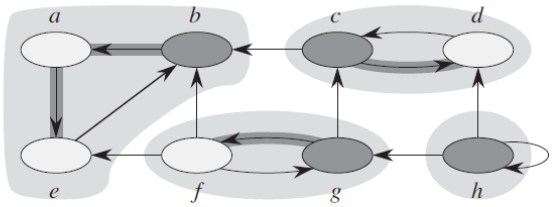
We now consider a classic application of depth-first-search: decomposing a directed graph into its strongly connected components. This section shows how to do so using two depth-first searches. Many algorithms that work with directed graphs begin with such a decomposition. After decomposing the graph into strongly connected components, such algorithms run separately on each one and then combine the solutions according to the structure of connections among components.

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 $v \rightsquigarrow u$; that is, vertices u and v are reachable from each other. The following figure shows an example.

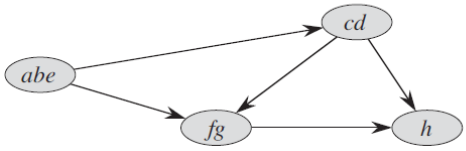
(a)



(b)



(c)



Our algorithm for finding strongly connected components of a graph $G = (V, E)$ uses the transpose of G, which we defined to be the graph $G^T = (V, E^T)$, where $E^T = \{(u,v) : (v, u) \in E\}$. That is, E^T consists of the edges of G with their directions reversed. Given an adjacency-list representation of G, the time to create G^T is $O(V + E)$. It is interesting to observe that G and G^T have the same strongly connected components: u and v are reachable from each other in G if and only if they are reachable from each other in G^T.

Figure (b) shows the transpose of the graph in figure (a), with the strongly connected components shaded.

The following linear-time (i.e., $\Theta(V + E)$-time) algorithm computes the strongly connected components of a directed graph $G = (V, E)$ using two depth-first searches, one on G and one on G^T.

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 $\underline{u}.f$ (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

The idea behind this algorithm comes from a key property of the component graph $G^{SCC} = (V^{SCC}, E^{SCC})$, which we define as follows. Suppose that G has strongly connected components $C_1, C_2, \ldots, C_k$. The vertex set V^{SCC} is $\{v_1, v_2, \ldots, v_k\}$, and it contains a vertex v_i for each strongly connected component C_i of G. There is an edge $(v_i, v_j) \in E^{SCC}$ if G contains a directed edge (x, y) for some $x \in C_i$ and some $y \in C_j$. Looked at another way, by contracting all edges whose incident vertices are within the same strongly connected component of G, the resulting graph is G^{SCC}. Figure (c) shows the component graph of the graph in (a).