13.1 Strongly Connected Components

A classic application of depth-first search is finding strongly connected components in a directed graph by decomposing it. In this section, we will use two depth-first searches in order to find the strongly connected components.

We define strongly components as a set of vertices, where each vertex is reachable from another vertex in the set. More precisely, 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 to v in C, we have both $u \rightsquigarrow v$ and $v \rightsquigarrow u$; that is vertices u and v are reachable from each other. Figure 22.9 shows an example.

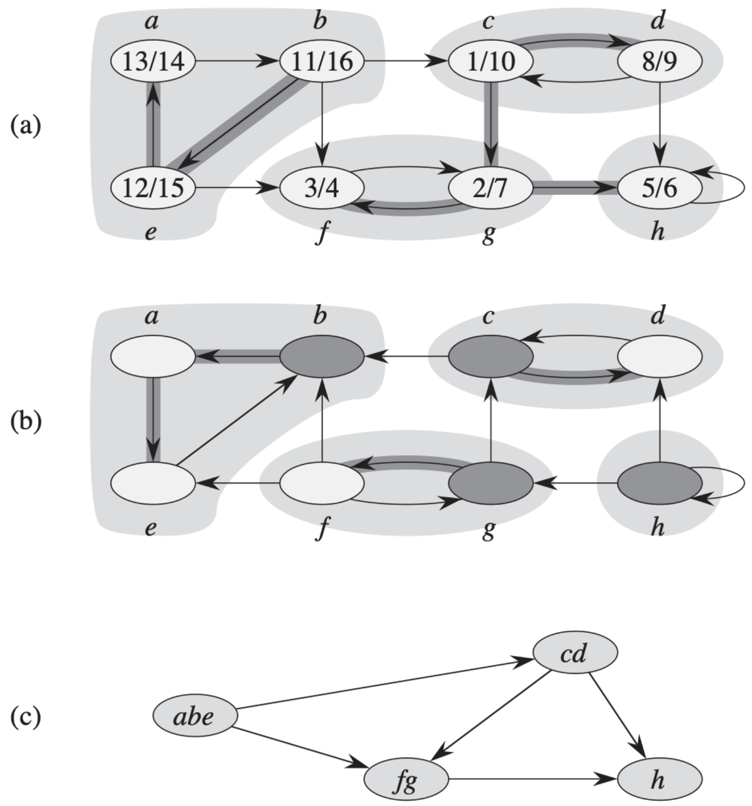

Figure 22.9 **(a)** A directed graph G. Each shaded region is a strongly connected component of G. Each vertex is labeled with its discovery and finishing times in a depth-first search, and tree edges are shaded. **(b)** The graph G^{T}, the transpose of G, with the depth-first forest computed in line 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.

Our algorithm for finding strongly connected components of a graph $G = (V, E)$ uses the transpose of G. A transpose of a directed graph $G = (V, E)$ is the graph $G^T = (V, E^T)$, where $E^T = \{(v, u) \in V \times V : (u, v) \in E\}$. Essentially, G^T is G with all its edges reversed. Given an adjacency-list representation of G, the time to create G^T is $O(V+E)$.

We can see that, interestingly, G and G^T have **exactly** 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 22.9(b) shows the transpose of the graph in Figure 22.9(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 $u.f$ (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 components
```

The idea behind this algorithm comes from a key property of the ***component graph*** $G^{\text{SCC}}=(V^{\text{SCC}},E^{\text{SCC}})$, which we define as follows. Supose that G has strongly connected components $C_1, C_2, ...C_k$. The vertex set V^{SCC} is $\{v_1, v_2, ...v_k\}$, and it contains a vertex v_i for each strongly connected component C_i of G. There is an edge $(v_i, v_j) \in E^{\text{SCC}}$ 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 in G^{SCC}. Figure 22.9(c) shows the component graph of the graph in Figure 22.9(a).

Here is an additional reading you may want to explore about strongly connencted components: Strongly connected components post on Programiz

13.2 Breadth-First Search

Breadth-first search is one of the simplest algorithms for searching a graph and the archetype for many graph algorithms. Prim's minimum-spanning-tree algorithm and Dijkstra's single-source shortest-paths algorithm use ideas similar to 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 (smaller 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 works on both directed and undirected graph.

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 $k+1$.

```
BFS($G, s$)
 1 for each vertex $u \in G.V - \{s\}$
 2       $u.color$ = WHITE
 3       $u.d = \infty$
 4       $u.\pi$ = NIL
 5 $s.color$ = GRAY
 6 $s.d = 0$
 7 $s.\pi$ = NIL
 8 $Q = \emptyset$
 9 ENQUEUE($Q, s$)
10 while $Q \neq \emptyset$
11       $u$ = DEQUEUE($Q$)
12       for each $v \in G.Adj[u]$
13             if $v.color$ == WHITE
14                   $v.color$ = GRAY
15                   $v.d = u.d + 1$
16                   $v.\pi = u$
17                   ENQUEUE($Q, v$)
18       $u.color$ = BLACK
```

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 newly discovered vertex turns gray, and is pushed into a queue.
- A gray vertex becomes black when all its neighbors are discovered.

Whenever the search discovers a white vertex v in the course of 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 ***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.

We store the color of each vertex $u \in V$ in the attribute $u.color$ and the predecessor of u in the attribute $u.\pi$. If u has no predecessor (for example, if $u = s$ or u has not been discovered), then $u.\pi$ = NIL. The attribute $u.d$ holds the distance from the source s to vertex u computed by the algorithm. (*Please note that u.d is a different attribute as compared to how we used this attribute in* DFS). The algorithm also uses a first-in, first-out queue Q to manage the set of gray vertices.

13.2.1 Analysis

The operations of enqueuing and dequeuing take $O(1)$ time, and so the total time devoted to queue operations in $O(V)$. 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 thus the total running timde of the BFS procedure is $O(V + E)$. Thus, breadth-first search runs in time linear in the size of the adjacency-list representation of G.

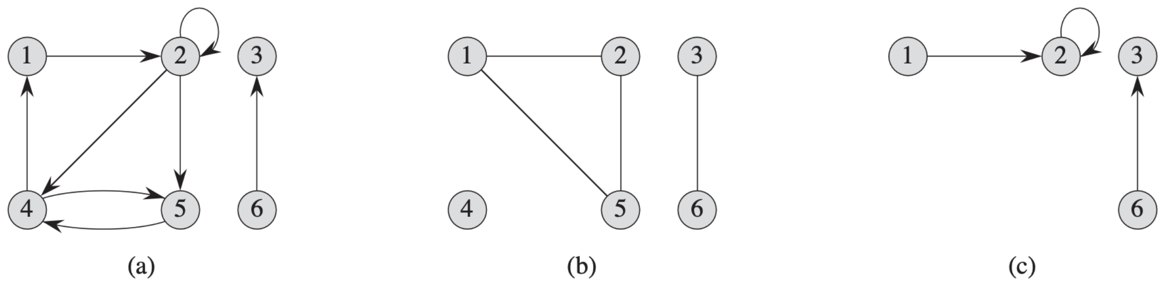

Figure B.2 Directed and undirected graphs. **(a)** A directed graph $G = (V, E)$, where $V = \{1,2,3,4,5,6\}$ and $E = \{(1,2),(2,2),(2,4),(2,5),(4,1),(4,5),(5,4),(6,3)\}$. The edge $(2,2)$ is a self-loop. **(b)** An undirected graph $G = (V, E)$, where $V = \{1,2,3,4,5,6\}$ and $E = \{(1,2),(1,5),(2,5),(3,6)\}$. The vertex 4 is isolated. **(c)** The subgraph of the graph in part (a) induced by the vertex set $\{1,2,3,6\}$.

The ***degree*** of a vertex in an undirected graph is the number of edges incident on it. For example, vertex 2 in Figure B.2(b) has a degree 2. A vertex whose degree is 0, such as vertex 4 in Figure B.2(b), is ***isolated***. In a directed graph, the ***out-degree*** of a vertex is the number of edges leaving it, and the ***in-degree*** of a vertex is the number of entering it. The ***degree*** of a vertex in a directed graph is its in-degree plus its out-degree. Vertex 2 in Figure B.2(a) has in-degree 2, out-degree 3, and degree 5.

(Handshaking) Lemma: If $G = (V, E)$ is an undirected graph, then

$$\sum_{v \in V} \text{degree}(v) = 2|E|$$

, given, each node is connected to another node in a graph. And, there are no self-loop. *(Proof of handshaking lemma has been left as an exercise to the readers.)*

When a node is dequeued in line 11 of BFS procedure, we explore the edges of a vertex equal to the degree of each dequeued vertex. Therefore, for all vertices, it is upper bounded by $2|E|$. Therefore, alternatively, BFS has a runtime $O(|V| + 2|E|) = O(|V| + |E|)$. Please note that the book uses E and $|E|$ interchangeably.

13.3 Shortest paths

Let's define the ***shortest-path distanece*** $\delta(s, v)$ from s to v as 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 a path of length $\delta(s, v)$ from s to v a ***shortest path*** from s to v.

13.4 Breadth-first trees

The procedure BFS builds a breadth-first tree as it searches the graph, as Figure 22.3 illustrates. The tree corresponds to the π attributes. More formally, for a graph $G = (V, E)$ with source s, we define the ***predecessor subgraph*** of G as $G_\pi = (V_{pi}, E_{pi})$, where

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

and

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

The predecessor subgraph G_π is a ***breadth-first tree*** if V_π consists of the vertices reachable from s and, for all $v \in V_\pi$, the subgraph G_π 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 $|E_\pi| = |V_\pi - 1$ (see Theorem B.2 from CLRS). We call the edges in E_π ***tree edges***.

The following procedure prints out the vertices on a shortest path from s to v, assuming that BFS has already computed a breadth-first tree:

```
PRINT-PATH($G, s, v$)
1  if $v == s$
2      print $s$
3  elseif $v.\pi ==$ NIL
4      print "no path from" $s$ "to" $v$ "exists"
5  else PRINT-PATH($G, s, v.\pi$)
6      print $v$
```

This procedure runs in time linear in the number of vertices in the path printed, since each recursive call is for a path one vertex shorter.