Lecture 13: Elementary Graph Algorithms

MCS/CS 401 Computer Algorithms I — July 18, 2022
Instructor: Danko Adrovic

Lecture Outline

- Elementary Graph Algorithms
  - Graph terminology
  - Representations of graphs
  - Breadth-first search
  - Depth-first search
  - Depth-first search on an undirected graph (simplified version)

1 Elementary Graph Algorithms

A graph is a fundamental structure in discrete mathematics and in computer science. In computer science, graphs are frequently used as way to represent and/or embed real world problems in a mathematical setting. Once embedded in mathematical settings, mathematical tools can be used to analyze the problems. Among these tools, *graph algorithms* for *searching* and general exploration of *structural properties* of the graph play a prominent role.

Before we can develop algorithms to run on graphs, we have to 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.

Once the representation of a graph has been established, we will focus on several graph algorithms for simple graph-searching. These are breadth-first search algorithm and the resulting breadth-first tree, depth-first search algorithm and its application, called topological sort of a directed acyclic graph, and a second application of depth-first search which finds the strongly connected components of a directed graph.

1.1 Graph terminology

Basic Graphs

- A node in a graph is called a *vertex*.
- A connection between two vertices is an *edge*.

A *graph* G is defined by a tuple (V, E) and usually denoted $G = (V, E)$:

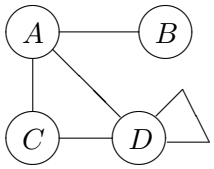
- V is the set of vertices, and
- E is the set of edges.

$|V|$ – number of vertices

$|E|$ – number of edges

An example:

$\deg(A) = 3$



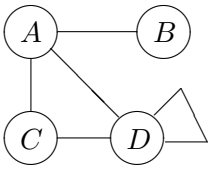
$V = \{A, B, C, D\}$

$E = \{(A, B), (A, C), (A, D), (C, D), (D, D)\}$

Two vertices are *adjacent* if there is an edge between them. A *path* is a sequence of successively adjacent vertices. For example, a path from B to D is B, A, D.

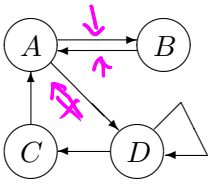
Undirected and Directed Graphs

In an *undirected* graph (which is the default), the edge (A, B) means: there is an edge from A to B and there is an edge from B to A.



$V = \{A, B, C, D\}$

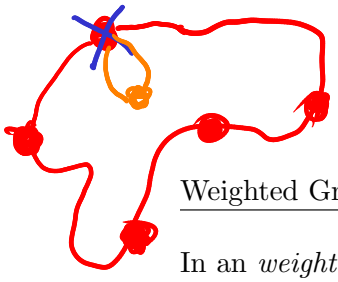
$E = \{(A, B), (A, C), (A, D), (C, D), (D, D)\}$



$V = \{A, B, C, D\}$

$E = \{(A, B), (B, A), (C, A), (A, D), (D, C), (D, D)\}$

In a *directed* graph or *digraph*, (A, B) means there is an edge from A to B and (B, A) means there is an edge from B to A.



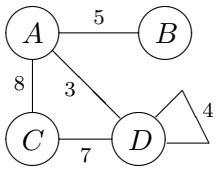
weighted Graph ←

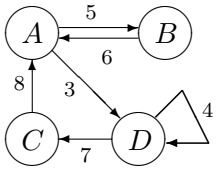
—u— DiGraph ←

Weighted Graphs

In an *weighted* graph, a value (a weight) is associated to every edge.

deg(A) = 5+3+8 = 16





$\deg_{in}(A) = 8+6 = 14$

$\deg_{out}(A) = 5+3 = 8$

Edges in a weighted graphs are triplets (u, v, w), with w the weight of the edge which connects u to v.

A *cycle* is a path where the first vertex equals the last one. For example: A, D, C, A is a cycle. A graph without cycles can be viewed as a tree.

The *degree* of a vertex is the number of edges that contain that vertex.

|V|

|E|

1.2 Representations of graphs

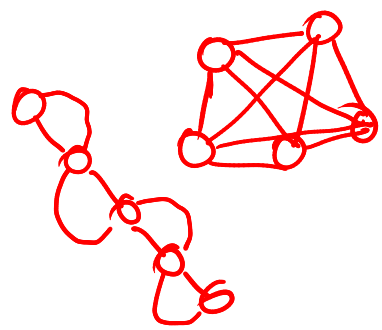
A graph G is a structure consisting of a set vertices (nodes) and edges (undirected or directed). We denote such as graph $G = (V, E)$, where V denotes the *set* of *vertices* and E denotes the set of *edges*. There are two standard ways to represent a graph $G = (V, E)$: as a collection of *adjacency-lists* or as an *adjacency-matrix*.

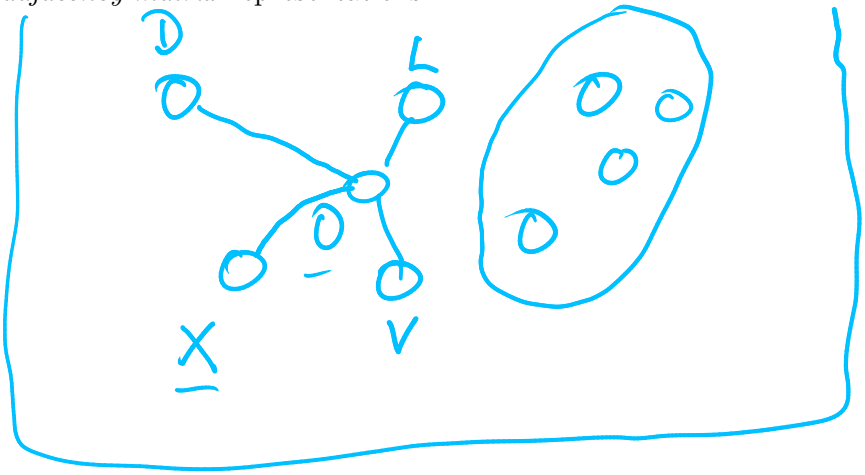
The adjacency-list representation provides a compact way to represent **sparse** graphs, that is, those for which $|E|$ is much less than $|V|^2$ and it is usually the method of choice. Majority of the graph algorithms covered in our textbook, however, assume that an input graph is represented in *adjacency list* form.

We may prefer an adjacency-matrix representation, however, when the graph is dense, that is, $|E|$ is close to $|V|^2$, or when we need to be able to tell quickly if there is an edge connecting two given vertices.

The following two images show an undirected graph (top) and directed graph (bottom) *adjacency lists* and *adjacency matrix* representations.

$|E| << |V|^2$





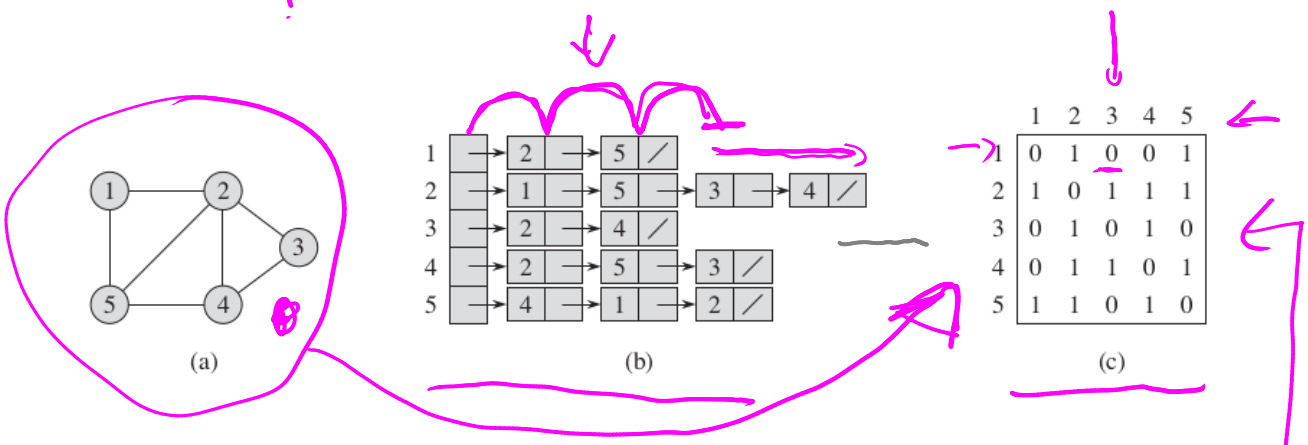


Figure 22.1 Two representations of an undirected graph. **(a)** An undirected graph G with 5 vertices and 7 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

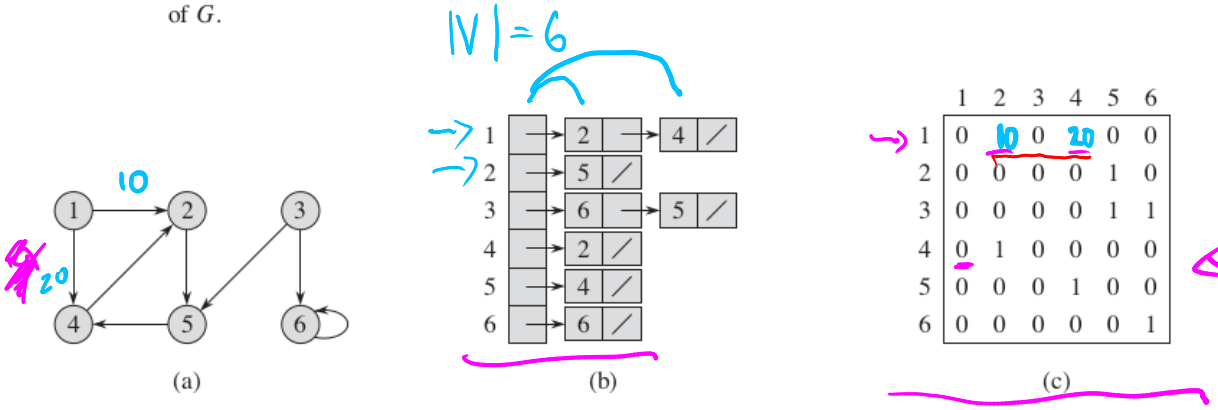


Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G with 6 vertices and 8 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

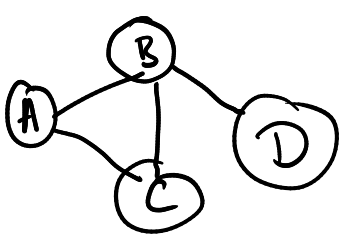
The adjacency-list representation of a graph $G = (V, E)$ consists of an array Adj of $|V|$ lists, one for each vertex in V. For each $u \in V$, the adjacency list $Adj[u]$ contains all the vertices v such that there is an edge (u, v) E. That is, $Adj[u]$ consists of all the vertices adjacent to u in G.

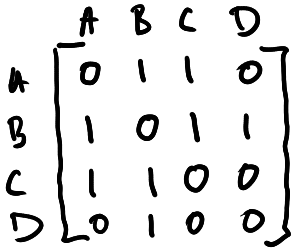
In addition, we can easily adapt adjacency lists to represent **weighted graphs**, that is, graphs for which each edge has an associated weight, typically given by a weight function $w : E \mapsto \mathbb{R}$. For example, let $G = (V, E)$ be a weighted graph with weight function w. We simply store the weight $w(u, v)$ of the edge $(u, v) \in E$ with vertex v in us adjacency list. The adjacency-list representation is very flexible in that we can modify it to support many other graph variants.

For both directed and undirected graphs, the adjacency-list representation has the desirable property that the amount of memory it requires is for graph representation is $\Theta(V + E)$.

For the adjacency-matrix representation of a graph $G = (V, E)$, we assume that the vertices are numbered $1, 2, 3, ..., |V|$ in some arbitrary manner. Then the adjacency-matrix representation of a graph G consists of a $|V| \times |V|$ matrix $A(a_{ij})$, such that

$$a(ij) = \begin{cases} 1 & if\ (i,j) \in E \\ 0 & otherwise \end{cases}$$



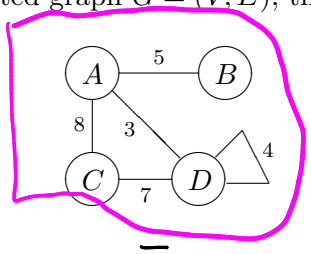


where indices i and j represent the vertices in the graph ((i, j) denotes an edge between i and j). In addition, indices i and j can be thought of as **rows** and **columns** of the adjacency-matrix. While the adjacency-list helps with the input format for an algorithm to run, the adjacency-matrix representation is quite powerful as it allows us to connect graphs with the field of *linear algebra*.

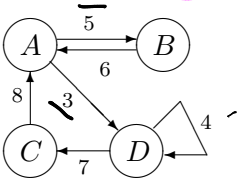
For both directed and undirected graphs, the adjacency matrix of a graph requires $\Theta(|V|^2)$of memory, that is, the bound is independent of the number of edges in the graph.

Adjacency Matrices for Weighted Graphs

For a weighted graph $G = (V, E)$, the *adjacency matrix* A is given by



	A	B	C	D
A	0	5	8	3
B	5	0	0	0
C	8	0	0	7
D	3	0	7	4



	A	B	C	D
A	0	5	0	3
B	6	0	0	0
C	8	0	0	0
D	0	0	7	4

1.3 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
```

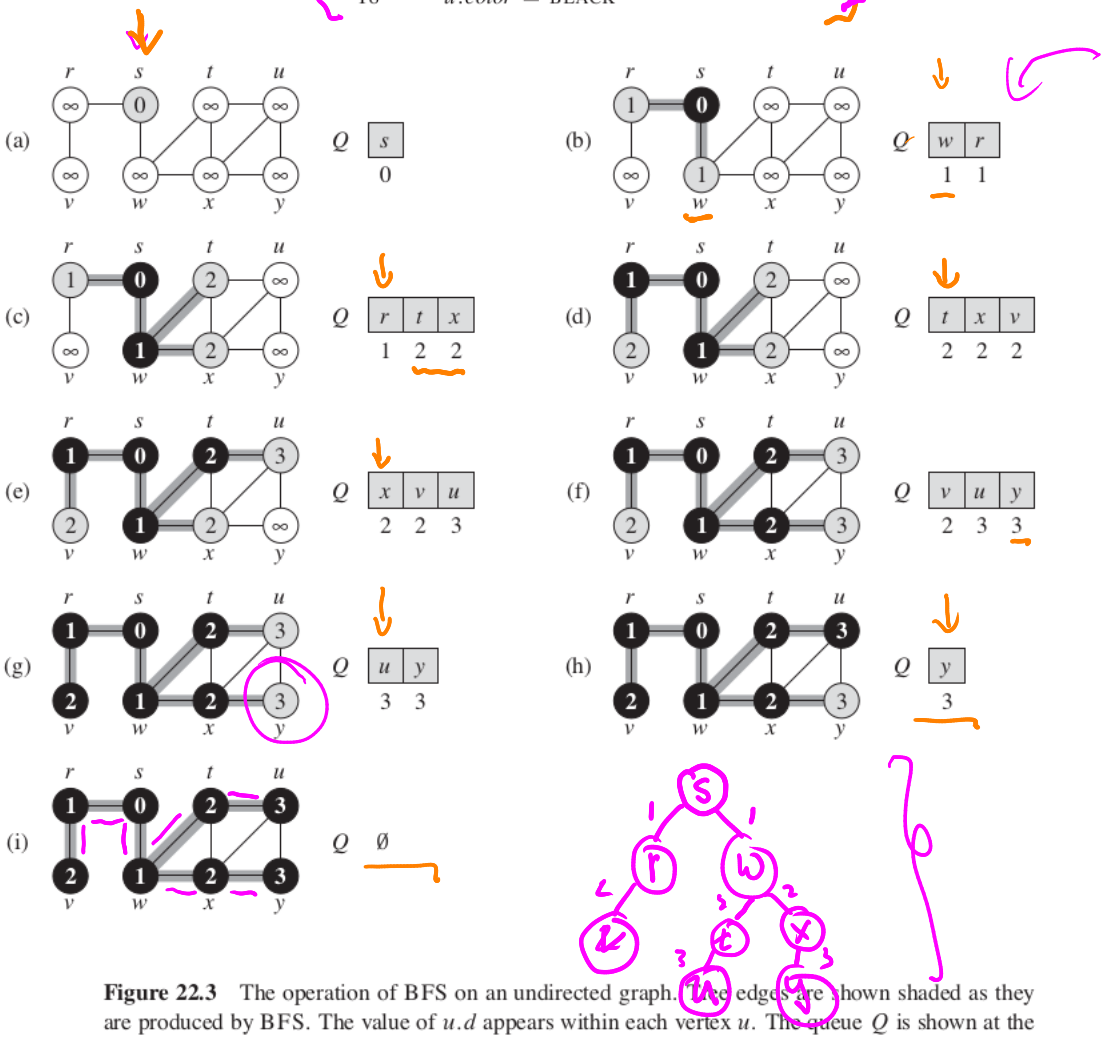


Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. The value of $u.d$ appears within each vertex u. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances appear below vertices in the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

1.4 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".

Classification of edges

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 following illustrates a DFS algorithm on a directed graph with cycles. In general, the algorithm is slightly easier on undirected graphs. 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.

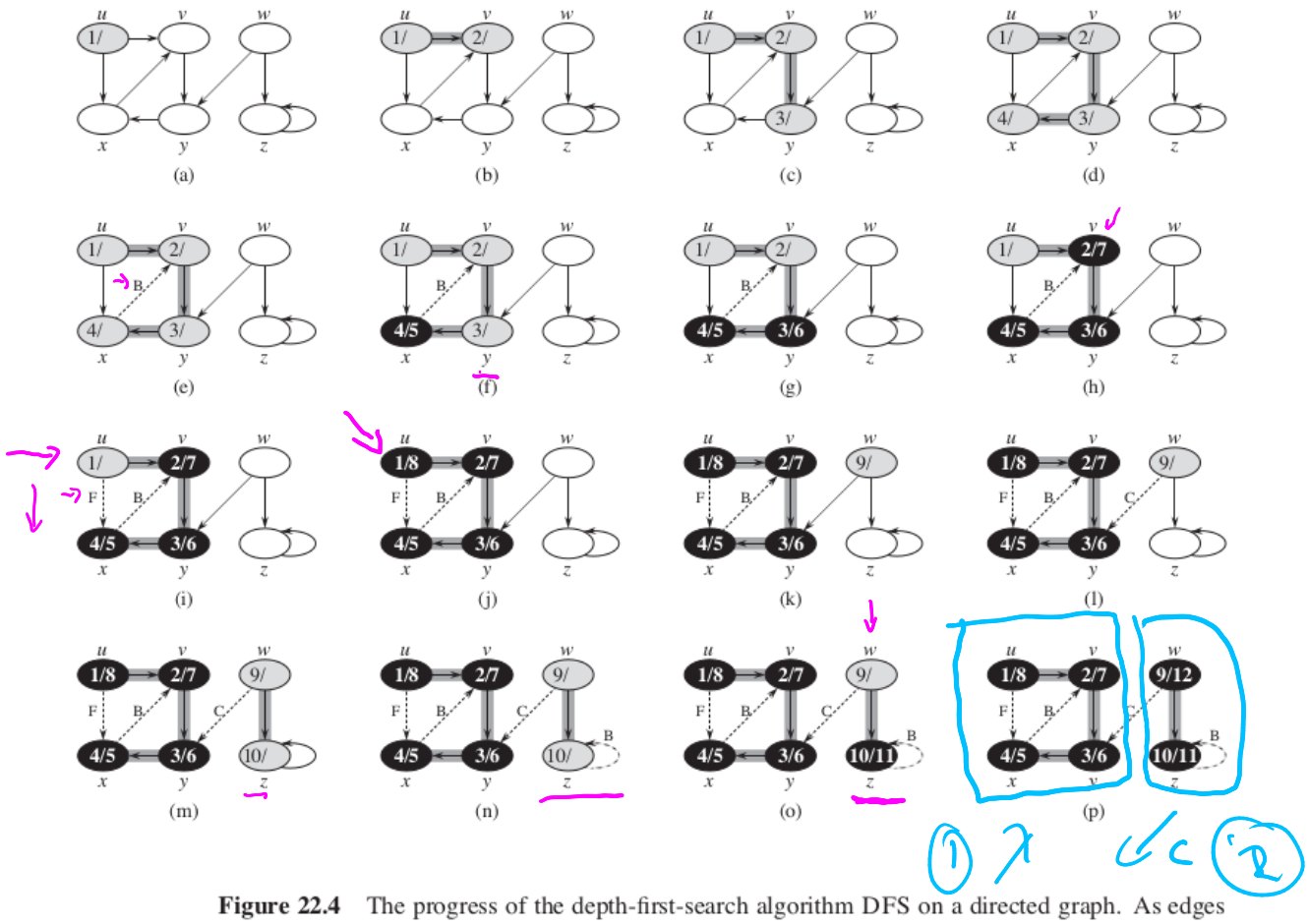
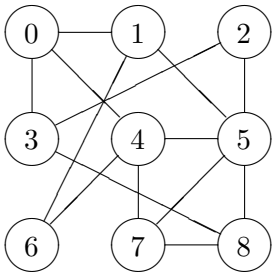


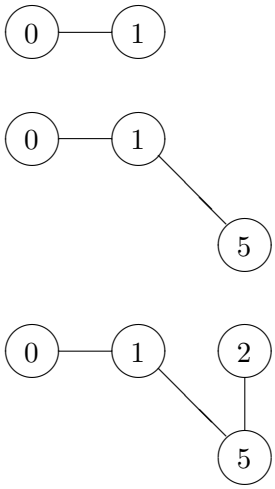
Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges.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

1.5 Depth-first search on an undirected graph (simplified version)

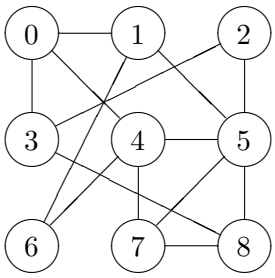
The following illustration gives an example of a DFS algorithm running on an undirected graph. In addition, when selecting a new vertex to explore, we use the convention that we always select the vertex with the smallest value, then then continue selecting in ascending order. This is merely a convention in this example for a clearer illustration.

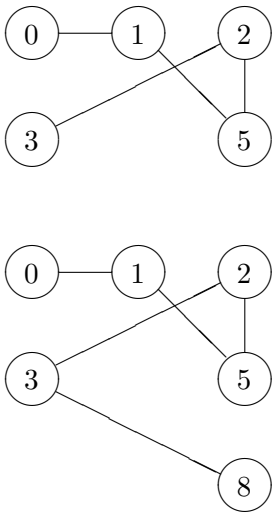
We start at vertex 0.



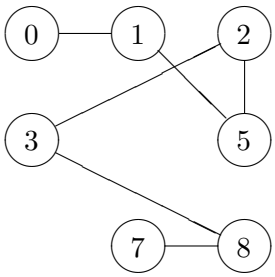


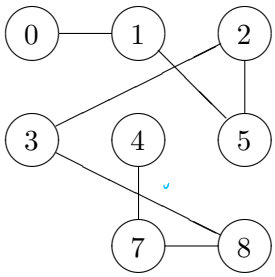
DFS continues



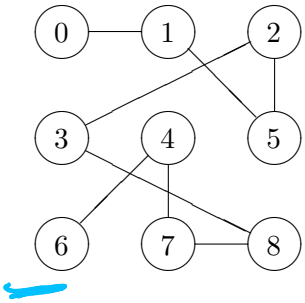


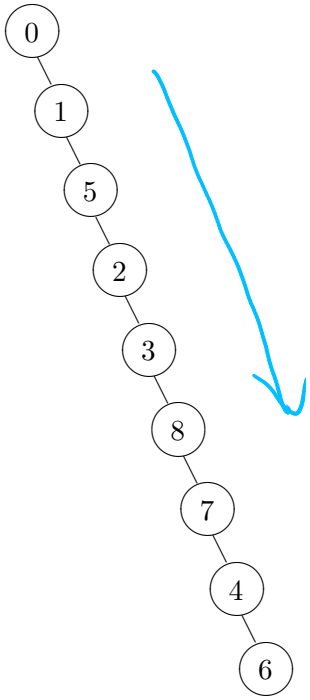
DFS continues





The resulting depth-first search tree





The *depth-first search tree of a graph* contains all the vertices of the graph and the edges visited in the depth-first search.