Graphs

- Graph G(V,E)
- V Vertices or Nodes
- E Edges or Links: pairs of vertices
- D Directed vs. Undirected edges
- Weighted vs Unweighted
- Graphs can be augmented to store extra info (e.g., city population, oil flow capacity, etc.)
- Paths and Cycles
- Subgraphs G'(V',E'), where V' is a subset of V and E' is a subset of E
- Trees and Spanning trees

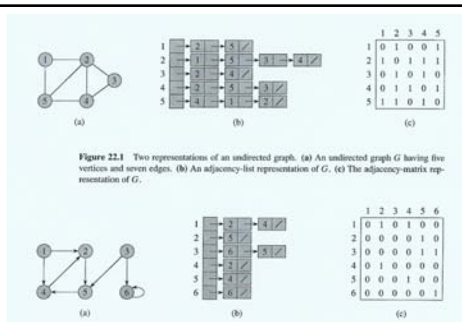


Figure 22.1 Two representations of an undirected graph. (a) An undirected graph G having five vertices and seven edges. (b) An adjacency-list representation of G. (c) The adjacency-matrix representation of G.

Figure 22.2 Two representations of a directed graph. (a) A directed graph G having six vertices and eight edges. (b) An adjacency-list representation of G. (c) The adjacency-matrix representation of G.

Graph Traversal

- Visit every vertex and every edge.
- Traversal has to be systematic so that no vertex or edge is missed.
- Just as tree traversals can be modified to solve several tree-related problems, graph traversals can be modified to solve several problems.

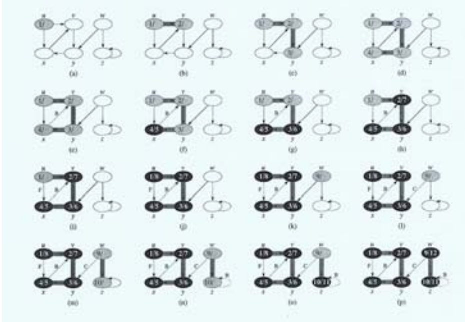


Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges.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


```
DFS(G)
For each vertex u ∈ V[G] do
    color[u] ← WHITE
    π[u] ← NIL
Time ← 0
For each vertex u ∈ V[G] do
    if color[u] = WHITE then
        DFS-VISIT(u)
```

```
DFS-VISIT(u)
VisitVertex(u)
Color[u] ← GRAY
Time ← Time + 1
d[u] ← Time
for each v ∈ Adj[u] do
    VisitEdge(u,v)
    if (v ≠ π[u]) then
        if (color[v] = WHITE) then
            π[v] ← u
            DFS-VISIT(v)
color[u] ← BLACK
F[u] ← Time ← Time + 1
```

Depth First Search

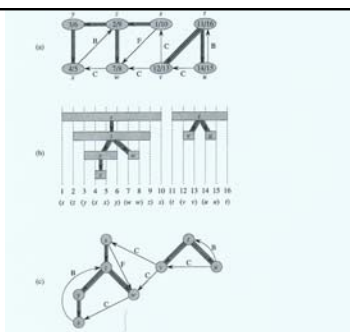


Figure 22.5 Properties of depth-first search. (a) The result of a depth-first search of a directed graph. Vertices are timestamped and edge types are indicated as in Figure 22.4. (b) Intervals for the discovery time and finishing time of each vertex correspond to the parenthesization shown. Each rectangle spans the interval given by the discovery and finishing times of the corresponding vertex. Tree edges are shown. If two intervals overlap, then one is nested within the other, and the vertex corresponding to the smaller interval is a descendant of the vertex corresponding to the larger. (c) The graph of part (a) redrawn with all tree and forward edges going down within a depth-first tree and all back edges going up from a descendant to an ancestor.

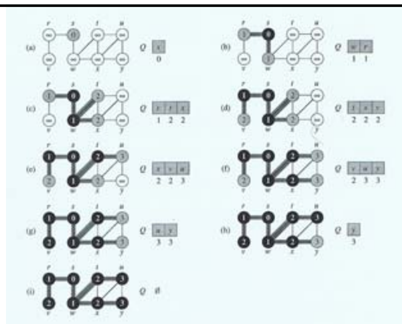


Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. Within each vertex u is shown $d[u]$. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances are shown next to vertices in the queue.

Breadth First Search

```
BFS(G,s)
1.  For each vertex u ∈ V[G] – {s} do
2.      color[u] ← WHITE
3.      d[u] ← ∞
4.      π[u] ← NIL
5.  Color[u] ← GRAY
6.  D[s] ← 0
7.  π[s] ← NIL
8.  Q ← Φ
9.  ENQUEUE(Q,s)
10. While Q ≠ Φ do
11.     u ← DEQUEUE(Q)
12.     VisitVertex(u)
13.     for each v ∈ Adj[u] do
14.         VisitEdge(u,v)
15.         if (color[v] = WHITE) then
16.             color[v] ← GRAY
17.             d[v] ← d[u] + 1
18.             π[v] ← u
19.             ENQUEUE(Q,v)
20.     color[u] ← BLACK
```


Figure 14.33
An activity-node graph

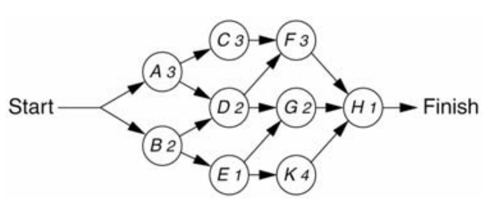


Figure 14.30A

A topological sort. The conventions are the same as those in Figure 14.21 (continued).

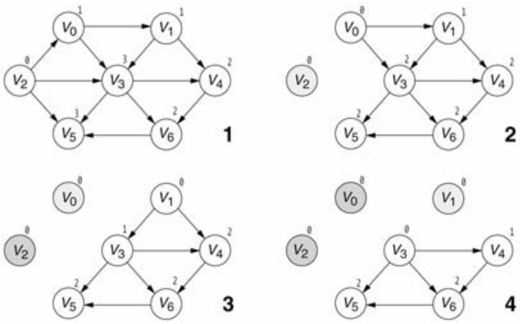


Figure 14.30B

A topological sort. The conventions are the same as those in Figure 14.21.

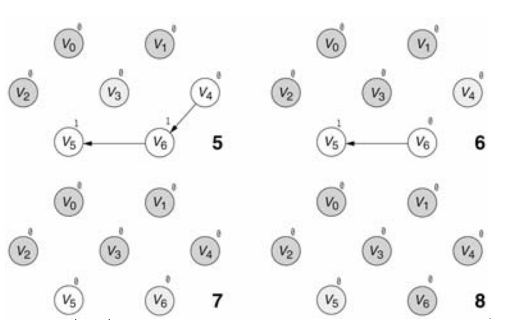


Figure 14.31A

The stages of acyclic graph algorithm. The conventions are the same as those in Figure 14.21 (*continued*).

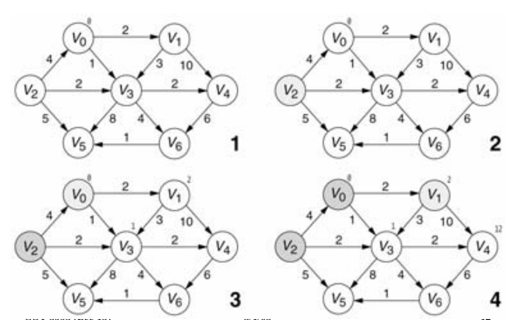


Figure 14.31B

The stages of acyclic graph algorithm. The conventions are the same as those in Figure 14.21.

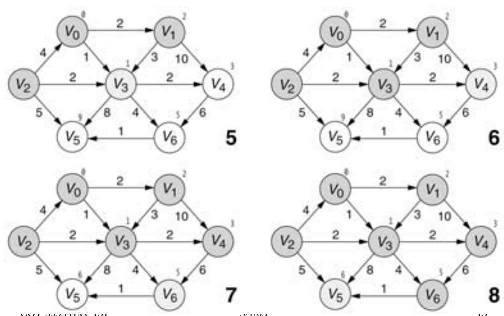


Figure 14.34

An event-node graph

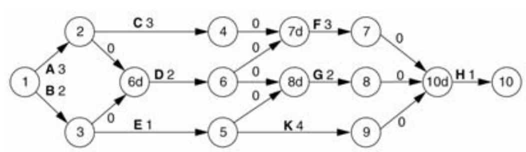


Figure 14.35

Earliest completion times

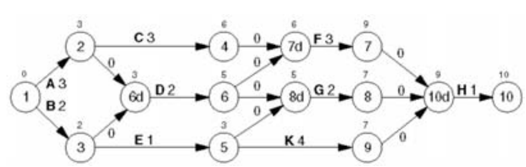


Figure 14.36
Latest completion times

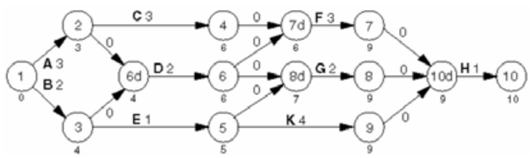
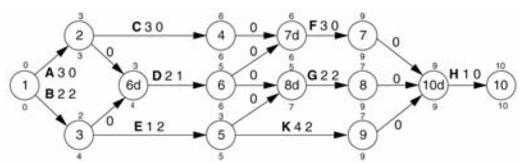


Figure 14.37
Earliest completion time, latest completion time, and slack (additional edge item)



Connectivity

- A (simple) undirected graph is connected if there exists a path between every pair of vertices.
- If a graph is not connected, then G'(V',E') is a connected component of the graph G(V,E) if V' is a maximal subset of vertices from V that induces a connected subgraph. (What is the meaning of maximal?)
- The connected components of a graph correspond to a partition of the set of the vertices. (What is the meaning of partition?)
- How to compute all the connected components?
  - Use DFS or BFS.

Minimum Spanning Tree

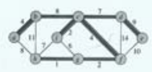


Figure 23.1 A minimum spanning tree for a connected graph. The weights on edges are shown, and the edges in a minimum spanning tree are shaded. The total weight of the tree shown is 37. This minimum spanning tree is not unique: removing the edge (b, c) and replacing it with the edge (a, h) yields another spanning tree with weight 37.

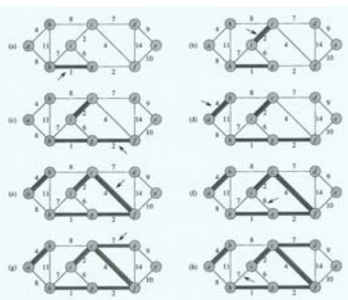
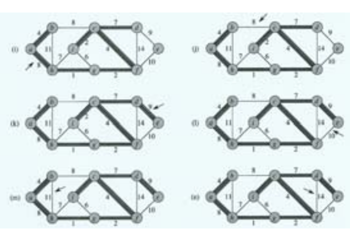


Figure 23.4 The execution of Kruskal's algorithm on the graph from Figure 23.1. Shaded edges belong to the forest A being grown. The edges are considered by the algorithm in sorted order by weight. An arrow points to the edge under consideration at each step of the algorithm. If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.



Minimum Spanning Tree

```
MST-KRUSKAL(G, w)
A ← ∅
for each vertex v ∈ V[G]
    do MAKE-SET(v)
sort the edges of E by nondecreasing weight w
for each edge (u, v) ∈ E, in order by nondecreasing weigh
    do if FIND-SET(u) ≠ FIND-SET(v)
        then A ← A ∪ {(u, v)}
             UNION(u, v)
return A
```

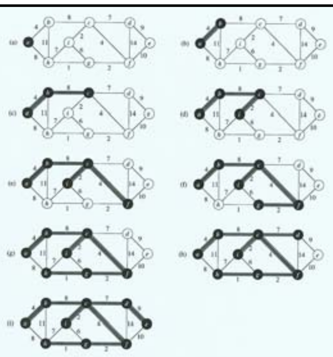



Figure 23.5 The execution of Prim's algorithm on the graph from Figure 23.1. The root vertex is a. Shaded edges are in the tree being grown, and the vertices in the tree are shown in black. At each step of the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing the cut is added to the tree. In the second step, for example, the algorithm has a choice of adding either edge (b, c) or edge (a, h) to the tree since both are light edges crossing the cut.


```
MST-KRUSKAL(G, w)
A ← ∅
for each vertex v ∈ V[G]
    do MAKE-SET(v)
sort the edges of E by nondecreasing weight w
for each edge (u, v) ∈ E, in order by nondecreasing weigh
    do if FIND-SET(u) ≠ FIND-SET(v)
        then A ← A ∪ {(u, v)}
             UNION(u, v)
return A

MST-PRIM(G, w, r)
Q ← V[G]
for each u ∈ Q
    do key[u] ← ∞
key[r] ← 0
π[r] ← NIL
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       for each v ∈ Adj[u]
           do if v ∈ Q and w(u, v) < key[v]
               then π[v] ← u
                    key[v] ← w(u, v)
```

Proof of Correctness: MST Algorithms

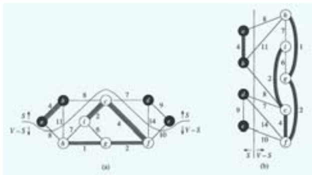


Figure 23.2 Two ways of viewing a cut $(S, V-S)$ of the graph from Figure 23.1. **(a)** The vertices in the set S are shown in black, and those in $V-S$ are shown in white. The edges crossing the cut are those connecting white vertices with black vertices. The edge (d, c) is the unique light edge crossing the cut. A subset A of the edges is shaded; note that the cut $(S, V-S)$ respects A, since no edge of A crosses the cut. **(b)** The same graph with the vertices in the set S on the left and the vertices in the set $V-S$ on the right. An edge crosses the cut if it connects a vertex on the left with a vertex on the right.

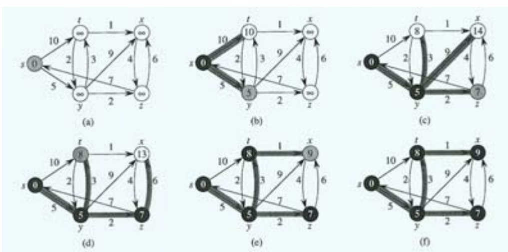


Figure 24.6 The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates are shown within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S, and white vertices are in the min-priority queue $Q = V - S$. **(a)** The situation just before the first iteration of the **while** loop of lines 4–8. The shaded vertex has the minimum d value and is chosen as vertex u in line 5. **(b)–(f)** The situation after each successive iteration of the **while** loop. The shaded vertex in each part is chosen as vertex u in line 5 of the next iteration. The d and π values shown in part (f) are the final values.

Dijkstra's Single Source Shortest Path Algorithm

```
DIJKSTRA(G, w, s)
1.  // INITIALIZE-SINGLE-SOURCE(G, s)
        for each vertex v ∈ V[G]
            do d[v] ← ∞
               π[v] ← NIL
        d[s] ← 0
2.  S ← ∅
3.  Q ← V[G]
4.  while Q ≠ ∅
5.      do u ← EXTRACT-MIN(Q)
6.         S ← S ∪ {u}
7.         for each v ∈ Adj[u]
8.             do // RELAX(u, v, w)
                   if d[v] > d[u] + w(u, v)
                     then d[v] ← d[u] + w(u, v)
                          π[v] ← u
```

```
DIJKSTRA(G, w, s)
// INITIALIZE-SINGLE-SOURCE(G, s)
    for each vertex v ∈ V[G]
        do d[v] ← ∞
           π[v] ← NIL
    d[s] ← 0
S ← ∅
Q ← V[G]
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each v ∈ Adj[u]
           do // RELAX(u, v, w)
              if d[v] > d[u] + w(u, v)
                 then d[v] ← d[u] + w(u, v)
                      π[v] ← u
```

```
MST-PRIM(G, w, r)
Q ← V[G]
for each u ∈ Q
    do key[u] ← ∞
key[r] ← 0
π[r] ← NIL
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       for each v ∈ Adj[u]
           do if v ∈ Q and w(u, v) < key[v]
                 then π[v] ← u
                      key[v] ← w(u, v)
```


All Pairs Shortest Path Algorithm

- Invoke Dijkstra's SSSP algorithm n times.
- Or use dynamic programming. How?

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Figure 25.4 The sequence of matrices $D^{(k)}$ and $\Pi^{(k)}$ computed by the Floyd-Warshall algorithm for the graph in Figure 25.1.

Figure 14.38
Worst-case running times of various graph algorithms

Type of Graph Problem	Running Time	Comments
Unweighted	$O(\|E\|)$	Breadth-first search
Weighted, no negative edges	$O(\|E\| \log \|V\|)$	Dijkstra's algorithm
Weighted, negative edges	$O(\|E\| \cdot \|V\|)$	Bellman–Ford algorithm
Weighted, acyclic	$O(\|E\|)$	Uses topological sort