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.

Biconnectivity: Generalizing Connectivity

- A tree is a minimally connected graph.
- Removing a vertex from a connected graph may make it disconnected.
- A graph is biconnected if removing a single vertex does not disconnect the graph.
- Alternatively, a graph is biconnected if for every pair of vertices there exists at least 2 disjoint paths between them.
- A graph is k-connected if for every pair of vertices there exists at least k disjoint paths between them. Alternatively, removal of any k-1 vertices does not disconnect the graph.

Biconnected Components

- If a graph is not biconnected, it can be decomposed into biconnected components.
- An articulation point is a vertex whose removal disconnects the graph.
- **Claim:** If a graph is not biconnected, it must have an articulation point. Proof?
- A biconnected component of a simple undirected graph G(V,E) is a maximal set of edges from E that induces a biconnected subgraph.

Biconnected Components

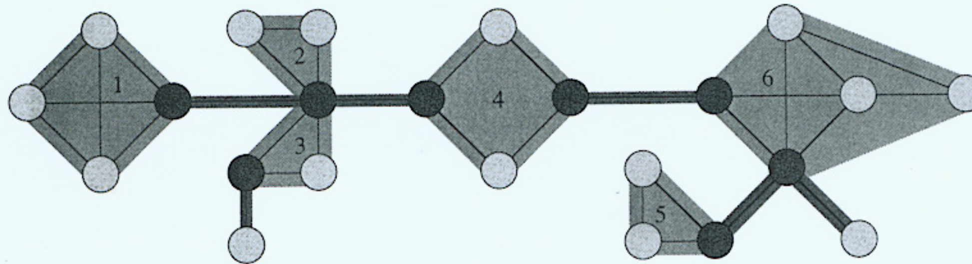
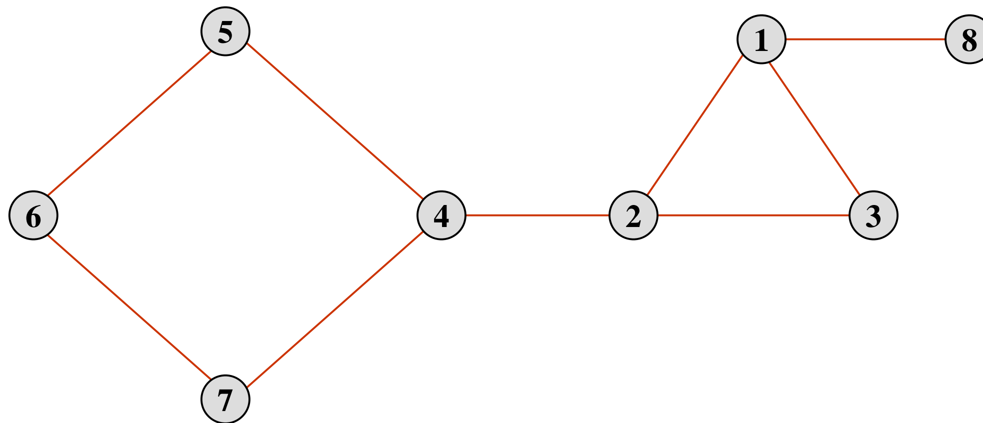


Figure 22.10 The articulation points, bridges, and biconnected components of a connected, undirected graph for use in Problem 22-2. The articulation points are the heavily shaded vertices, the bridges are the heavily shaded edges, and the biconnected components are the edges in the shaded regions, with a *bcc* numbering shown.

```
BCC(G, u) // Compute the biconnected components of G
          // starting from vertex u
1.  Color[u] ← GRAY
2.  Low[u] ← d[u] ← Time ← Time + 1
3.  Put u on stack S
4.  for each v ∈ Adj[u] do
5.      if (v ≠ π[u]) and (color[v] ≠ BLACK) then
6.          if (TopOfStack(S) ≠ u) then put u on stack S
7.          Put edge (u,v) on stack S
8.          if (color[v] = WHITE) then
9.              π[v] ← u
10.             BCC(G, v)
11.             if (Low[v] >= d[u]) then // u is an articul. pt.
12.                 // Output next biconnected component
13.                 Pop S until u is reached
14.                 Push u back on S
15.             Low[u] = min { Low[u], Low[v] }
16.         else Low[u] = min { Low[u], d[v] } // back edge
17. color[u] ← BLACK
18. F[u] ← Time ← Time + 1
```

BCC Example



Minimum Spanning Tree

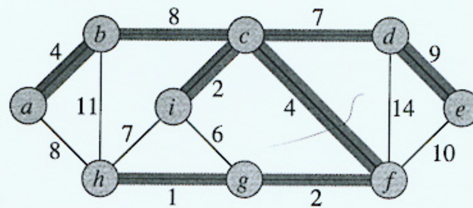


Figure 23.1 A minimum spanning tree for a connected graph. The weights on edges are shown, and the edges in a minimum spanning tree are shaded. The total weight of the tree shown is 37. This minimum spanning tree is not unique: removing the edge (b, c) and replacing it with the edge (a, h) yields another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

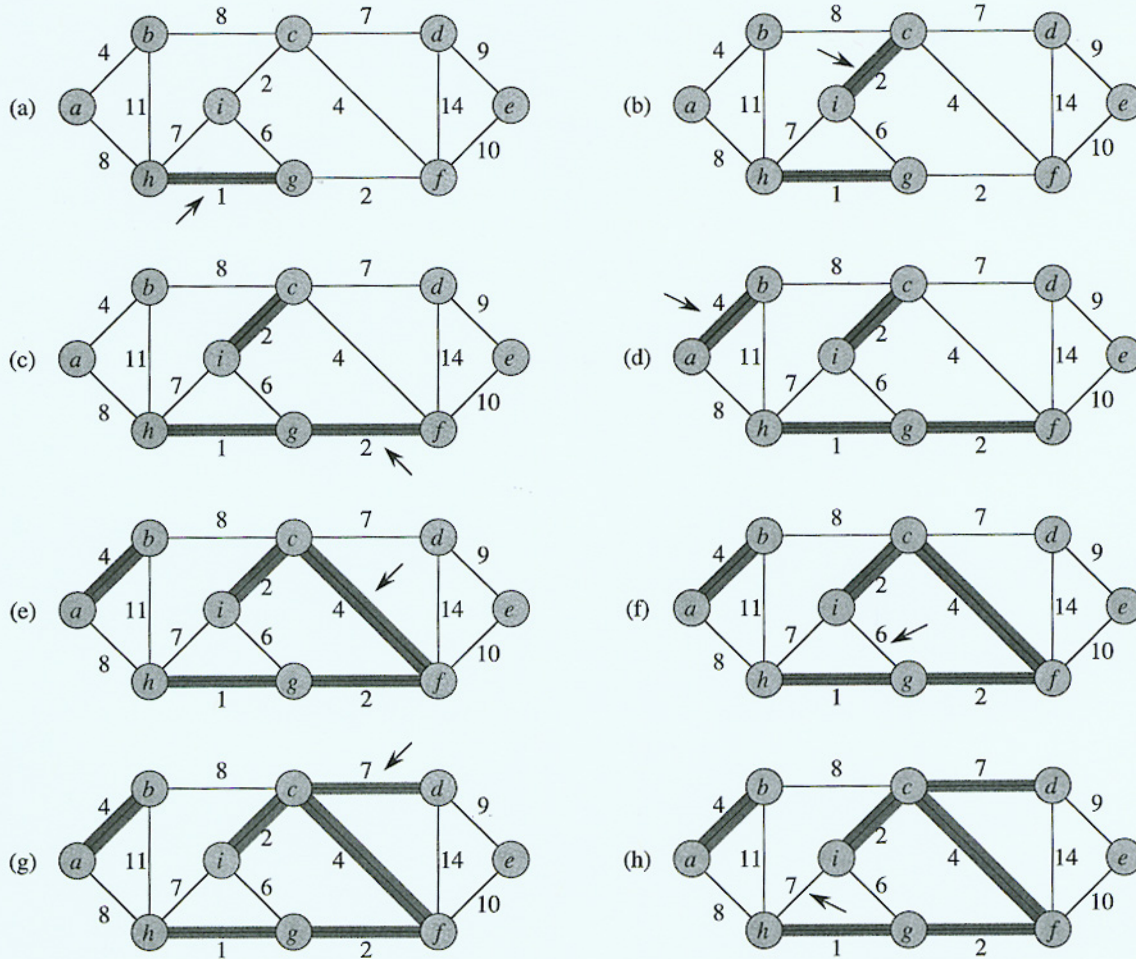
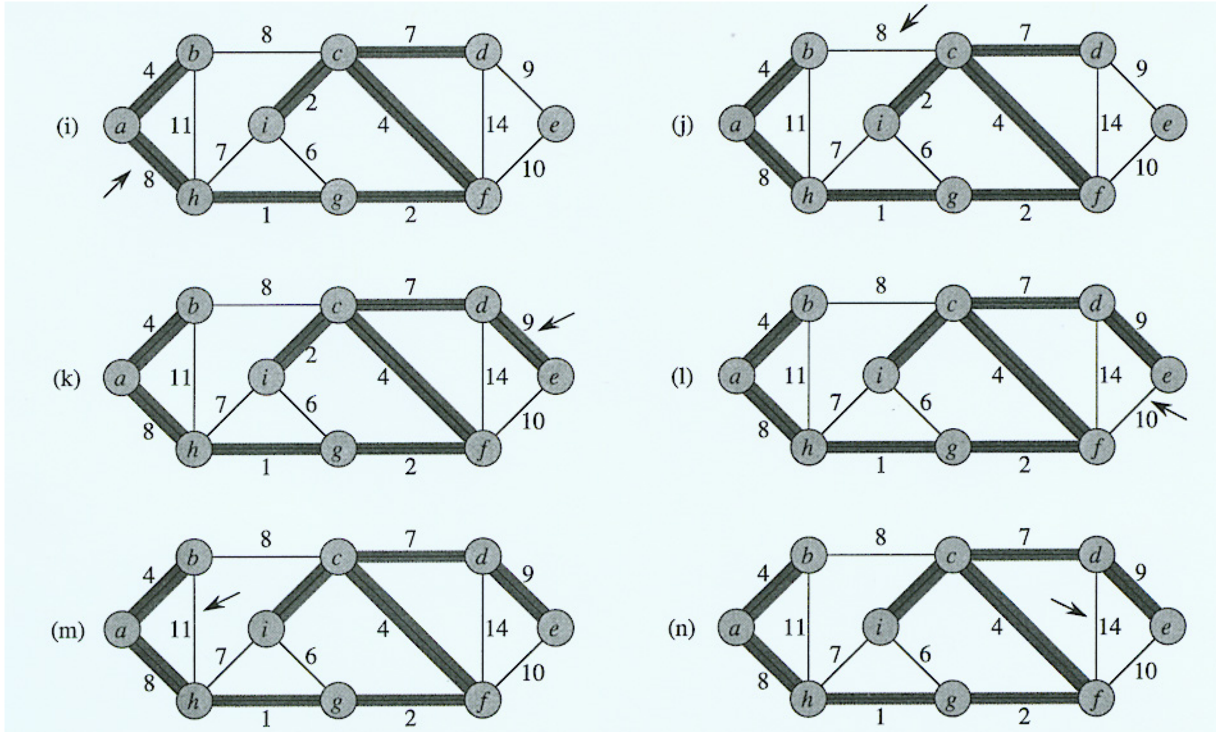


Figure 23.4 The execution of Kruskal's algorithm on the graph from Figure 23.1. Shaded edges belong to the forest A being grown. The edges are considered by the algorithm in sorted order by weight. An arrow points to the edge under consideration at each step of the algorithm. If the edge joins two distinct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


(i)
a b c d e f g h i
4 8 7 9 11 2 4 14 7 6 8 10 1 2
(j)
a b c d e f g h i
4 8 7 9 11 2 4 14 7 6 8 10 1 2
(k)
a b c d e f g h i
4 8 7 9 11 2 4 14 7 6 8 10 1 2
(l)
a b c d e f g h i
4 8 7 9 11 4 14 7 6 8 10 1 2
(m)
a b c d e f g h i
4 8 7 9 11 4 14 7 6 8 10 1 2
(n)
a b c d e f g h i
4 8 7 9 11 4 14 7 6 8 10 1 2

Minimum Spanning Tree

MST-KRUSKAL(G, w)

1. $A \leftarrow \emptyset$
2. **for** each vertex $v \in V[G]$
3. **do** MAKE-SET(v)
4. sort the edges of E by nondecreasing weight w
5. **for** each edge $(u, v) \in E$, in order by nondecreasing weigh
6. **do if** FIND-SET$(u) \neq$ FIND-SET(v)
7. **then** $A \leftarrow A \cup \{(u, v)\}$
8. UNION(u, v)
9. **return** A

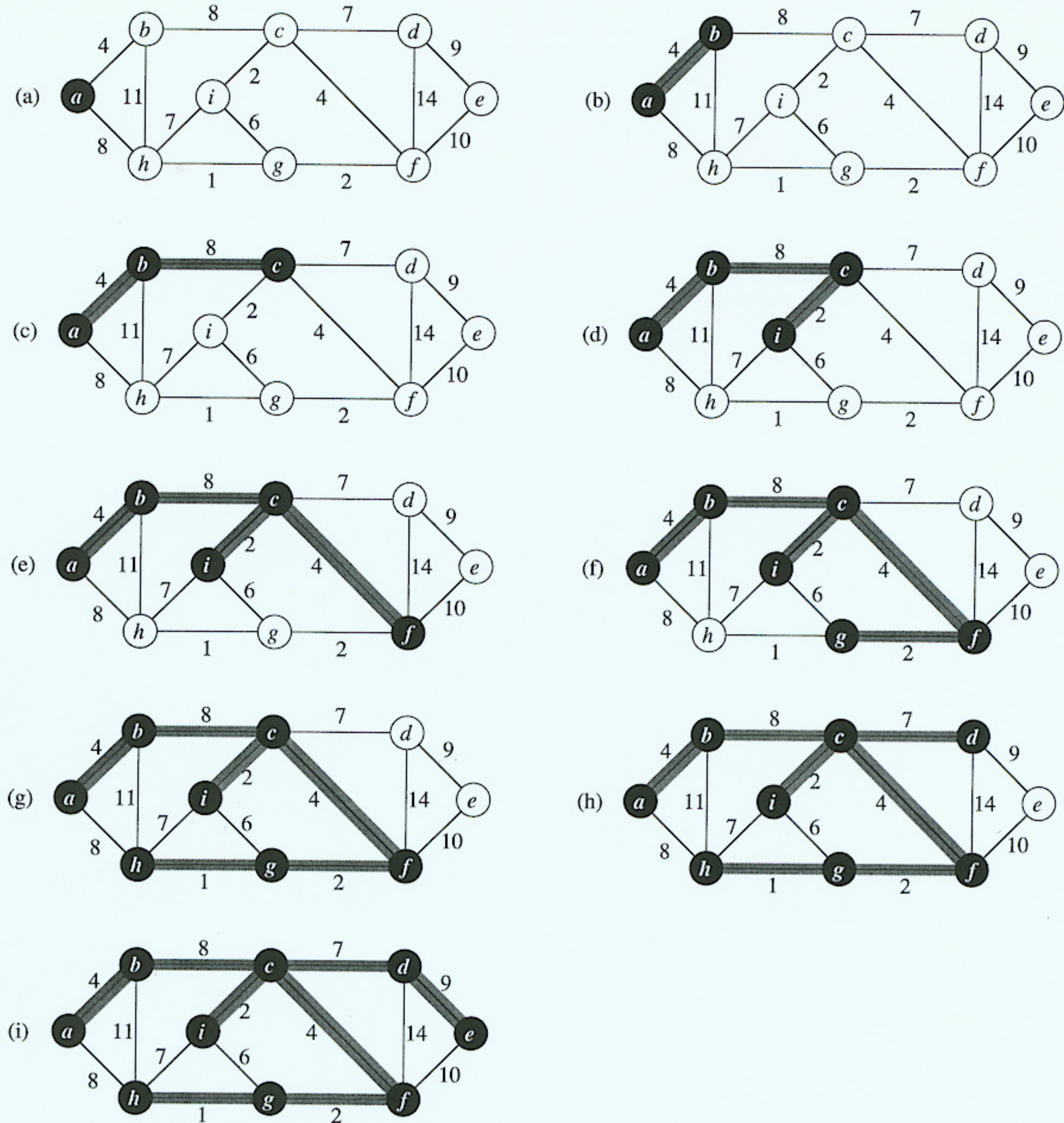


Figure 23.5 The execution of Prim's algorithm on the graph from Figure 23.1. The root vertex is a. Shaded edges are in the tree being grown, and the vertices in the tree are shown in black. At each step of the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing 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)
1.   A ← ∅
2.   for each vertex v ∈ V[G]
3.        do MAKE-SET(v)
4.   sort the edges of E by nondecreasing weight w
5.   for each edge (u, v) ∈ E, in order by nondecreasing weigh
6.        do if FIND-SET(u) ≠ FIND-SET(v)
7.              then A ← A ∪ {(u, v)}
8.                   UNION(u, v)
9.   return A
```

```
MST-PRIM(G, w, r)
1.   Q ← V[G]
2.   for each u ∈ Q
3.        do key[u] ← ∞
4.   key[r] ← 0
5.   π[r] ← NIL
6.   while Q ≠ ∅
7.          do u ← EXTRACT-MIN(Q)
8.              for each v ∈ Adj[u]
9.                   do if v ∈ Q and w(u, v) < key[v]
10.                        then π[v] ← u
11.                             key[v] ← w(u, v)
```

Proof of Correctness: MST Algorithms

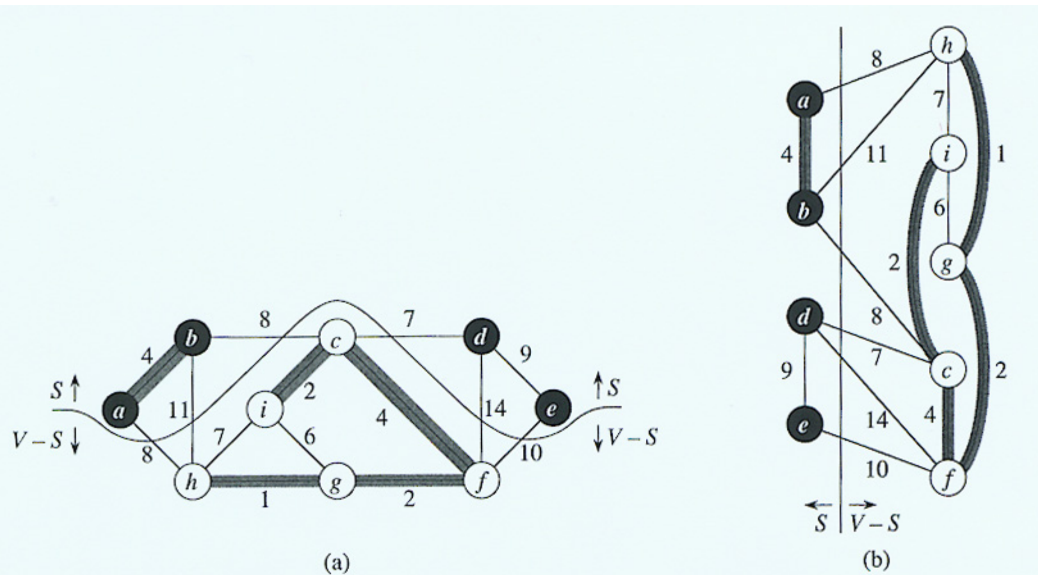


Figure 23.2 Two ways of viewing a cut $(S, V - S)$ of the graph from Figure 23.1. **(a)** The vertices in the set S are shown in black, and those in $V - S$ are shown in white. The edges crossing the cut are those connecting white vertices with black vertices. The edge (d, c) is the unique light edge 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.

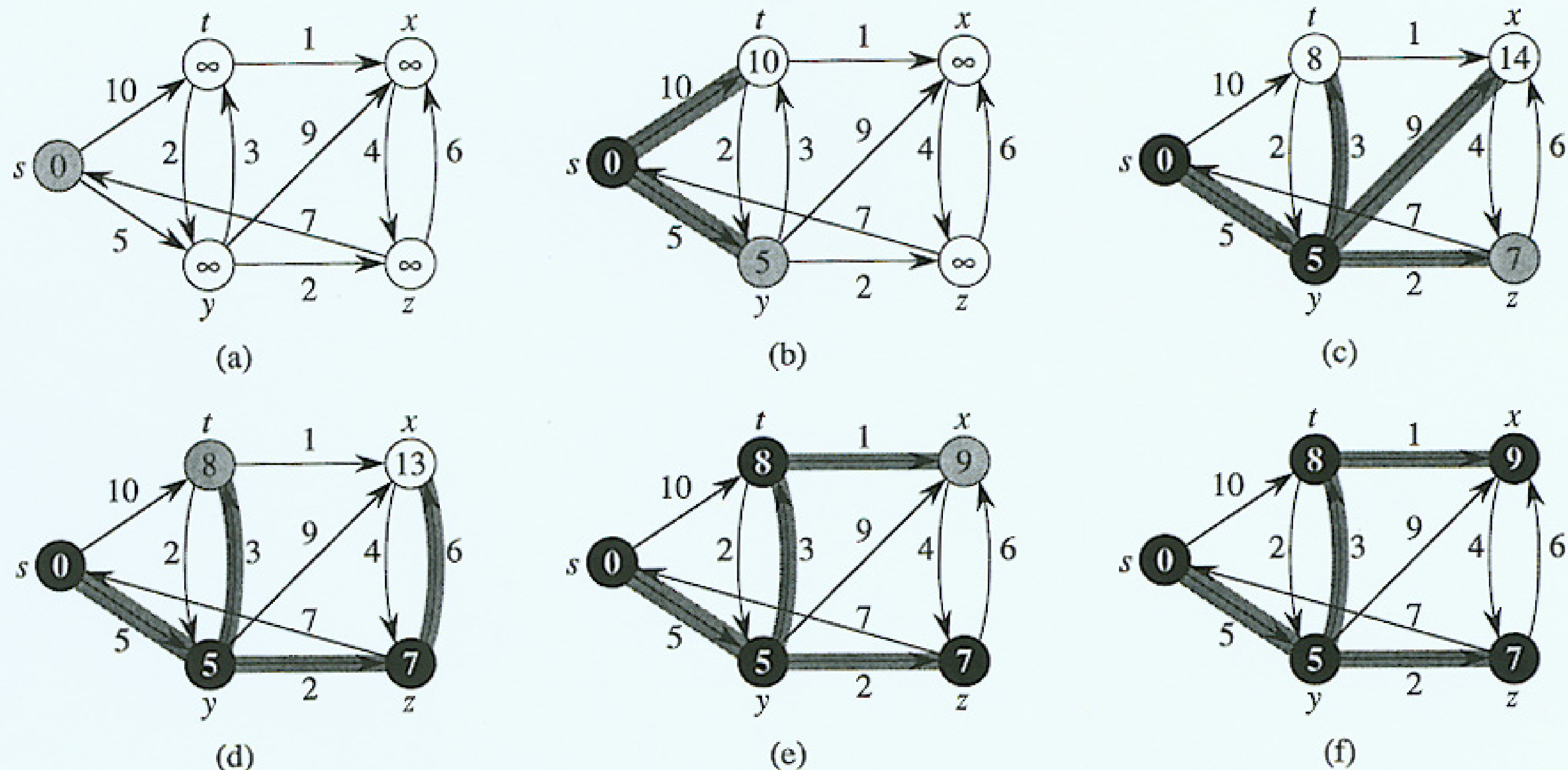


Figure 24.6 The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates are shown within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S, and white vertices are in the min-priority queue $Q = V - S$. **(a)** The situation 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)
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
```

```
MST-PRIM(G, w, r)
1.   Q ← V[G]
2.   for each u ∈ Q
3.       do key[u] ← ∞
4.   key[r] ← 0
5.   π[r] ← NIL
6.   while Q ≠ ∅
7.       do u ← EXTRACT-MIN(Q)
8.          for each v ∈ Adj[u]
9.              do if v ∈ Q and w(u, v) < key[v]
10.                   then π[v] ← u
11.                        key[v] ← w(u, v)
```

All Pairs Shortest Path Algorithm

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

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

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

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

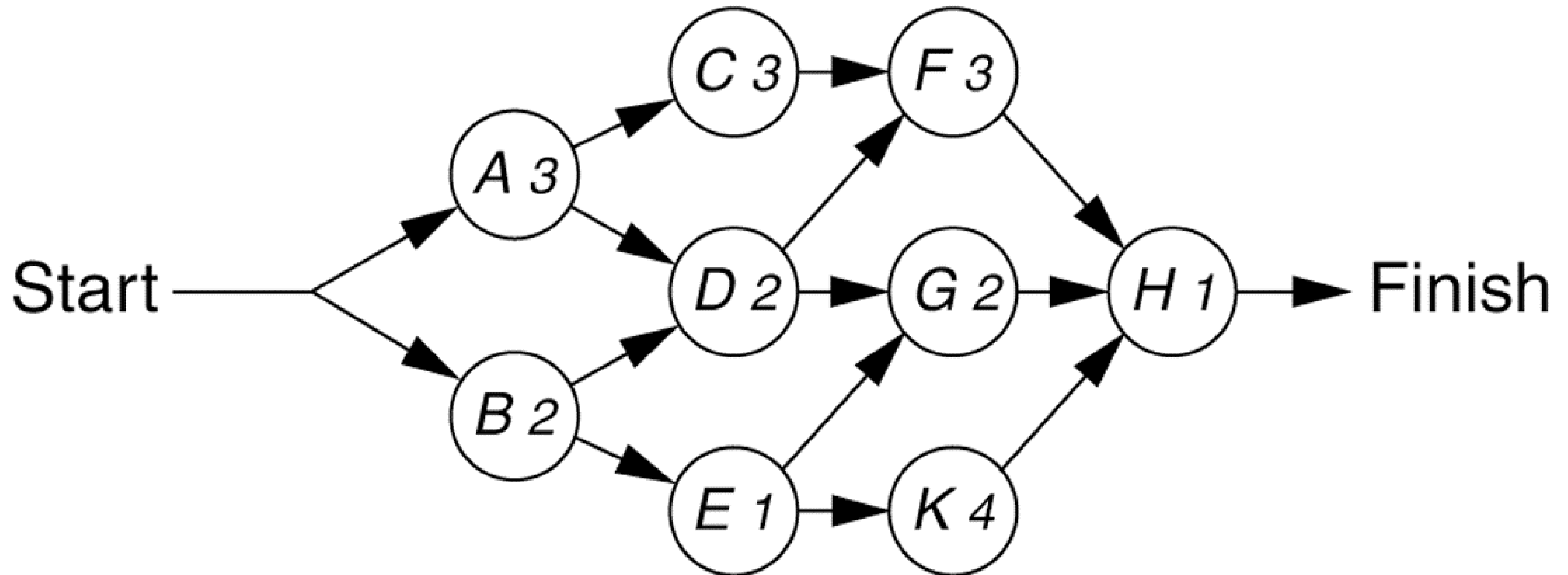
$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL}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

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

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

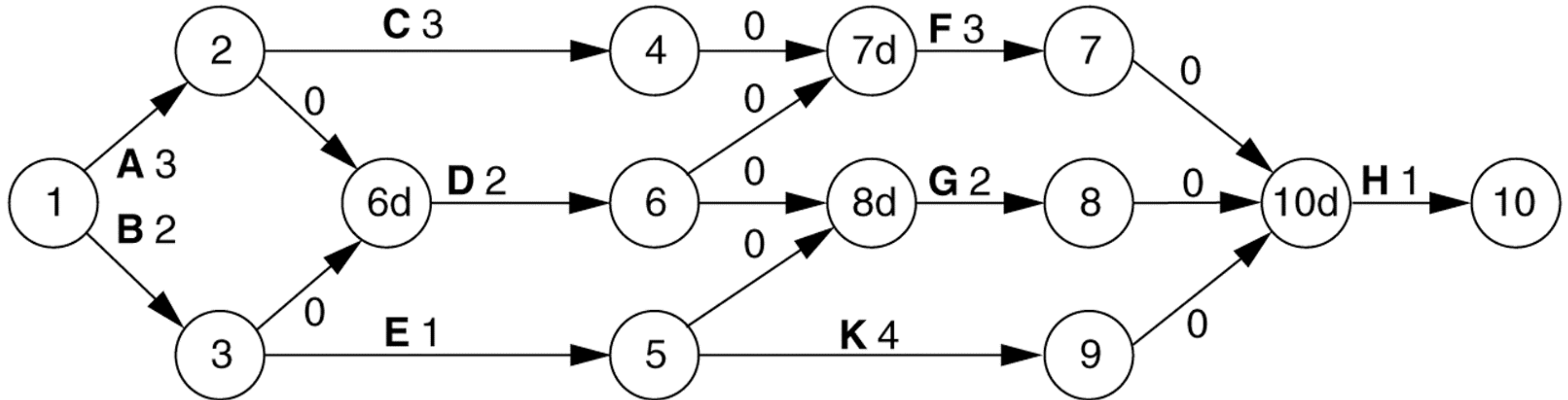
Figure 25.4 The sequence of matrices $D^{(k)}$ and $\Pi^{(k)}$ computed by the Floyd-Warshall algorithm for the graph in Figure 25.1.

Figure 14.33
An activity-node graph



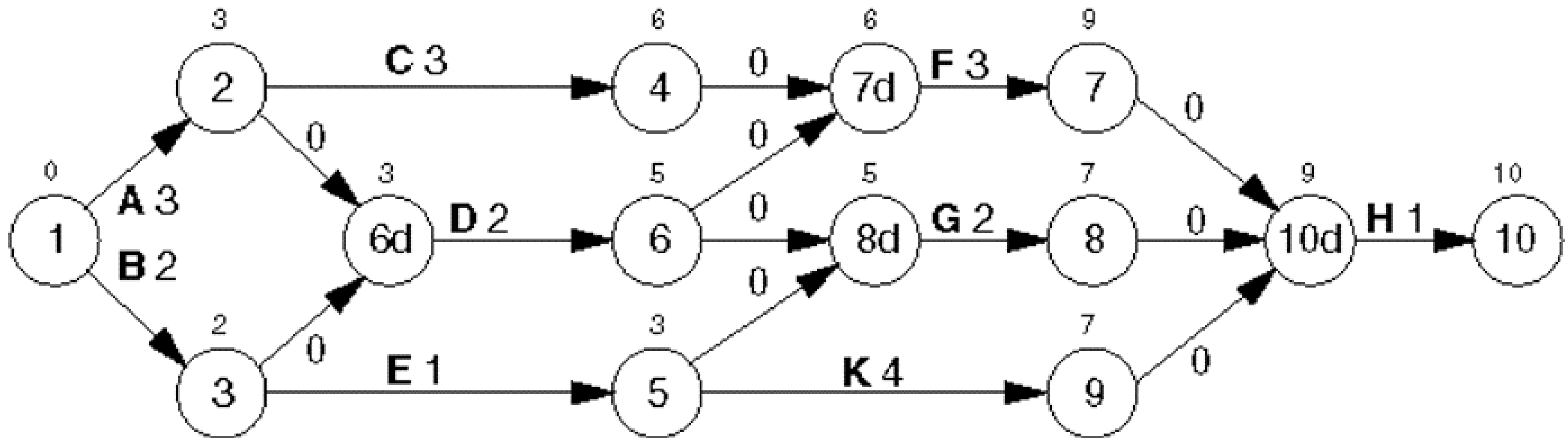
Data Structures & Problem Solving using JAVA/2E Mark Allen Weiss © 2002 Addison Wesley

Figure 14.34
An event-node graph



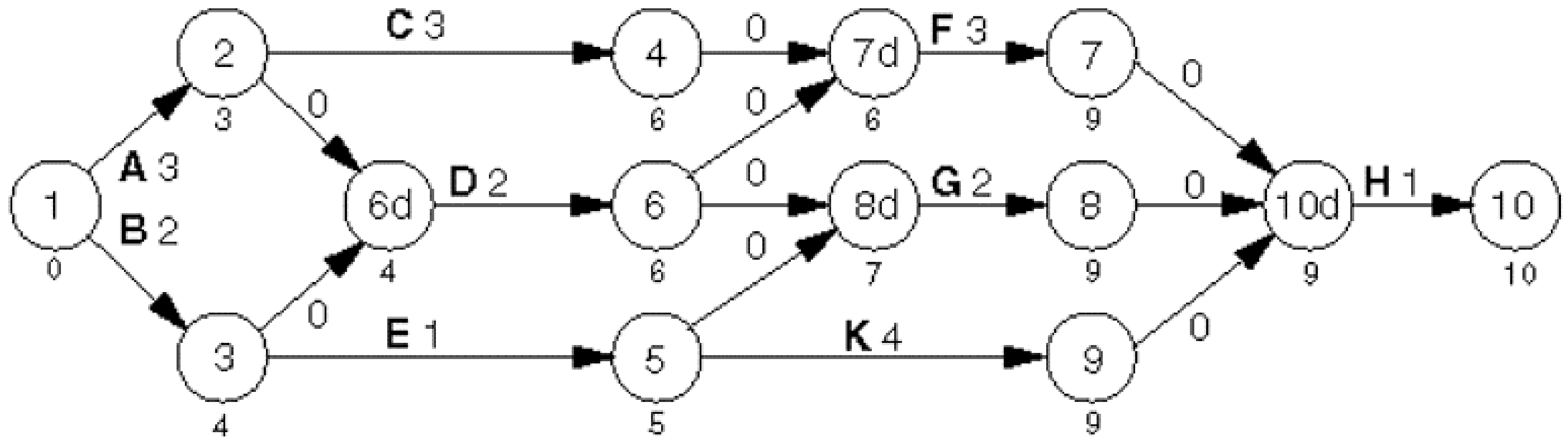
Data Structures & Problem Solving using JAVA/2E Mark Allen Weiss © 2002 Addison Wesley

Figure 14.35
Earliest completion times



Data Structures & Problem Solving using JAVA/2E Mark Allen Weiss © 2002 Addison Wesley

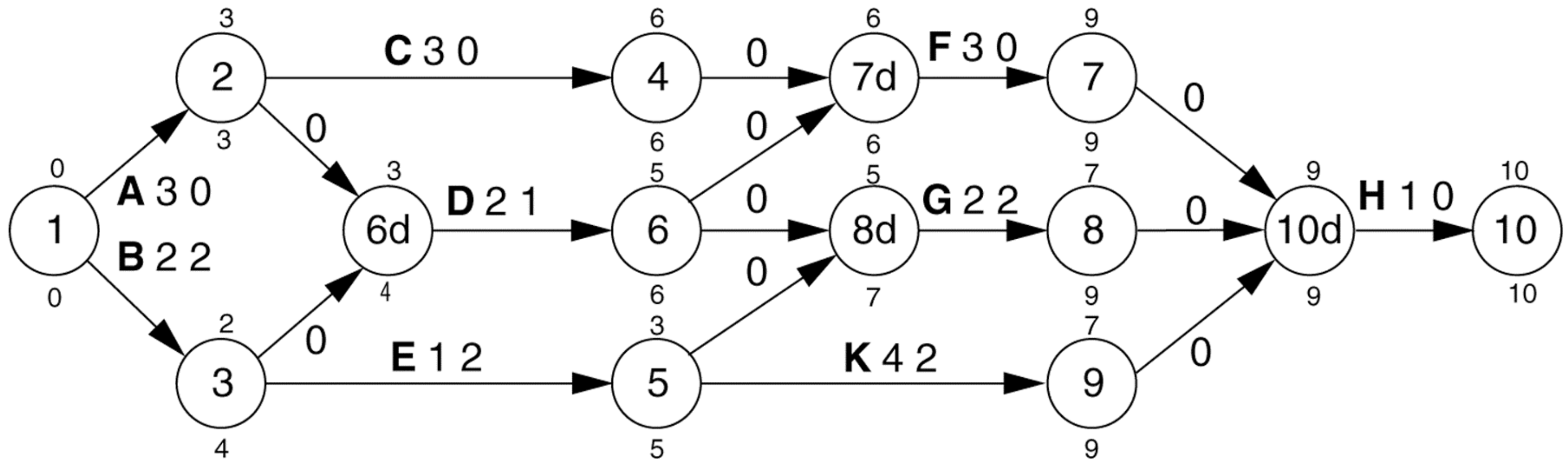
Figure 14.36
Latest completion times



Data Structures & Problem Solving using JAVA/2E Mark Allen Weiss © 2002 Addison Wesley

Figure 14.37

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



Data Structures & Problem Solving using JAVA/2E Mark Allen Weiss © 2002 Addison Wesley

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

 © 2002 Addison Wesley

Amortized Analysis

- In amortized analysis, we are looking for the time complexity of a sequence of n operations, instead of the cost of a single operation.
- Cost of a sequence of n operations = $n\ S(n)$, where $S(n)$ = worst case cost of each of the n operations
- ***Amortized Cost*** = $T(n)/n$, where $T(n)$ = worst case total cost of the n operations in the sequence.
- Amortized cost can be small even when some operations in that sequence are expensive. Often, the worst case may not occur in every operation. The cost of expensive operations may be 'paid for' by charging to other less expensive operations.

Problem 1: Stack Operations

- Data Structure: **Stack**
- Operations:
  - *Push(s,x)* : Push object *x* into stack *s*.
    - Cost: T(push)= O(1).
  - *Pop(s) :* Pop the top object in stack *s*.
    - Cost: T(pop)=O(1).
  - *MultiPop(s,k) ;* Pop the top *k* objects in stack *s*.
    - Cost: T(mp) = O(size(s)) worst case
- ***Assumption:*** Start with an empty stack
- ***Simple analysis:*** For N operations, the maximum size of stack is N. Since the cost of *MultiPop* under the worst case is O(N), which is the largest in all three operations, the total cost of N operations must be less than N x T(mp) = $O(N^2)$.

Amortized analysis: Stack Operations

- **Intuition**: Worst case cannot happen all the time!
- **Idea**: pay a dollar for every operation, and then count carefully.
- Suppose we pay 2 dollars for each *Push* operation, one to pay for the operation itself, and another for "future use" (we pin it to the object on the stack).
- When we do *Pop* or *MultiPop* operations to pop objects, instead of paying from our pocket, we pay the operations with the extra dollar pinned to the objects that are being popped.
- So the total cost of N operations must be less than 2 x N
- **Amortized cost** = T(N)/N = 2.

Problem 2: Binary Counter

- Data Structure: binary counter b.
- Operations: Inc(b).
  - Cost of Inc(b) = number of bits flipped in the operation.
- What's the total cost of N operations when this counter counts up to integer N?
- ***Approach 1: simple analysis***
  - The size of the counter is log(N). The worst case will be that every bit is flipped in an operation, so for N operations, the total cost under the worst case is O(Nlog(N))

Approach 2: Binary Counter

- **Intuition**: Worst case cannot happen all the time!

  000000

  000001

  000010

  000011

  000100

  000101

  000110

  000111

Bit 0 flips every time, bit 1 flips every other time, bit 2 flips every fourth time, etc. We can conclude that for bit k, it flips every 2^k time.

So the total bits flipped in N operations, when the counter counts from 1 to N, will be = ?

$$T(N) = \sum_{k=0}^{\log N} \frac{N}{2^k} < N \sum_{k=0}^{\infty} \frac{1}{2^k} = 2N$$

So the amortized cost will be T(N)/N = 2.

Approach 3: Binary Counter

- For k bit counters, the total cost is

  $t(k) = 2 \times t(k-1) + 1$
- So for N operations, $T(N) = t(\log(N))$.
- $t(k) = ?$
- $T(N)$ can be proved to be bounded by $2N$.

Amortized Analysis: Potential Method

- For the n operations, the data structure goes through states: D_0, D_1, D_2, …, D_n with costs c_1, c_2, …, c_n
- Define potential function $\Phi(D_i)$: represents the potential energy of data structure after i_{th} operation.
- The amortized cost of the i_{th} operation is defined by:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

- The total amortized cost is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{N} \left(c_i + \Phi(D_i) - \Phi(D_{i-1}) \right) = \Phi(D_n) - \Phi(D_0) + \sum_{i=1}^{n} c_i$$

$$\sum_{i=1}^{n} c_i = -\left(\Phi(D_n) - \Phi(D_0) \right) + \sum_{i=1}^{n} \hat{c}_i$$

Potential Method - Cont'd

- If $\Phi(D_n) \geq \Phi(D_0)$

  then $$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$$

  which then acts as an upper bound for the total cost.

  So we need to define a suitable potential function such that this function is always **non-negative**.

Potential Method: Stack

- Define $\Phi(D)$ = # of items on stack
- $\Phi(D_0) = 0$
- $\Phi(D_n) \geq 0$

$$\hat{c}_{push} = c_{push} + 1 = 2$$

$$\hat{c}_{pop} = c_{pop} - 1 = 0$$

$$\hat{c}_{multipop(k)} = c_{multipop(k)} - k = k - k = 0$$

$$\sum^{N} c < \sum^{N} \hat{c} = \sum \hat{c}_{push} + \sum \hat{c}_{multipop} + \sum \hat{c}_{pop} = \sum \hat{c}_{push} < 2N$$

Potential Method: Binary Counter

- Define $\Phi(D)$ = # of 1's in counter
- $\Phi(D_0) = 0$
- $\Phi(D_n) \geq 0$

$$\hat{c} = c + \Delta\Phi = (k+1) + (1-k) = 2$$

$$\sum^{N} c < \sum^{N} \hat{c} = 2N$$