ALGORITHMS: CLRS IV 22-26

- Definition of Graphs: CLRS 22
  - Graph Representations
  - Graph Traversal
  - DFS (Topological Sort), BFS (Fewest Hops)
  - Sub-graphs, Embedded Trees
- Minimum Spanning Tree (MST) CLRS 23
  - Prim's --- Start from in node
  - Kruskal's --- Start from smallest weight

ALGORITHMS: CLRS IV 22-26

- Shortest Distance (SD): --- One to All CLRS 24
  - Dijkstra's --- Start from in node
  - Bellman-Ford (Negative cycle finding) --- Optimality
- All to All (SD, Matrix Mult,Transitive Closure) CLRS 25
  - Multiple Bellman-Ford --- Add one arc a time
  - Repeated Squaring --- Double number of arcs
  - Floyd-Warshall --- Add one node at a time
  - Linear Programming --- Difference constraints
  - Johnson's --- Map to all positive arcs
- Capacity Graphs CLRS 26
  - Theorem Min cut = Max Flow
  - Ford-Fulkerson's — Max Flow

Graphs

in the "real world"

- networks are graphs
- object hierarchies are graphs
- circuit layouts are graphs
- computer programs are graphs

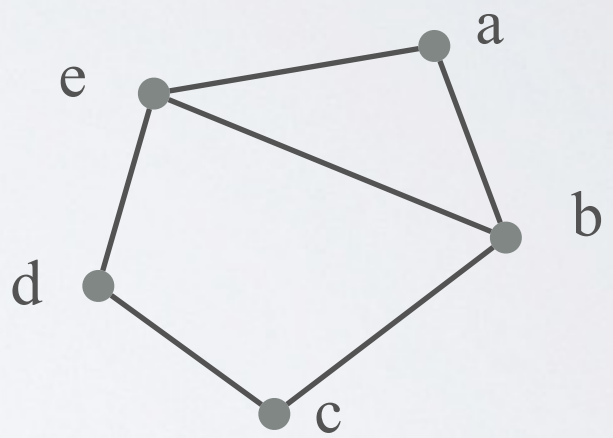
Graphs

GRAPH G(V,E) = G(N, A)

Undirected graph: G(V,E) such that

V is a set of vertices (or nodes N)

E is a set of unordered edges (or Arcs A)



degree = 3

Vertex Set: V = (a,b,c,d,e)

Edge Set: E = ({a,b}, {b,c}, {c,d},{d,e},{e,a},{e,b})

Graphs : *G(V,E)*

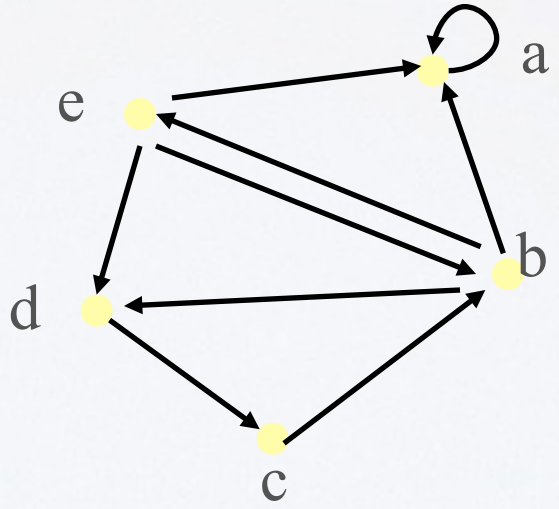
Directed graph: G(V,E) such that

V is a set of vertices *(or nodes)*

E ⊆ V × V is a set of edges *(or arcs)*

V = (a,b,c,d,e)

E = ((a,a), (b,a), (b,d), (b,e), (c,b), (d,c),(e,a),(e,b),(e,d))



in-degree = 2

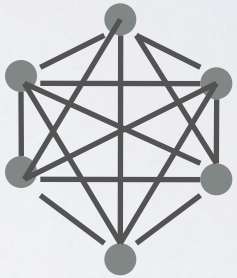
out-degree = 3

Specific graphs

Definition

complete graph on *n* vertices (K_n):

undirected graph containing an edge between each pair of distinct vertices

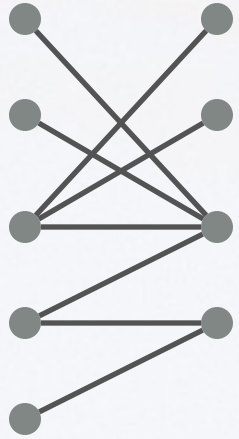


Definition

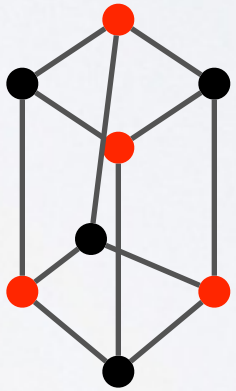
bipartite graph: (V,E)

an undirected graph whose vertex set V can be partitioned in two disjoint, nonempty sets V_1 and V_2 such that **every** edge connects a vertex in V_1 to a vertex in V_2.

Red/Black Coloring Problem!



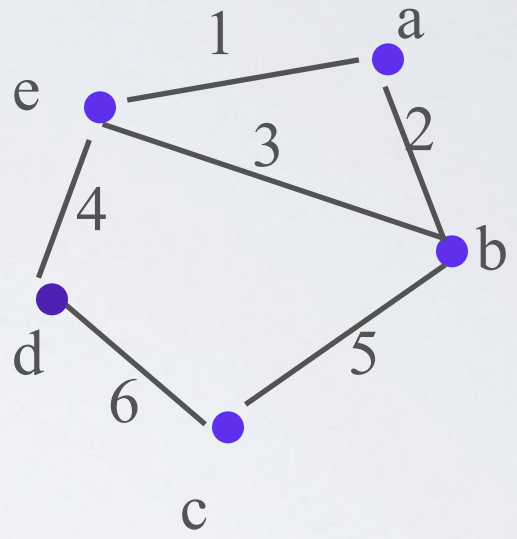
V_1 V_2



…

V_1 V_2

Graph Representations I



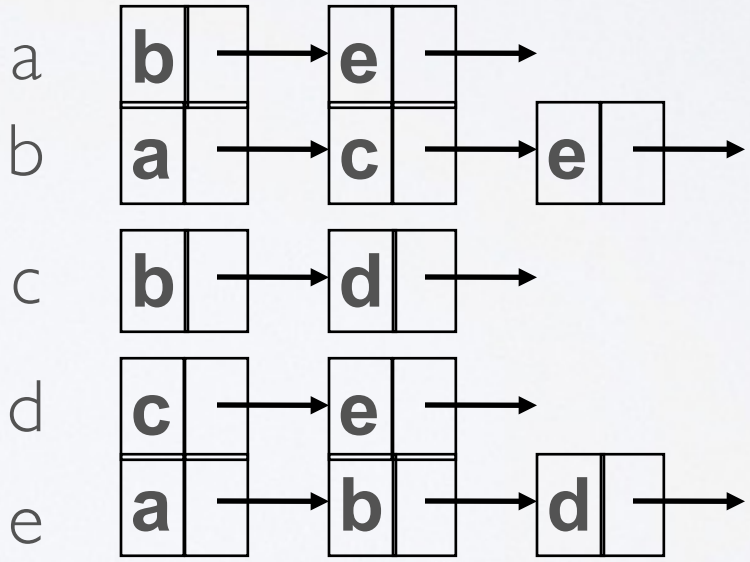
V vertices
E edges

1. Edge list: {a,b}, {b,c}, {c,d},{d,e},{e,a}, {e,b}

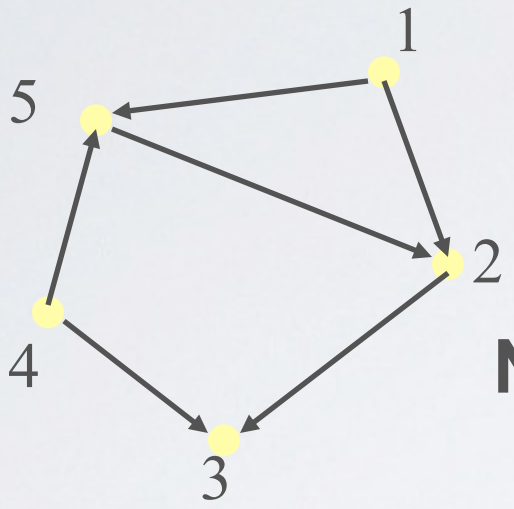
2. Adjacency list

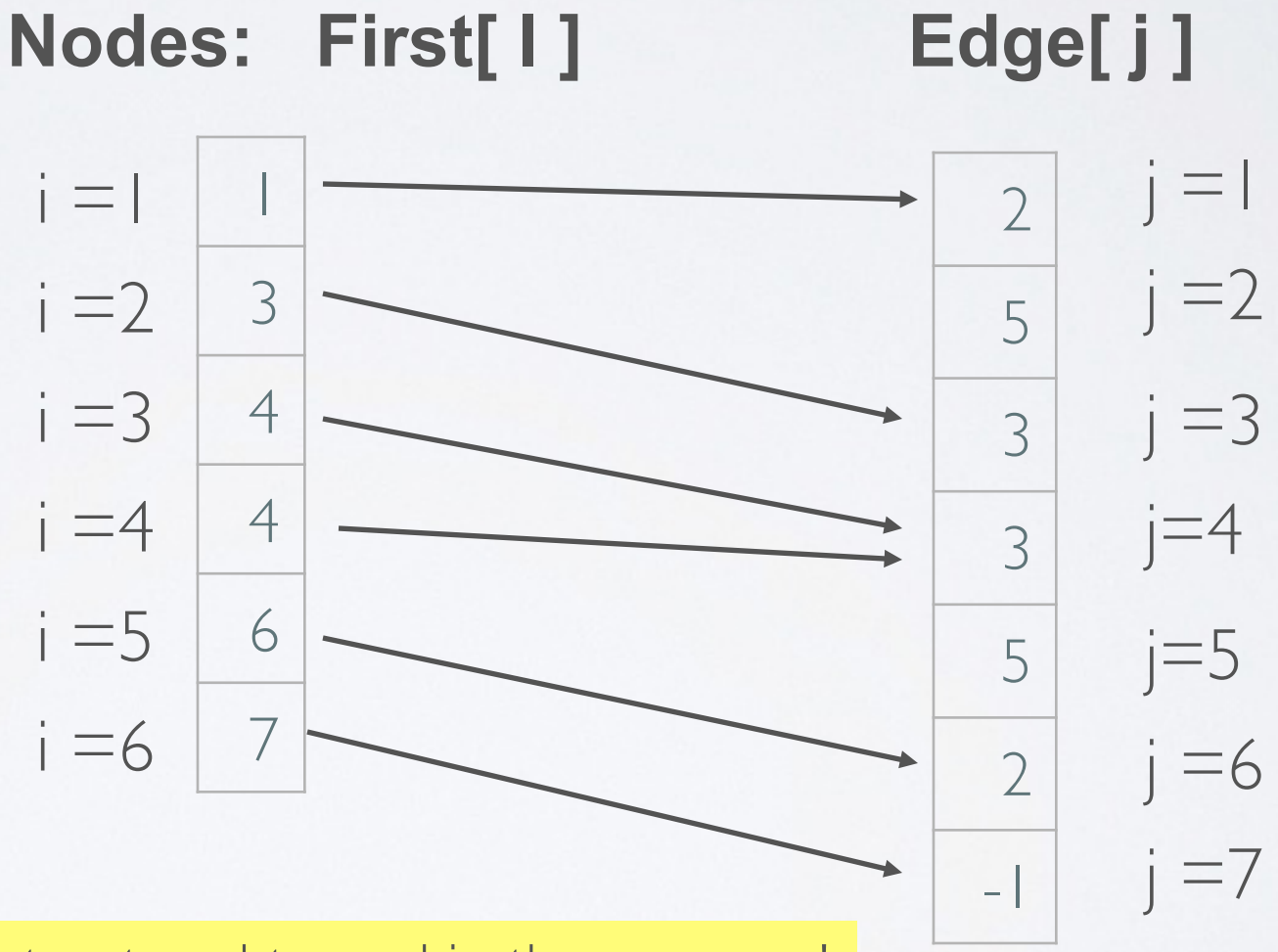
Vertex	Adjacencies
a	b, e
b	a, c, e
c	b, d
d	c, e
e	a, b, d

Linked List Form



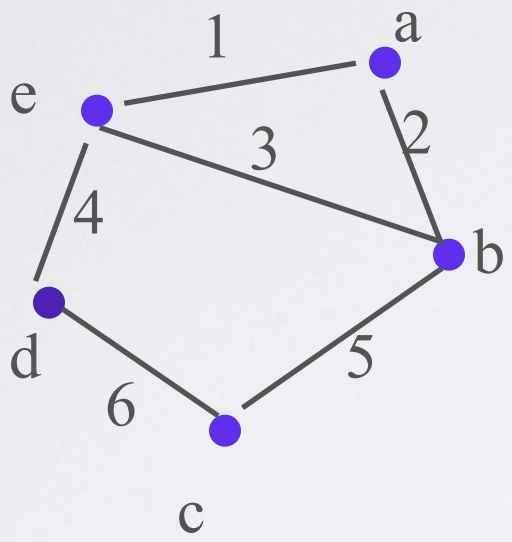
Forward Star/Neighbor array in Graph





Just the adjacency list put end to end in the arc array!

Graph Representations II



$A^2 \ldots A^\infty$

3. Adjacency matrix:

A=

	a	b	c	d	e
a	0	1	0	0	1
b	1	0	1	0	1
c	0	1	0	1	0
d	0	0	1	0	1
e	1	1	0	1	0

4. Incidence matrix:
(boundary operator)
(Transpose is co-boundary)

M =

	1	2	3	4	5	6
a	1	1	0	0	0	0
b	0	1	1	0	1	0
c	0	0	0	0	1	1
d	0	0	0	1	0	1
e	1	0	1	1	0	0

Note: $A = M M^T$ (in Boolean arithmatic)

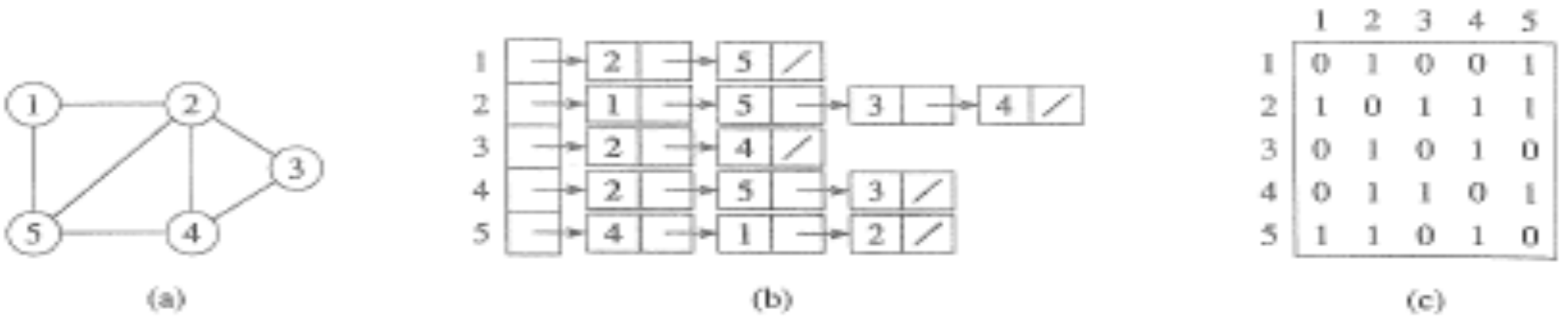


Figure 22.1 Two representations of an undirected graph. **(a)** An undirected graph G having five vertices and seven edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

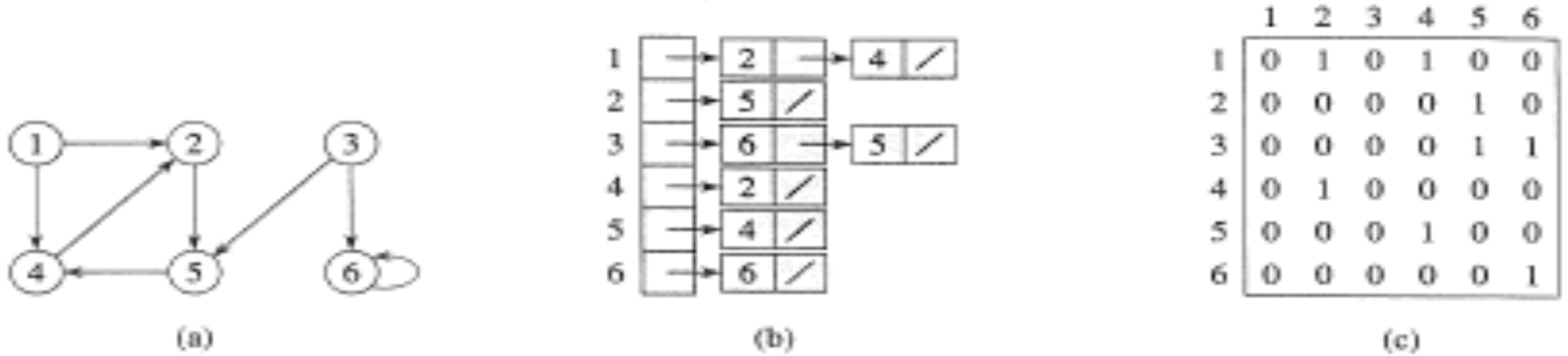


Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G having six vertices and eight edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

Some definitions

Definition

Path (in an undirected graph):
a sequence of edges joined end-to-end
(usually denoted by their vertices, in order)

e.g. ({1,3},{3,4},{4,2},{2,5}) = <1,3,4,2,5>

Def'n

Simple path: path that does not use the same vertex more than once.

e.g. <5,2,4,3,1> but **not** <5,2,4,5,3,1>

Def'n

Cycle: path that starts and ends at the same vertex:

e.g. <1,2,4,3,1> or <5,2,4,3,1,2,5>

TRAVERSALS DFS & BFS

- *Depth First Search (DFS): Uses a Stack*
  - Select a start node: mark visited and push
  - (2) Pick unmarked neighbor: mark visited and push
  - Else pop and mark done
  - Return to 2
- *Breadth First Search (BFS): Uses a Queue*
  - *Select a start node set d = 0: mark visited and enqueue*
  - *(2) Dequeue mark done*
  - *All unmarked neighbors set d = d+1: mark all visited and enqueue*
  - *Return to 2*

(For a DAG this is a "reverse order topological sort")

Set predecessor: Flag to reconstruct Tree.

(The distance label is the minimum number of hops to each node from start)

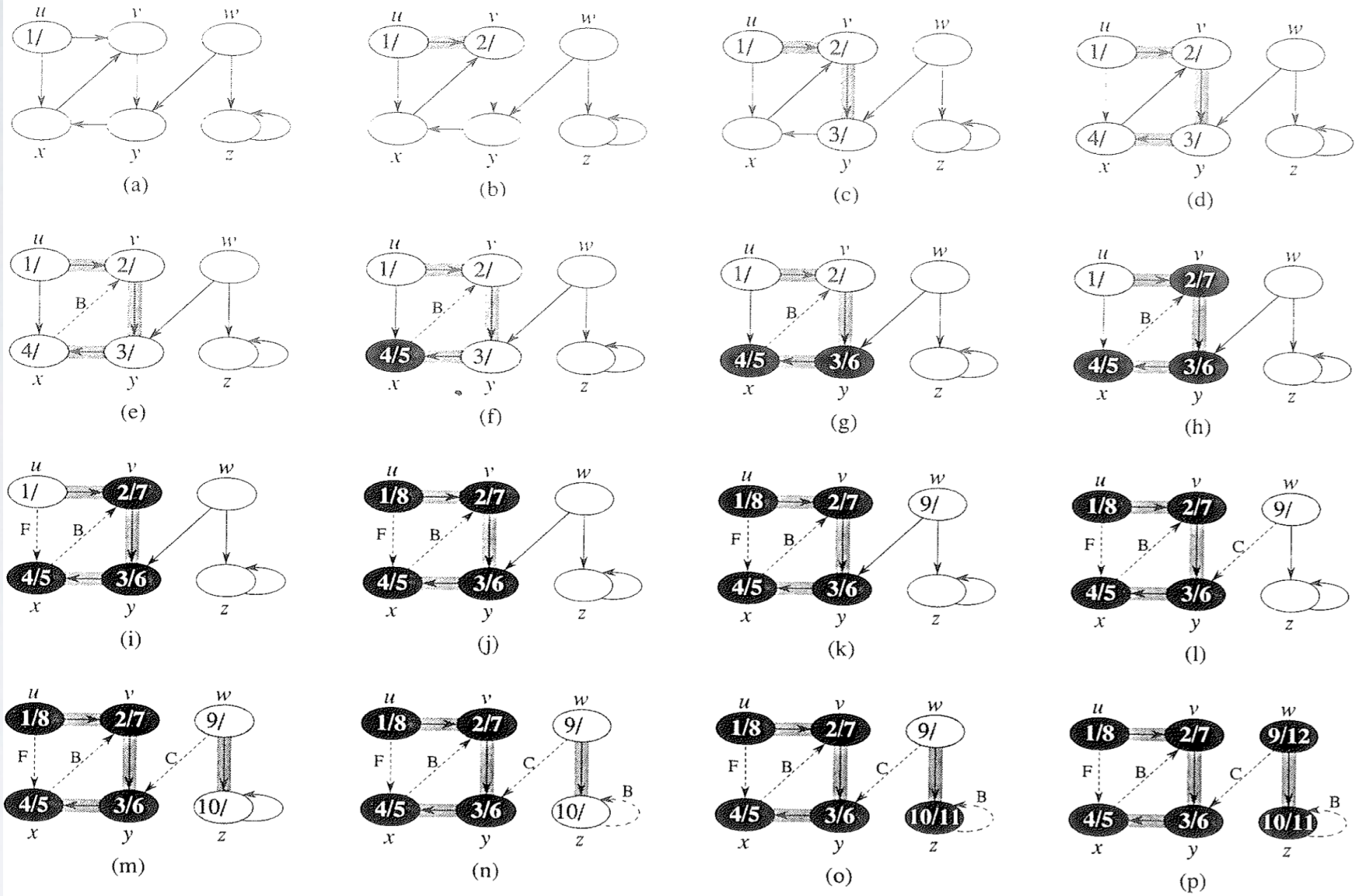


Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges.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

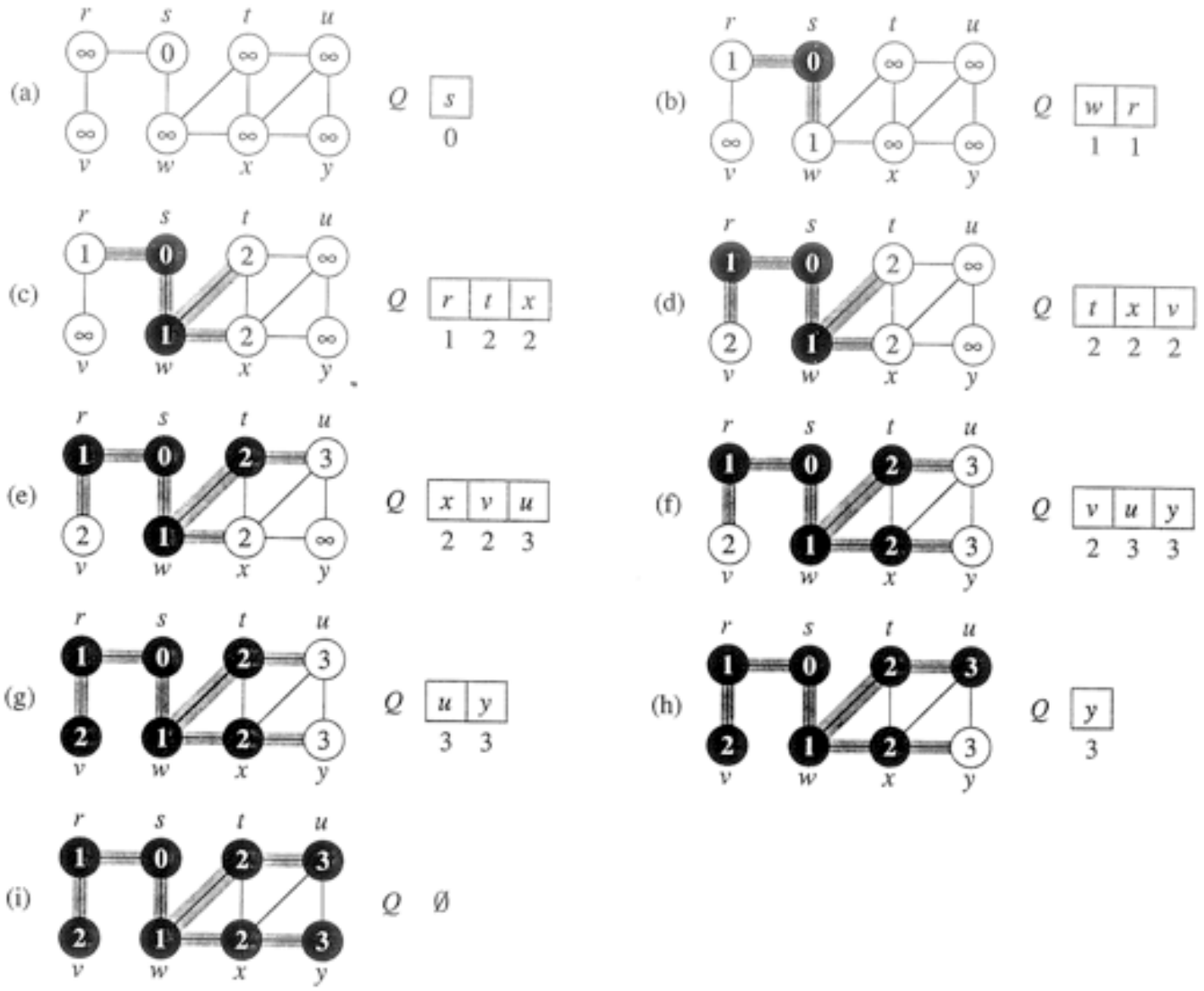
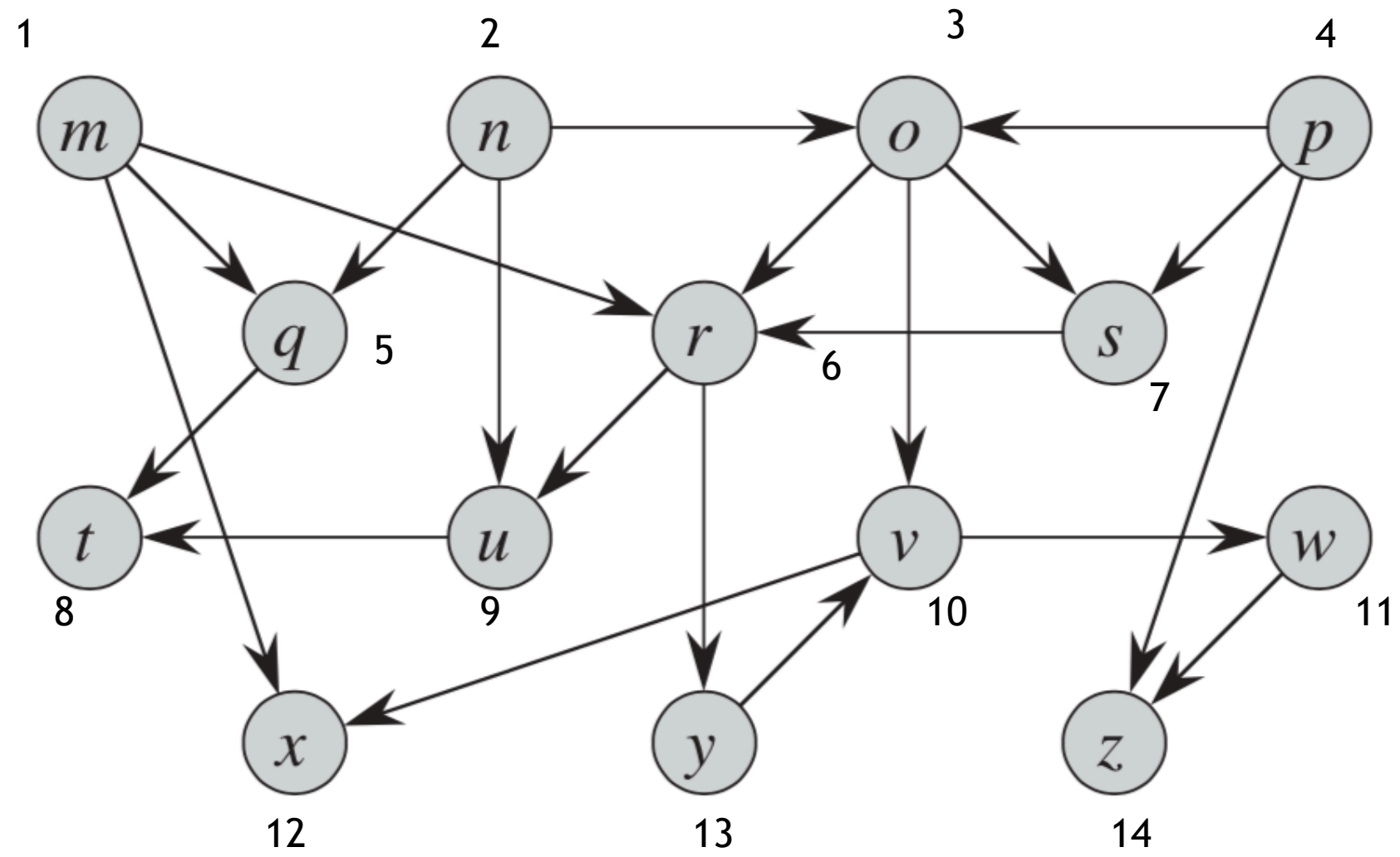


Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. Within each vertex u is shown $d[u]$. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances are shown next to vertices in the queue.

DAG Topological Sort (Execution order)



GRAPH OPTIMIZATIONS

Min Spanning Tree <==> Min Distance

Primes <==> Dijkstra's

Kruskals <==> Bellmann Ford

MINIMUM SPANNING TREE (MST)— GENERAL THEORY

- *Find* T_0 *for* $Min[\Sigma_{a \text{ in } T} w(a)]$, *for all* T in G
- *Find promising set of Arcs:* A_0 *in* T_0
- *While* A_0 *not a spanning tree find safe edge (n,n'),*

  add $A_0 = A_0 + (n,n')$

  - *Theorem: Safe edge is a minimum weight arc on cut.*

    Proof by contradiction: Consider the resultant tree using an edge E that is not the minimum. Now add back in the minimum edge for this cut. Since you now have |N| edges the new edge must give a cycle. Exchange with the min-edge to get smaller weight.

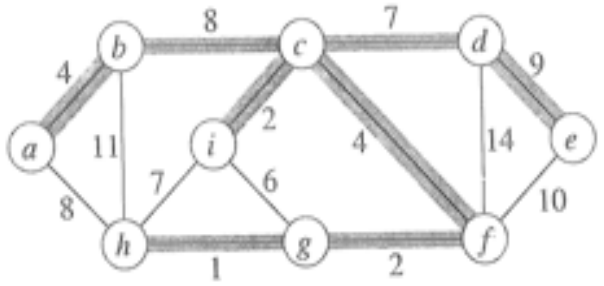


Figure 23.1 A minimum spanning tree for a connected graph. The weights on edges are shown, and the edges in a minimum spanning tree are shaded. The total weight of the tree shown is 37. This minimum spanning tree is not unique: removing the edge (b, c) and replacing it with the edge (a, h) yields another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

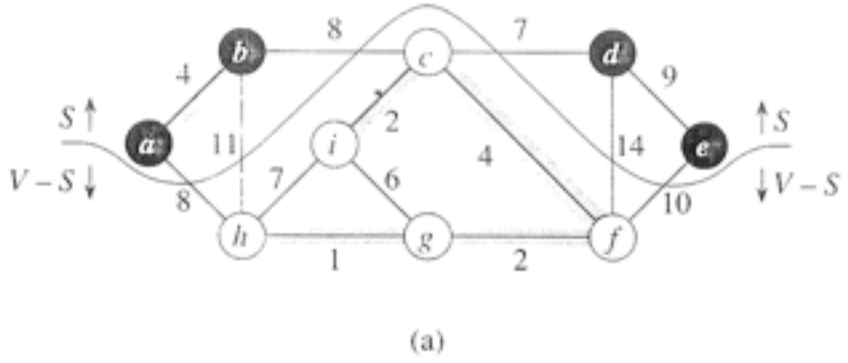


Figure 23.2 Two ways of viewing a cut $(S, V - S)$ of the graph from Figure 23.1. **(a)** The vertices in the set S are shown in black, and those in $V - S$ are shown in white. The edges crossing the cut are those connecting white vertices with black vertices. The edge (d, c) is the unique light edge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

FIRST IMPLEMENTATION

- Prim's Algorithm: grow from seed node

1. Initialize: Selected set S = empty, T = empty,

   w(i) = "infty", except w(1) = 0 ,p(i) = -1, i = 1,…n; O(N)

2. Insert w(i) into min heap, i = 1, …, n (Heap is Candidate set)

3. Take Min w(n) from heap O(N log N)

   S= S + {n} ; T = T + (p(n),n) p(v) = n;

4. For all arcs {n,v} v not in S insert into heap. O(A log N)

5. Select node n as top of heap and return to step 3 until heap is empty.

Complexity: O(N log N + A log(N)) (with out heap O(A N))

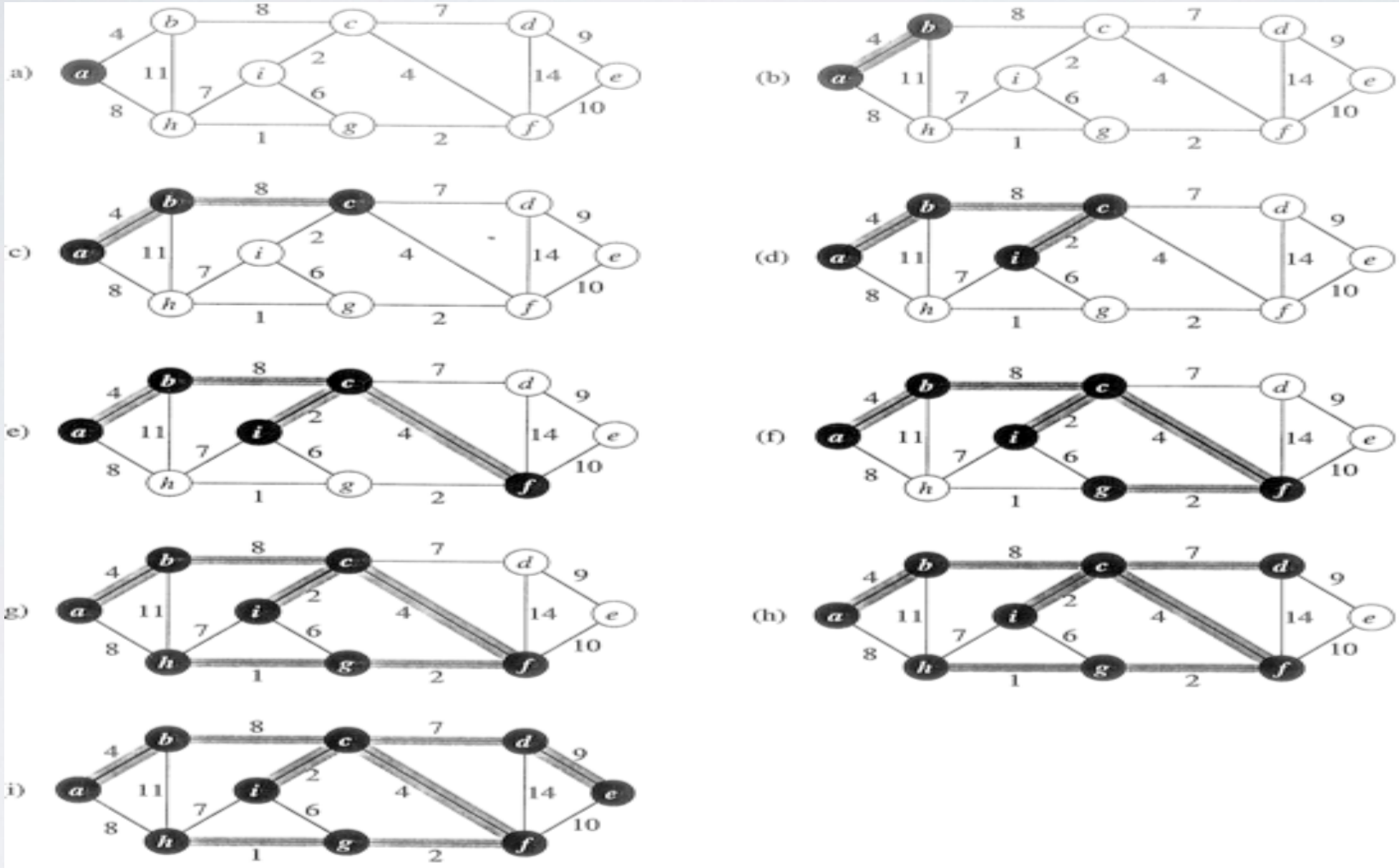


Figure 23.5 The execution of Prim's algorithm on the graph from Figure 23.1. The root vertex is a. Shaded edges are in the tree being grown, and the vertices in the tree are shown in black. At each step of the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing 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.

SECOND IMPLEMENTATION

- *Kruskal's Algorithm: Lightest edge (least weight) first and*
  add if no cycle. *(Need fast cycle identification: Cluster algorithm)*

1. Initialize T = empty; *initialize connected component labels for n nodes as 1, …, n*
2. Sort arcs in A by weight.
3. Select best arc a in list; remove from list if end nodes of arcs belong to different components t
4. Add a to tree T: T = T+{a}; Merge the connected components at the two ends of arc a.
5. Repeat 3 until there is only 1 component

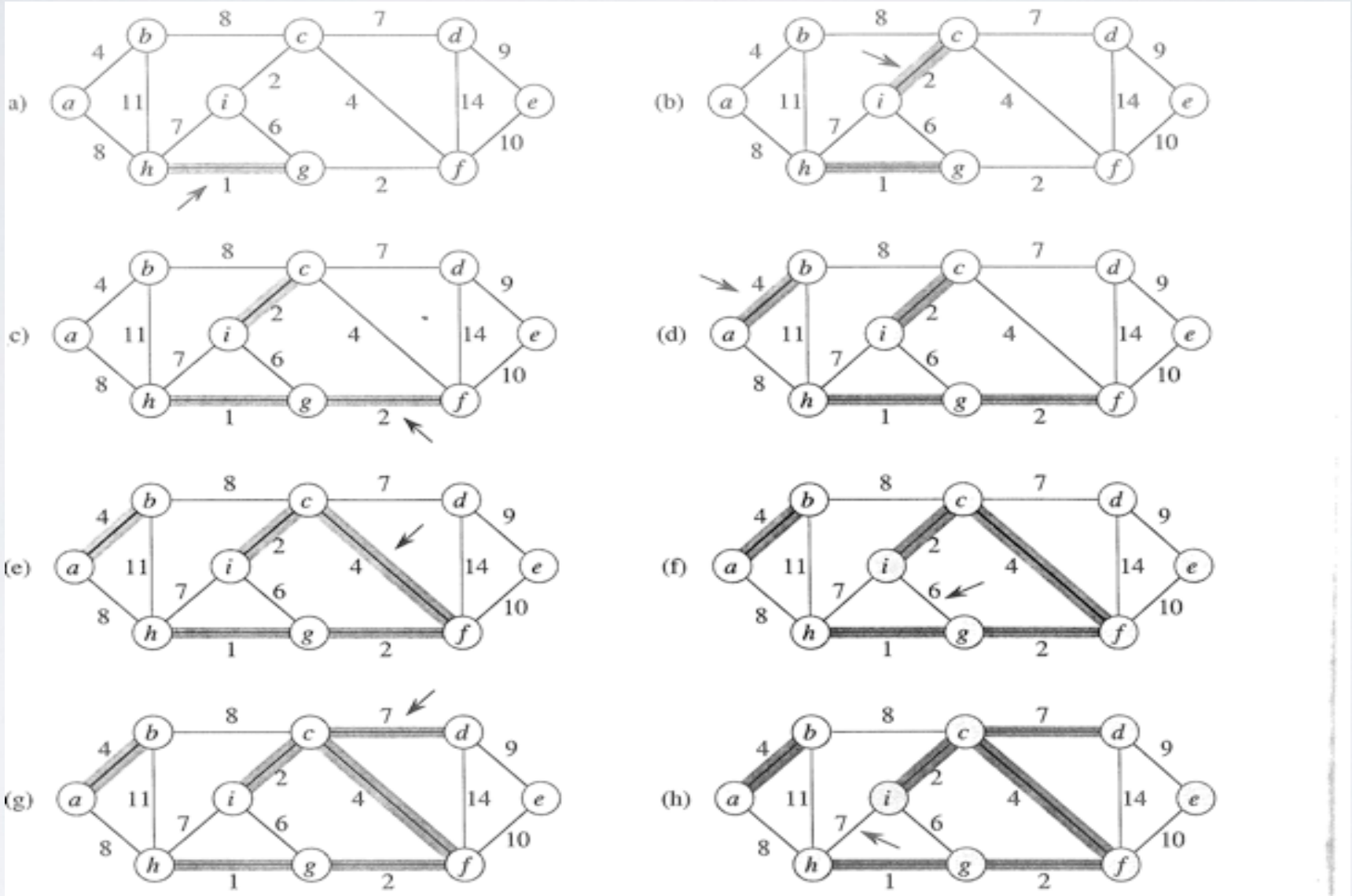
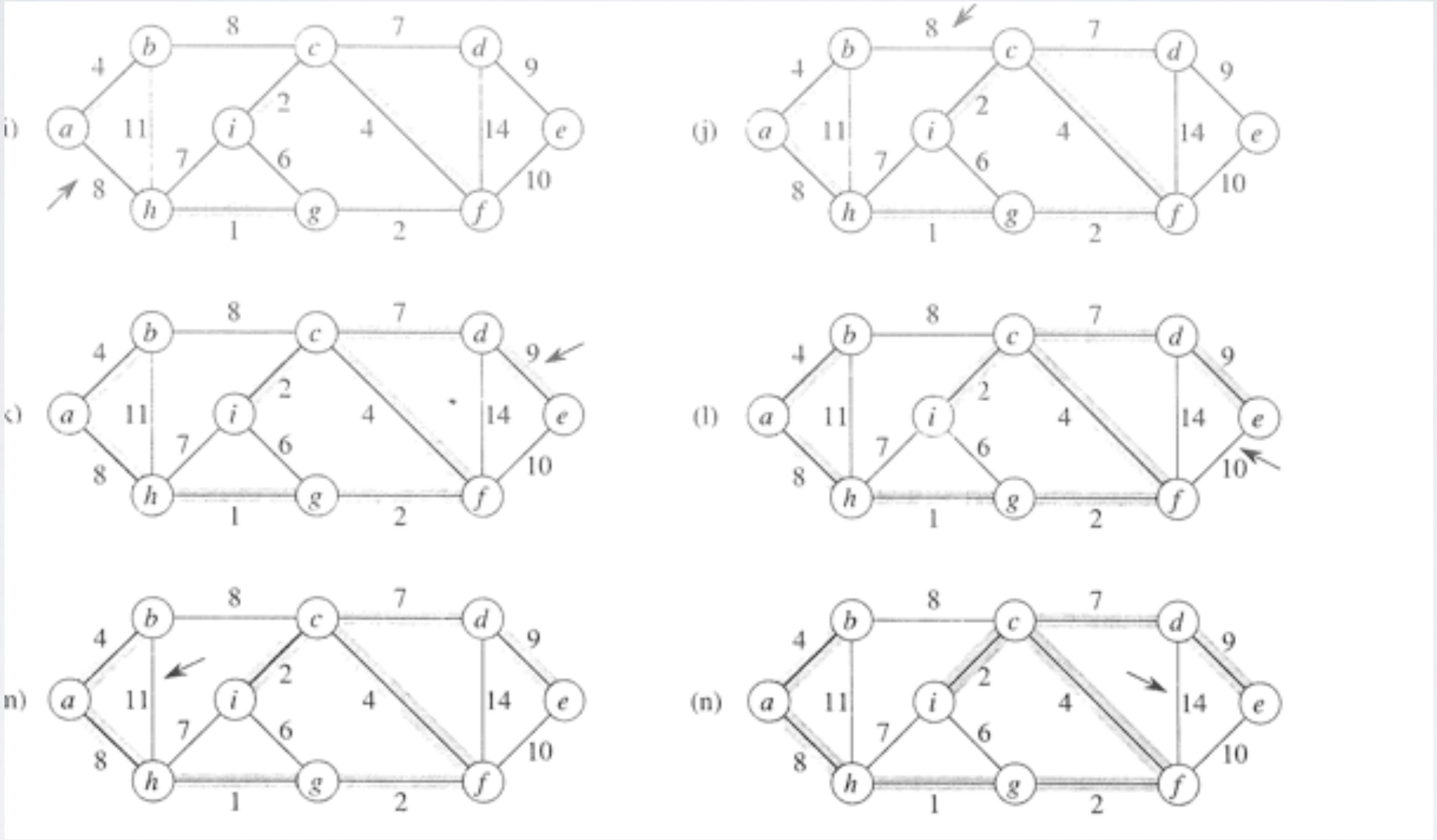


Figure 23.4 The execution of Kruskal's algorithm on the graph from Figure 23.1. Shaded edges belong to the forest A being grown. The edges are considered by the algorithm in sorted order by weight. An arrow points to the edge under consideration at each step of the algorithm. If the edge joins two distinct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

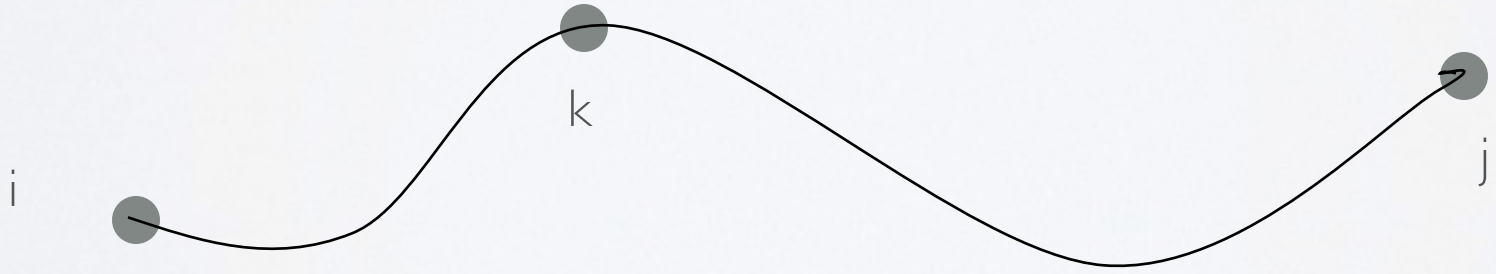

(i)
(j)
(k)
(l)
(m)
(n)
a
b
c
d
e
f
g
h
i

Shortest Path (Minimum Distance)

- d(i,j) = Min[$\Sigma_{a\ 2\ p}$ w(a)], for all paths p(i > j):
- 1 to All: This results in a tree!
- Bellman's Principle of Optimality:

Theorem: Any node on a shortest path is shortest distance from the start (or end) node.

d(i,j) = d(i,k) + d(k,j) shortest => d(i,k) & d(k,j) shortest



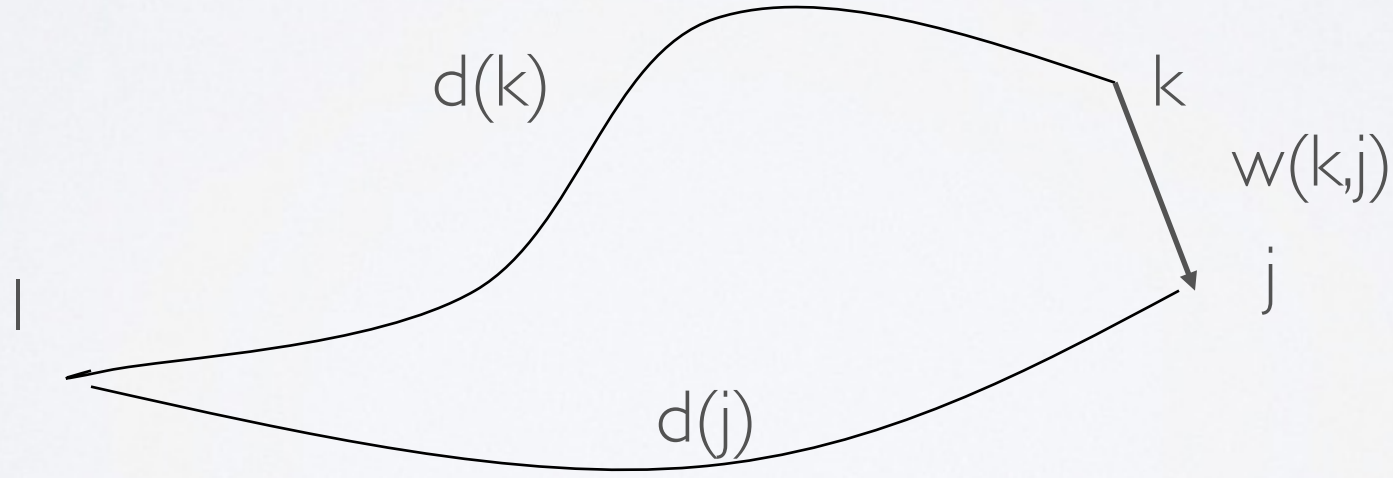
GENERIC ALGORITHM

- Relaxation Principle:

The distances (from the start node) can be improved

by the iterative replacement:

=> d(j) = MIN[d(j), d(k) + w(k,j)] any arcs (k,j)



FIRST IMPLEMENTATION :

- Dijkstra's algorithm (like Prim's) O(A+ A log(N))
  - grow from seed node: add one node at at time!

1. Start with the origin node (node 1) as node x. Initialize the distance d(i) to nodes i = 2, …, n as infinite. Initialize the predecessor array p(i) to nodes i=2, …, n as 0. Initialize the set of unmarked candidate nodes as C = {2,…,n} and

   selected set S ={1}. Initialize d(1) = 0.

2. Examine each of node v in S for outgoing arcs. O(A)

   a. Let a = (v,x) be an outgoing arc.

   b. If x is in C and d(v) + w(a) < d(x) then

      d(x) = d(v) + w(a); p(x) = v

3. If C is empty, stop; else, select x in C such that d(x) is minimal

Use heap and restore heap in log(N).

Binomial heap:
O(A + N log N)

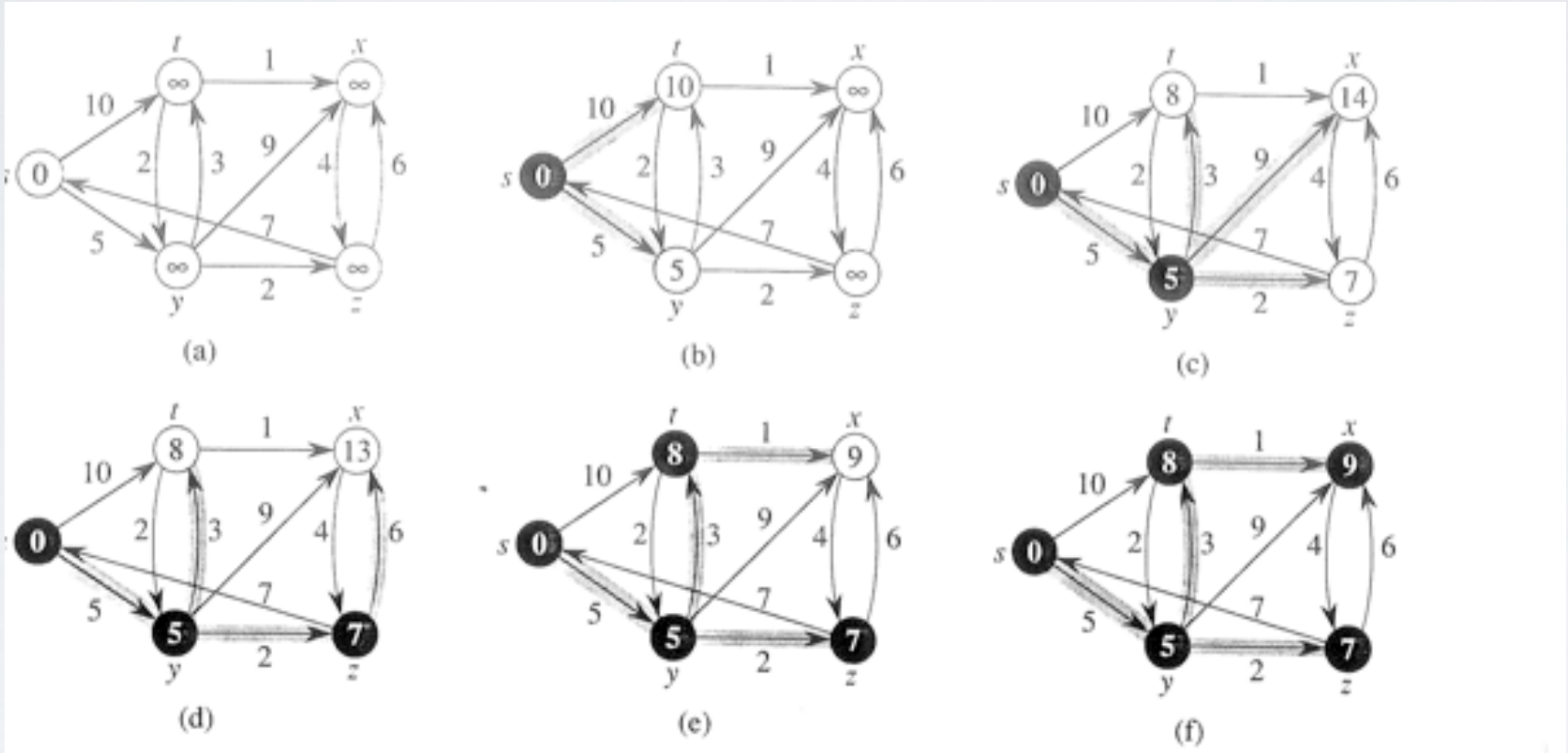


Figure 24.6 The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates are shown within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S, and white vertices are in the min-priority queue $Q = V - S$. **(a)** The situation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

Shortest Path is not necessarily unique but Dijkstra's get a solution if there are no negative edges

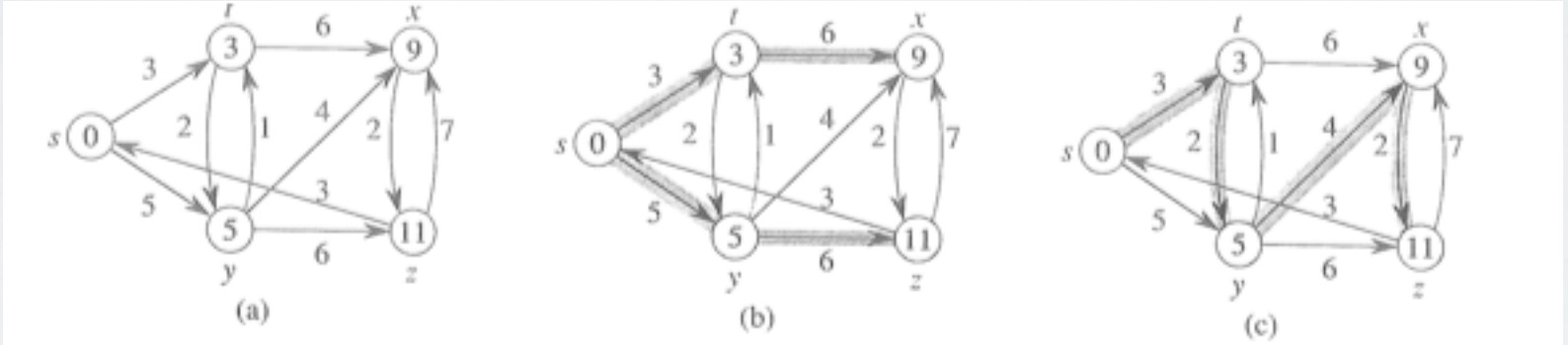


Figure 24.2 **(a)** A weighted, directed graph with shortest-path weights from source s. **(b)** The shaded edges form a shortest-paths tree rooted at the source s. **(c)** Another shortest-paths tree with the same root.

Proof by induction:
Selected tree is min. Add min to Candidate.
Suppose a min path outside it. Not possible because of Bellman's principle of optimality

SECOND IMPLEMENTATION

- Bellman-Ford's (like Kruskal's) Relaxation O(N A) with Queue
- *Add one arc at a time! like Matrix algebra: d = d + w*d*

```
Initialize: d(j)) = infty and  p(j) = 0 and set d(1) = 0
for L = 1, ... INI - 1  //Number of arcs
      { for j = 1, ... N and all arcs (k,j)
         if d(j) >  d(k) + w(k,j) {    //i.e. d(j) = MIN[d(j), d(k) + w(k,j)]
            d(j)  =  d(k) + w(k,j) ;p(j) = k }
    }                                  // see CRLS 24.1
```

Exact for all min distances requiring L links

- Unlike Dijkstras can have negative links!
- No negative cycles if it stops: no d(j) > d(k) + w(k,j) (negative cycle finding)

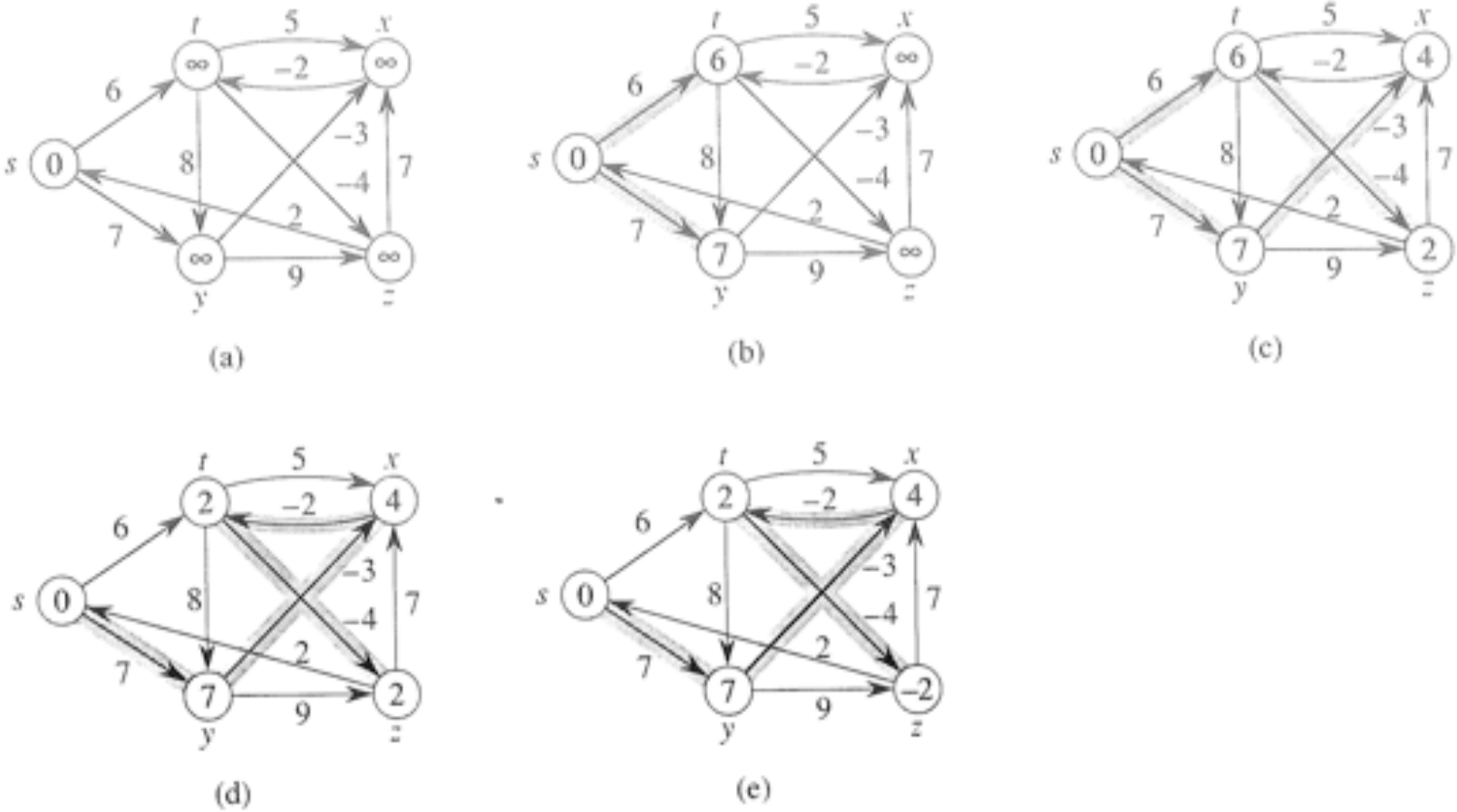


Figure 24.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values are shown within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $\pi[v] = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

NEGATIVE ARC FINDING

Bellman-Ford :

Each iteration adds one arc SO after L passes it give all min distances correctly if they only need L arc. At L =N -1 it gives them all correctly and stops unless there is negative cycle.

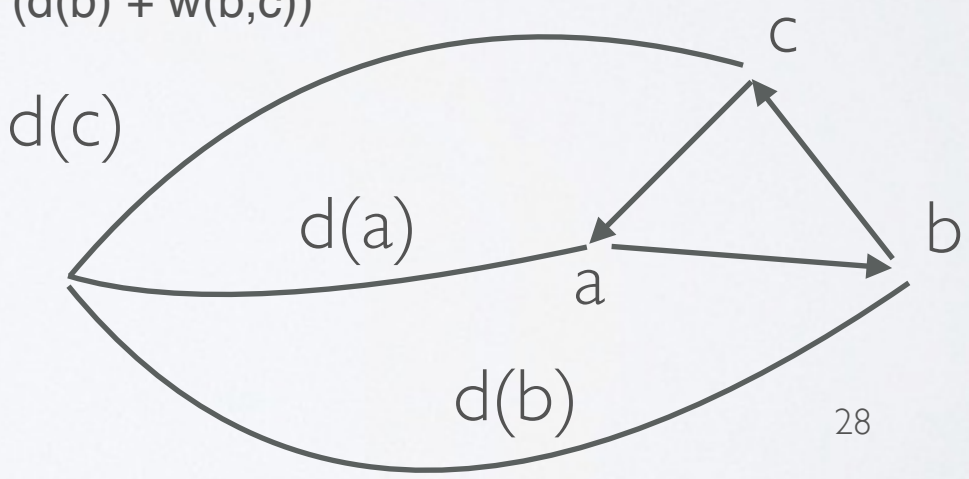
When L =N th if it find some case d(i) > d(k) + w(k,i)
a negative cycle is detected! Proof. Consider a negative 3 cycle.

w(c,a) + w(a,b) + w(b,c) < 0 ==>

By contradiction: Suppose all d(i) <= d(k) + w(k,i) ==>

d(a) + d(b) + d(c) <= (d(c) + w(c,a)) + (d(a) + w(a,b)) + (d(b) + w(b,c))
false!

Q.E.D.



ALL TO ALL SHORTEST PATH

- Matrix of weights W = w(i,j) find Matrix distances D =d(i,j)
  - *Relaxation Method:*

    Initialize: d(i,j)) = infty , p(i,j) = - 1 set d(i,i) = 0

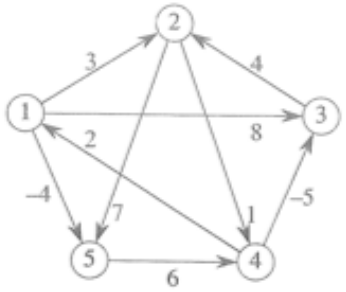
    iterate: d(i,j) = Min[d(i,j), d(i,k) + w(k,j)]

  - with operators overloaded: MIN => + and + => *

    get algebra of matrix multiplication: D = I + D * W

    Start with D = I get " series expansion for 1/(1 – W)

- Do N times 1 to All algorithm
  - Run Dijkstra from every origin: complexity O(|N||A|+|N||A|log|N|)
- Matrix Product $O(N^4)$
- Repeated Squaring $O(N^3 Log(N))$ Exponentiation
- Floyd-Warshall Algorithm $O(N^3)$ "Add a node at a time. Like Matrix Multiply"
- Johnson's Algorithm



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

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

Figure 25.1 A directed graph and the sequence of matrices $L^{(m)}$ computed by SLOW-ALL-PAIRS-SHORTEST-PATHS. The reader may verify that $L^{(5)} = L^{(4)} \cdot W$ is equal to $L^{(4)}$, and thus $L^{(m)} = L^{(4)}$ for all $m \geq 4$.

$$\begin{aligned} L^{(1)} &= W\,, \\ L^{(2)} &= W^2 &&= W \cdot W\,, \\ L^{(4)} &= W^4 &&= W^2 \cdot W^2 \\ L^{(8)} &= W^8 &&= W^4 \cdot W^4\,, \\ &\vdots \\ L^{(2^{\lceil \lg(n-1) \rceil})} &= W^{2^{\lceil \lg(n-1) \rceil}} &&= W^{2^{\lceil \lg(n-1) \rceil - 1}} \cdot W^{2^{\lceil \lg(n-1) \rceil - 1}}\,. \end{aligned}$$

Since $2^{\lceil \lg(n-1) \rceil} \geq n - 1$, the final product $L^{(2^{\lceil \lg(n-1) \rceil})}$ is equal to $L^{(n-1)}$.

The following procedure computes the above sequence of matrices by using this technique of ***repeated squaring***.

Floyd Warshall's algorithm:

1. Initialize: d(i,j) = w(i,j) and p(i,j) = i

   else w(i,j) = infinite, p(i,j) = -1, if w(i,j) is not in A.

2. For k = 1 to N // Outer loop on internal nodes! (Note D = D*D)

   For i = 1 to N

   For j = 1 to N

   if(d(i,j) > min(d(i,j),d(i,k) + d(k,j)){

   d(i,j) = d; p(i,j)= p(k,j)

   }

Properties:

1. Complexity: $O(N^3)$
2. d(i,j) is non-increasing.
3. After iteration k, d(i,j) is shortest path distance for paths which use only nodes 1, …, k as intermediate nodes.
4. Easy implementation with a matrix…very popular.

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

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

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

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

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

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

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.

REFERENCE 0TH NODE METHOD

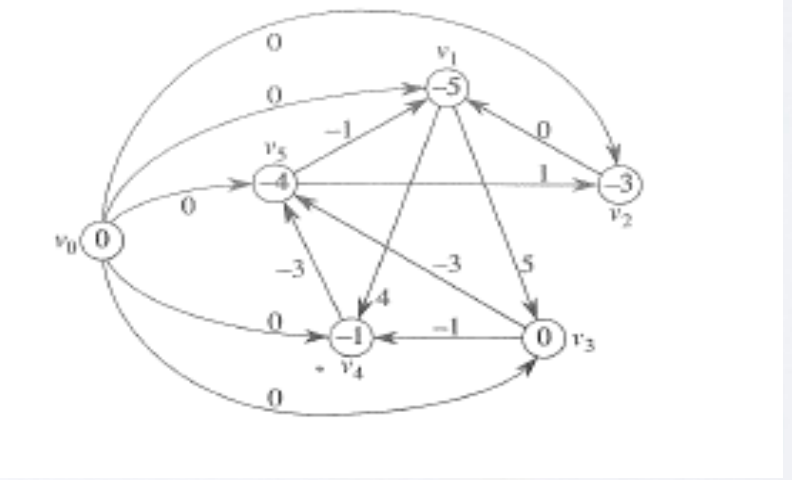
- Linear Programming: $A\,x \cdot b$ maximize $\Sigma_i\, c_i\, x_i$

  Feasible solution to difference constraints

  $x_i - x_j \cdot w(i,j)$ ➔ $d(i) \cdot d(j) + w(j,i)$

- If Bellman-Ford terminates from node 0 a solution exits!

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} \leq \begin{bmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{bmatrix}$$



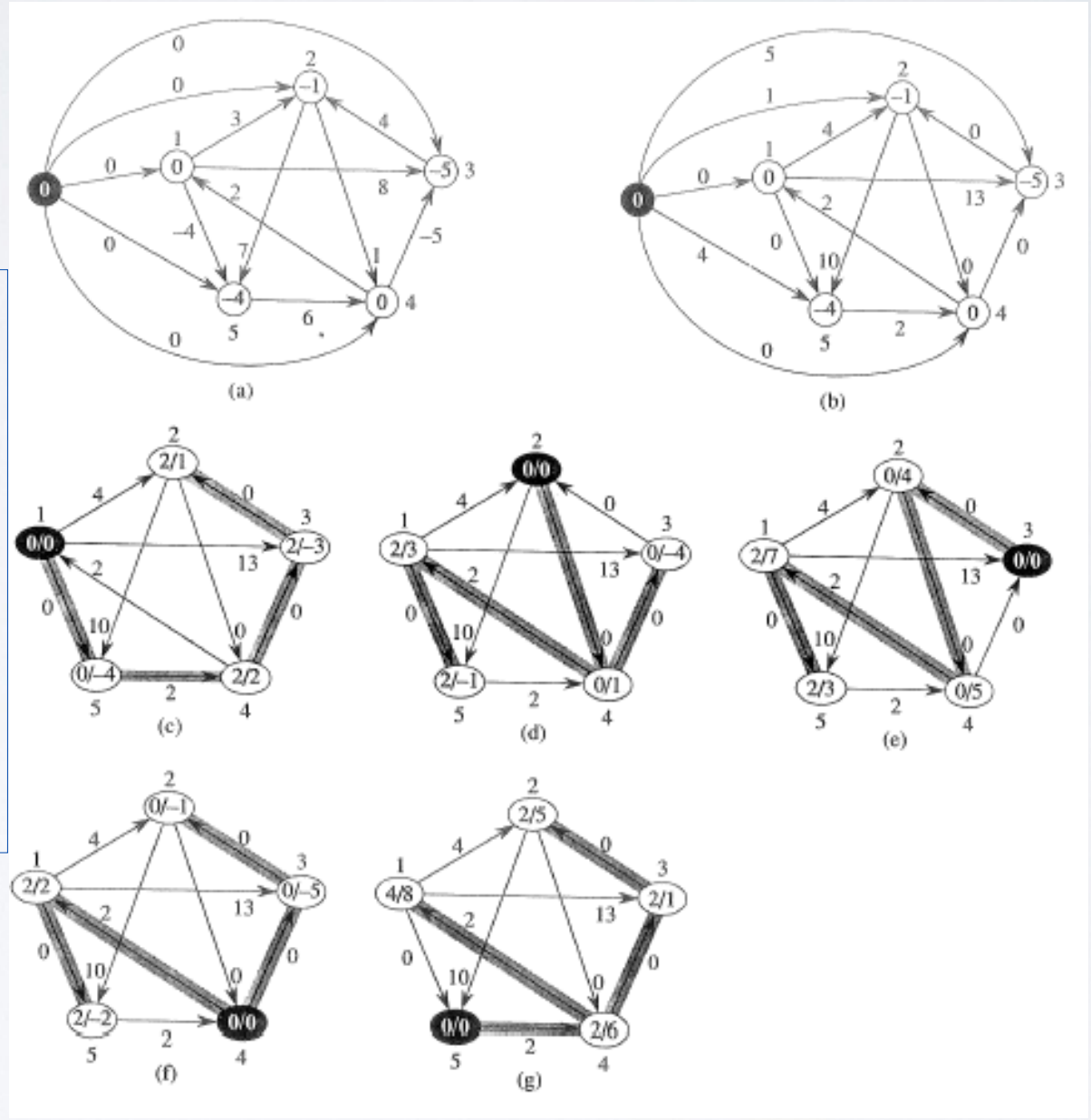
Johnson's Sparse Graph Algorithm: O(N^2 log N + N A)

1. Preprocess with Bellman-Ford from 0 node and re-weight

$W(i,j) = w(i,j) + d(0,i) - d(0,j)$

#2. Solve N-times with Dijkstras to get D(i,j) and convert

$D(i,j) = d(i,j) + d(0,j) - d(0,i)$

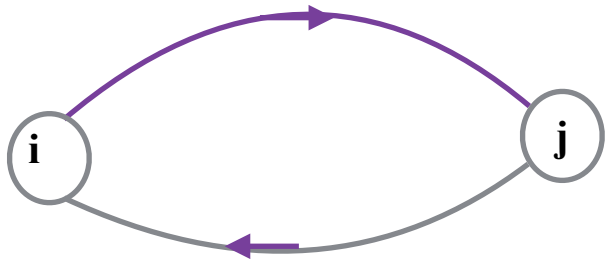


Capacity Graphs

- Capacity graph: link i > j has capacity c(i,j)

- Max Flow Problem: Max flow into source node (s) out of terminal node (t)
  - constraints on flow i to j: f(i,j) = - f(j,i) $\sum_j f(i,j) = 0$

    capacity: 0 < f(i,j) < c(i,j) ;



If there is return arc -c(j,i) · f(i,j) · c(i,j)

Augmentation, Flow, Residual

- *Initialize residual graph G(N,A1), A1 = A.*

- *Initialize all f(i,j) = 0.* *0 => r(i,j) => c(i,j)*

  *a. Find path P from s to t (**augmenting path)***

  b. Find minimum capacity arc on path P. Denote capacity by C.

  c. Add C units of flow f(i,j) ! f(i,j) + C to each arc on path Modify residual graph (N,A1) accordingly.

  -f(i,j) => r(i,j) · c(i,j) => f(i,j)

  d. Repeat a-c until no path P from s to t in G(N,A1)

- *Properties:*

  a) Total flow increases and terminate when no augmentation possible: O(Max(c) |A|)

  b) With minimum hop path for augmentation O(|N| $|A|^2$)

Solution without backtracking

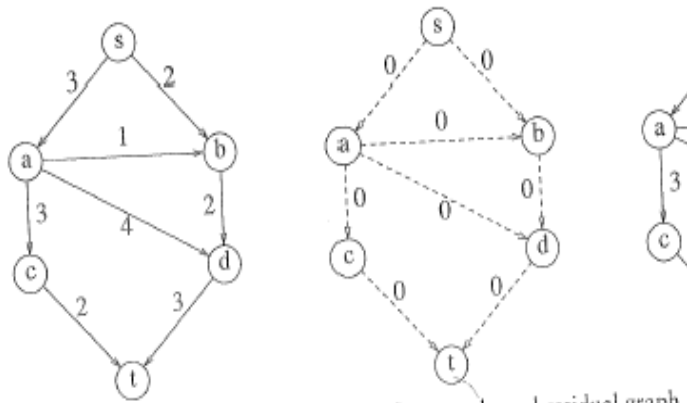


Figure 9.40 Initial stages of the graph, flow graph, and residual graph

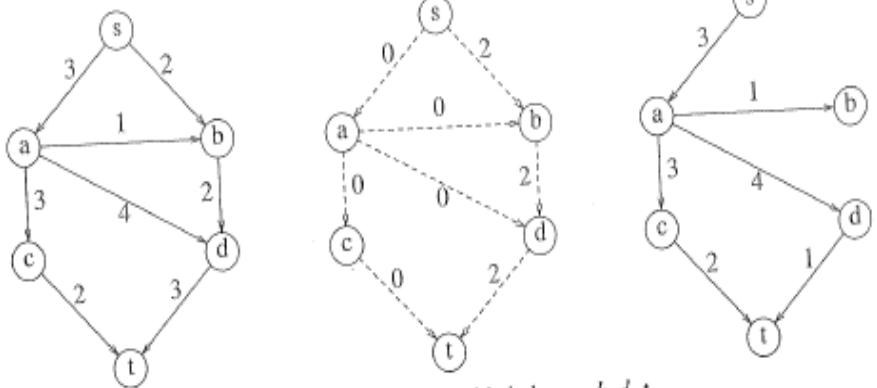


Figure 9.41 G, G_f, G_r after two units of flow added along s, b, d, t

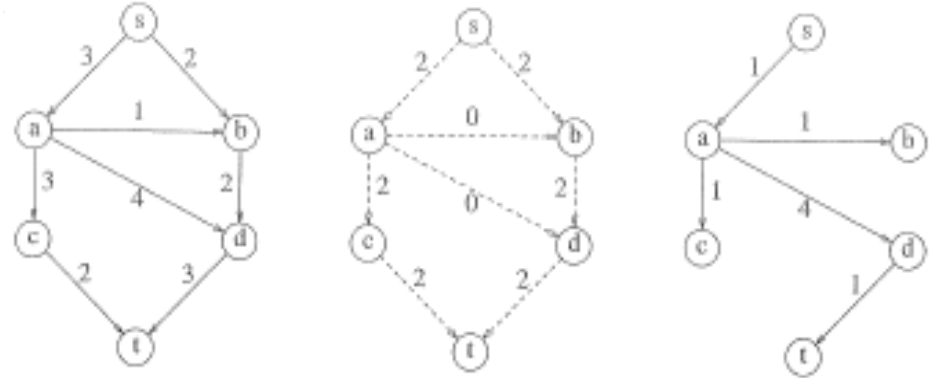


Figure 9.42 G, G_f, G_r after two units of flow added along s, a, c, t

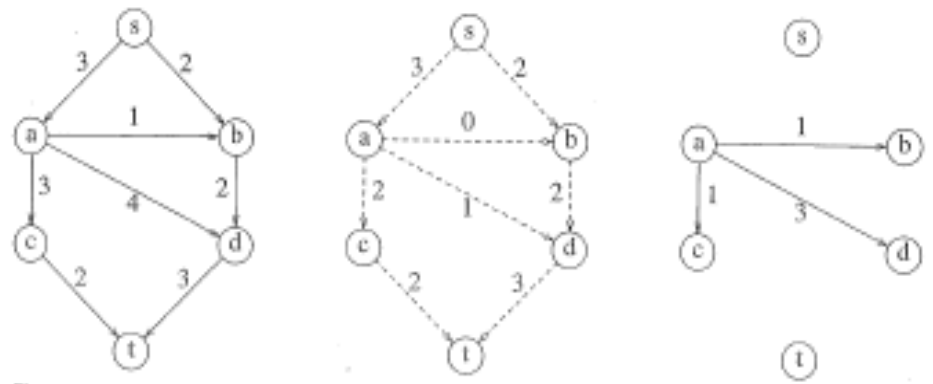


Figure 9.43 G, G_f, G_r after one unit of flow added along s, a, d, t—algorithm terminates

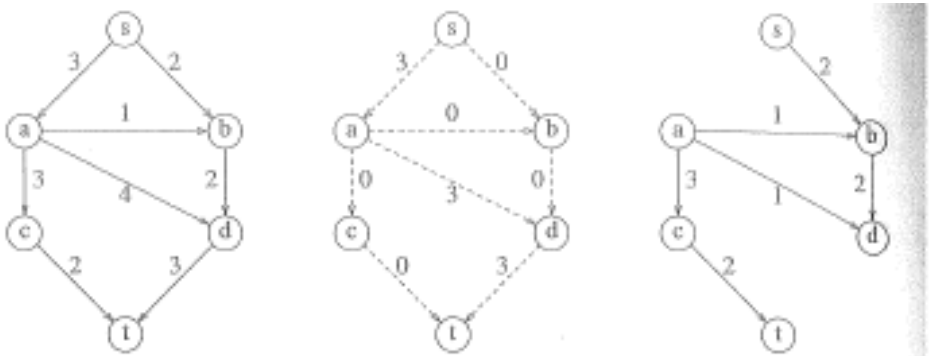


Figure 9.44 G, G_f, G_r if initial action is to add three units of flow along s, a, d, t—algorithm terminates with suboptimal solution

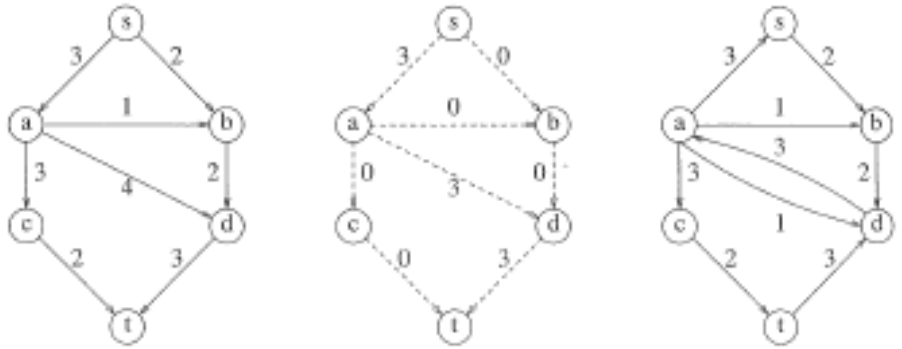


Figure 9.45 Graphs after three units of flow added along s, a, d, t using correct algorithm

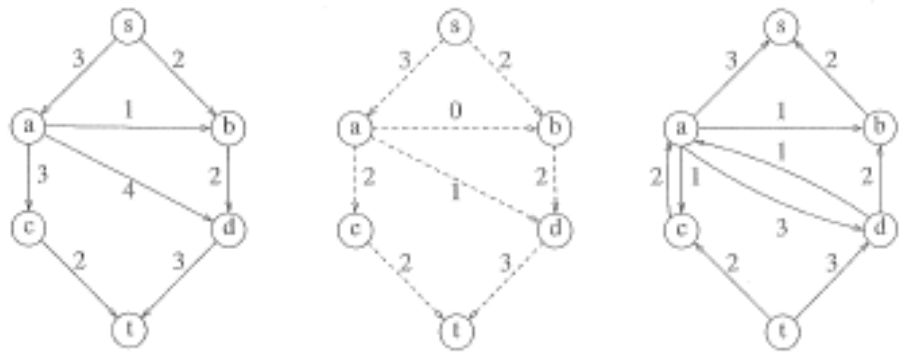


Figure 9.46 Graphs after two units of flow added along s, b, d, a, c, t using correct algorithm

Solution with backtracking

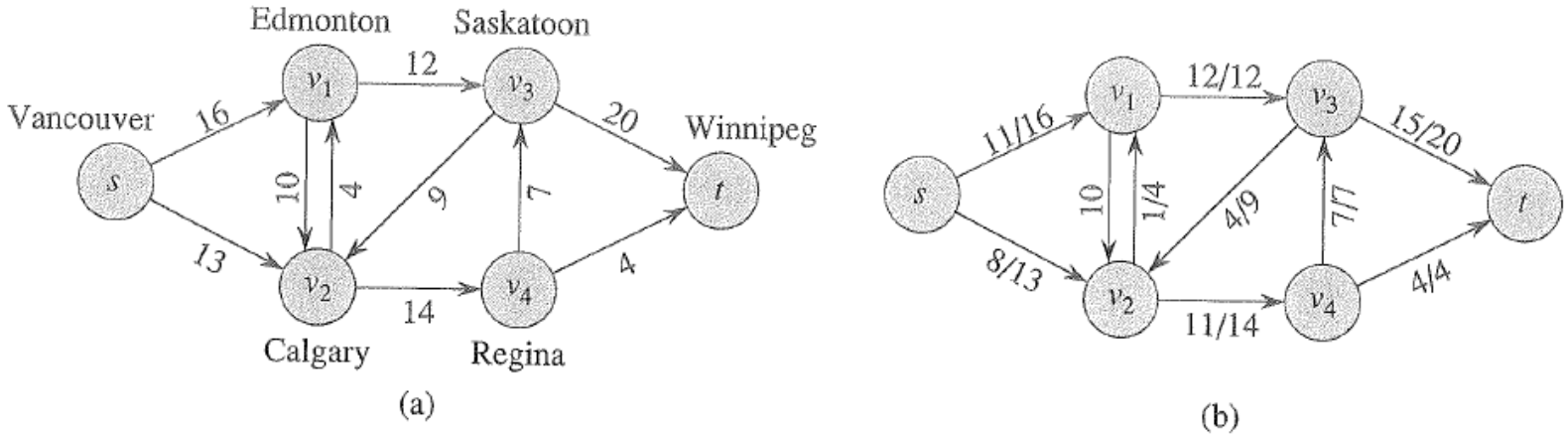
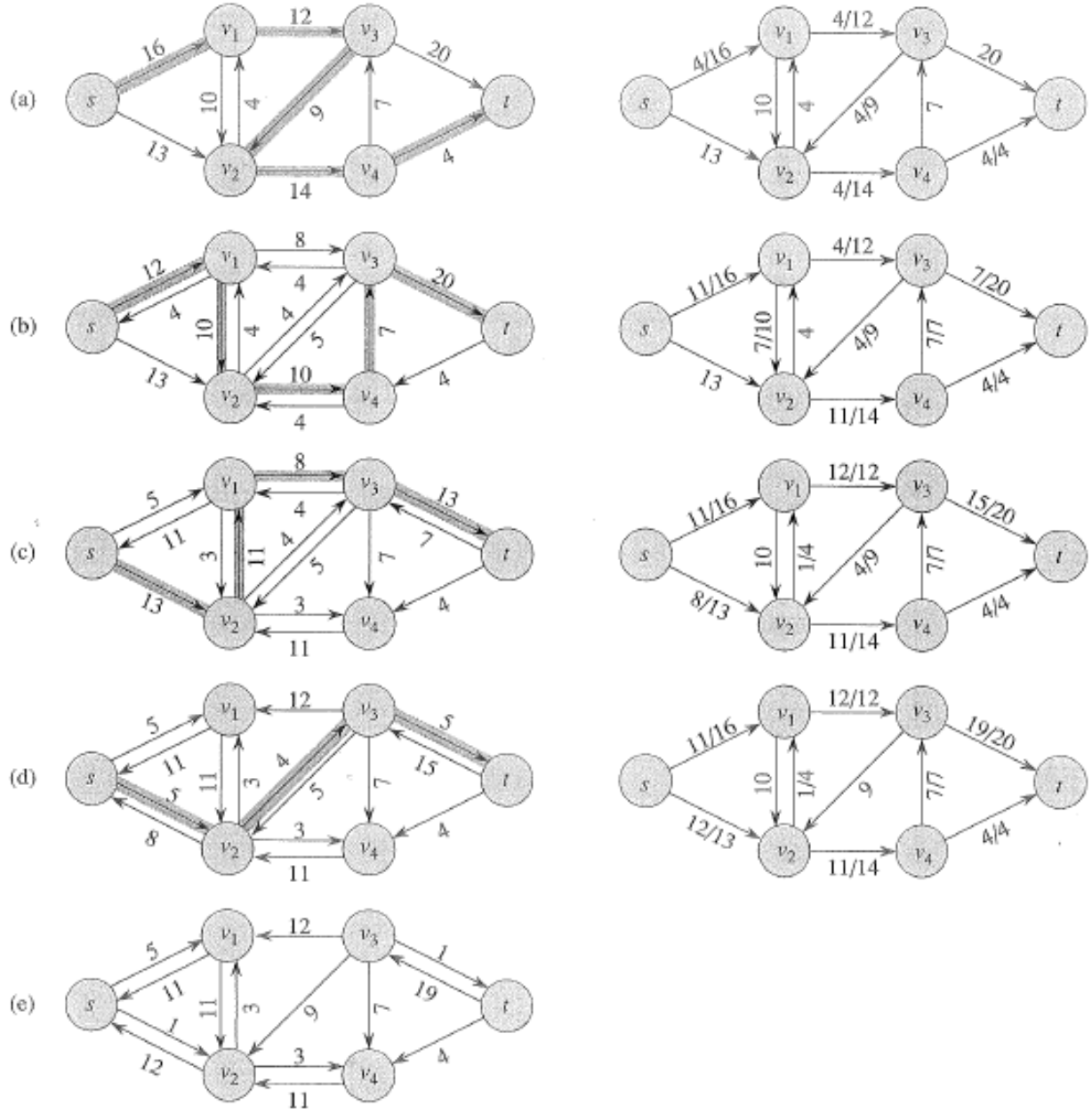
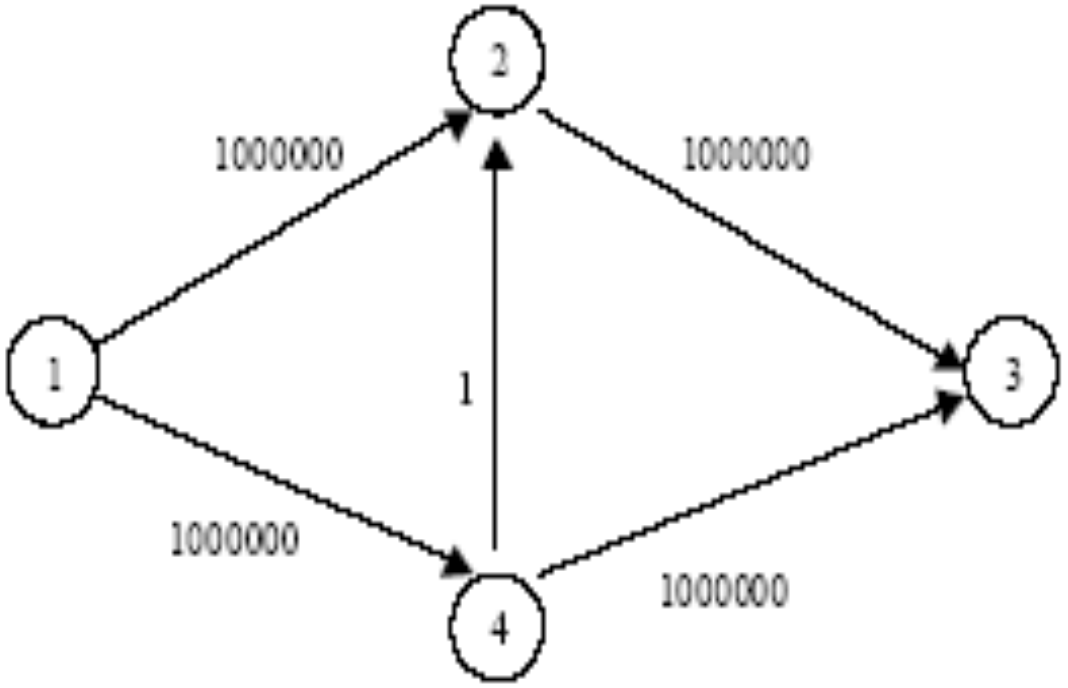


Figure 26.1 **(a)** A flow network $G = (V, E)$ for the Lucky Puck Company's trucking problem. The Vancouver factory is the source s, and the Winnipeg warehouse is the sink t. Pucks are shipped through intermediate cities, but only $c(u, v)$ crates per day can go from city u to city v. Each edge is labeled with its capacity. **(b)** A flow f in G with value $|f| = 19$. Only positive flows are shown. If $f(u, v) > 0$, edge (u, v) is labeled by $f(u, v)/c(u, v)$. (The slash notation is used merely to separate the flow and capacity; it does not indicate division.) If $f(u, v) \leq 0$, edge (u, v) is labeled only by its capacity.

Residual Graph *Flow/Capacity Graph*



Classical Bad Case