Priority Queue

- **Operations**
  - MAXIMUM(S) ,
  - INSERT(S, x)
  - EXTRACT-MAX(S)
  - INCREASE-KEY(S, x, k)
- **Implementation**
  - Use a HEAP.
  - MAXIMUM:
    - Return value stored at root.
  - INSERT:
    - Insert at last leaf and percolate up tree.
  - EXTRACT-MAX:
    - Delete root of heap and call HEAPIFY.
  - INCREASE-KEY:
    - Change value and percolate up tree.

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

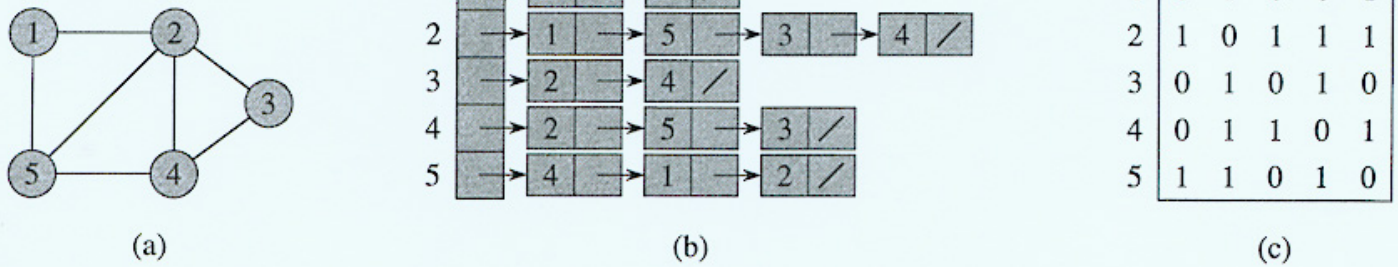


Figure 22.1 Two representations of an undirected graph. **(a)** An undirected graph G having five vertices and seven edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

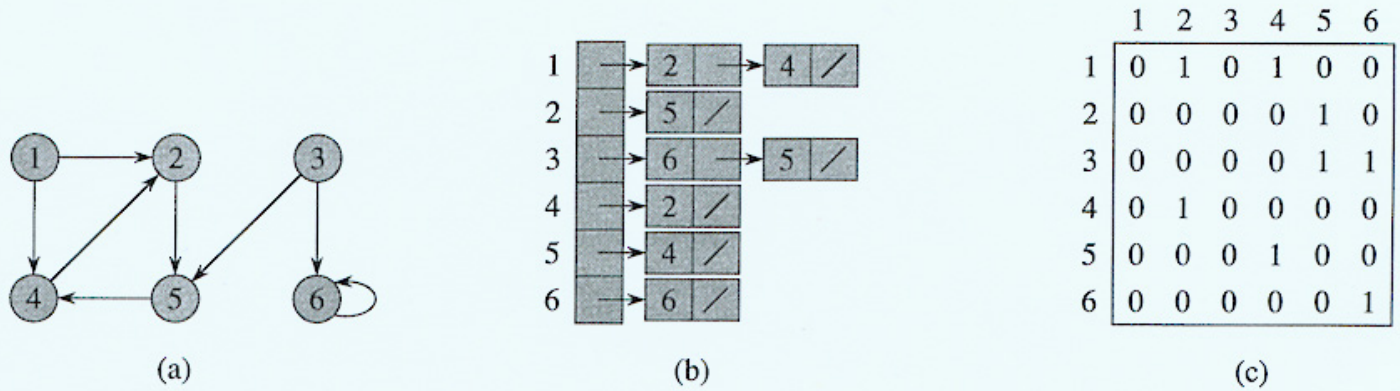


Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G having six vertices and eight edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

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.

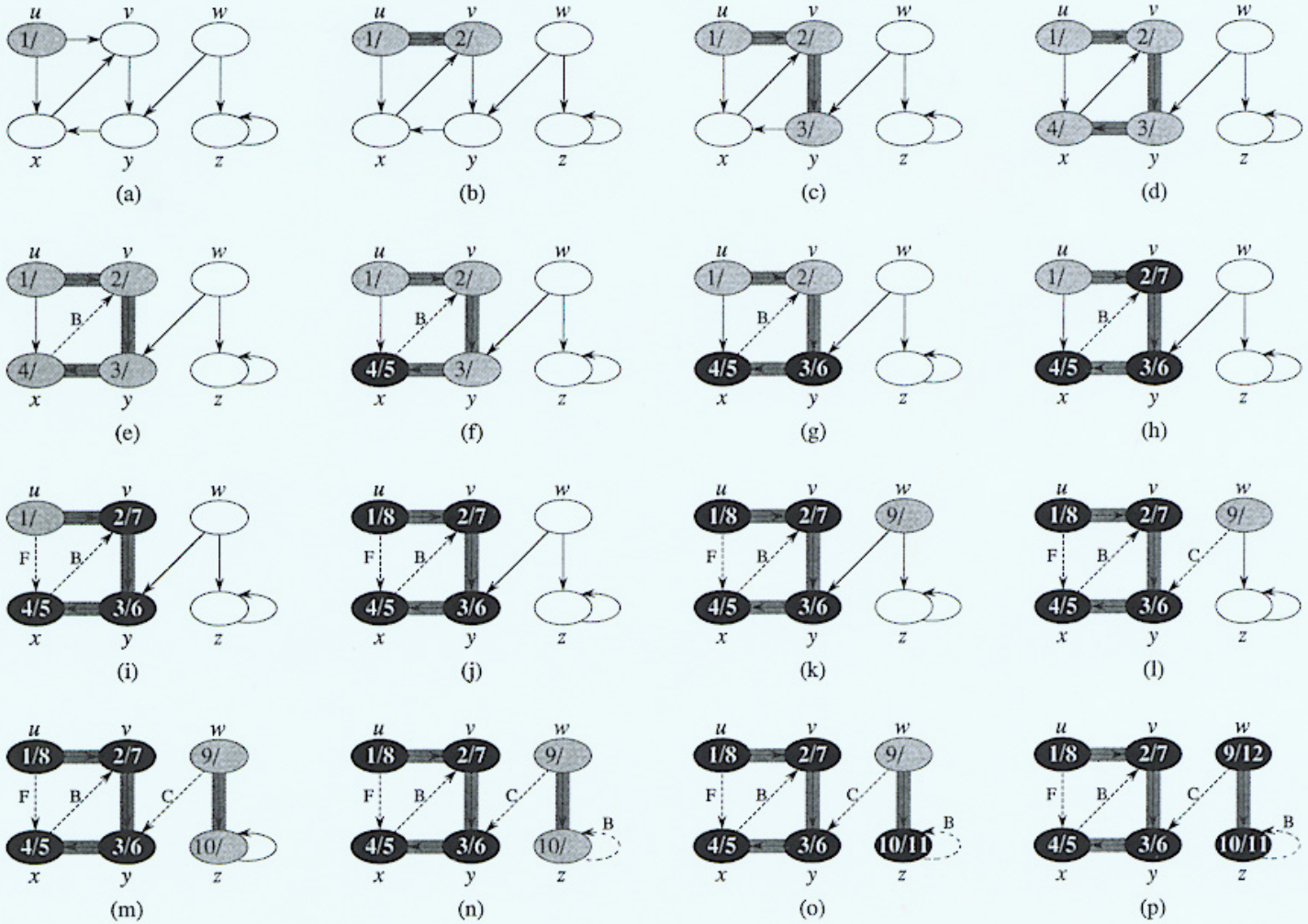


Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges.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

```
DFS(G)
1.  For each vertex u ∈ V[G] do
2.        color[u] ← WHITE
3.        π[u] ← NIL
4.  Time ← 0
5.  For each vertex u ∈ V[G] do
6.        if color[u] = WHITE then
7.              DFS-VISIT(u)
```

Depth First Search

```
DFS-VISIT(u)
1.  VisitVertex(u)
2.  Color[u] ← GRAY
3.  Time ← Time + 1
4.  d[u] ← Time
5.  for each v ∈ Adj[u] do
6.      VisitEdge(u,v)
7.      if (v ≠ π[u]) then
8.          if (color[v] = WHITE) then
9.              π[v] ← u
10.             DFS-VISIT(v)
11. color[u] ← BLACK
12. F[u] ← Time ← Time + 1
```

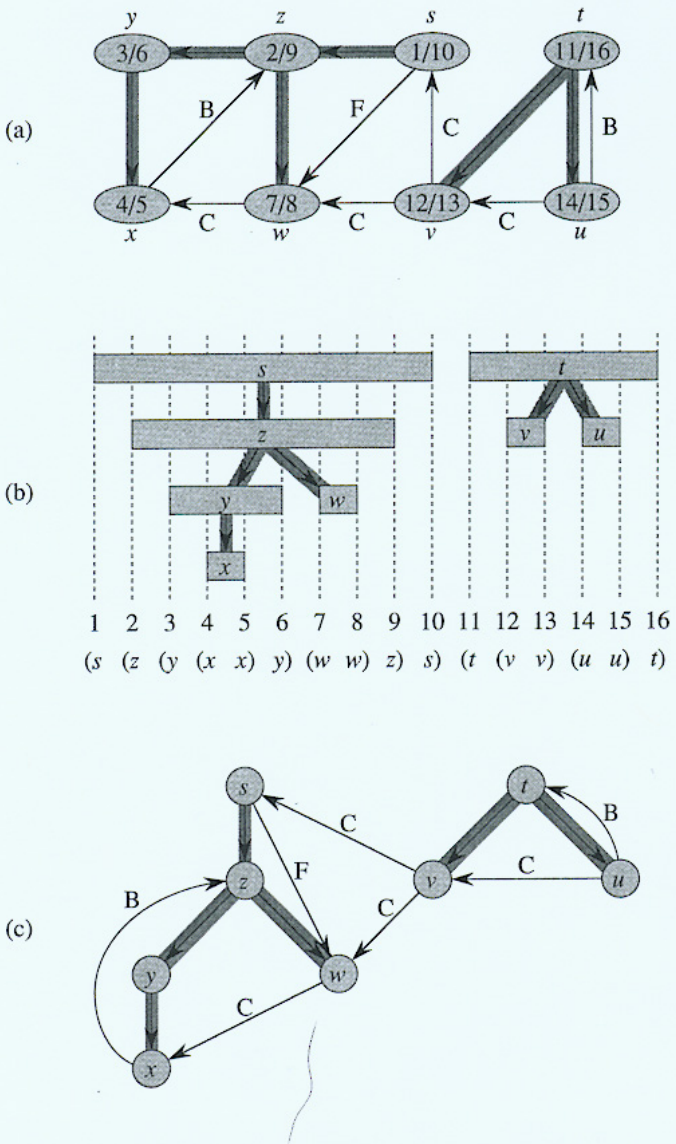


Figure 22.5 Properties of depth-first search. **(a)** The result of a depth-first search of a directed graph. Vertices are timestamped and edge types are indicated as in Figure 22.4. **(b)** Intervals for the discovery time and finishing time of each vertex correspond to the parenthesization shown. Each 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.

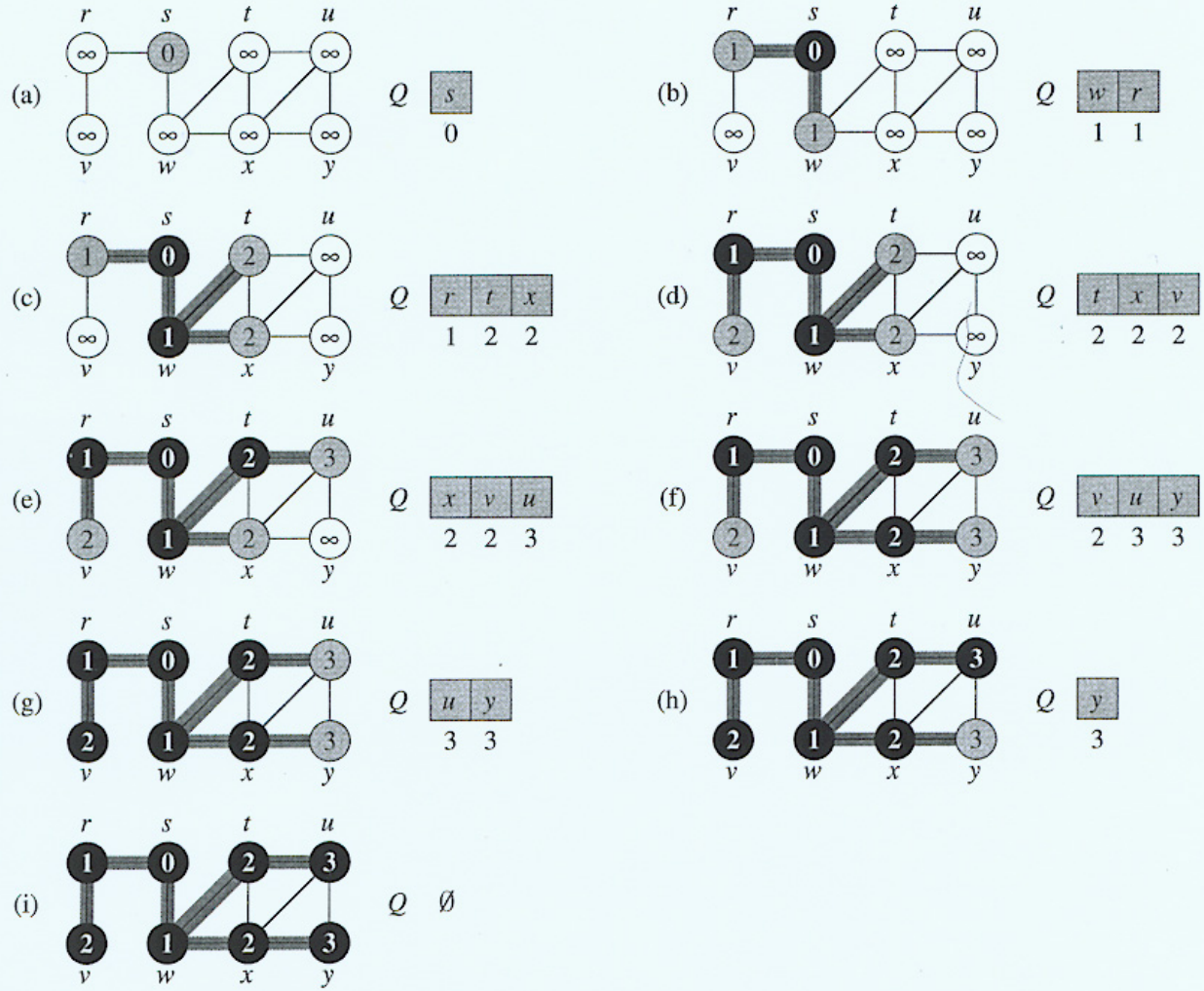


Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. Within each vertex u is shown $d[u]$. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances are shown next to vertices in the queue.

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 14.30A

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

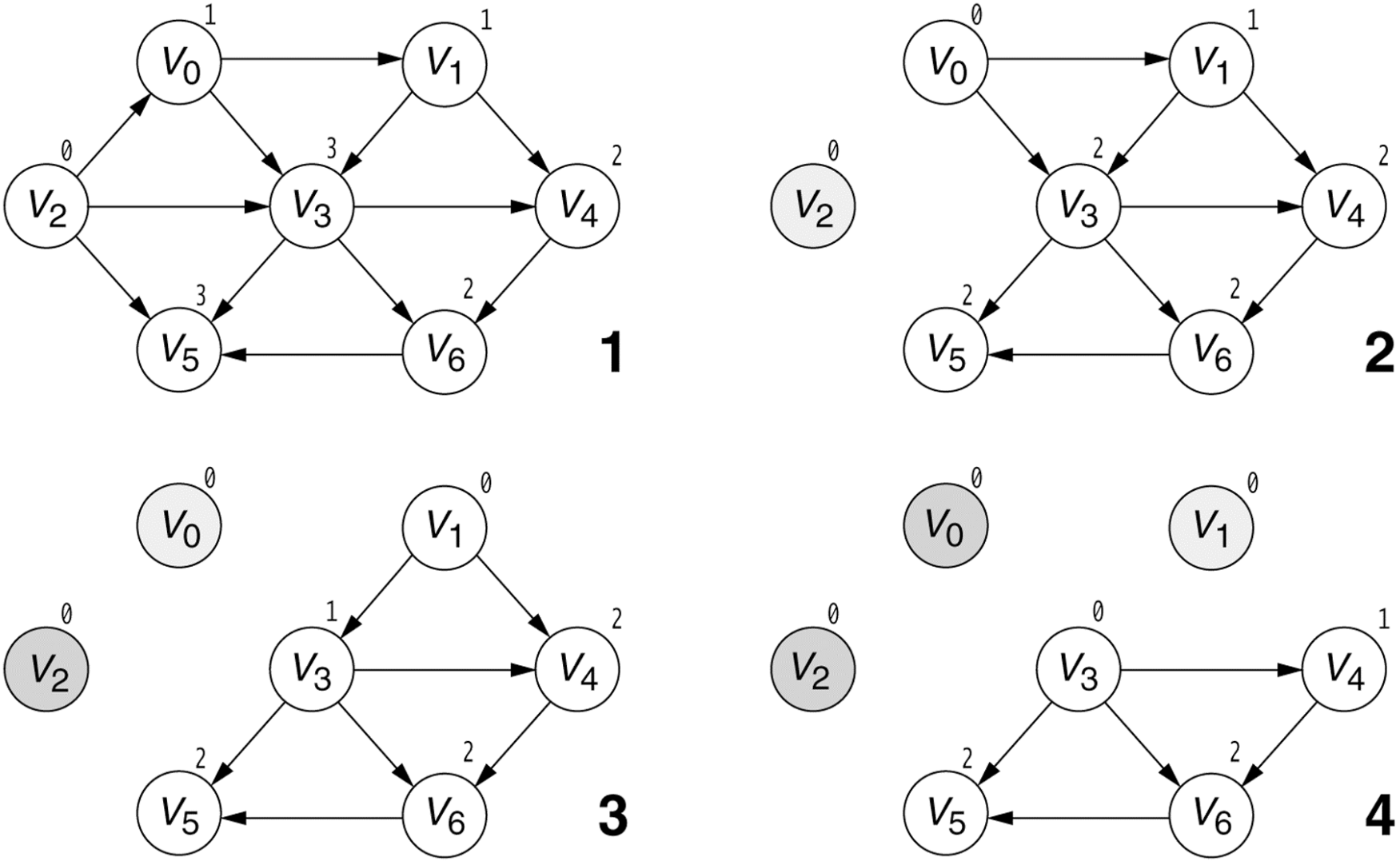


Figure 14.30B

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

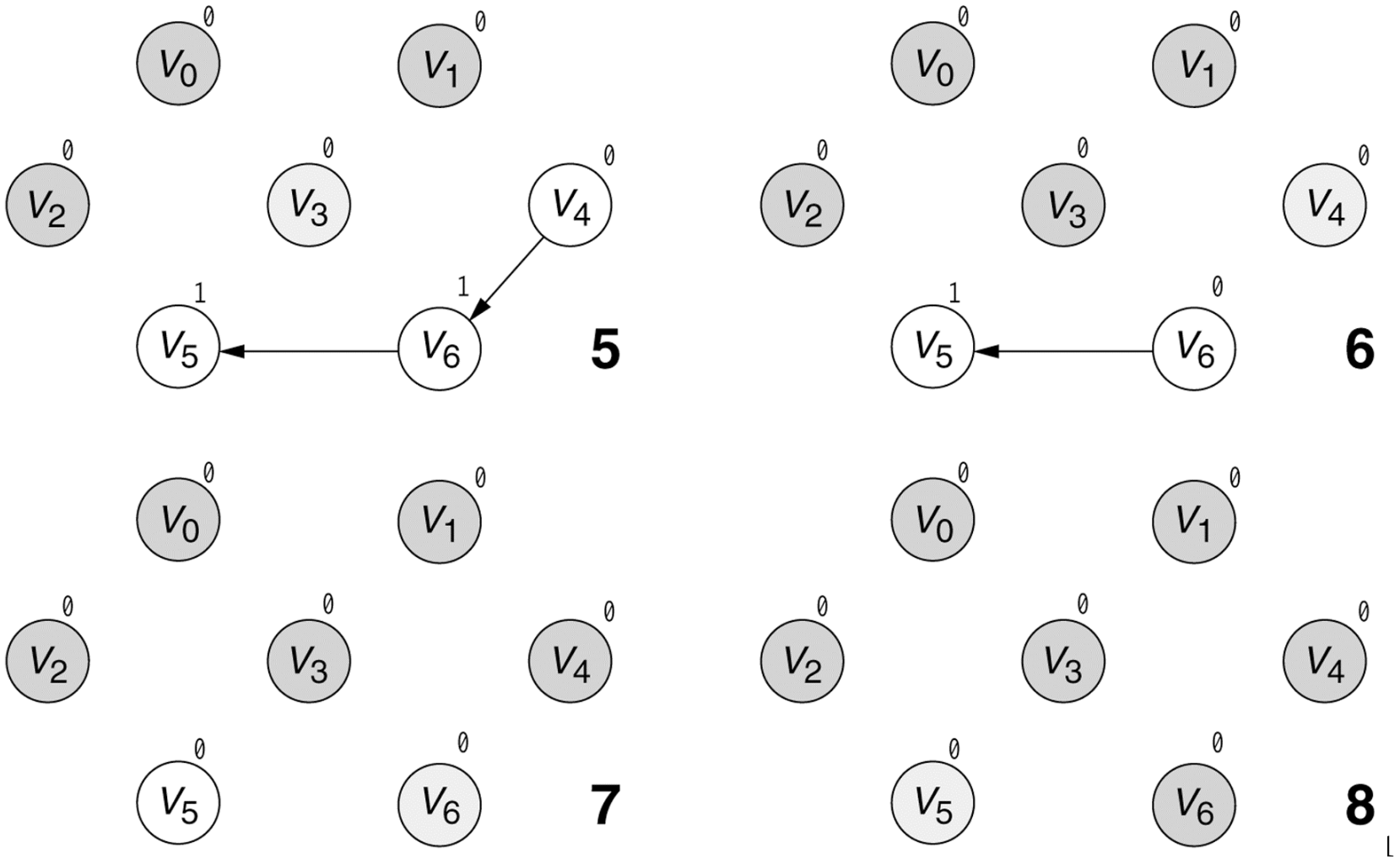


Figure 14.31A

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

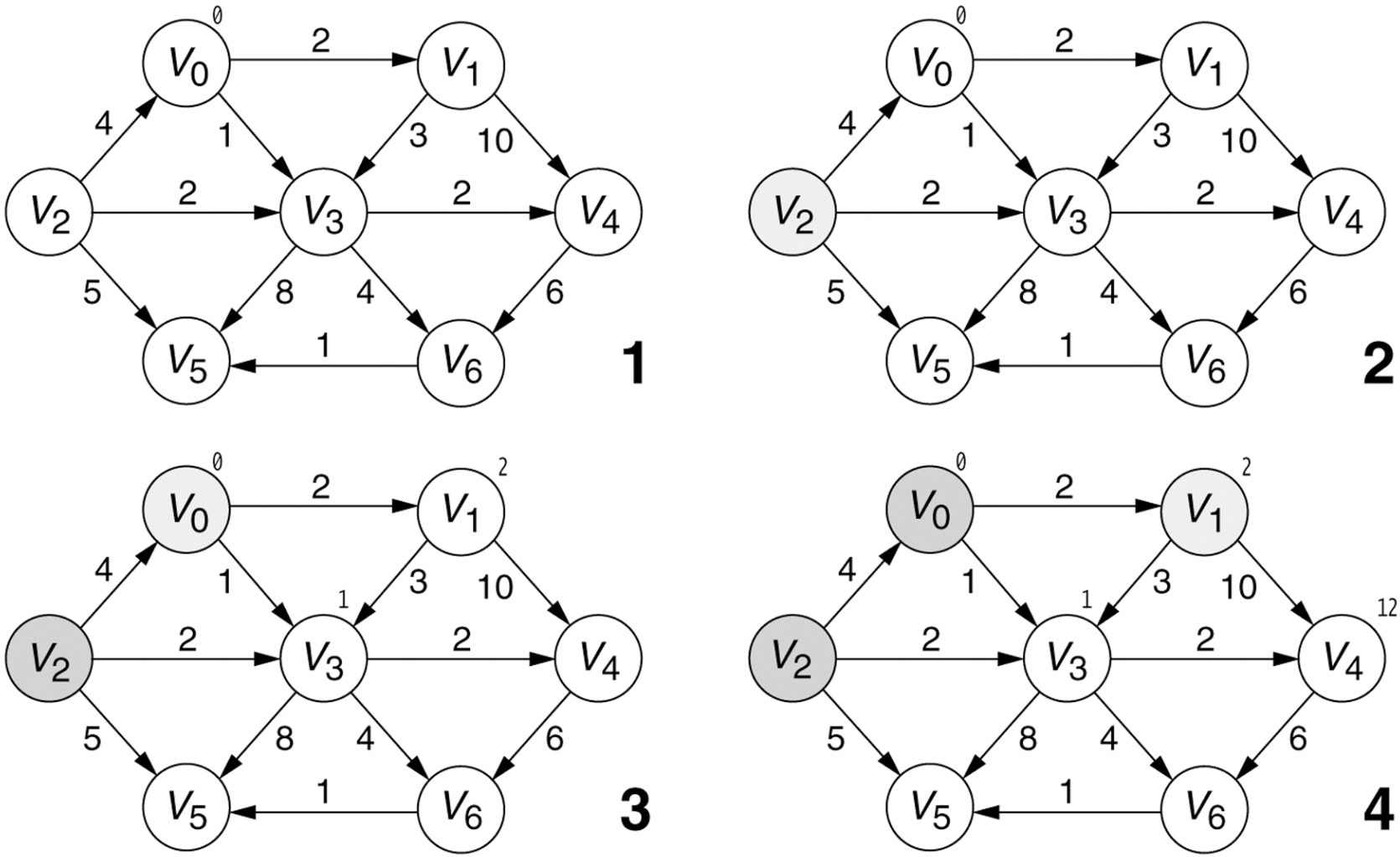
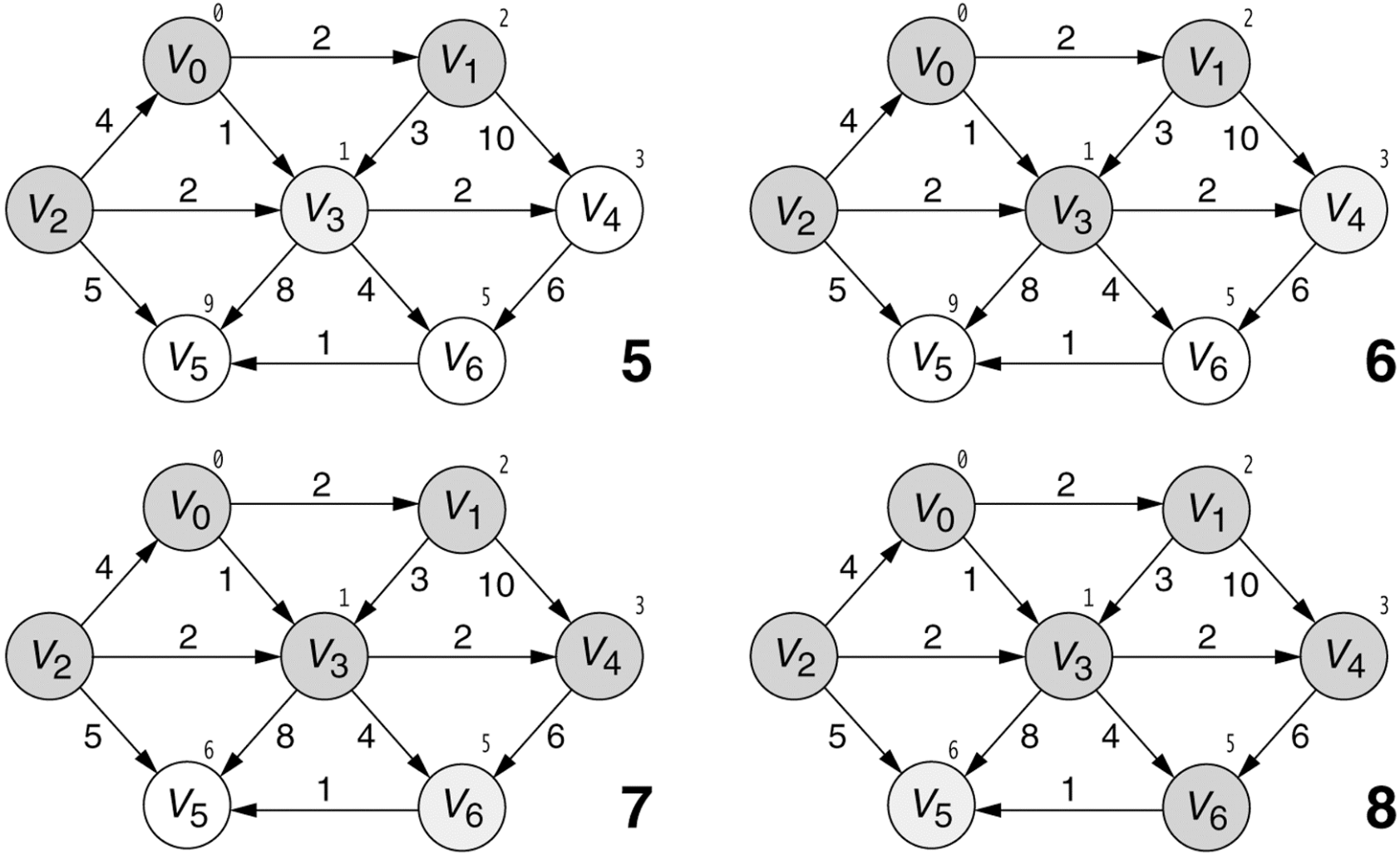


Figure 14.31B

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



5 6 7 8

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

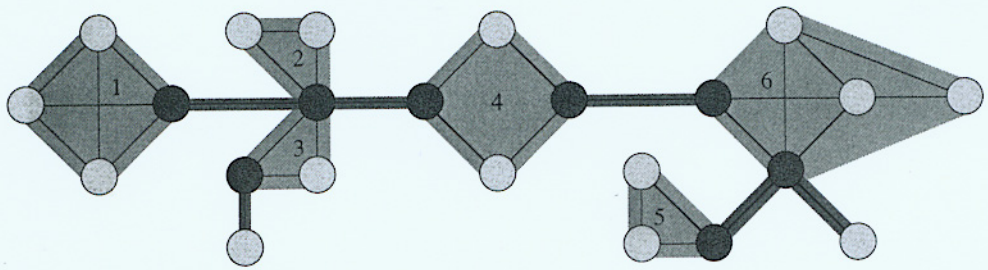


Figure 22.10 The articulation points, bridges, and biconnected components of a connected, undirected graph for use in Problem 22-2. The articulation points are the heavily shaded vertices, the bridges are the heavily shaded edges, and the biconnected components are the edges in the shaded regions, with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

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

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

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

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
- $\Phi(D_0) = 0$
- $\Phi(D_n) \geq 0$

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

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