Data Structures

Giri Narasimhan
Office: ECS 254A
Phone: x-3748
giri@cs.fiu.edu

Graphs

- Graphs model networks of various kinds: roads, highways, oil pipelines, airline routes, dependency relationships, etc.

- Graph G(V,E)
- V Vertices or Nodes
- E Edges or links connect vertices
- Directed vs. Undirected edges

Graph Representations

- Adjacency Matrix
- Adjacency List

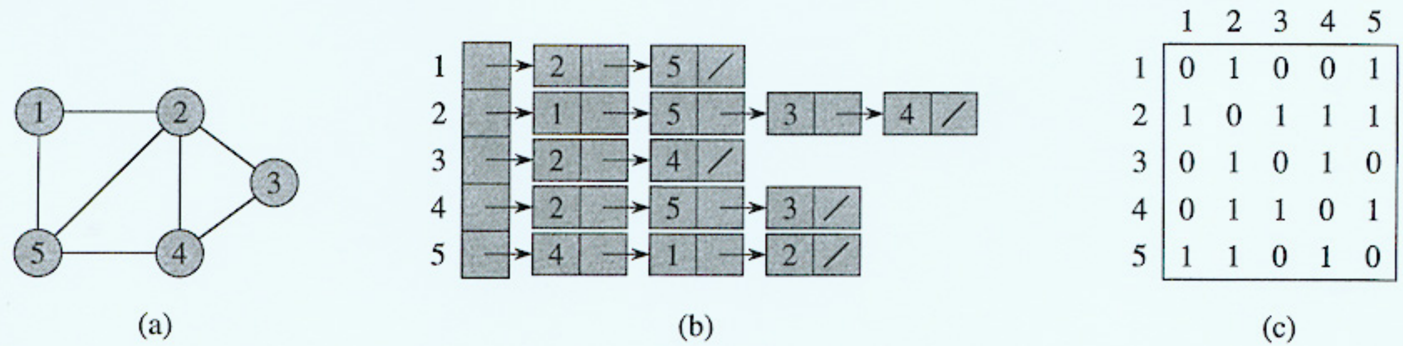


Figure 22.1 Two representations of an undirected graph. **(a)** An undirected graph G having five vertices and seven edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

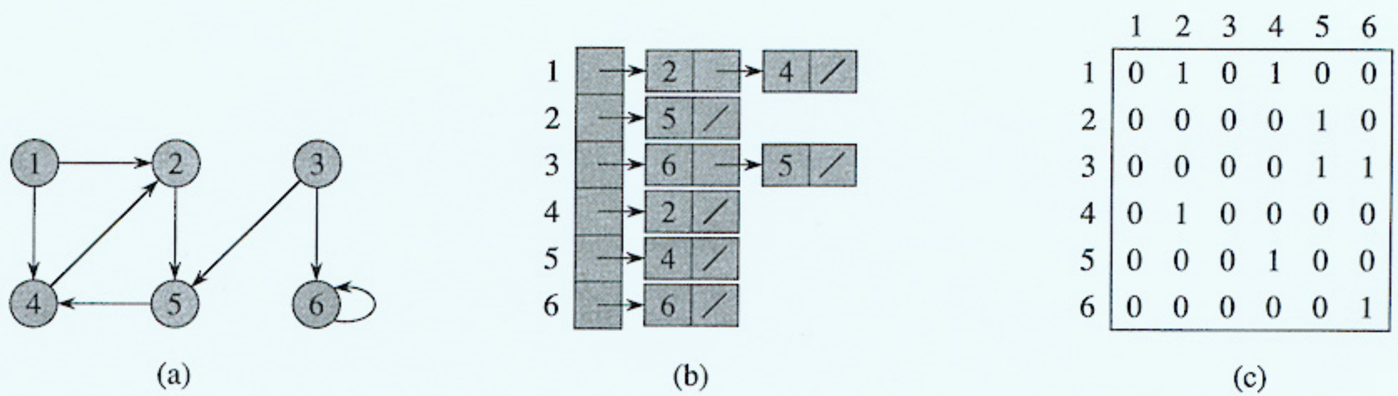


Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G having six vertices and eight edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

Vertex and Edge classes

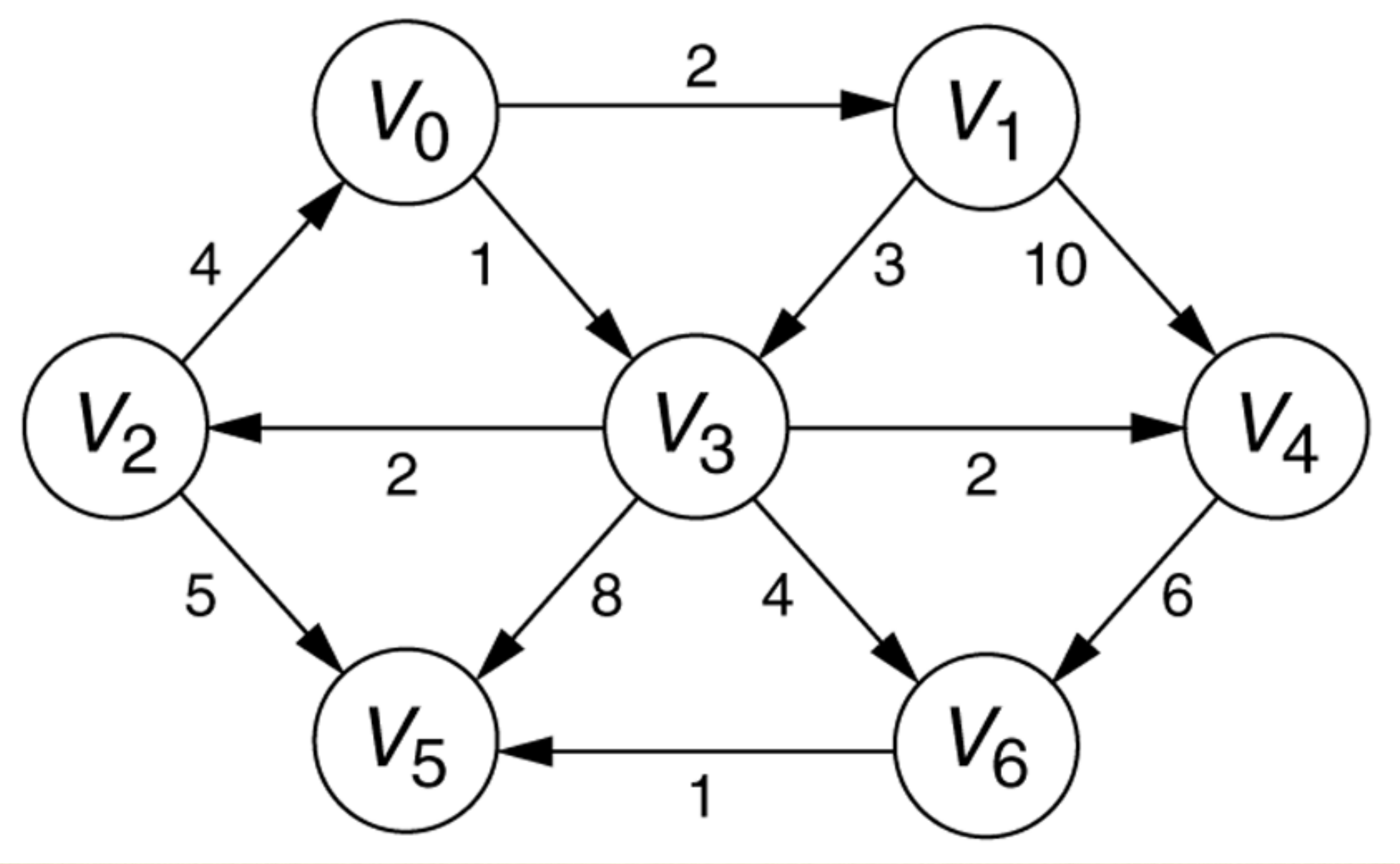
```
Class Edge {
    public Vertex dest;
    public double weight;

    public Edge (Vertex d,
                 double w) {
        dest = d;
        weight = w;
    }
}
```

```
Class Vertex {
    public String Name;
    public AnyType extraInfo;
    public List adj;
    public int dist; // double?
    public Vertex prev;
    public Vertex (String s) {
        Name = s;
        adj = new LinkedList();
        reset();
    }
    public reset () {
        dist=INFNT; path=null;
    }
}
```

Graphs

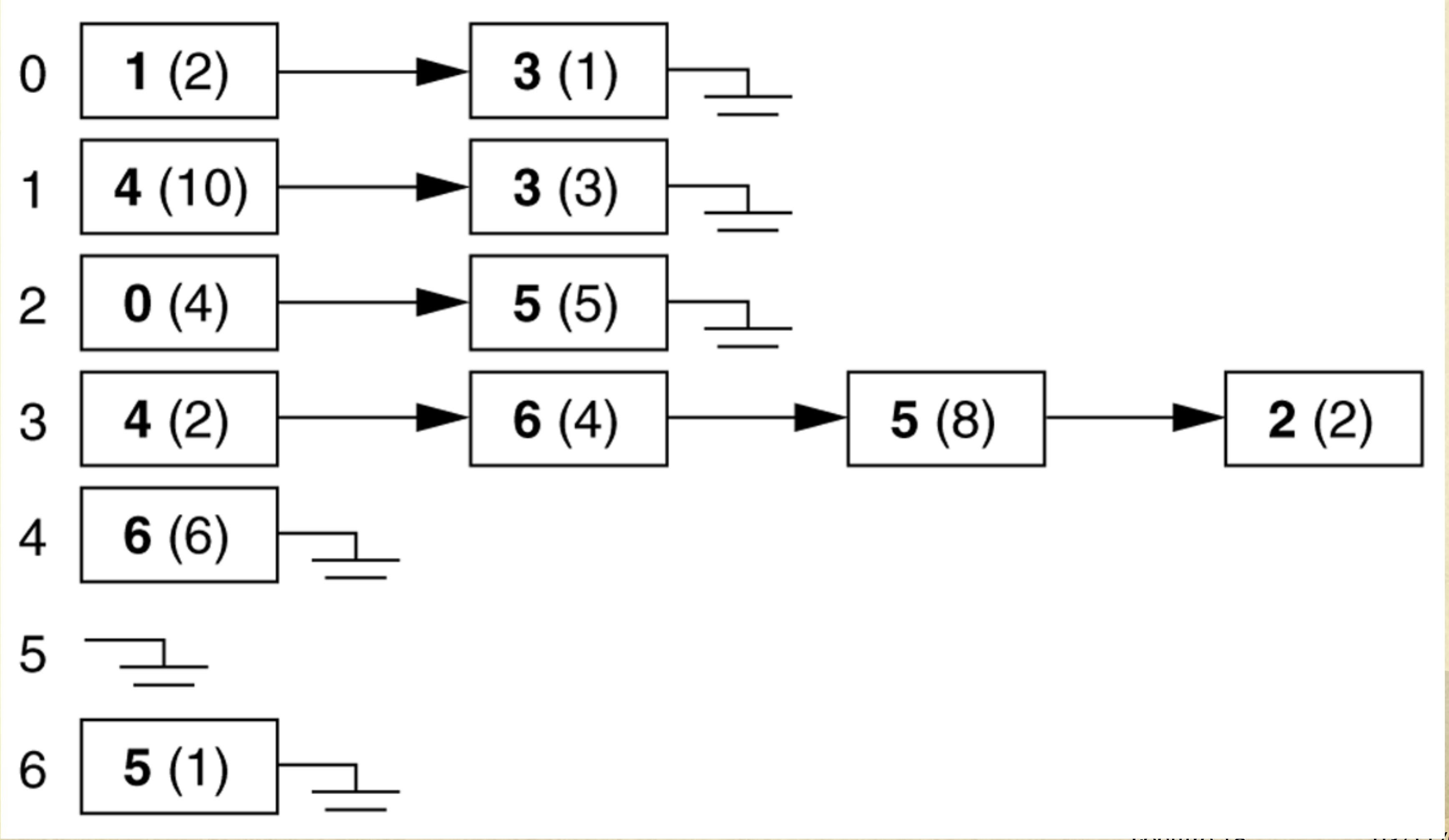
- Graphs can be augmented to store extra info (e.g., city population, oil flow capacity, etc.)
- Weighted vs. Unweighted
- Paths and Cycles

Figure 14.1
A directed graph.



© 2002 Addison Wesley

Figure 14.2
Adjacency list representation of the graph shown in Figure 14.1; the nodes in list *i* represent vertices adjacent to *i* and the cost of the connecting edge.

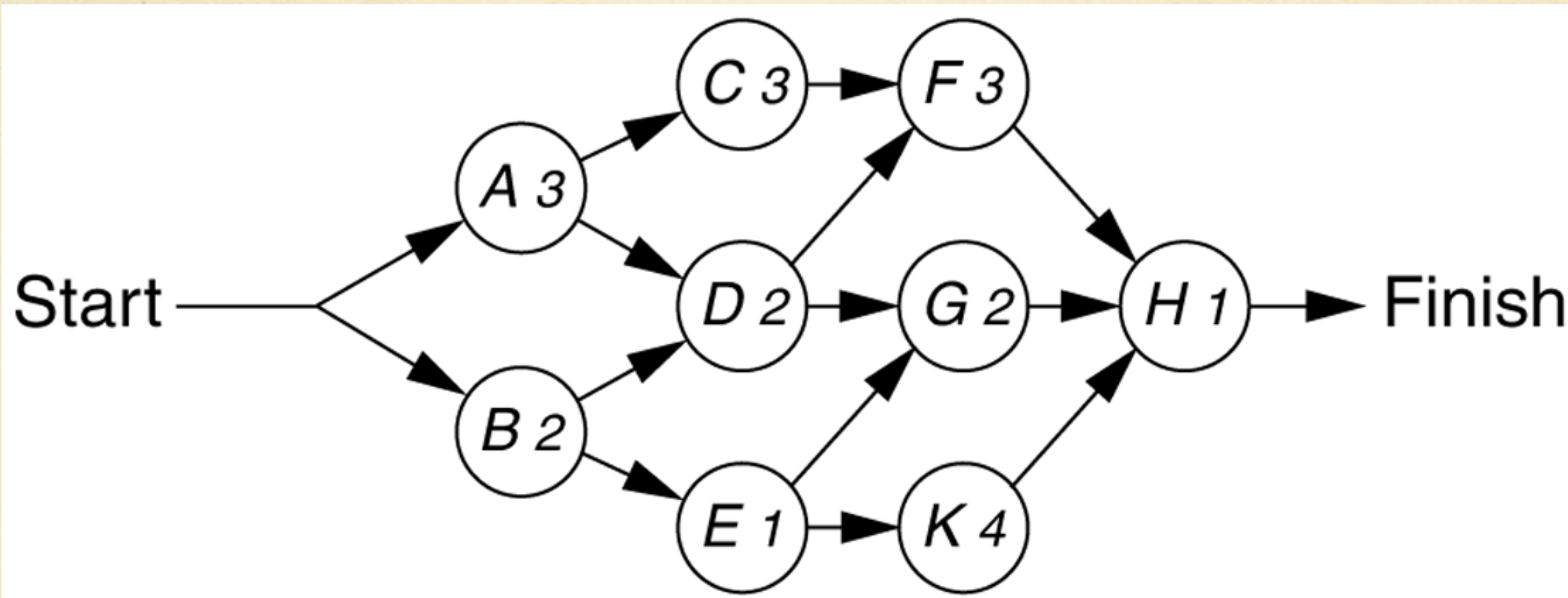


 © 2002 Addison Wesley

Adjacency Lists

- Constructing adjacency lists
  - Input: list of edges
  - Output: adjacency list for all vertices
  - Time: O(L), where L is length of list of edges.
- Check if edge exists
  - Input: edge (u,v)
  - Output: does the edge exist in the graph G?
  - Time: $O(d_u)$, where d_u is the number of entries in u's adjacency list. In the worst case it is O(N), where N is the number of vertices
- Need a MAP data structure to map vertex name or ID to (internal) vertex number.

Figure 14.33
An activity-node graph



© 2002 Addison Wesley

Topological Sort Example

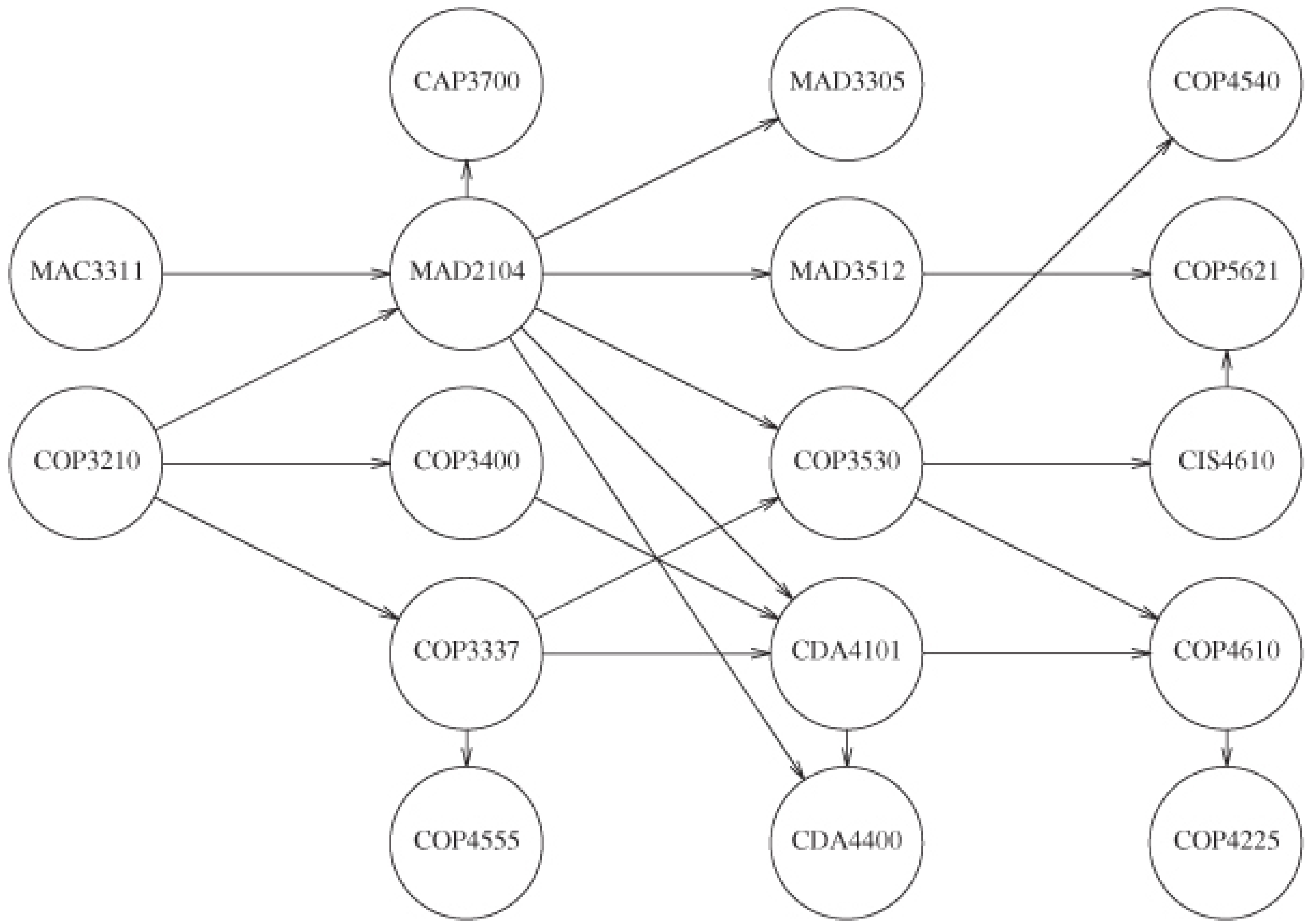
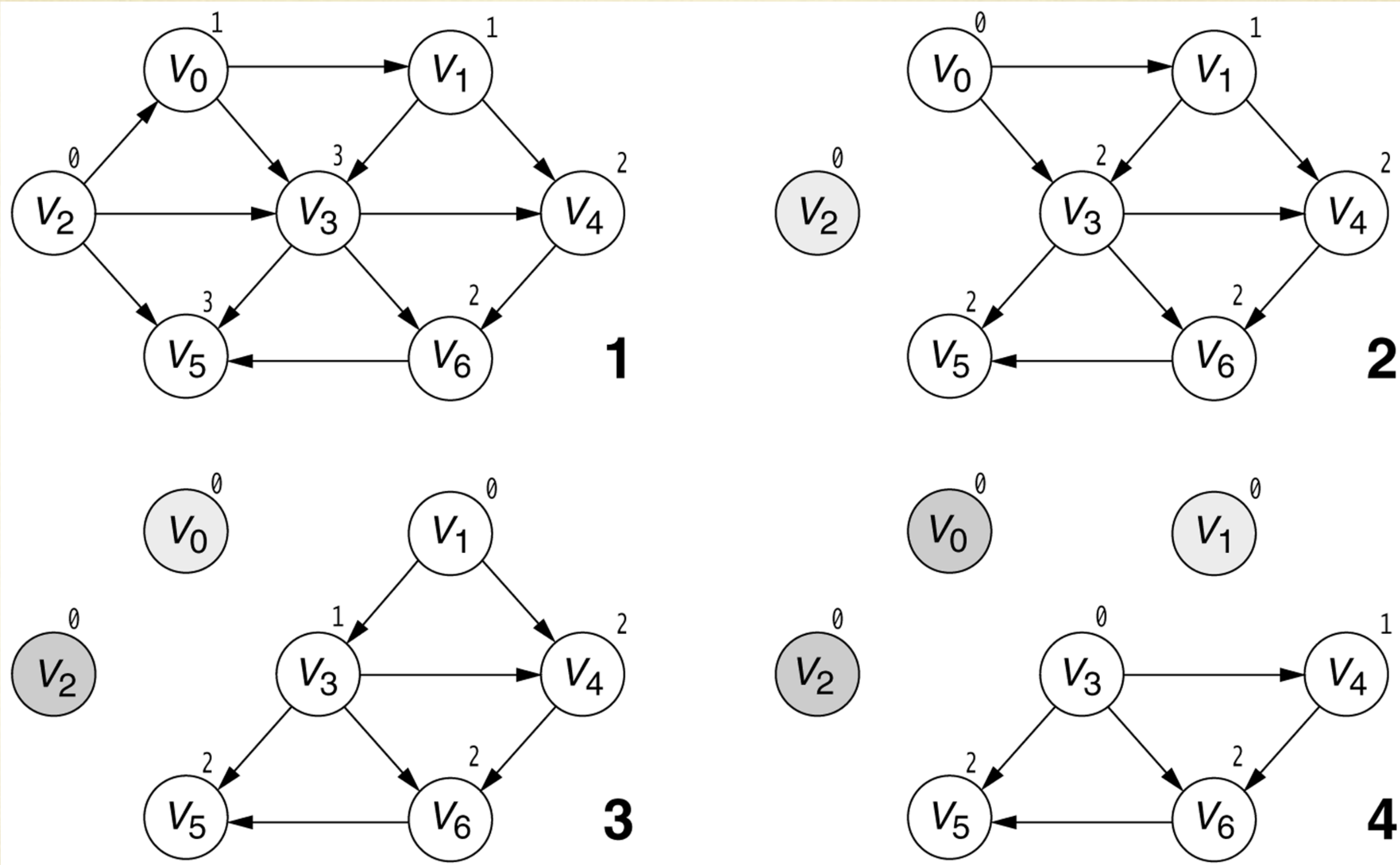


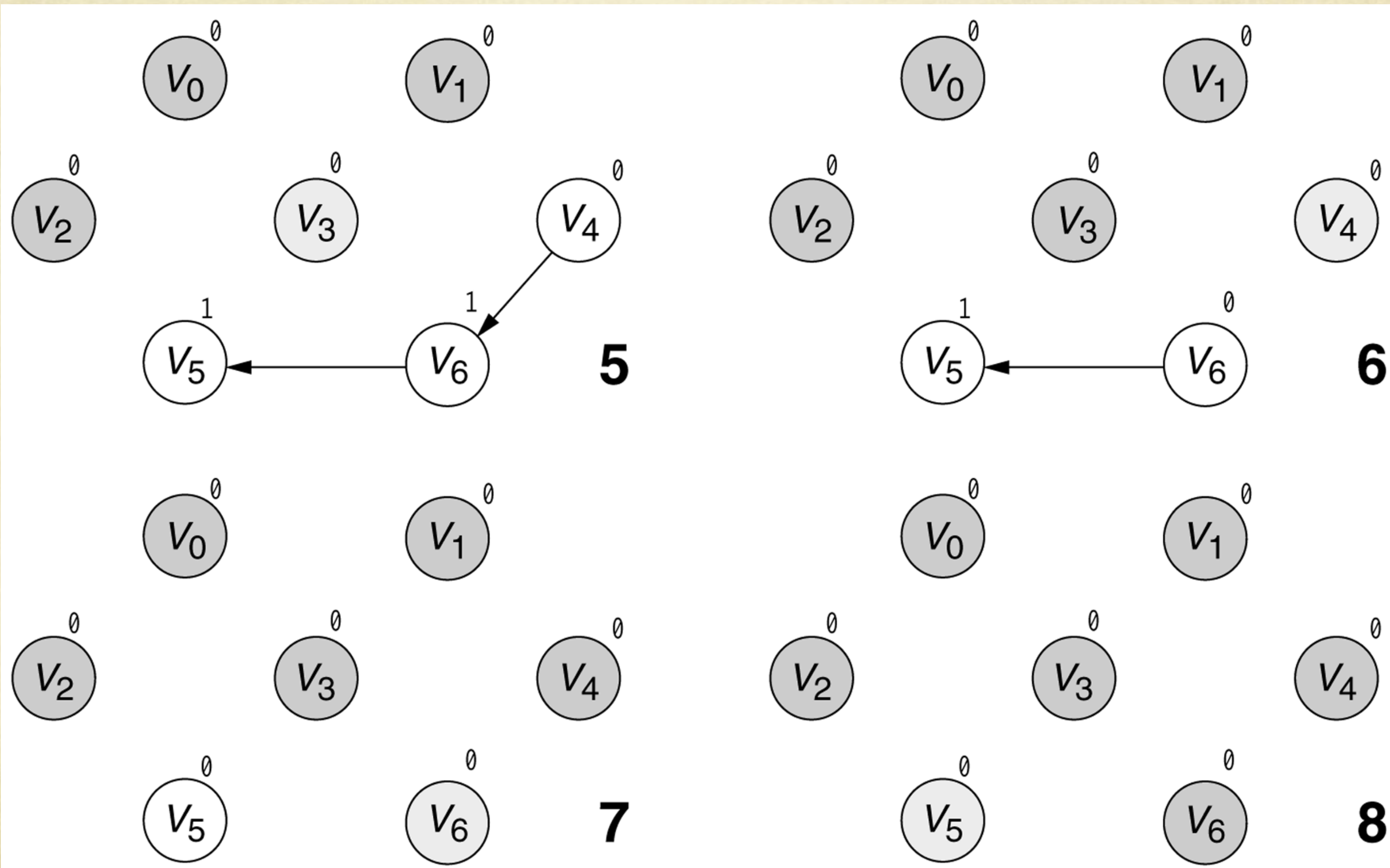
Figure 9.3 An acyclic graph representing course prerequisite structure

Figure 14.30A
A topological sort. The conventions are the same as those in Figure 14.21 (continued).



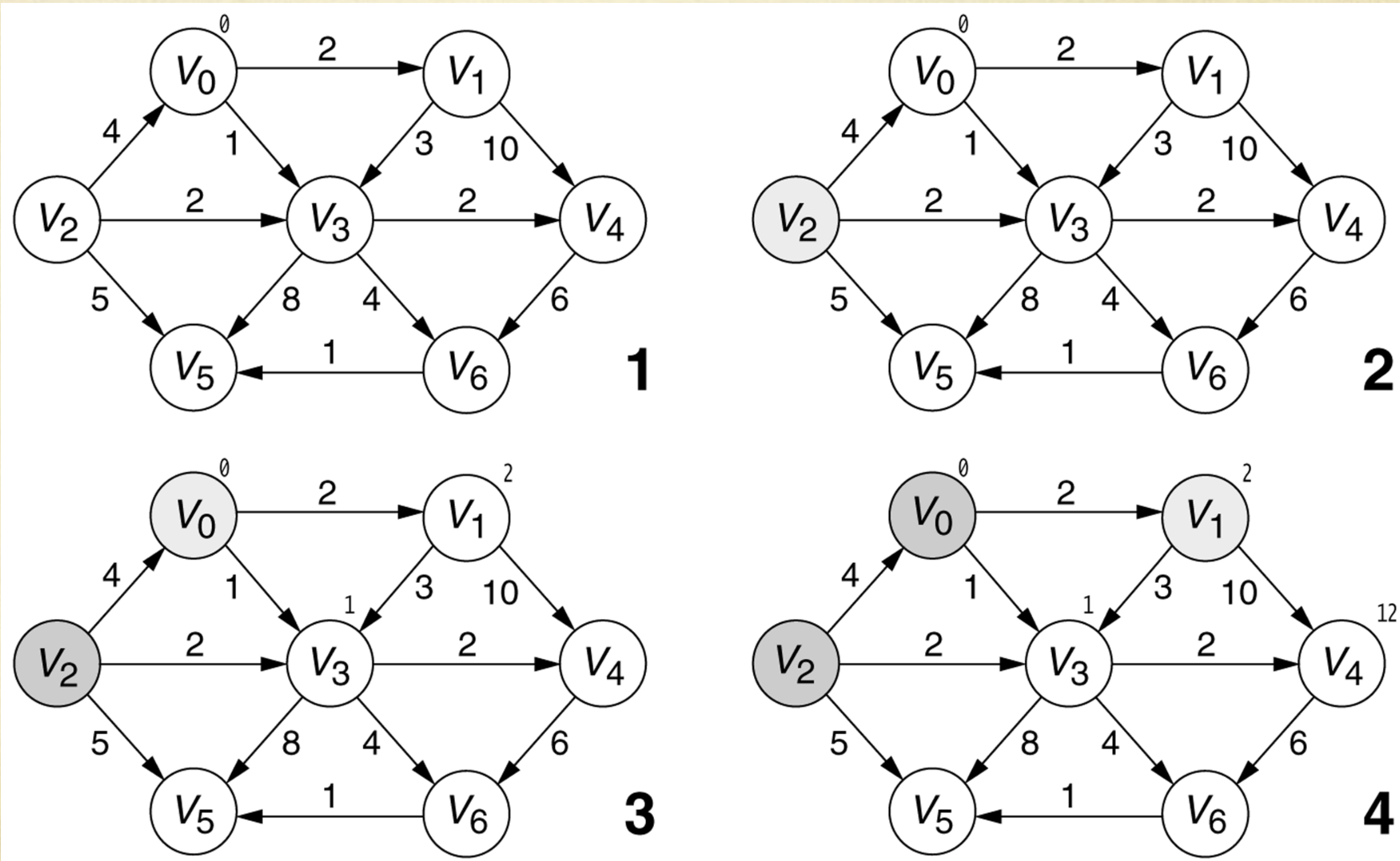
 © 2002 Addison Wesley

Figure 14.30B
A topological sort. The conventions are the same as those in Figure 14.21.



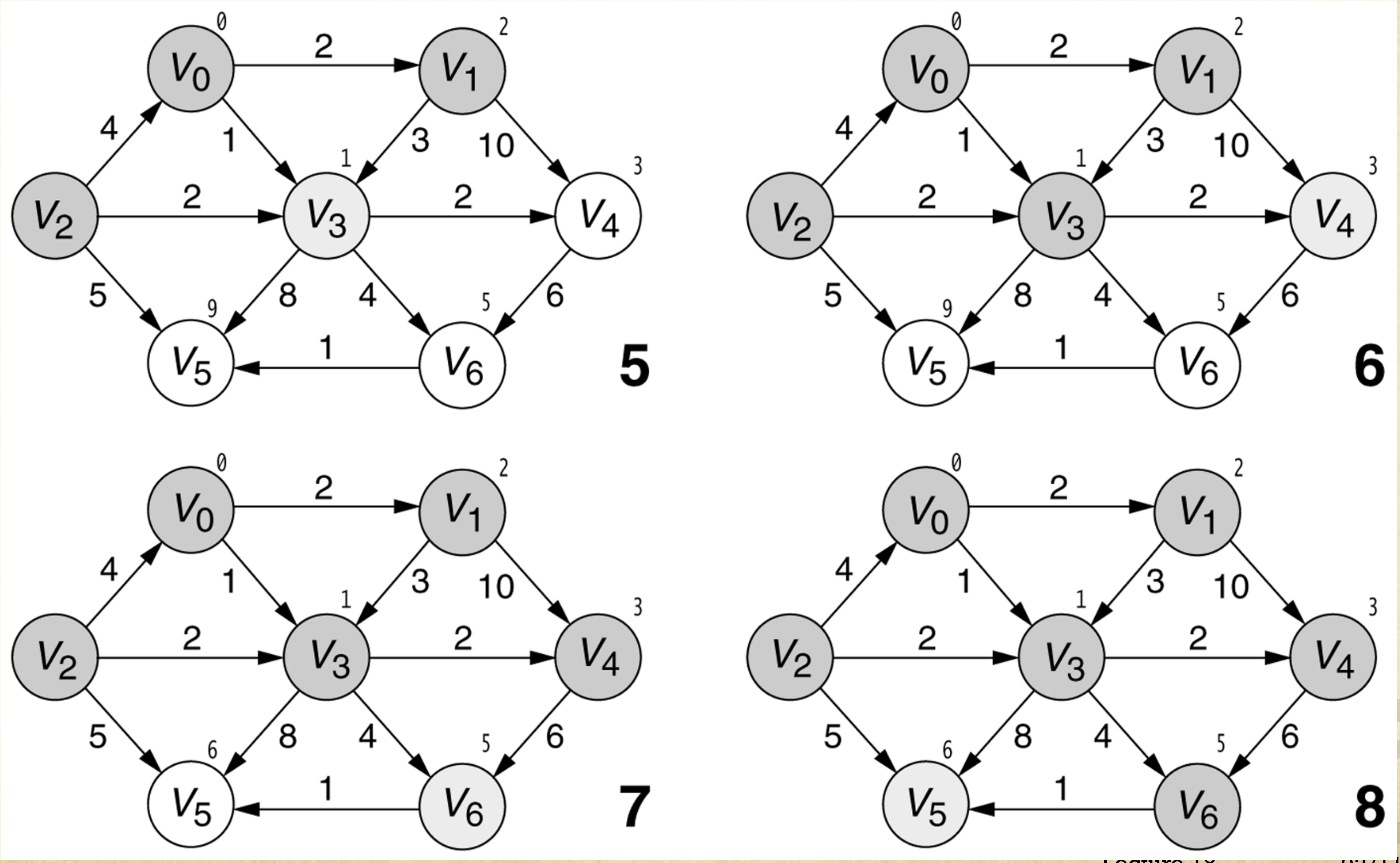
 © 2002 Addison Wesley

Figure 14.31A
The stages of acyclic graph algorithm. The conventions are the same as those in Figure 14.21 (*continued*).



 © 2002 Addison Wesley

Figure 14.31B
The stages of acyclic graph algorithm. The conventions are the same as those in Figure 14.21.



 © 2002 Addison Wesley

Topological Sort

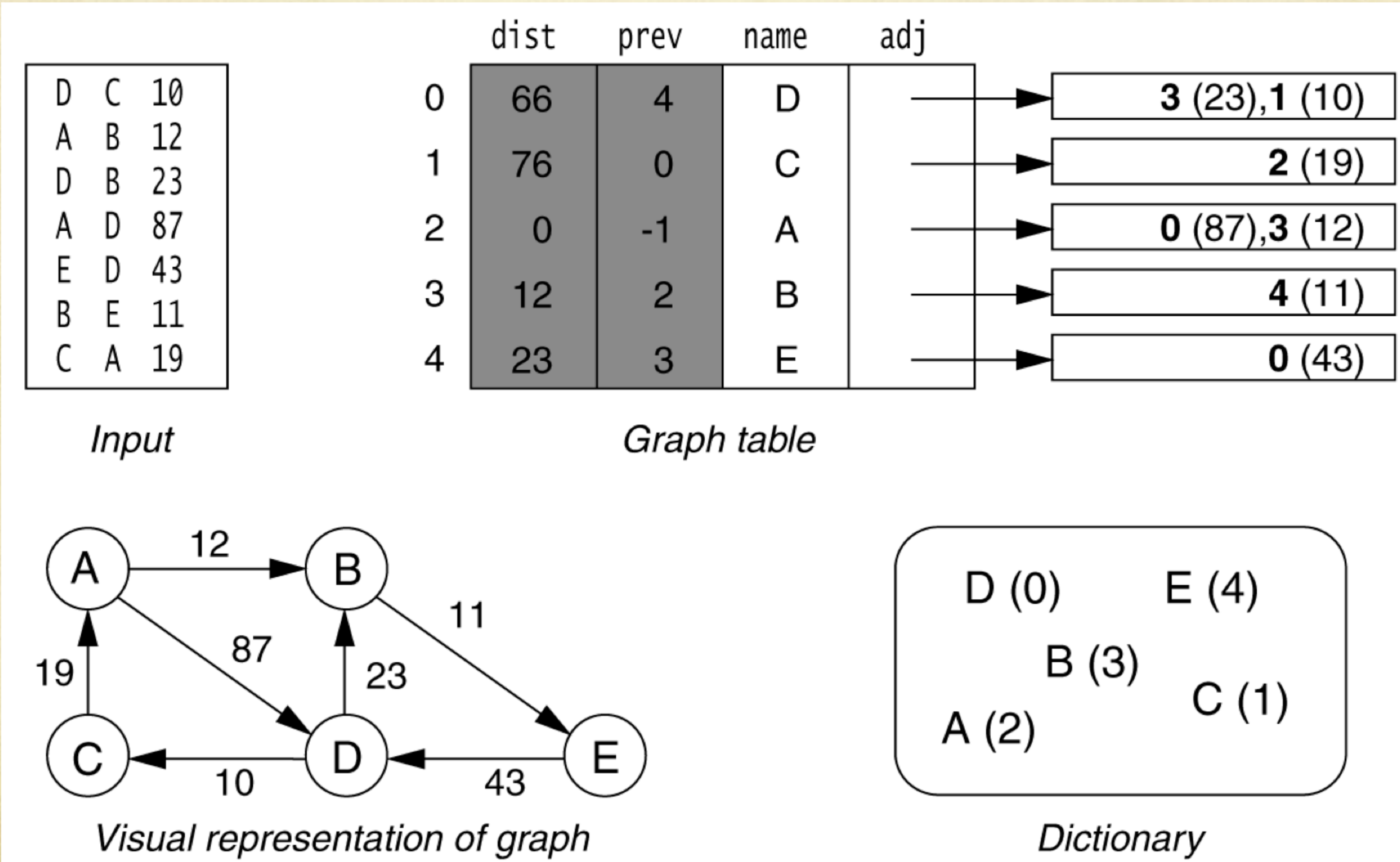
```
void topSort () {
        for( int j = 0; j < N; j++) {
                Vertex v = findVertexOfIndegZero();
                if (v == null)
                        return; //  Cycle found
                v.topologicalNum = j;
                for each vertex w adjacent to v
                        w.inDegree--; // use extraInfo field
        }
}
```

Time Complexity = O(n + m)

Shortest Paths

- Suppose we are interested in the shortest paths (and their lengths) from vertex "Miami" to all other vertices in the graph.
- We need to augment the data structure to store this information.

Figure 14.4
An abstract scenario of the data structures used in a shortest-path calculation, with an input graph taken from a file. The shortest weighted path from A to C is A to B to E to D to C (cost is 76).



 © 2002 Addison Wesley

Figure 14.21A

Searching the graph in the unweighted shortest-path computation. The darkest-shaded vertices have already been completely processed, the lightest-shaded vertices have not yet been used as *v*, and the medium-shaded vertex is the current vertex, *v*. The stages proceed left to right, top to bottom, as numbered *(continued).*

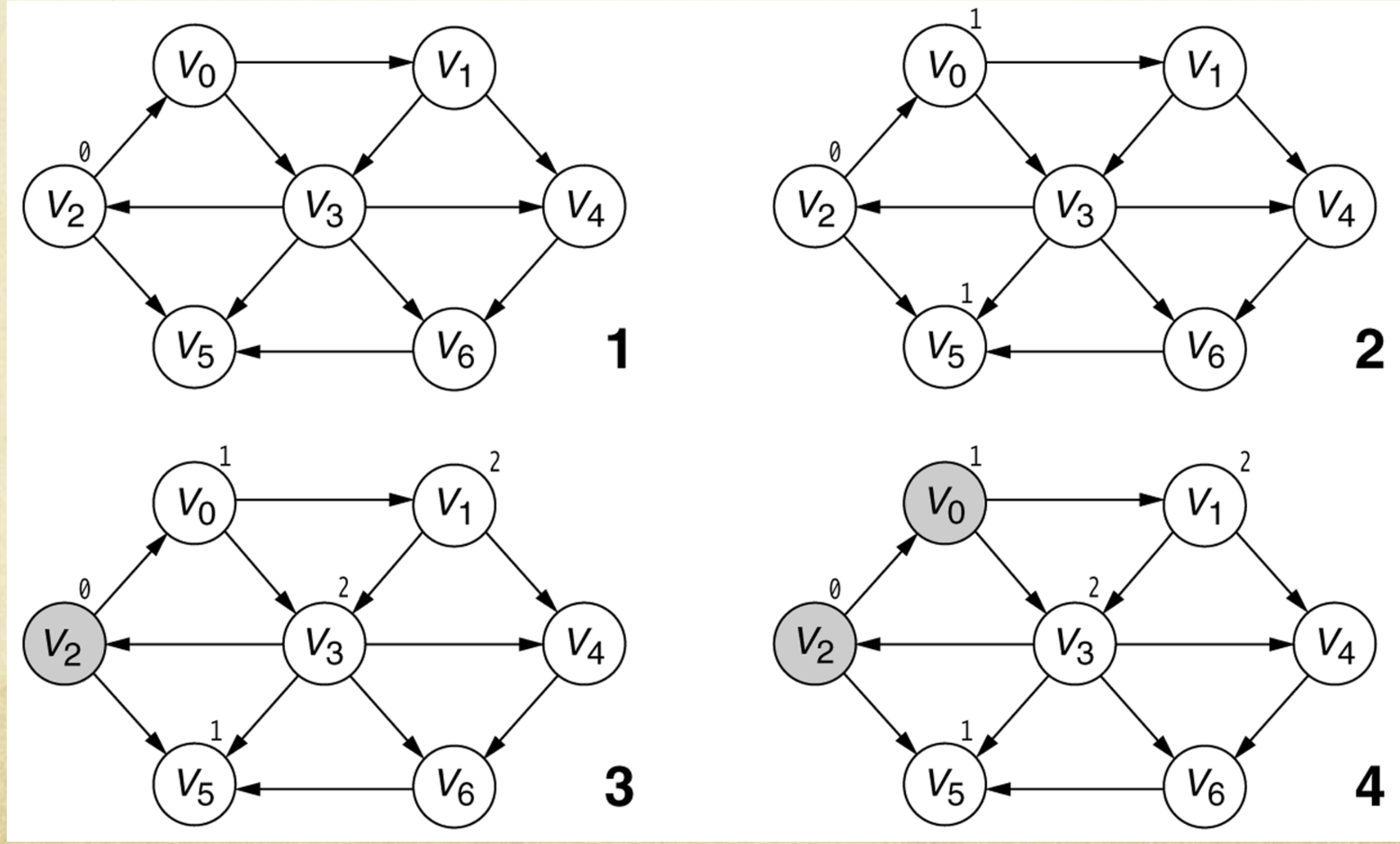
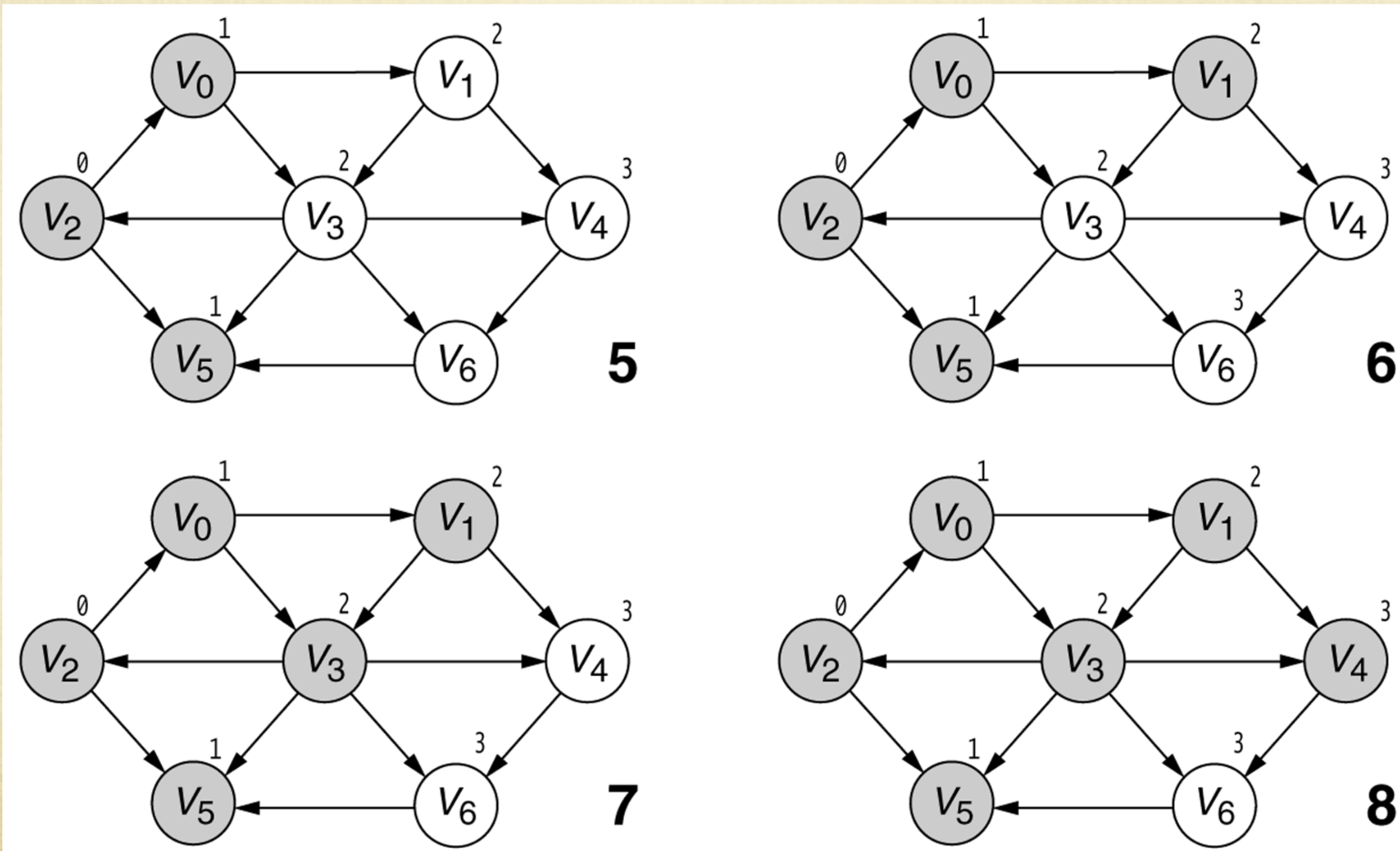


Figure 14.21B

Searching the graph in the unweighted shortest-path computation. The darkest-shaded vertices have already been completely processed, the lightest-shaded vertices have not yet been used as *v*, and the medium-shaded vertex is the current vertex, *v*. The stages proceed left to right, top to bottom, as numbered.



© 2002 Addison Wesley

Figure 14.16

The graph, after the starting vertex has been marked as reachable in zero edges

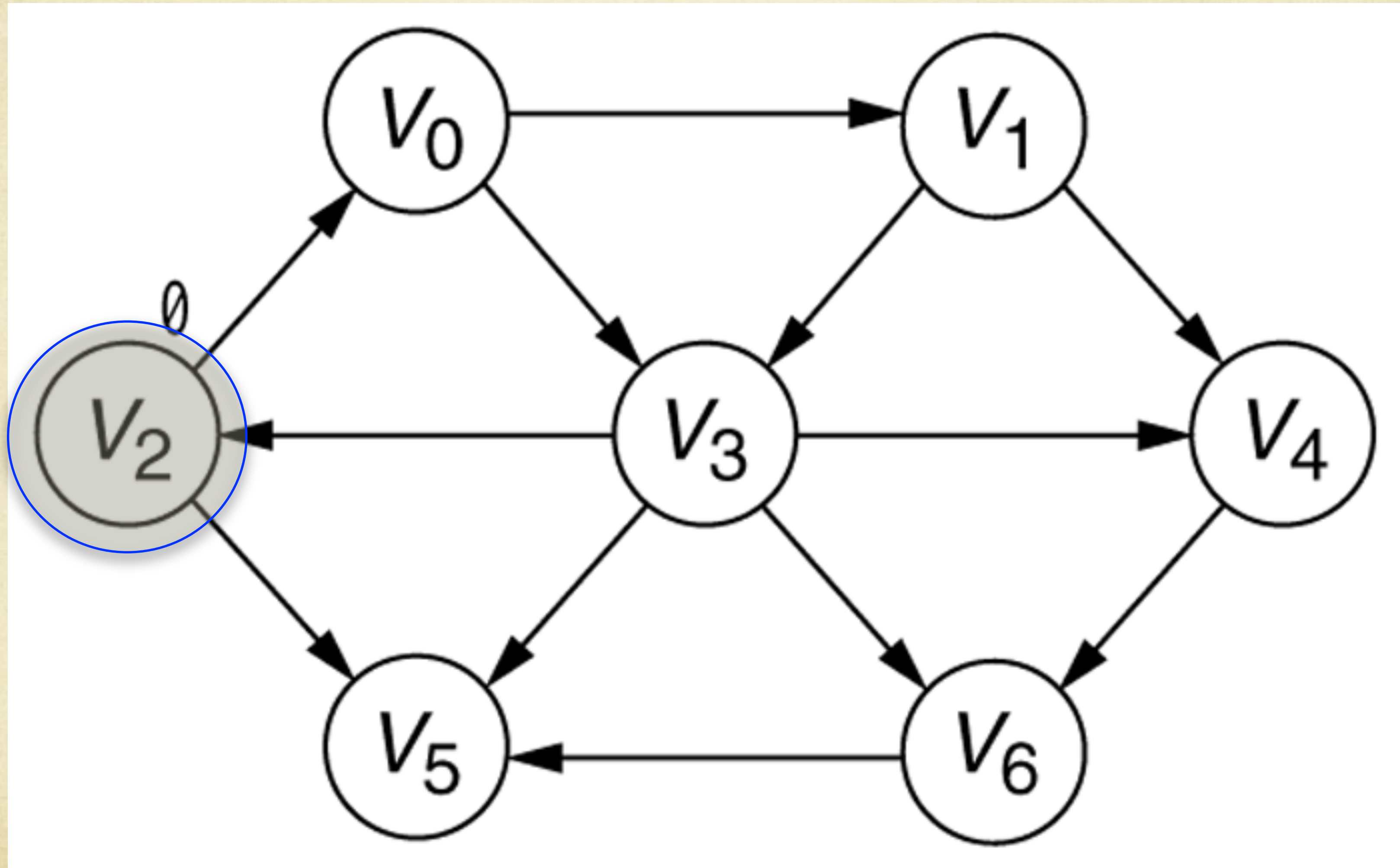
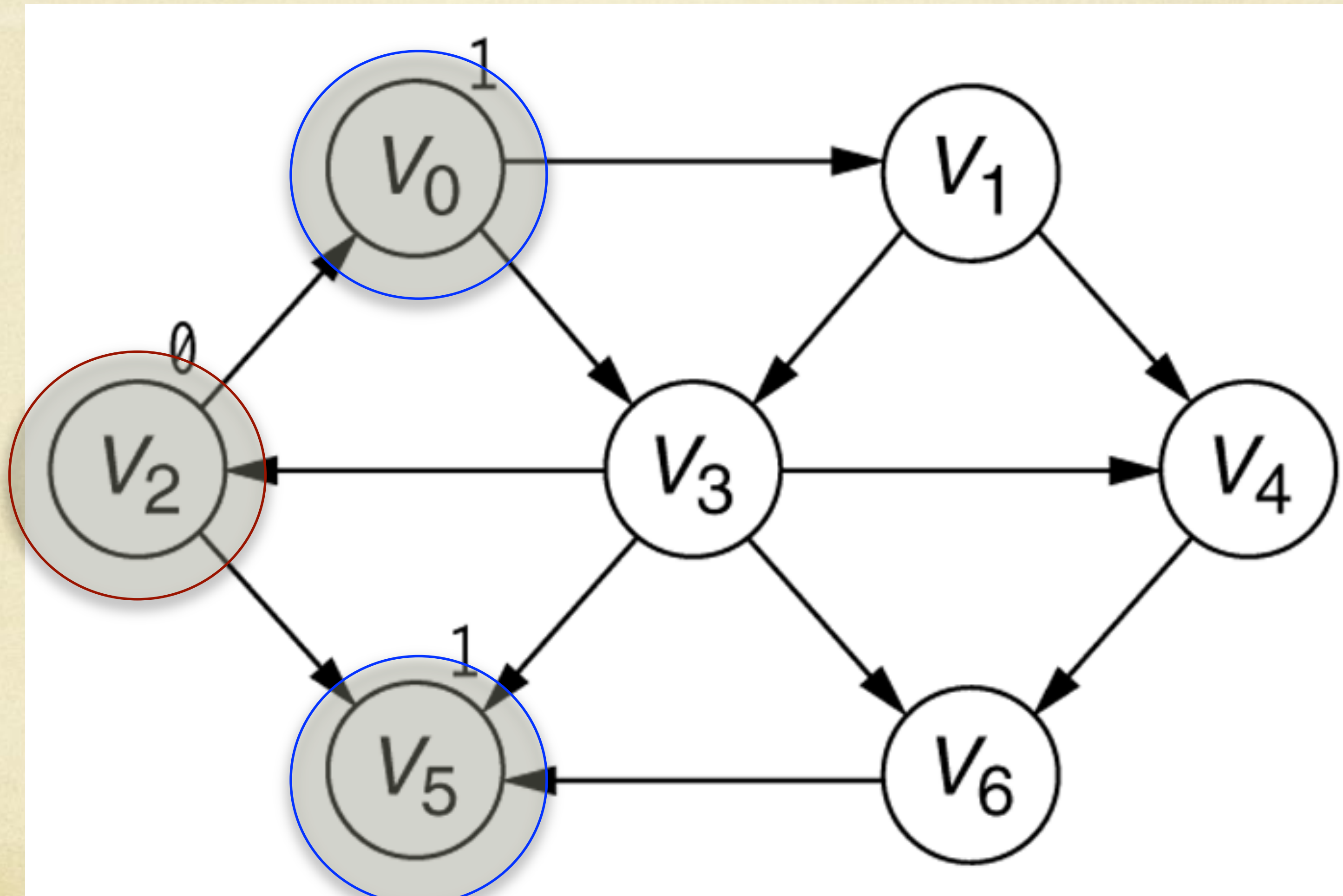


Figure 14.17
The graph, after all the vertices whose path length from the starting vertex is 1 have been found



© 2002 Addison Wesley

Figure 14.18
The graph, after all the vertices whose shortest path from the starting vertex is 2 have been found

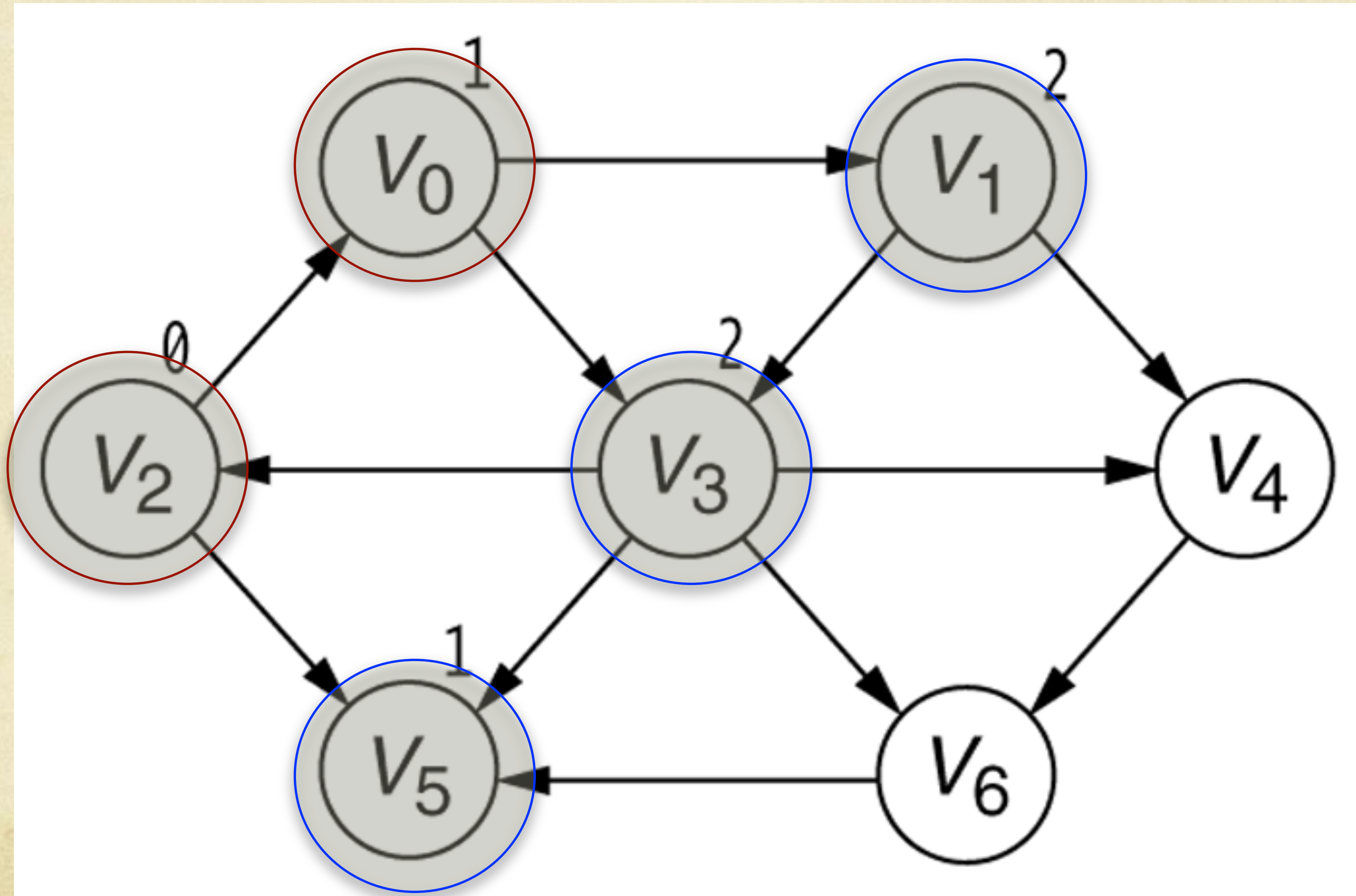
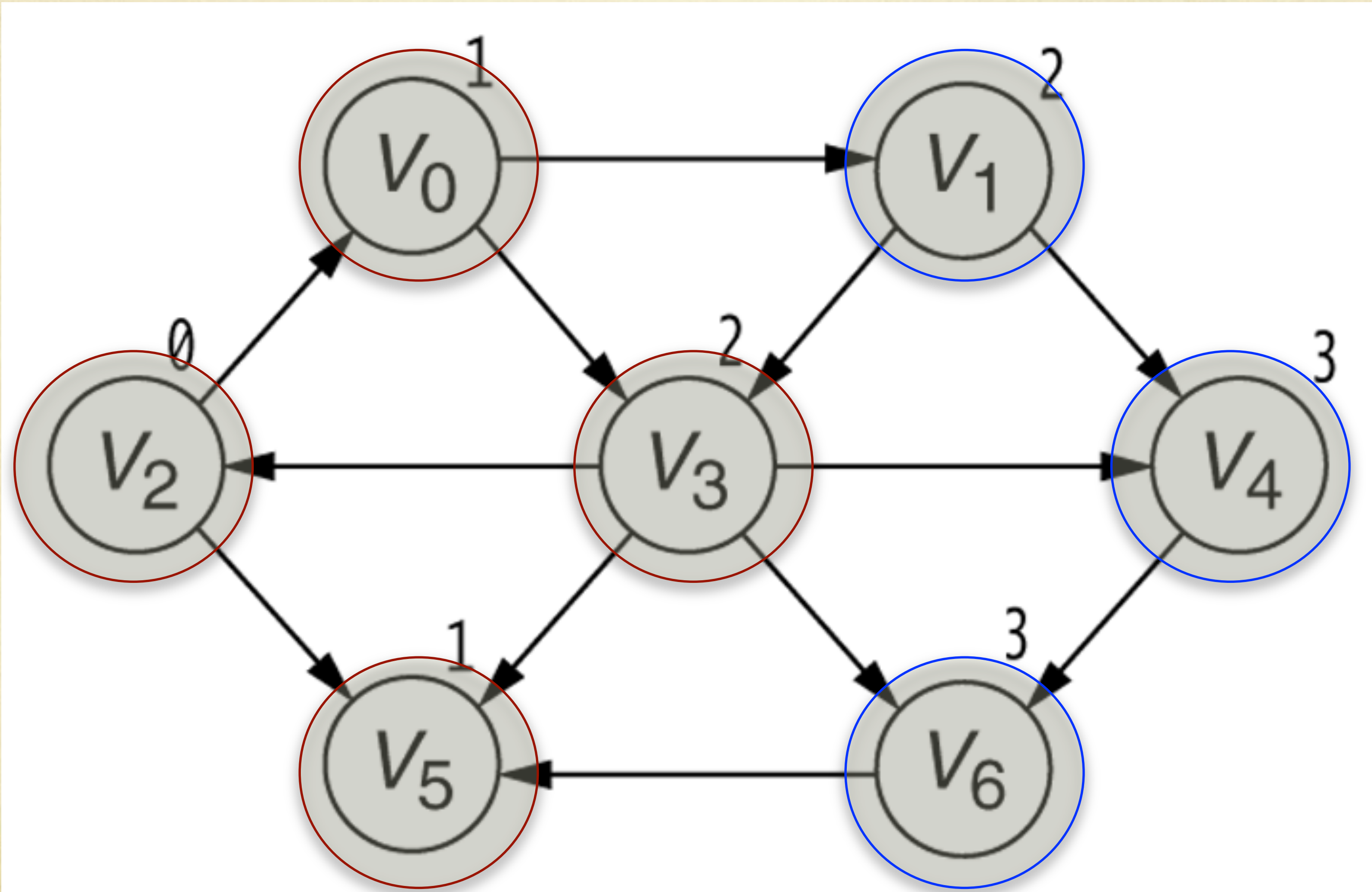


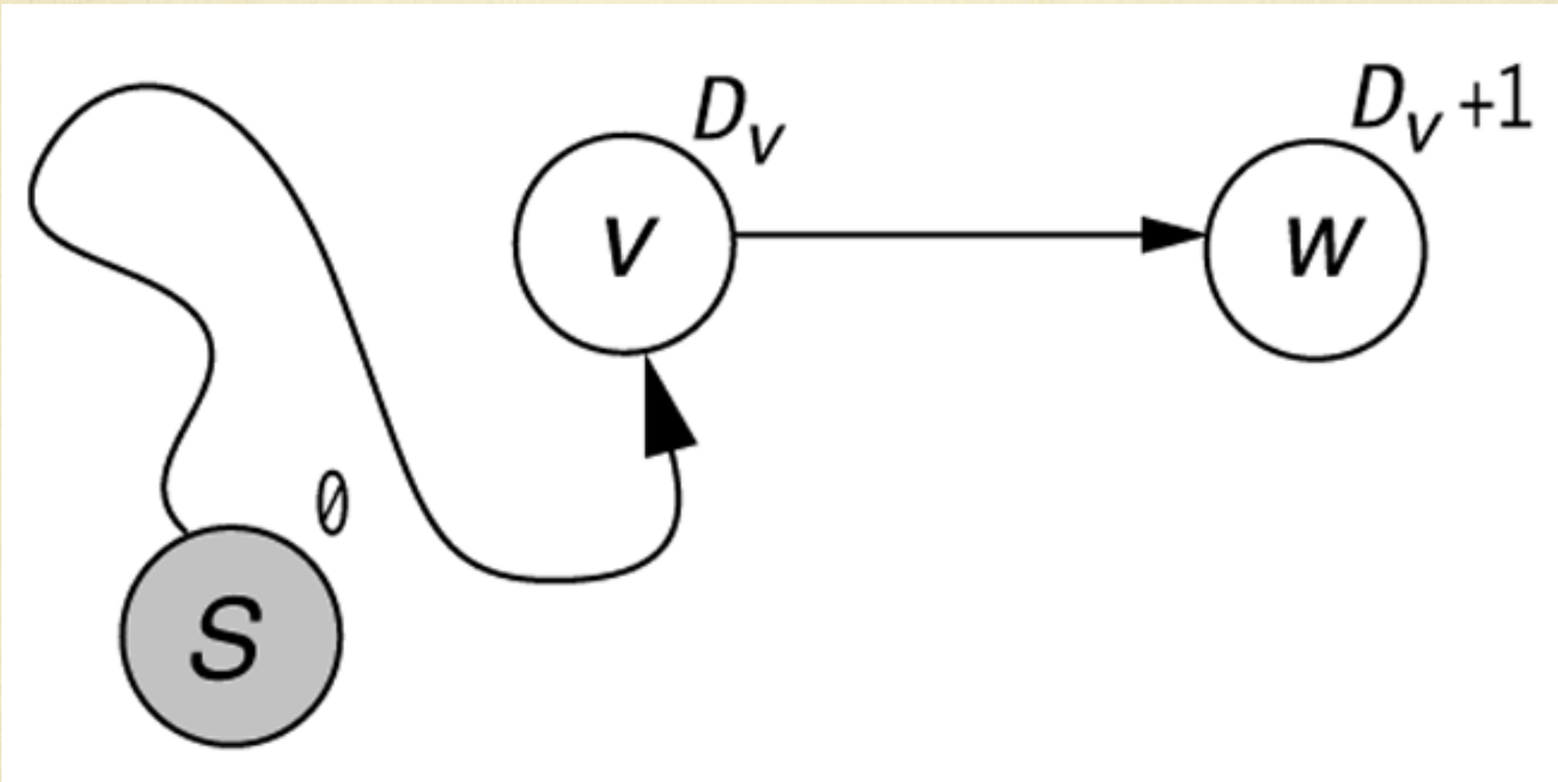
Figure 14.19
The final shortest paths



 © 2002 Addison Wesley

Figure 14.20

If *w* is adjacent to *v* and there is a path to *v*, there also is a path to *w*



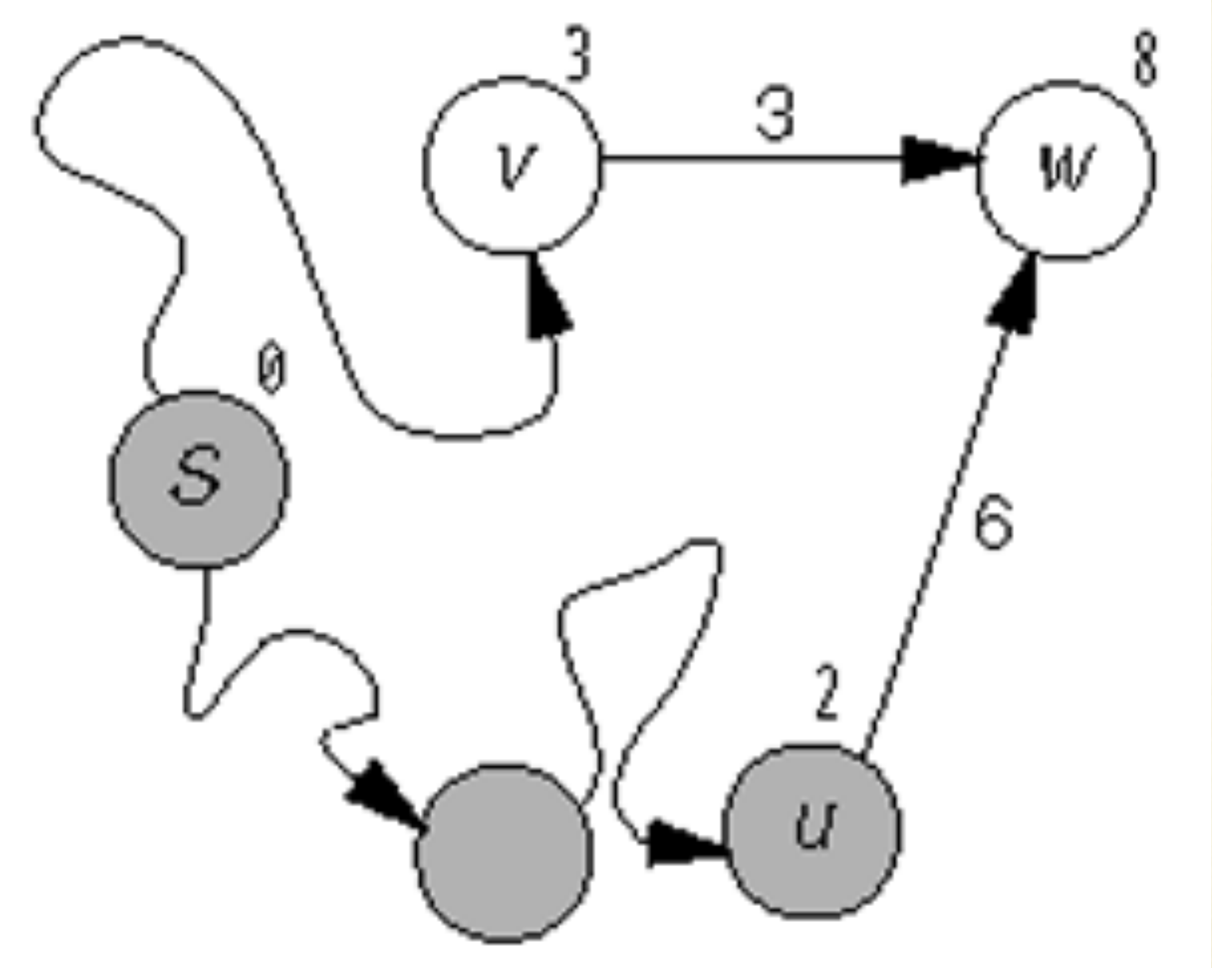
 © 2002 Addison Wesley

Unweighted SP algorithm

```
Void BFS (Vertex s) { // same as unweighted SP
    Queue <Vertex> Q = new Queue <>;
    for each Vertex v except s { v.dist = INFNT;}
    s.dist = 0; s.prev = null;
    Q.enqueue(s);
    while ( !Q.isEmpty() ) {
        v = Q.dequeue();
        for each vertex w adjacent to v
            if (w.dist == INFNT) {
                w.dist = v.dist + 1;
                w.prev = v;
                Q.enqueue(w);
            }
    }
}
```

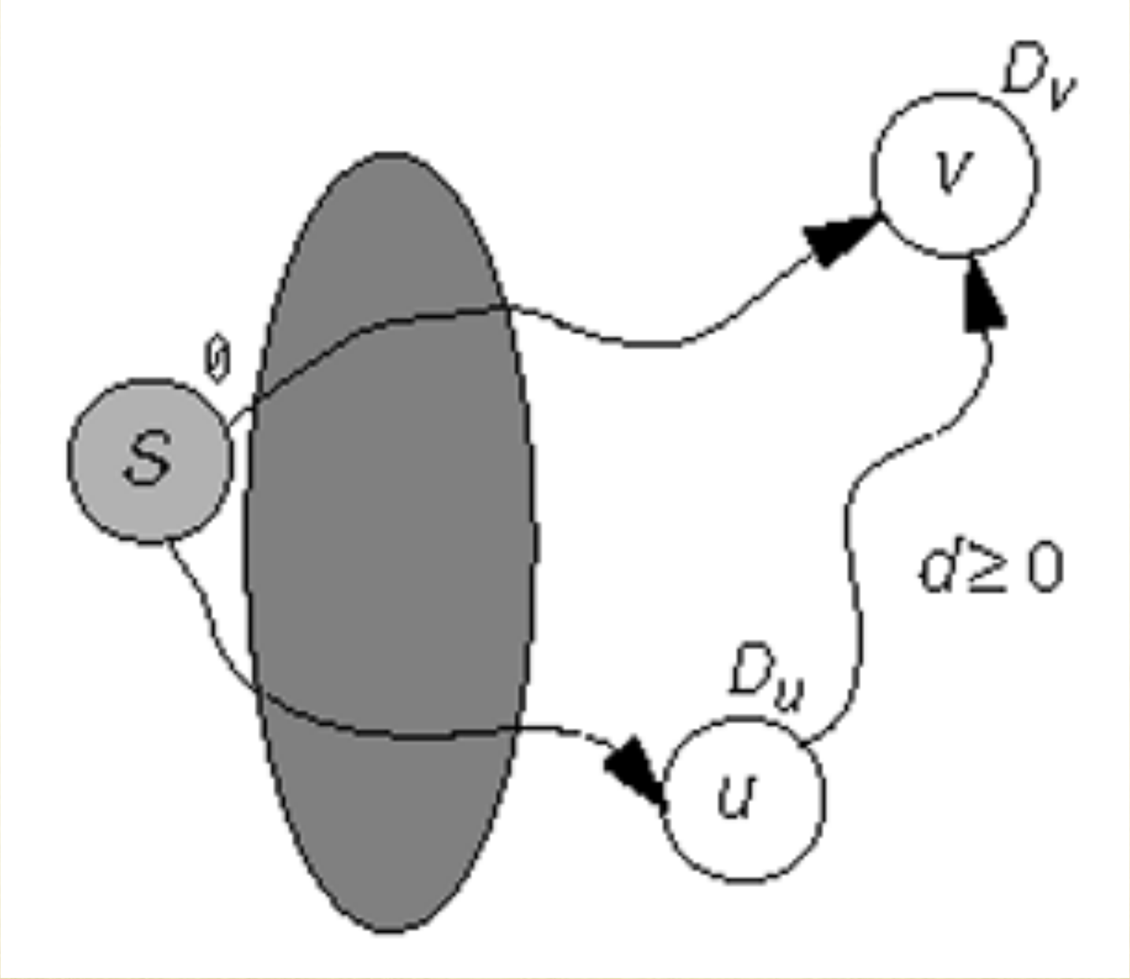
Time Complexity = O(n + m)

Figure 14.23
The eyeball is at *v* and *w* is adjacent, so D_w should be lowered to 6.



 © 2002 Addison Wesley

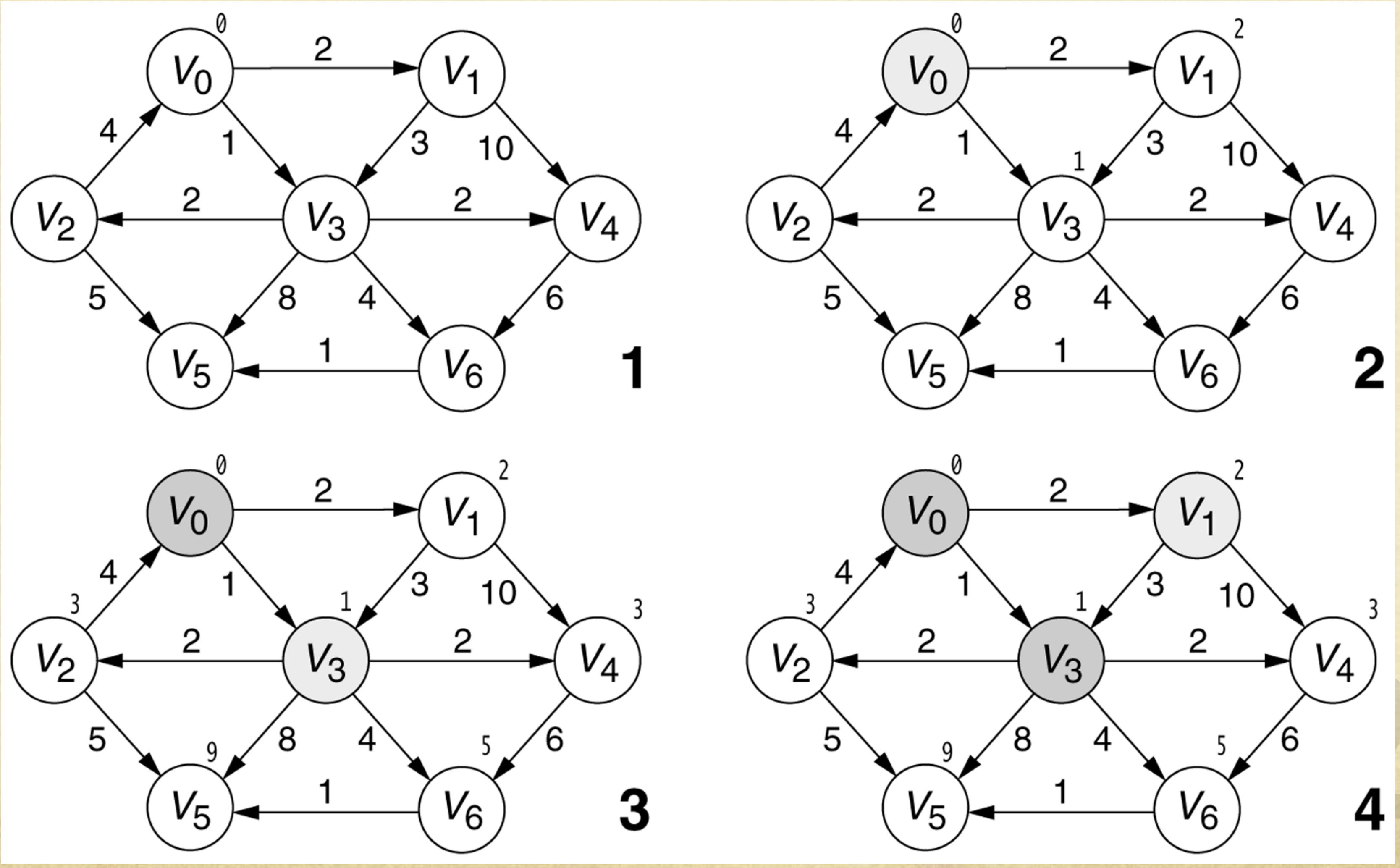
Figure 14.24
If D_v is minimal among all unseen vertices and if all edge costs are nonnegative, D_v represents the shortest path.



© 2002 Addison Wesley

Figure 14.25A

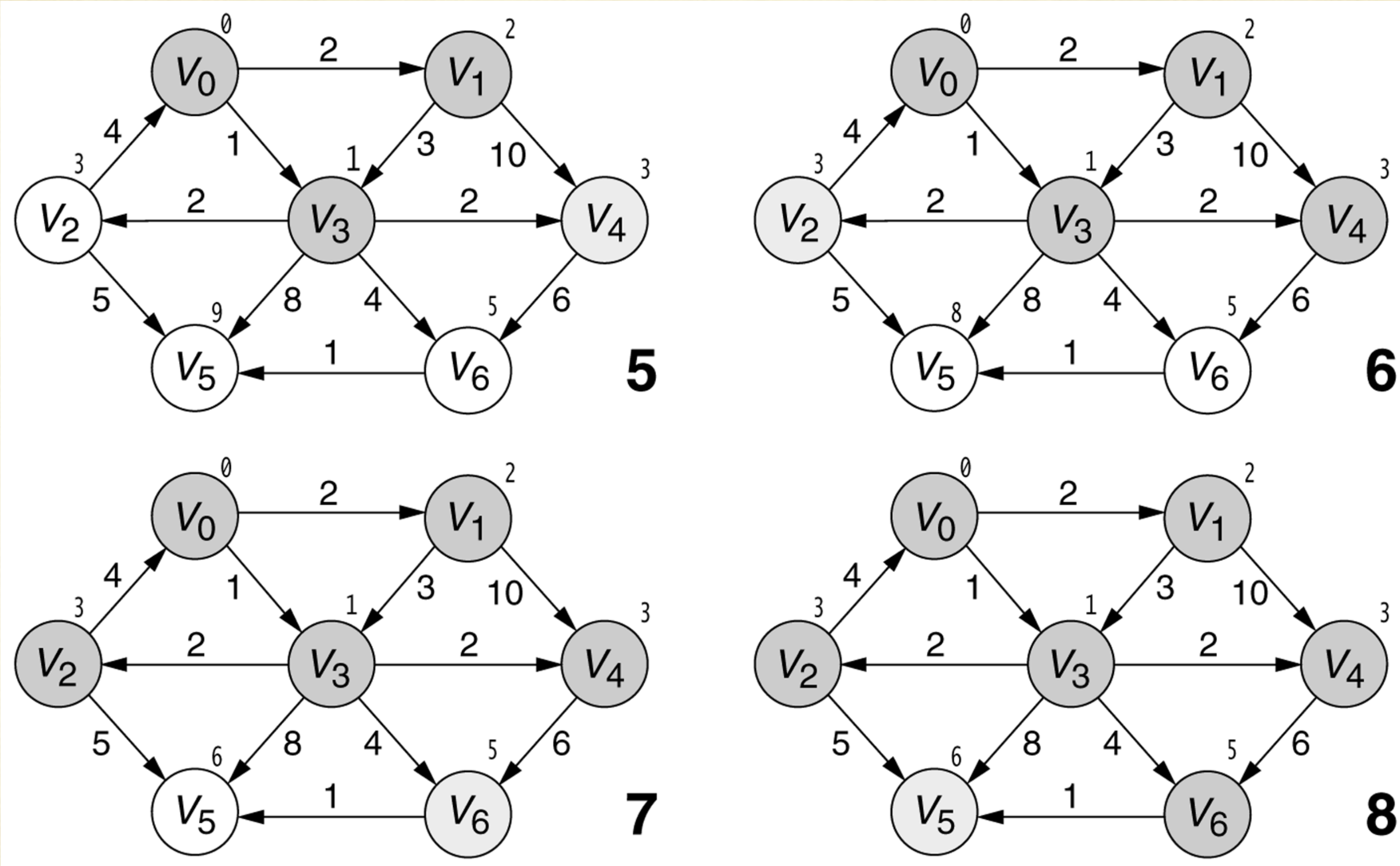
Stages of Dijkstra's algorithm. The conventions are the same as those in Figure 14.21 (*continued*).



03/11/04

 © 2002 Addison Wesley

Figure 14.25B
Stages of Dijkstra's algorithm. The conventions are the same as those in Figure 14.21.



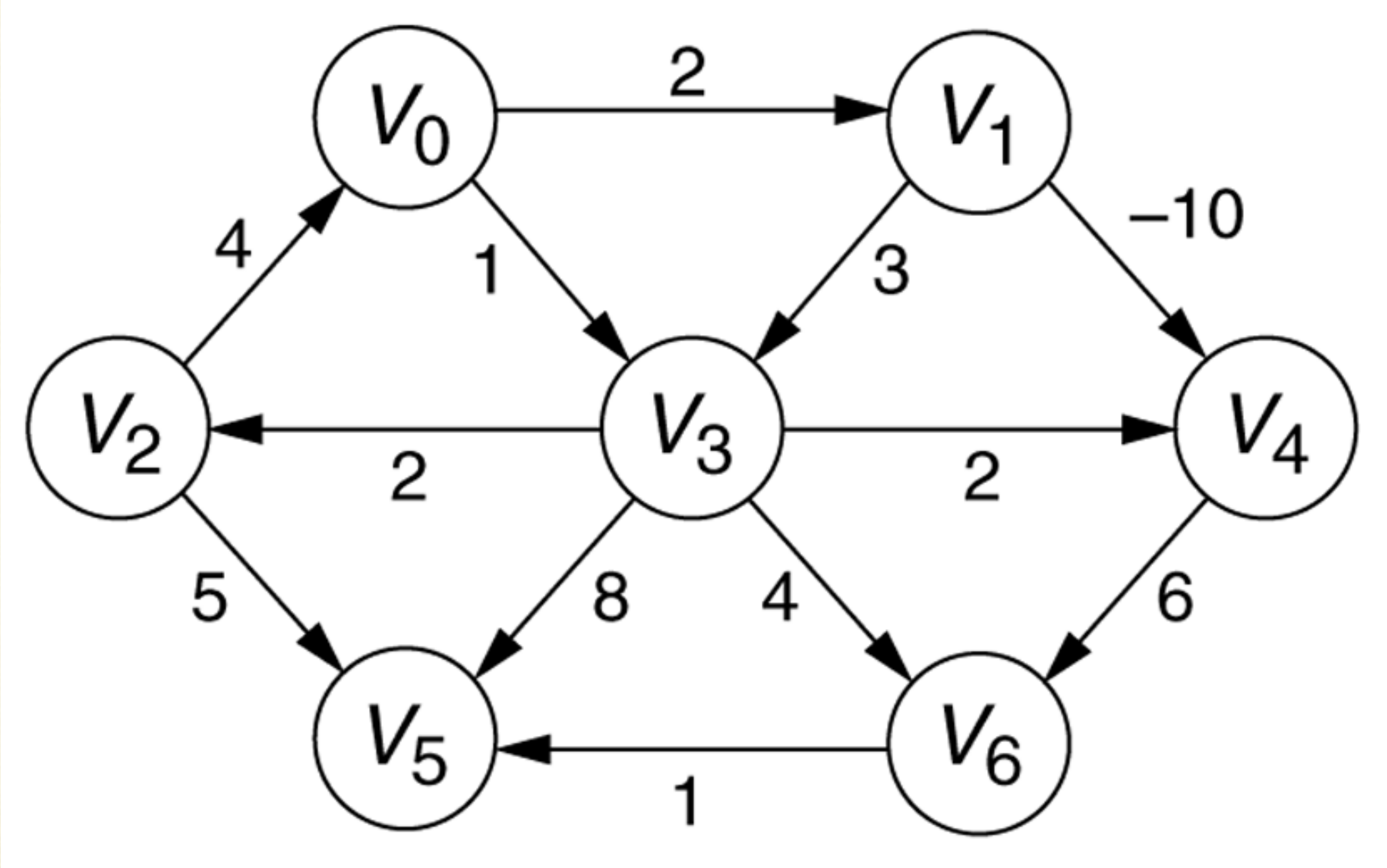
Dijkstra's SP algorithm

```
void Dijkstra (Vertex s) { // same as weighted SP
    PriorityQueue <Vertex> Q = new PriorityQueue <>;
    for each Vertex v except s { v.dist = INFNT; Q.insert(v); }
    s.dist = 0; s.prev= null;
    Q.insert(s);
    while ( !Q.isEmpty() ) {
        v = Q.deleteMin();
        for each vertex w adjacent to v
            if (w.dist > v.dist + weight of edge (v,w)) {
                w.dist = v.dist + weight of edge (v,w);
                w.prev= v;
                Q.updatePriority(w, v.dist + weight of edge (v,w));
            }
    }
}
```

Time Complexity = O(n log n + m + m log n) = O(m log n)

Figure 14.28

A graph with a negative-cost cycle



 © 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