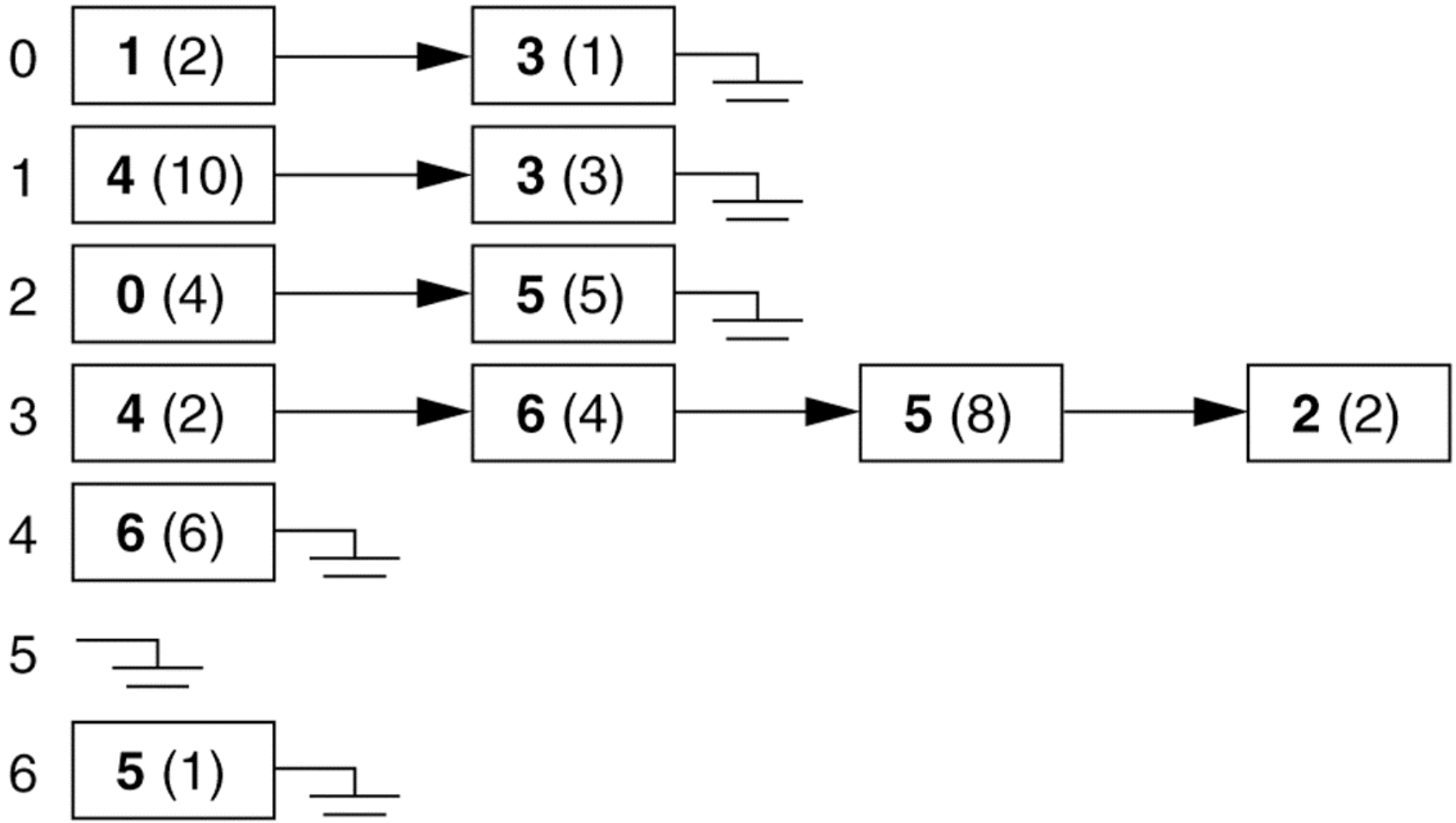
Figure 14.2

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.

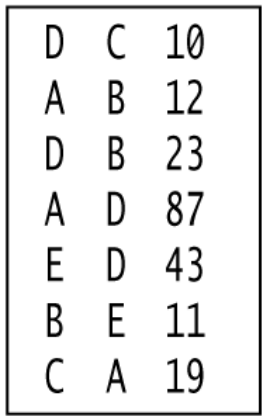


Shortest Paths

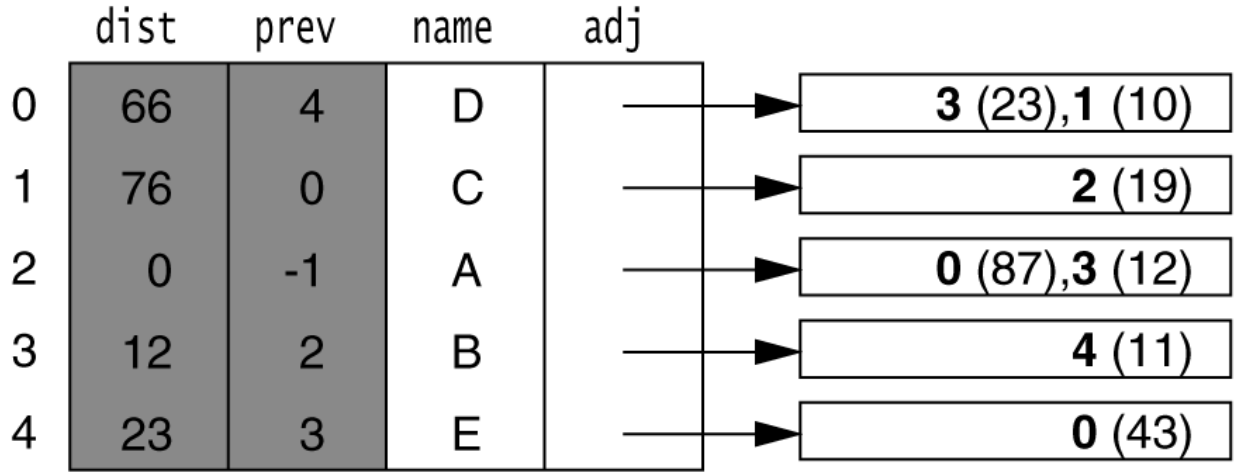
- Suppose we are interested in the shortest paths (and their lengths) from vertex "Miami" to all other vertices in the graph.
- We need to augment the data structure to store this information.

Figure 14.4

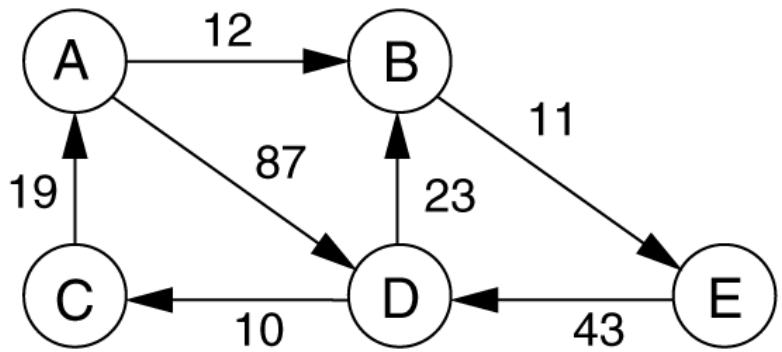
An abstract scenario of the data structures used in a shortest-path calculation, with an input graph taken from a file. The shortest weighted path from A to C is A to B to E to D to C (cost is 76).



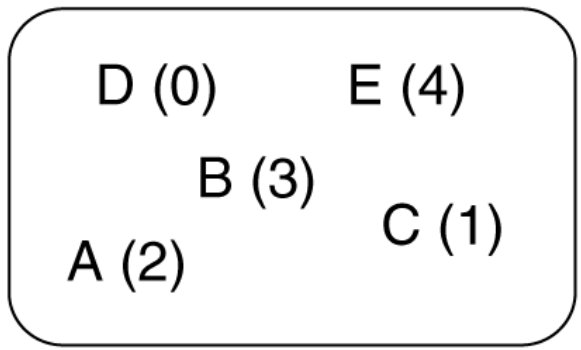
Input



Graph table



Visual representation of graph



Dictionary

Vertex Object

```
// Represents a vertex in the graph.
class Vertex
{   public String     name;   // Vertex name
    public List       adj;    // Adjacent vertices
    public double     dist;   // Cost
    public Vertex     prev;   // Previous vertex on shortest path
    public int        scratch;// Extra variable used in algorithm

    public Vertex( String nm )
      { name = nm; adj = new LinkedList( ); reset( ); }

    public void reset( )
      { dist = Graph.INFINITY; prev = null; pos = null; scratch = 0; }

    public PriorityQueue.Position pos;  // Used for dijkstra2
}
```

Edge Object

```
// Represents an edge in the graph.
class Edge
{
    public Vertex     dest;   // Second vertex in Edge
    public double     cost;   // Edge cost

    public Edge( Vertex d, double c )
    {
        dest = d;
        cost = c;
    }
}
```

Graph Object

```
// Graph class: evaluate shortest paths.
//
// CONSTRUCTION: with no parameters.
//
// ******************PUBLIC OPERATIONS*********
// void addEdge( String v, String w, double cvw )
//                          --> Add additional edge
// void printPath( String w )   --> Print path after alg is run
// void unweighted( String s )  --> Single-source unweighted
// void dijkstra( String s )    --> Single-source weighted
// void negative( String s )    --> Single-source negative
    weighted
// void acyclic( String s )     --> Single-source acyclic
```

getVertex Method

```
public class Graph {
   public static final double INFINITY = Double.MAX_VALUE;
   private Map vertexMap = new HashMap( ); // Maps String to Vertex

   /**  If vertexName is not present, add it to vertexMap.
    * *  In either case, return the Vertex.  */
   private Vertex getVertex( String vertexName )
   {
      Vertex v = (Vertex) vertexMap.get( vertexName );
      if( v == null )
      {
         v = new Vertex( vertexName );
         vertexMap.put( vertexName, v );
      }
      return v;
   }
```

addEdge Method

```
/**
 * Add a new edge to the graph.
 */
public void addEdge( String sourceName, String destName,
 double cost )
{
    Vertex v = getVertex( sourceName );
    Vertex w = getVertex( destName );
    v.adj.add( new Edge( w, cost ) );
}
```

printPath Method

```
/**
 * Recursive routine to print shortest path to dest
 * after running shortest path algorithm. The path
 * is known to exist.
 */
private void printPath( Vertex dest )
{
    if( dest.prev != null )
    {
        printPath( dest.prev );
        System.out.print( " to " );
    }
    System.out.print( dest.name );
}
```

clearAll Method

```
/**
 * Initializes the vertex output info prior to running
 * any shortest path algorithm.
 */
private void clearAll( )
{
    for( Iterator itr = vertexMap.values( ).iterator( );
 itr.hasNext( ); )
        ( (Vertex)itr.next( ) ).reset( );
}
```

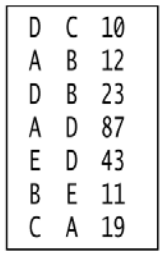
Unweighted Shortest Path Problem

```
/**
   * A main routine that:
   * 1. Reads a file containing edges (supplied as a command-
   line parameter);
   * 2. Forms the graph;
   * 3. Repeatedly prompts for two vertices and
   *    runs the shortest path algorithm.
   * The data file is a sequence of lines of the format
   *    source destination.
   */
```

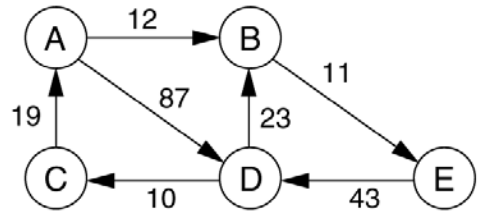
Figure 14.5

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).

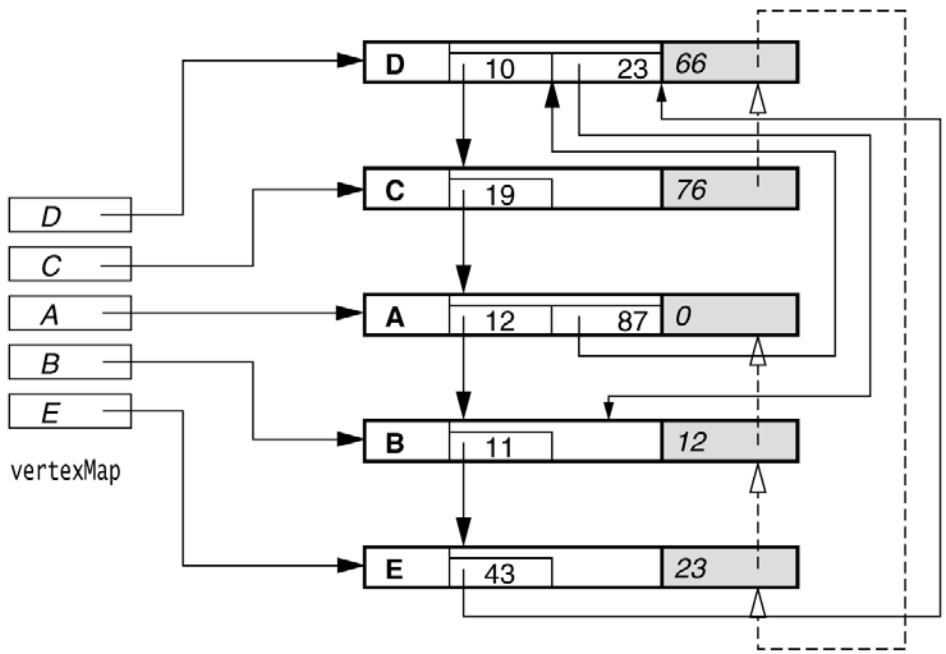
Legend: Dark-bordered boxes are Vertex objects. The unshaded portion in each box contains the name and adjacency list and does not change when shortest-path computation is performed. Each adjacency list entry contains an Edge that stores a reference to another Vertex object and the edge cost. Shaded portion is dist and prev, filled in after shortest path computation runs.
Dark arrows emanate from vertexMap.
Light arrows are adjacency list entries.
Dashed arrows are the prev data member that results from a shortest-path computation.



Input



Visual representation of graph



04/03/03

Figure 14.16

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

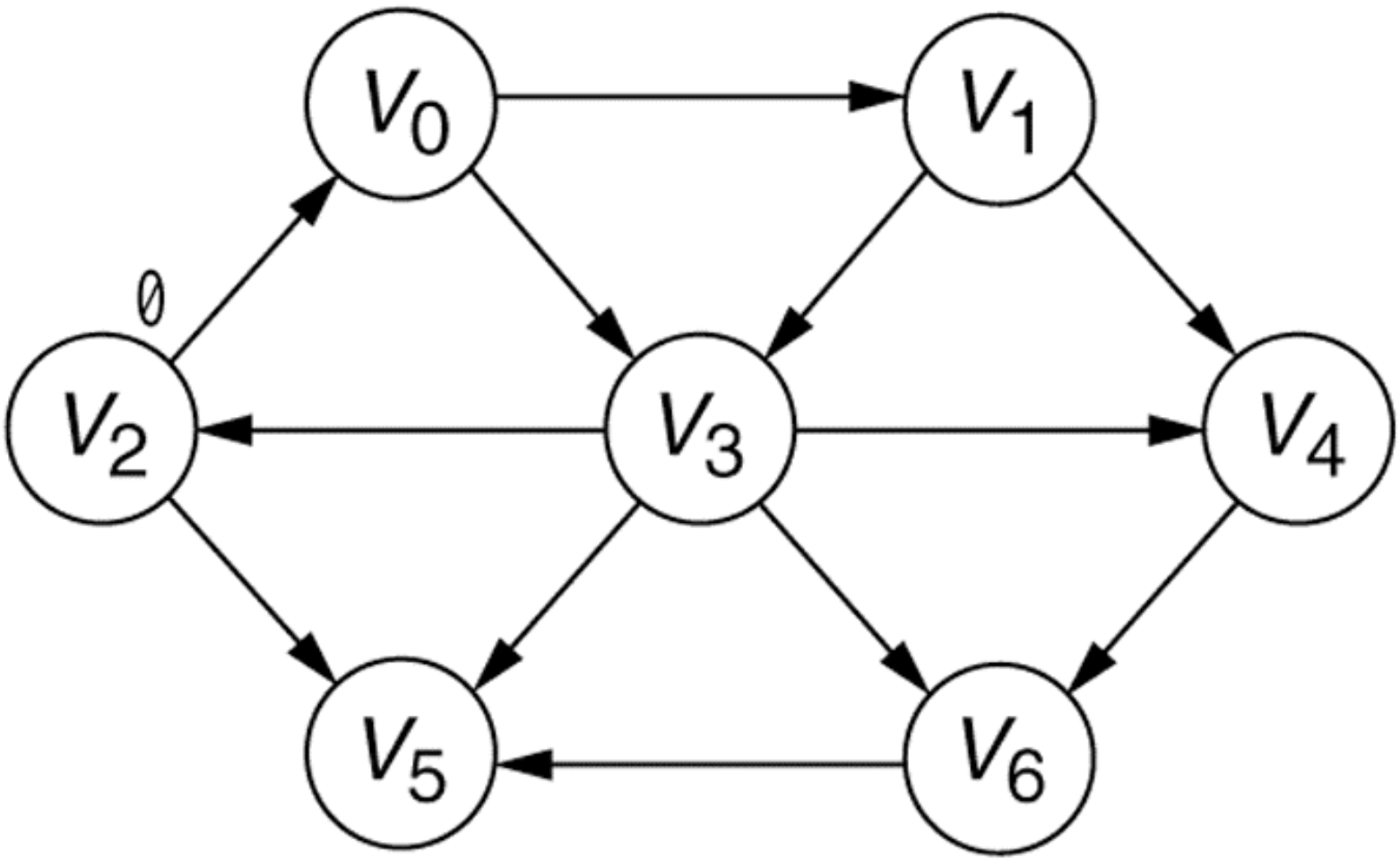


Figure 14.17

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

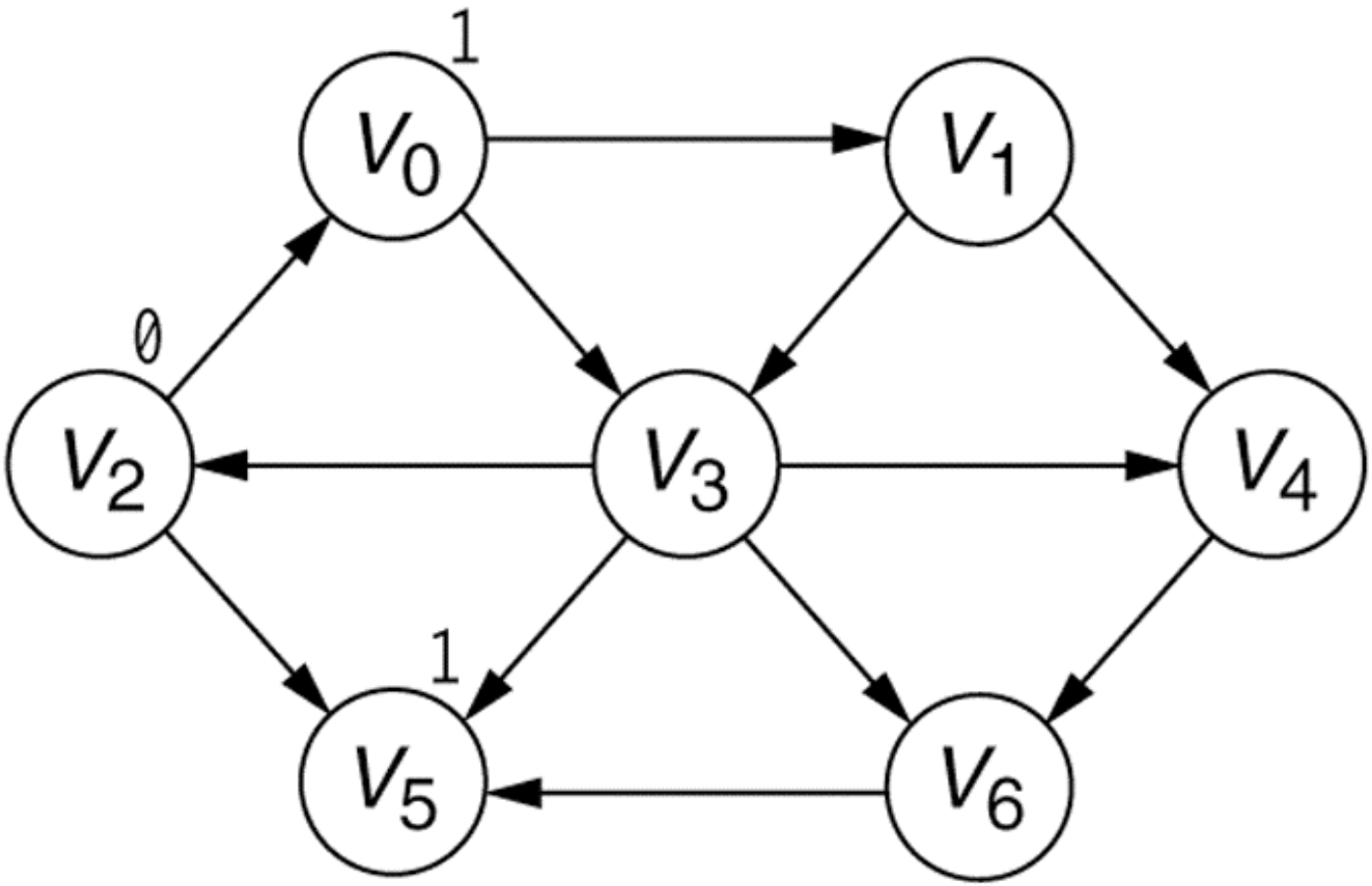


Figure 14.18

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

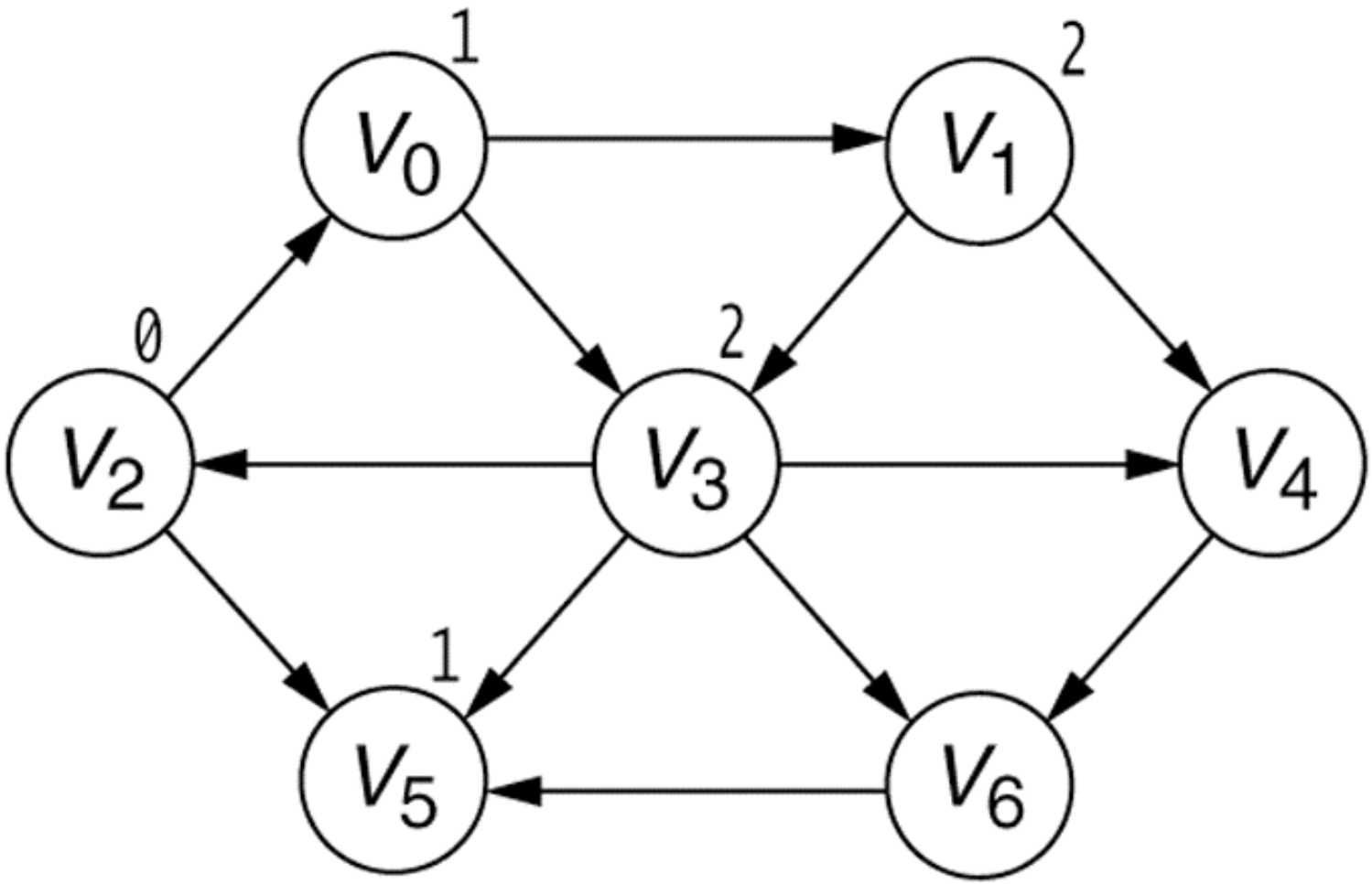


Figure 14.19
The final shortest paths

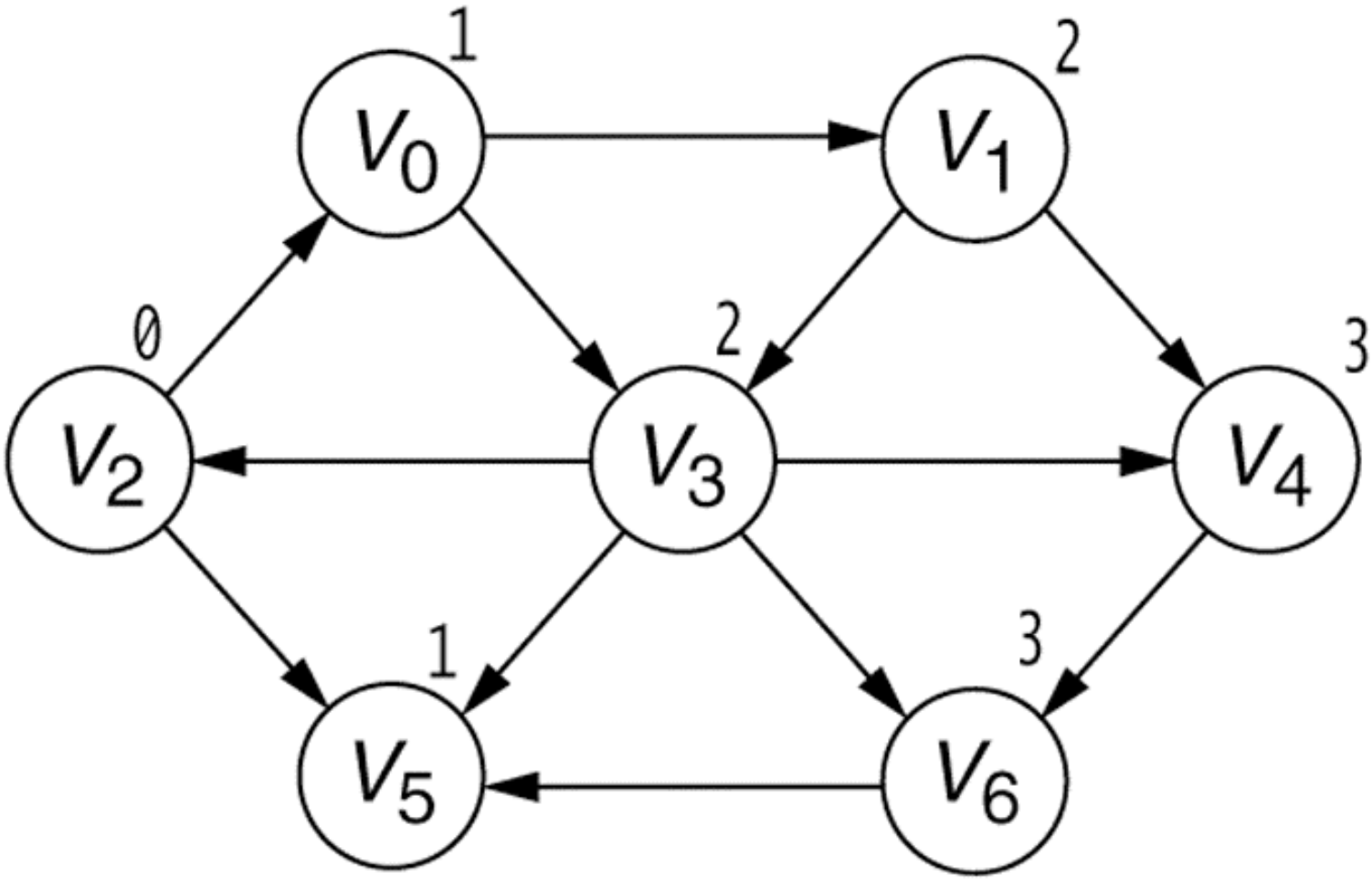


Figure 14.20

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

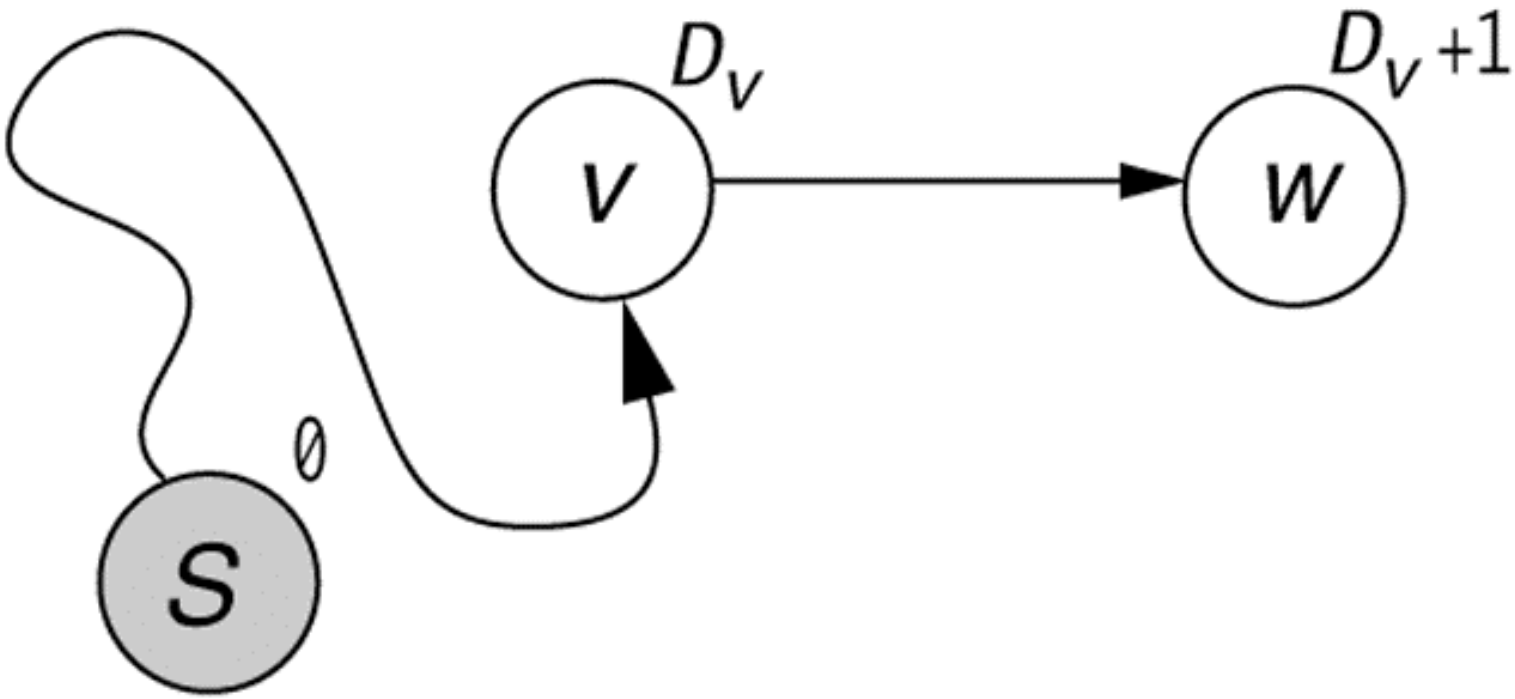


Figure 14.21A

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).*

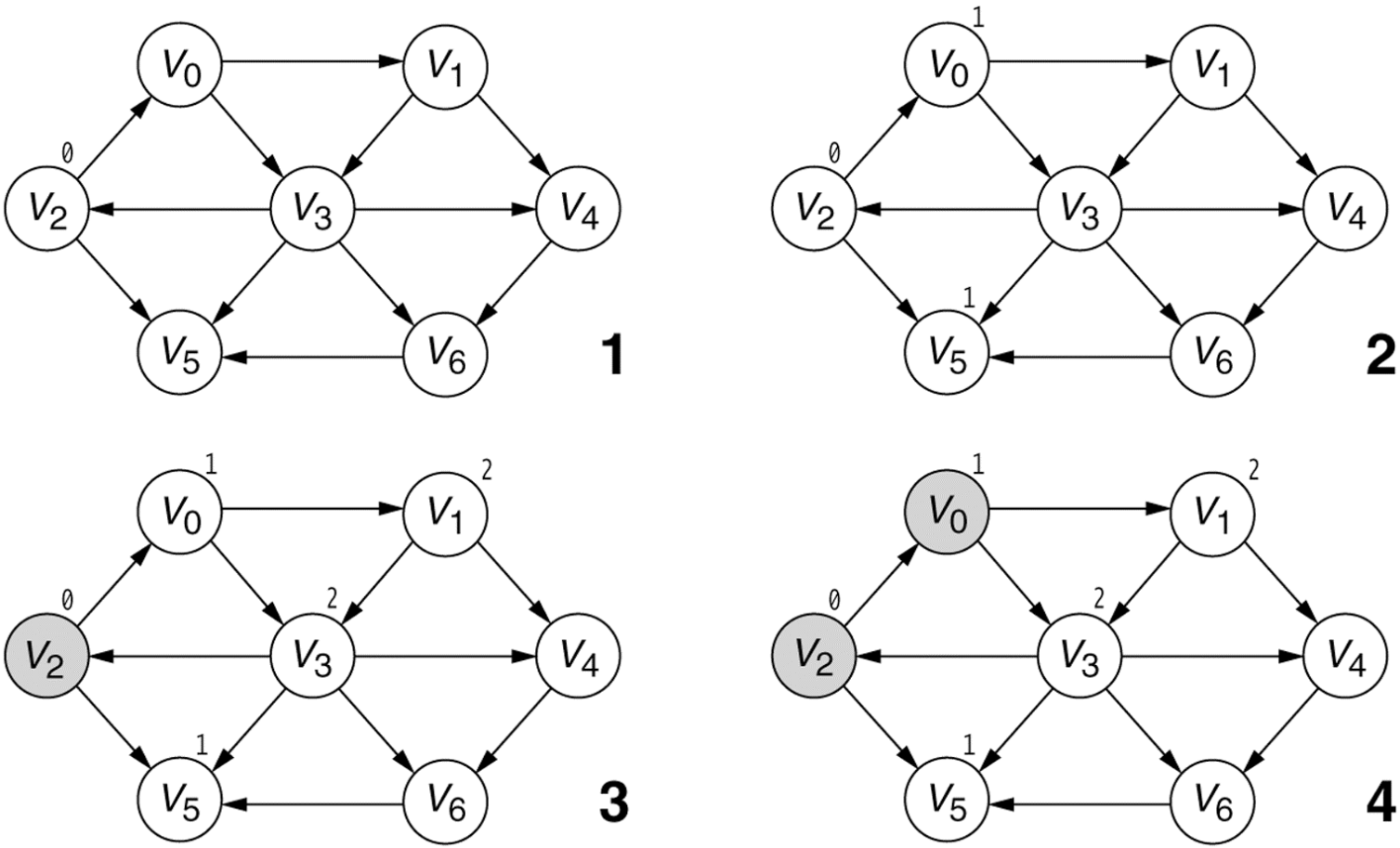
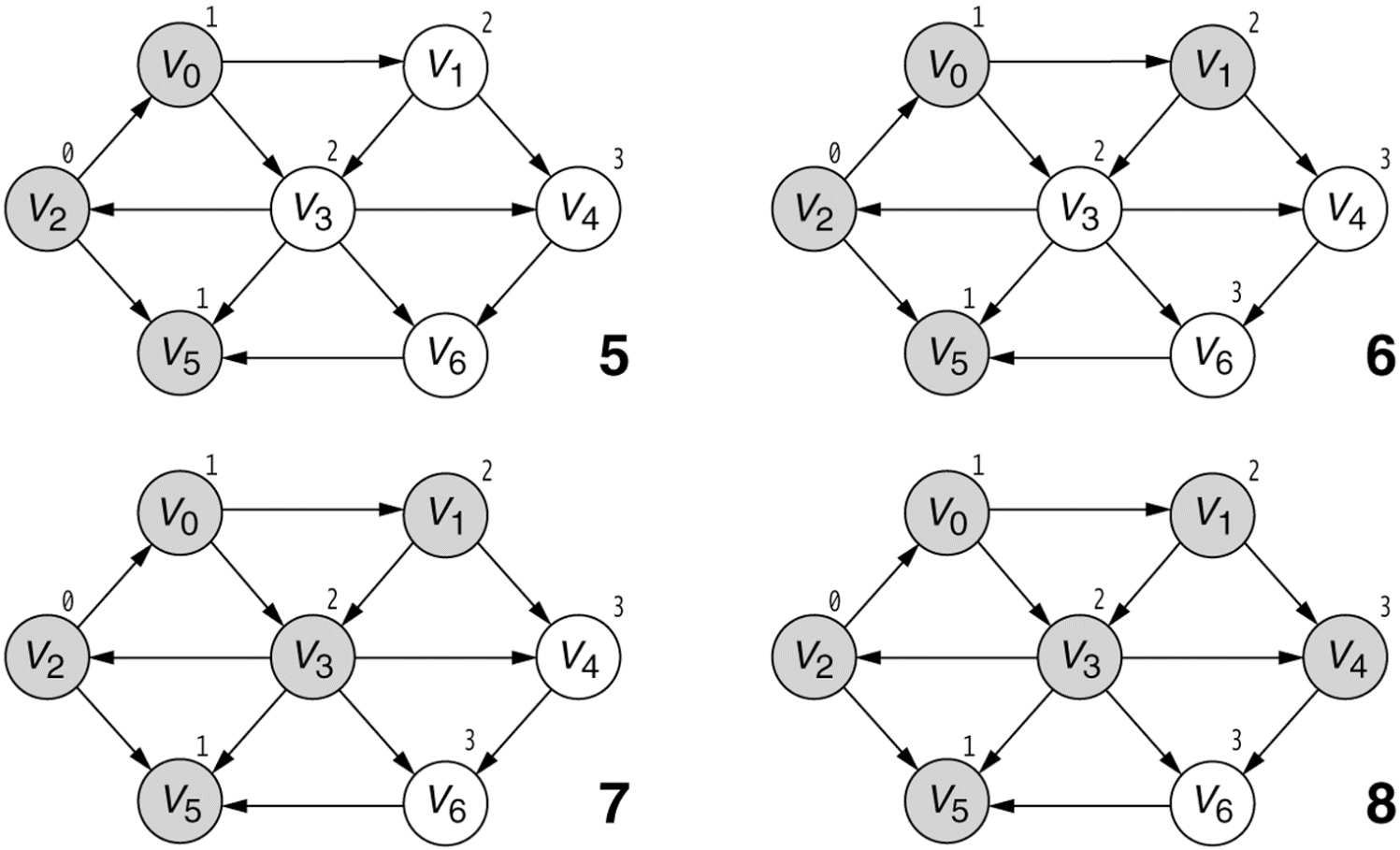


Figure 14.21B

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.




```
public static void main( String [ ] args )    {
    Graph g = new Graph( );
    BufferedReader graphFile=new BufferedReader(FileReader( args[0] ) );
     // Read the edges and insert
     String line;
     while( ( line = graphFile.readLine( ) ) != null )
     {
          StringTokenizer st = new StringTokenizer( line );
             if( st.countTokens( ) != 3 )  {  // some error message
              }
              String source  = st.nextToken( );
              String dest    = st.nextToken( );
              int    cost    = Integer.parseInt( st.nextToken( ) );
              g.addEdge( source, dest, cost );
     }
    // Read the queries
     BufferedReader in = new BufferedReader( new InputStreamReader(
                                 System.in ) );
      while( processRequest( in, g ) )  ; // while loop body is empty
}
```

processRequest Method

```
public static boolean processRequest( BufferedReader in,
   Graph g )
  {
      String startName = null, destName = null, alg = null;
      System.out.print( "Enter start node:" );
      if( ( startName = in.readLine( ) ) == null )   return false;
      System.out.print( "Enter destination node:" );
      if( ( destName = in.readLine( ) ) == null )    return false;

      g.unweighted( startName );  // changes with algorithm
      g.printPath( destName );
      return true;
  }
```

SP – unweighted graphs

```
public void unweighted( String startName ) {
    clearAll( );
    Vertex start = (Vertex) vertexMap.get( startName );
    LinkedList q = new LinkedList( );
    q.addLast( start ); start.dist = 0;
    while( !q.isEmpty( ) )        {
        Vertex v = (Vertex) q.removeFirst( );
        for( Iterator itr = v.adj.iterator( ); itr.hasNext( ); )    {
            Edge e = (Edge) itr.next( );
            Vertex w = e.dest;
            if( w.dist == INFINITY ) {
                w.dist = v.dist + 1;
                w.prev = v;
                q.addLast( w );
            }
        }
    }
}
```

Figure 14.23

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

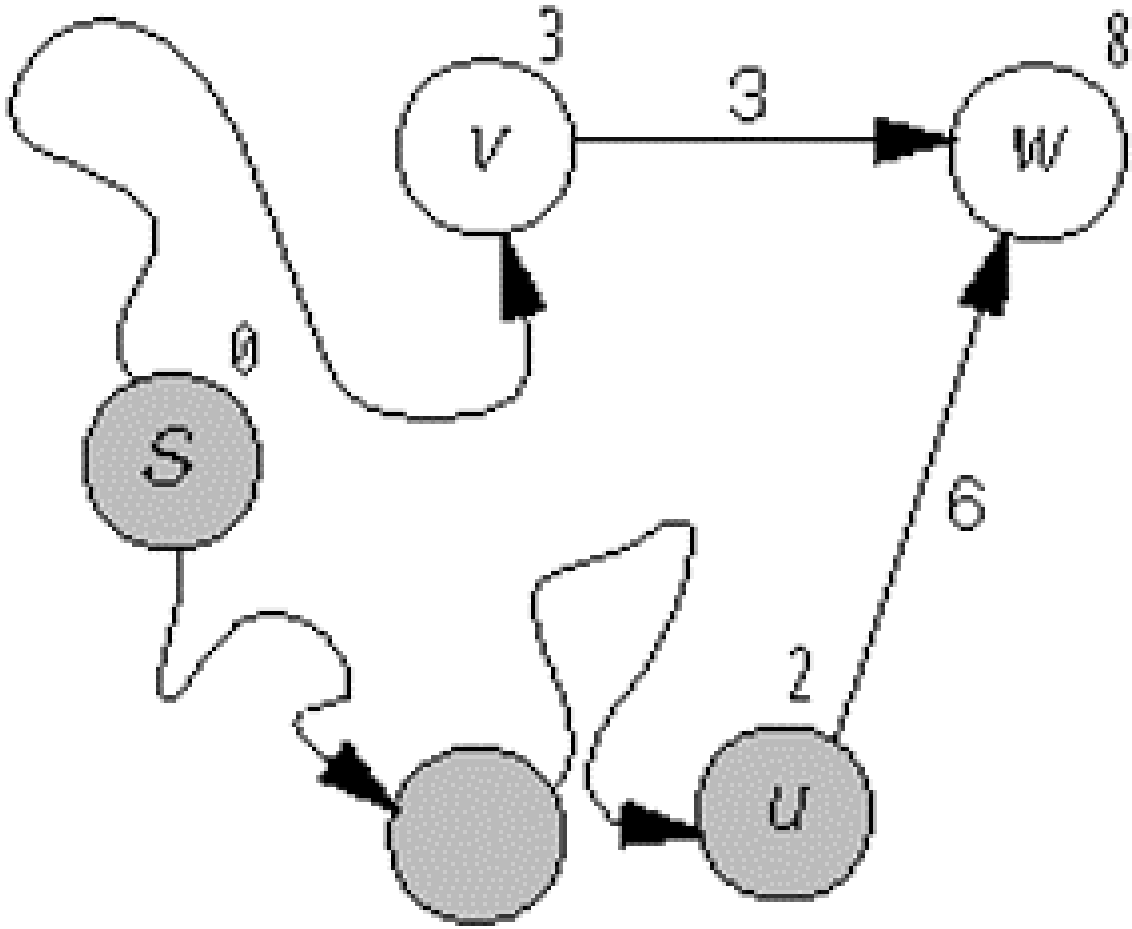


Figure 14.24

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

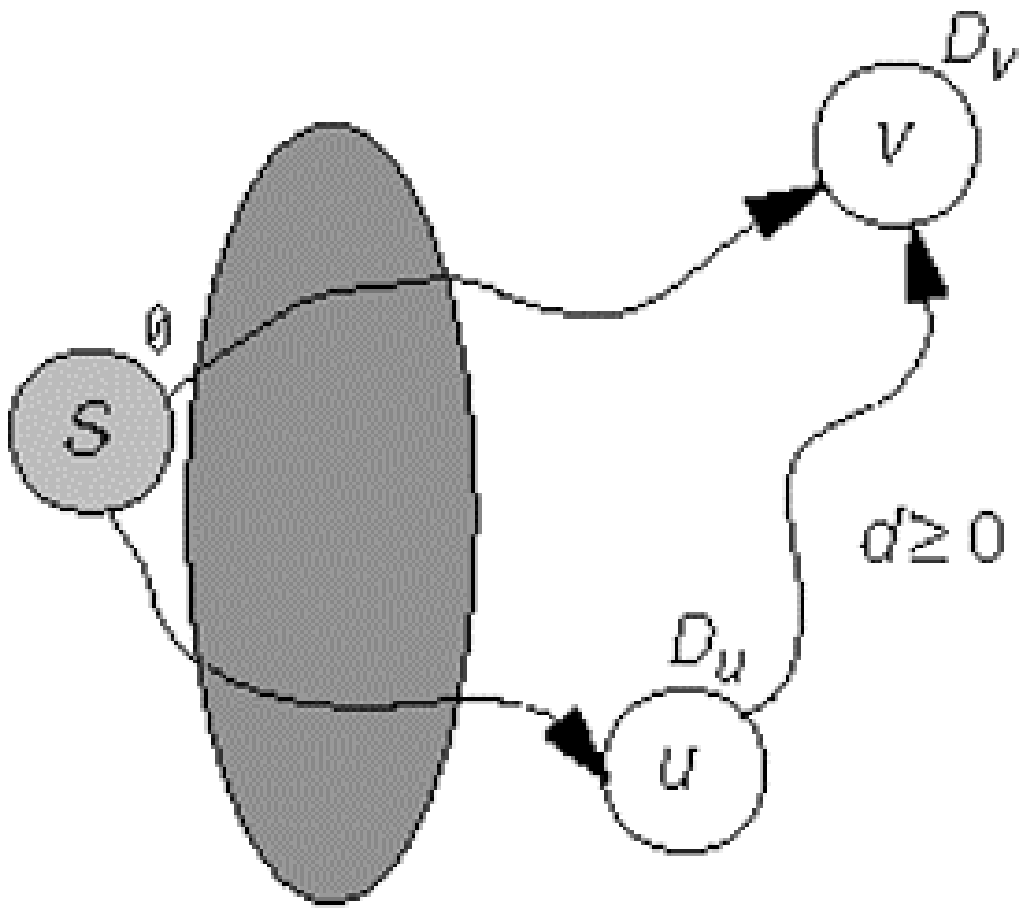


Figure 14.25A

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

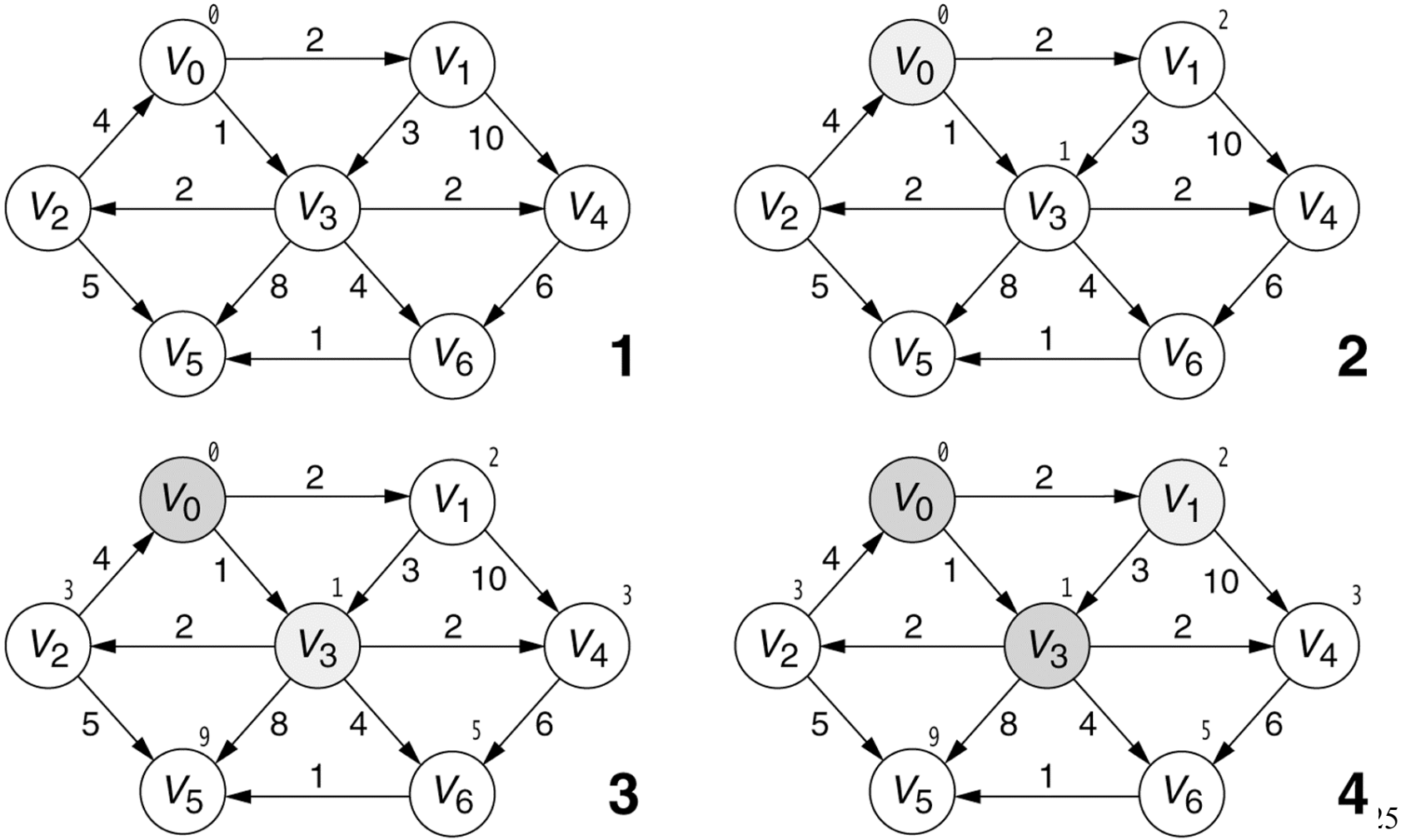
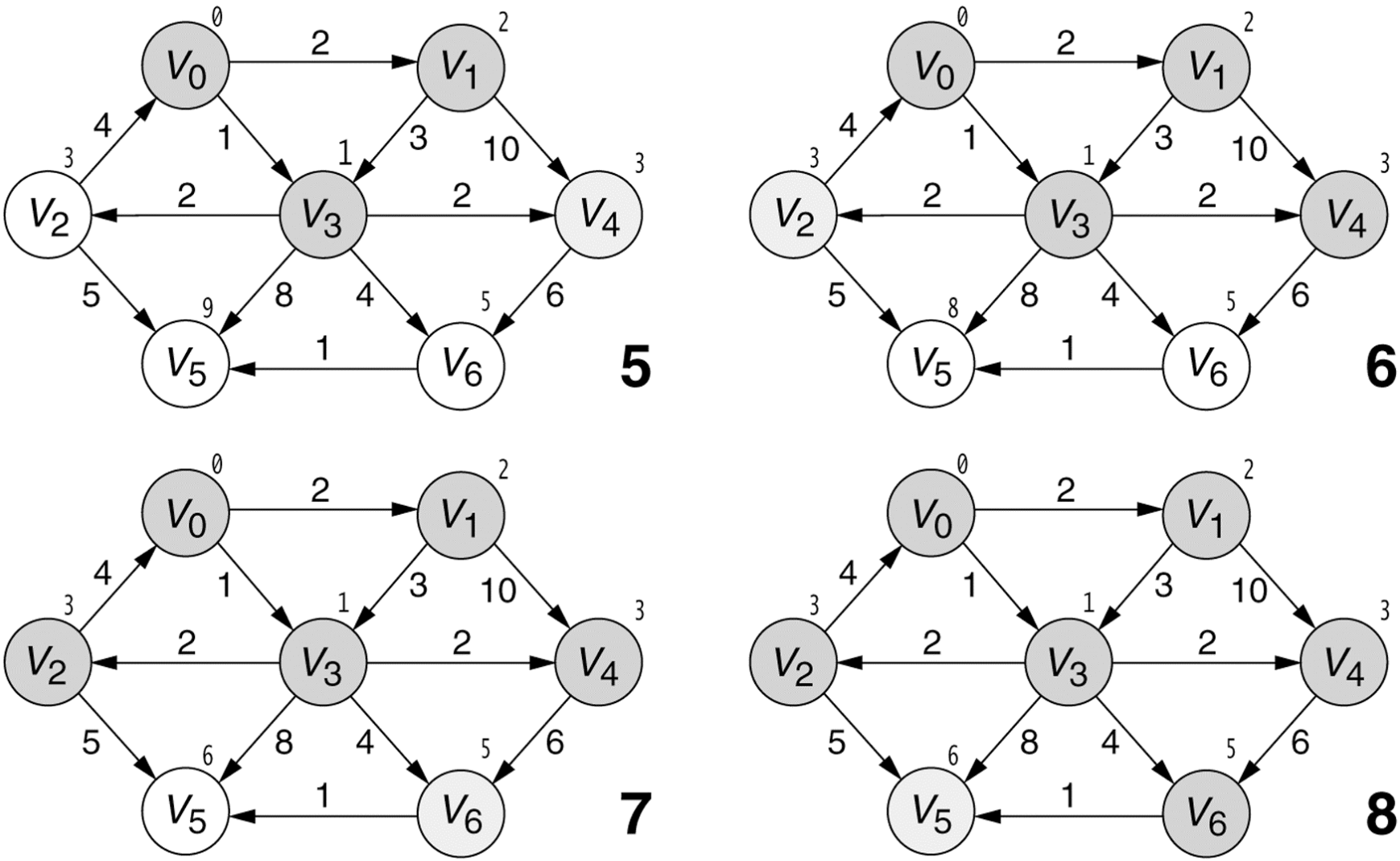


Figure 14.25B

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



Class Path

```
// Represents an entry in the priority queue for Dijkstra's algorithm.
class Path implements Comparable
{
    public Vertex     dest;   // w
    public double     cost;   // d(w)
    public Path( Vertex d, double c )
    {
        dest = d;
        cost = c;
    }
    public int compareTo( Object rhs )
    {
        double otherCost = ((Path)rhs).cost;
        return cost < otherCost ? -1 : cost > otherCost ? 1 : 0;
    }
}
```

```
public void dijkstra( String startName )  {
      PriorityQueue pq = new BinaryHeap( );
      Vertex start = (Vertex) vertexMap.get( startName );
      clearAll( );
      pq.insert( new Path( start, 0 ) ); start.dist = 0;
      int nodesSeen = 0;
      while( !pq.isEmpty( ) && nodesSeen < vertexMap.size( ) )   {
         Path vrec = (Path) pq.deleteMin( );
         Vertex v = vrec.dest;
         if( v.scratch != 0 ) continue; // already processed v
         v.scratch = 1;  nodesSeen++;
         for( Iterator itr = v.adj.iterator( ); itr.hasNext( ); )   {
            Edge e = (Edge) itr.next( );
            Vertex w = e.dest;
            double cvw = e.cost;
            if( w.dist > v.dist + cvw )   {
               w.dist = v.dist +cvw;  w.prev = v;
               pq.insert( new Path( w, w.dist ) );
            }
         }
      }
   }
```

Figure 14.30A

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

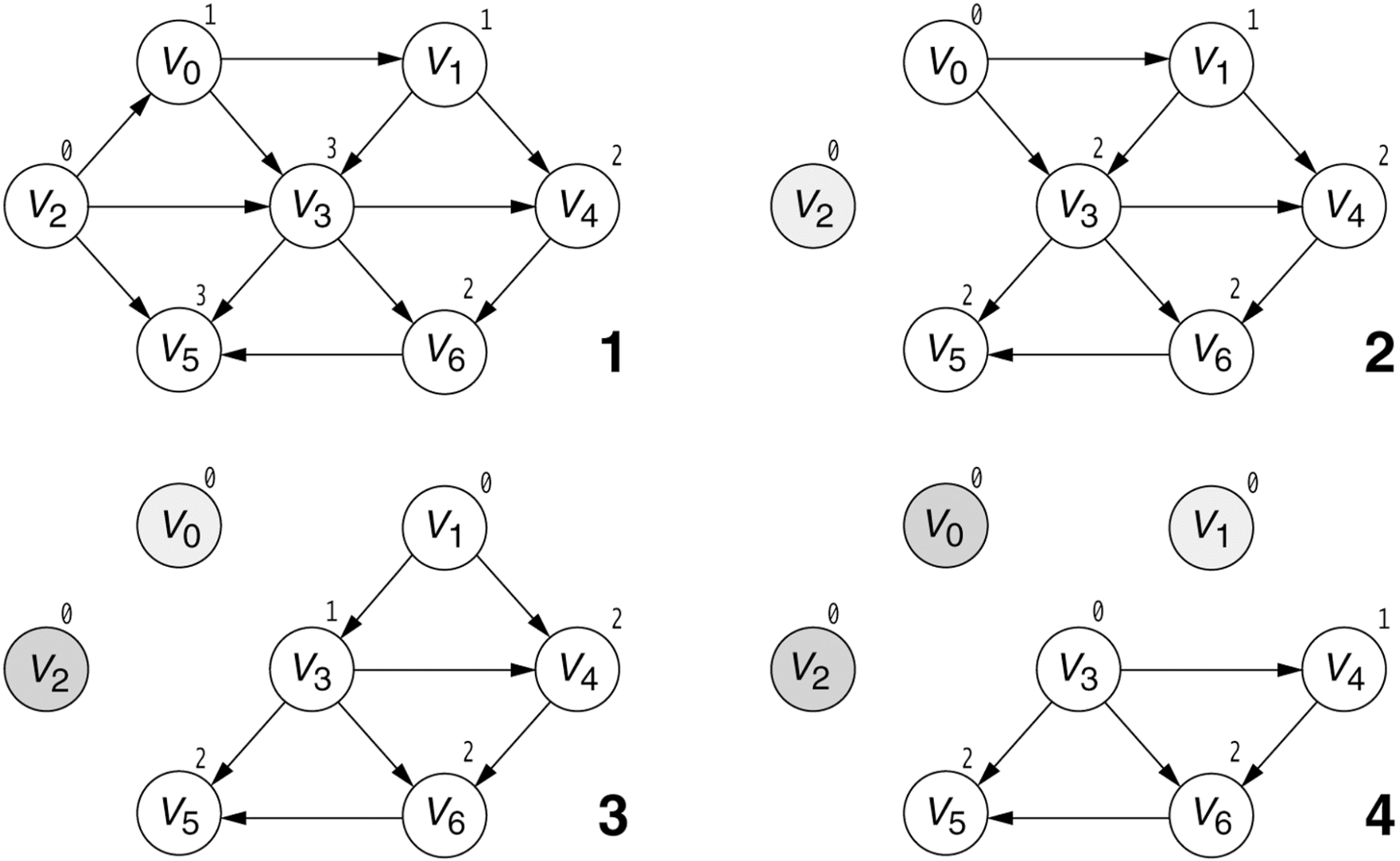


Figure 14.30B

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

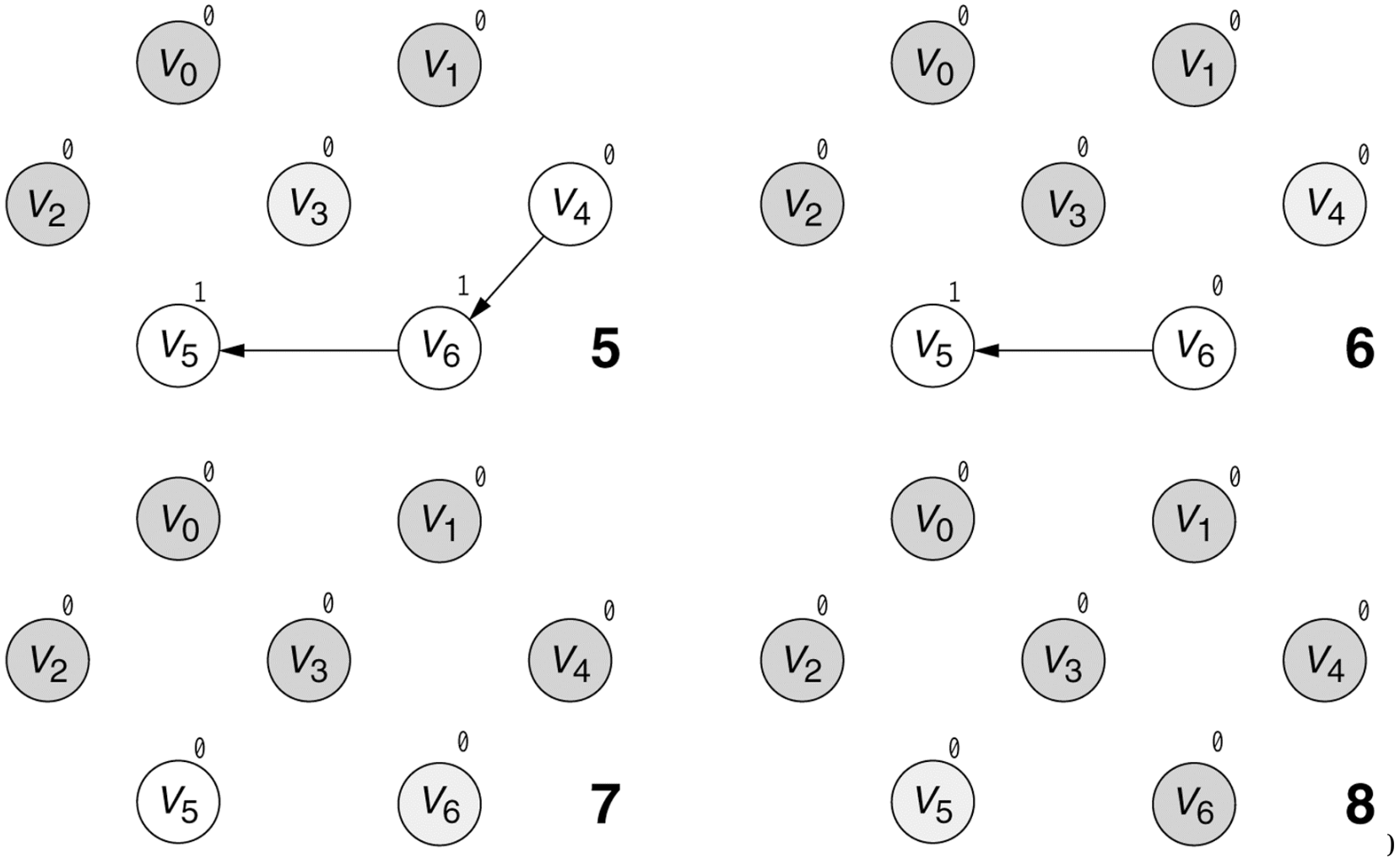


Figure 14.31A

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

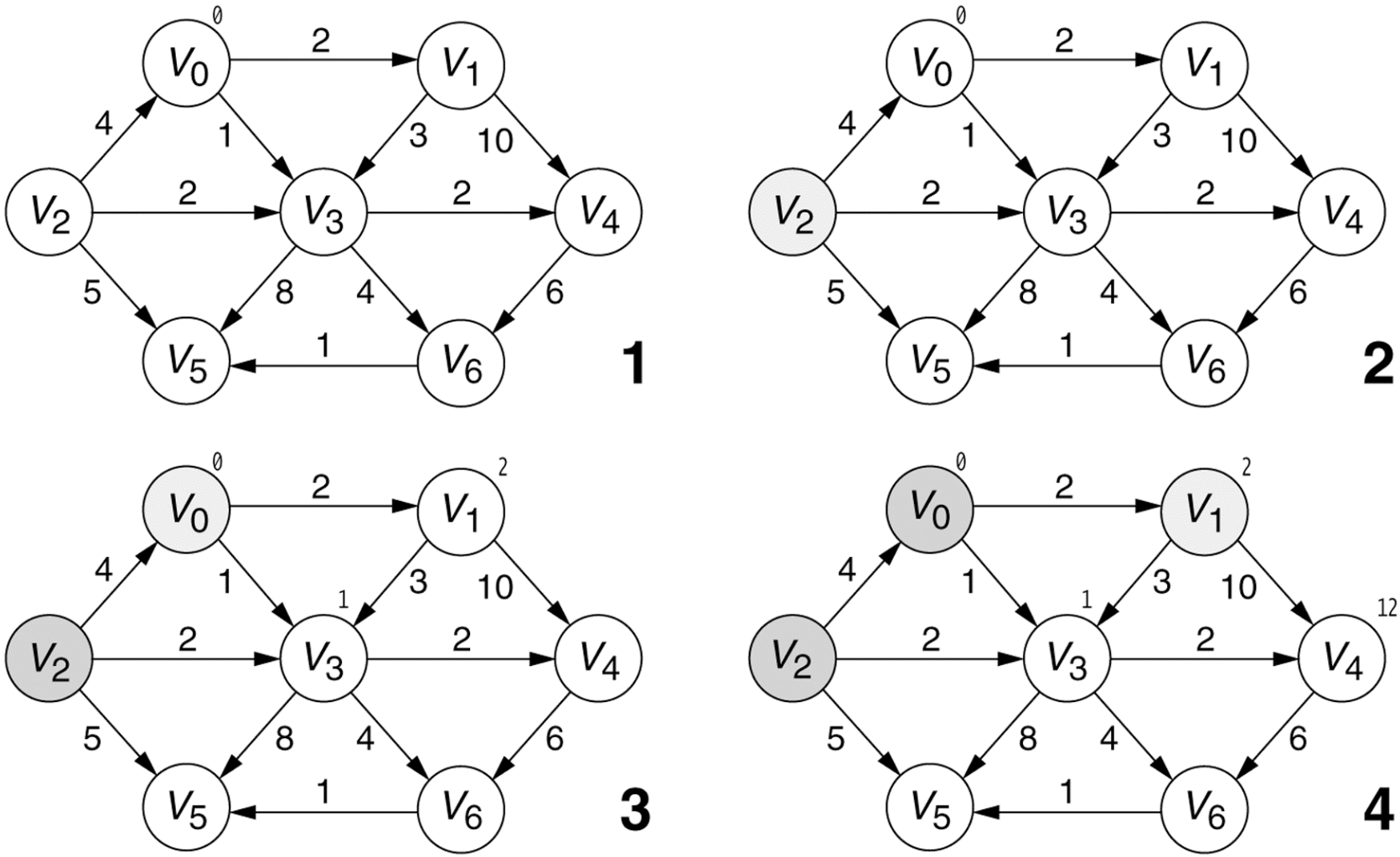


Figure 14.31B

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

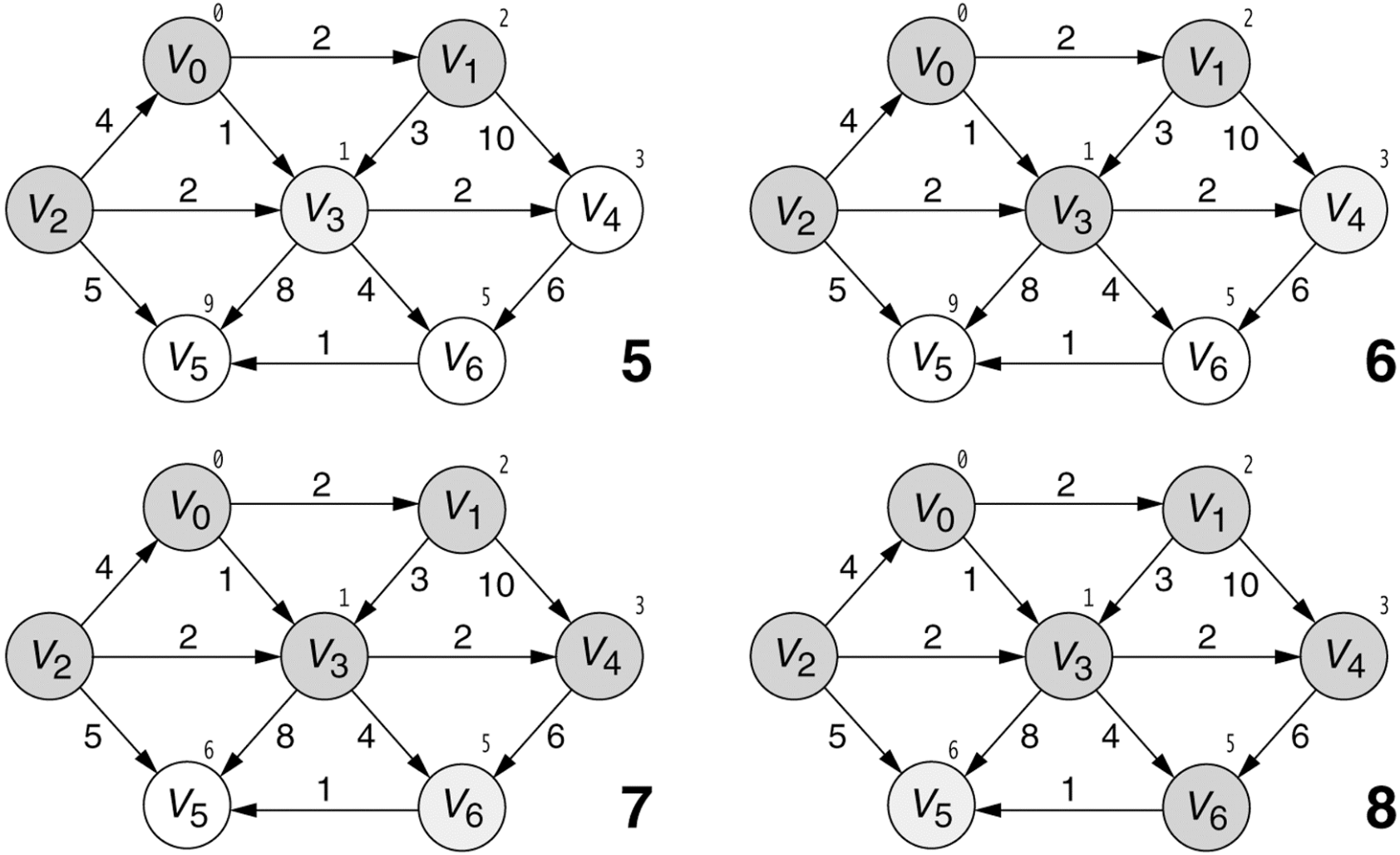


Figure 14.33
An activity-node graph

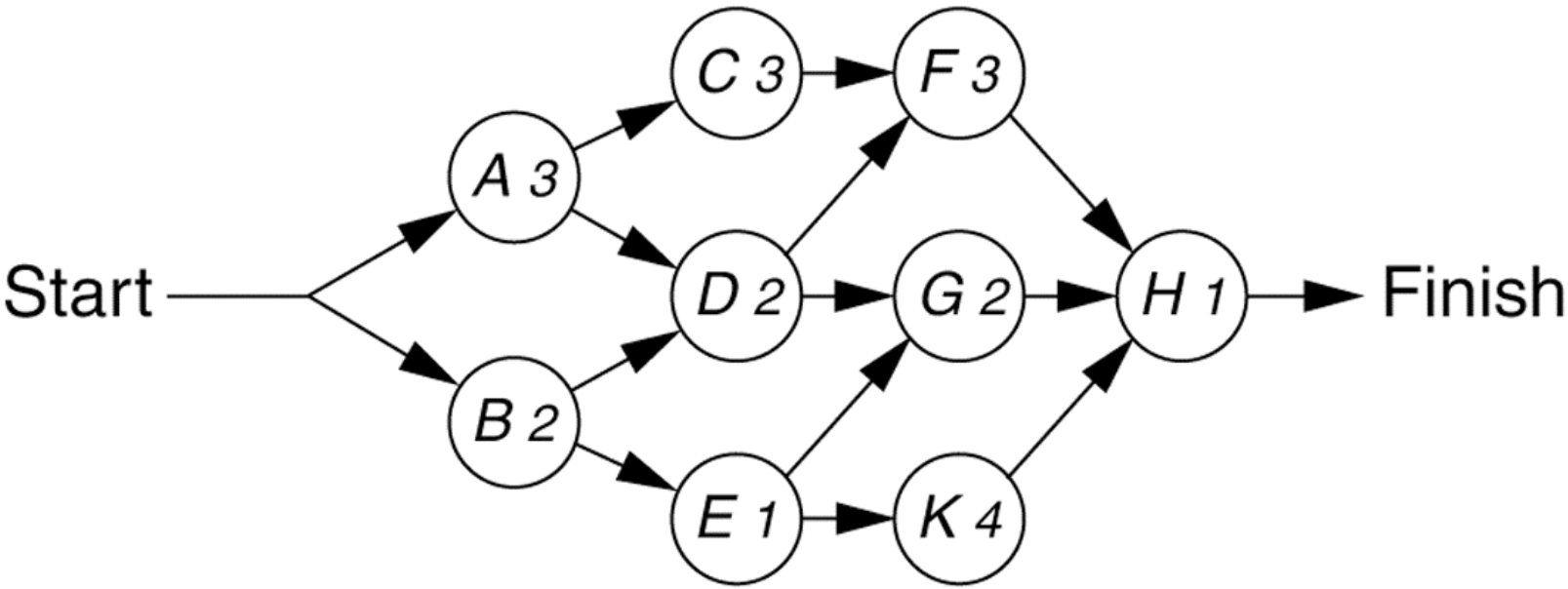


Figure 14.34
An event-node graph

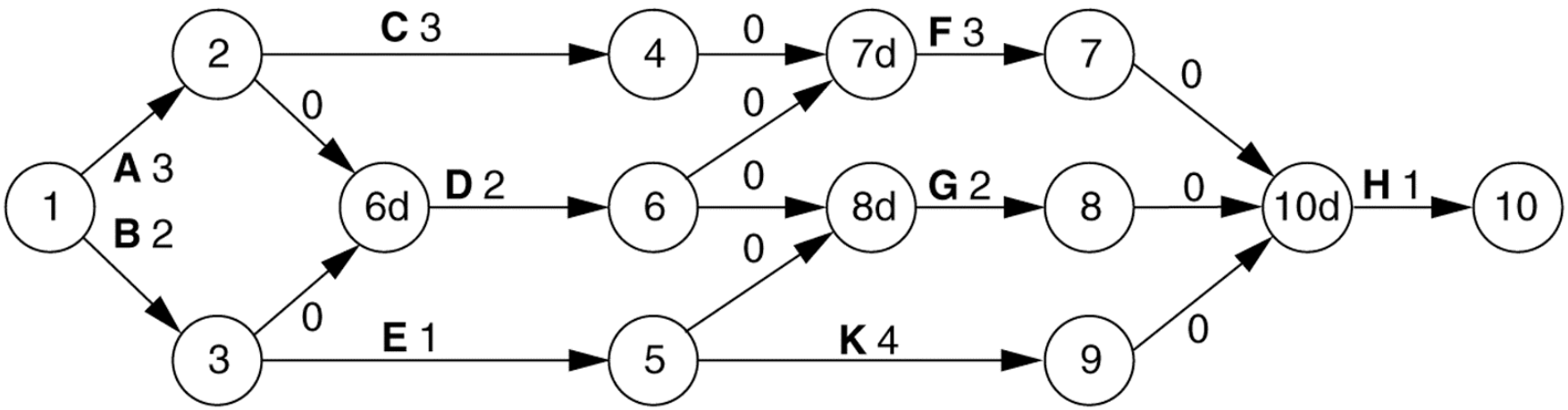


Figure 14.35
Earliest completion times

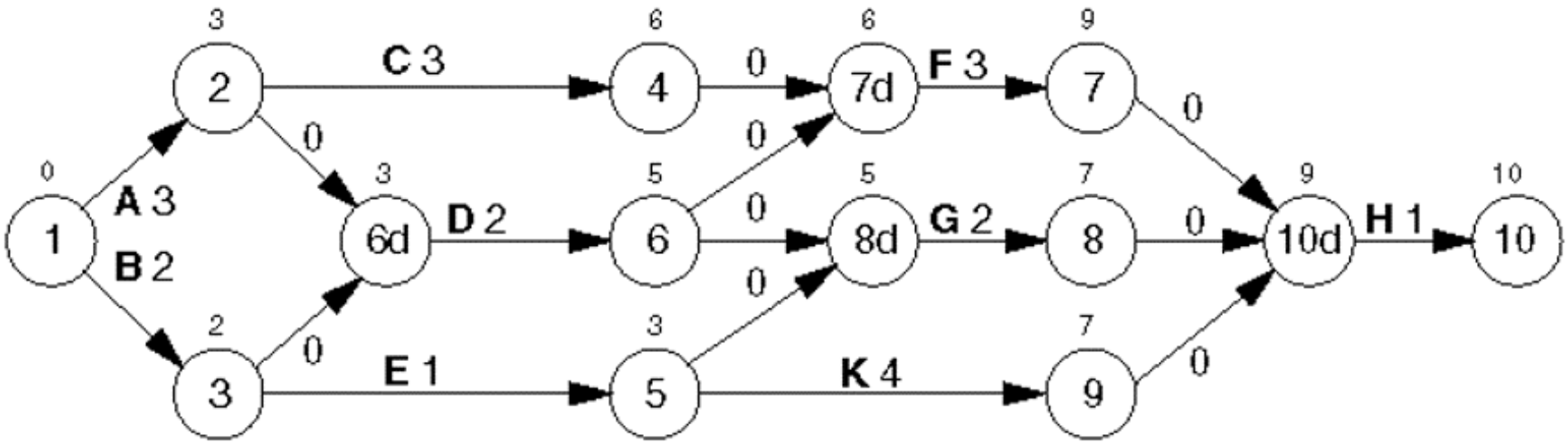


Figure 14.36
Latest completion times

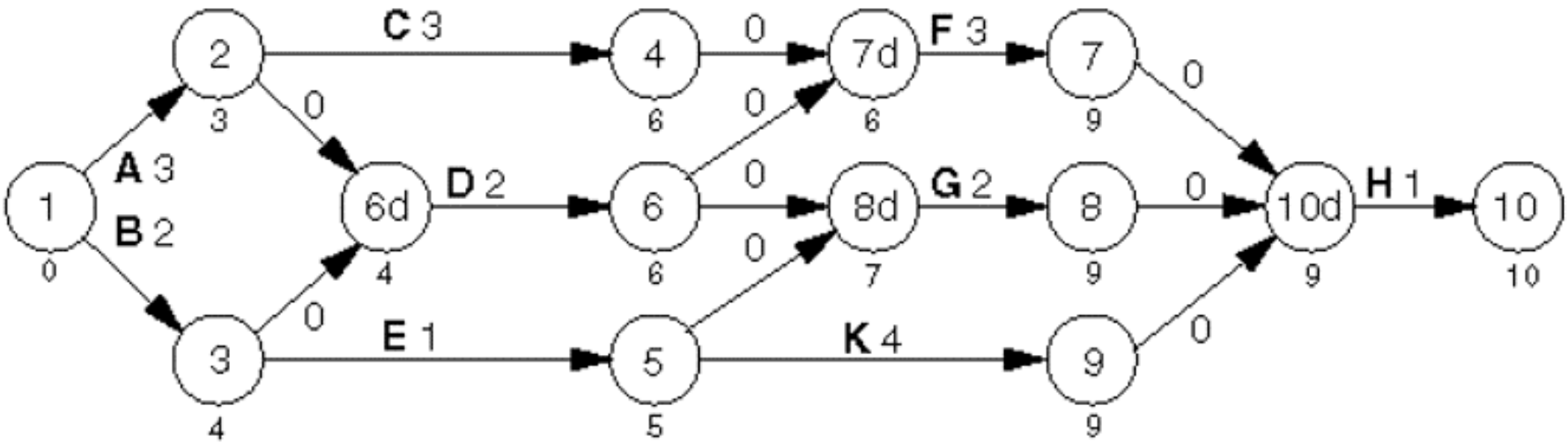


Figure 14.37

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

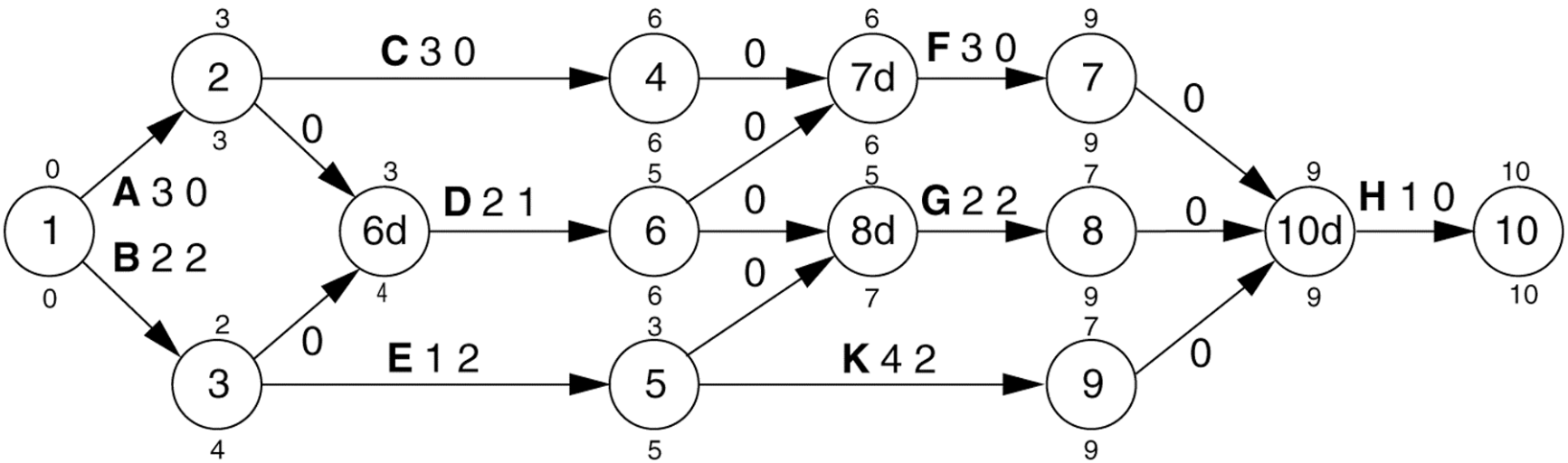


Figure 14.38
Worst-case running times of various graph algorithms

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

