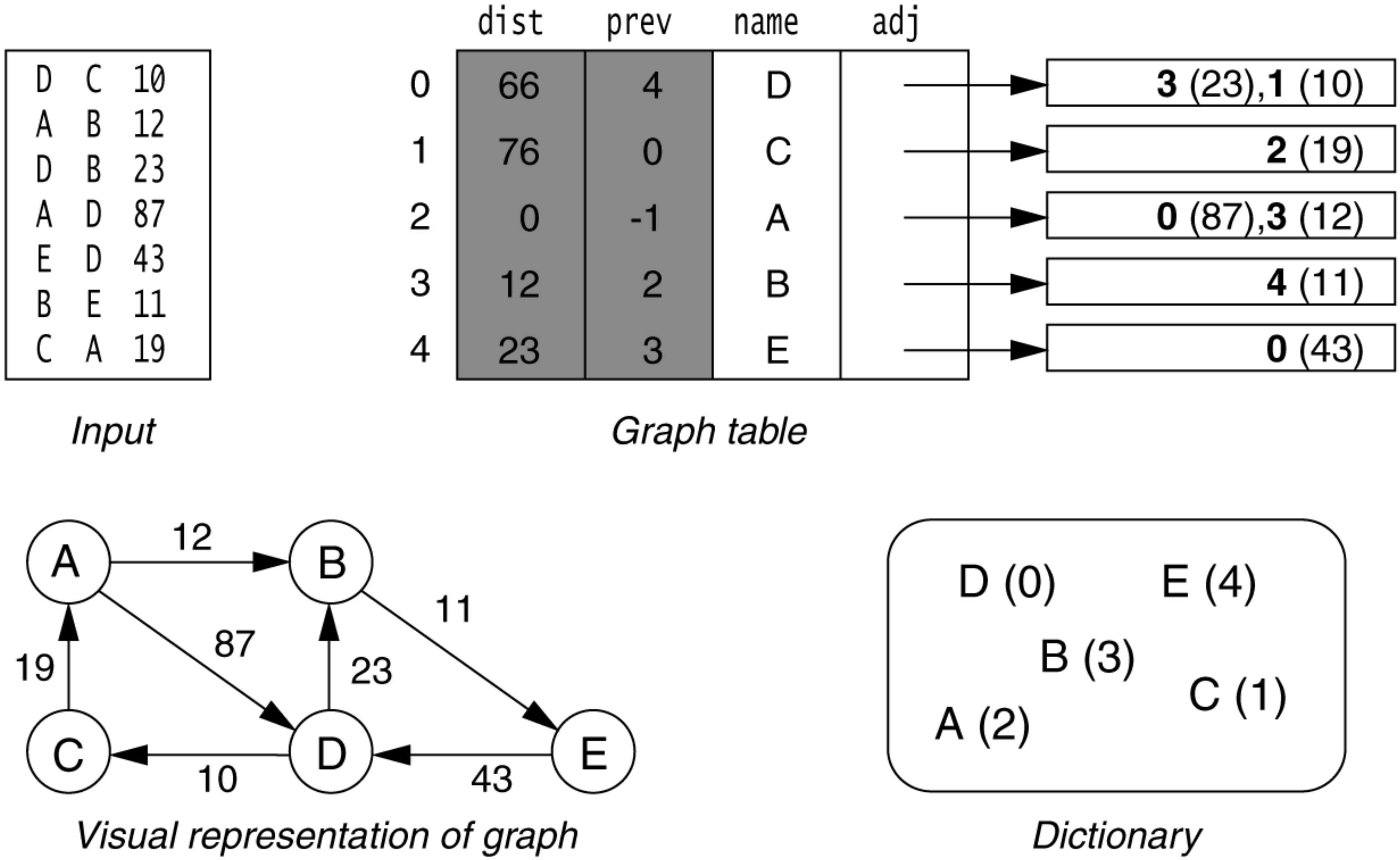
COP 3530, Course Outcomes

- be familiar with basic techniques of algorithm analysis
- master analyzing simple non-recursive algorithms
- be familiar with analyzing recursive algorithms
- master the implementation of stacks and queues
- master the implementation of linked data structures such as linked lists and unbalanced binary search trees
- be familiar with advanced data structures such as balanced search trees, hash tables, priority queues, and the disjoint set union/find data structure
- be familiar with several sub-quadratic sorting algorithms, including quicksort, mergesort, and heapsort
- be familiar with some graph algorithms such as shortest path and minimum spanning tree
- master the standard data structure library of a major programming language (e.g. java.util in Java 1.2)
- master analyzing problems and writing program solutions to problems using above techniques in a major programming language (e.g. Java)

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).



```
D  C  10
A  B  12
D  B  23
A  D  87
E  D  43
B  E  11
C  A  19
```

Input

	dist	prev	name	adj
0	66	4	D	**3** (23),**1** (10)
1	76	0	C	**2** (19)
2	0	-1	A	**0** (87),**3** (12)
3	12	2	B	**4** (11)
4	23	3	E	**0** (43)

Graph table

Visual representation of graph

D (0) E (4)
B (3)
C (1)
A (2)

Dictionary

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.

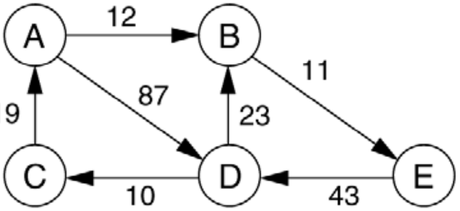
Dark arrows emanate from vertexMap.

Light arrows are adjacency list entries.

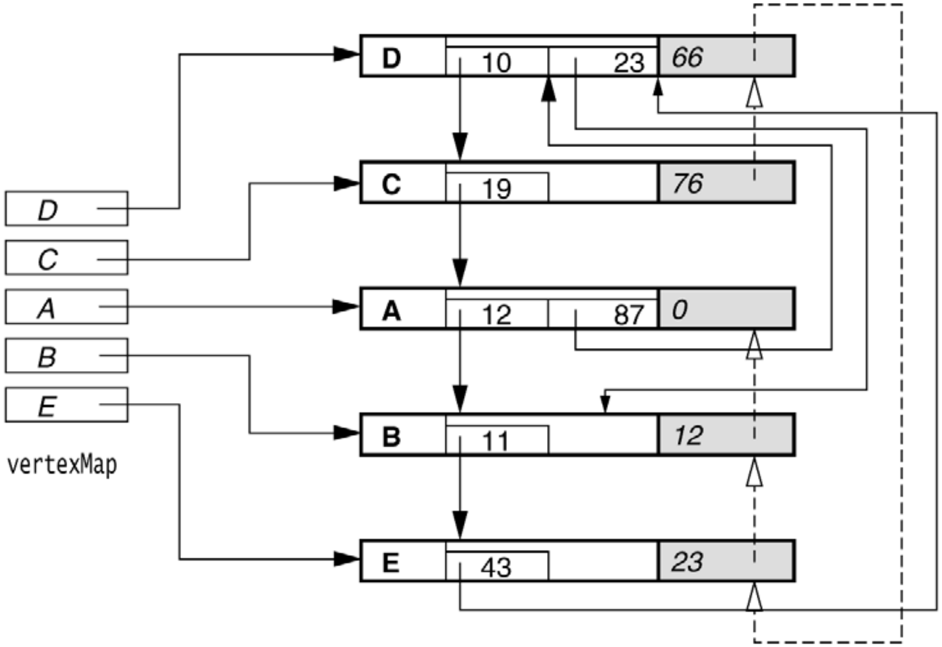
Dashed arrows are the prev data member that results from a shortest-path computation.

```
D  C  10
A  B  12
D  B  23
A  D  87
E  D  43
B  E  11
C  A  19
```

Input



Visual representation of graph



```
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;
  }
```

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.
   */
```

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 14.25A

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

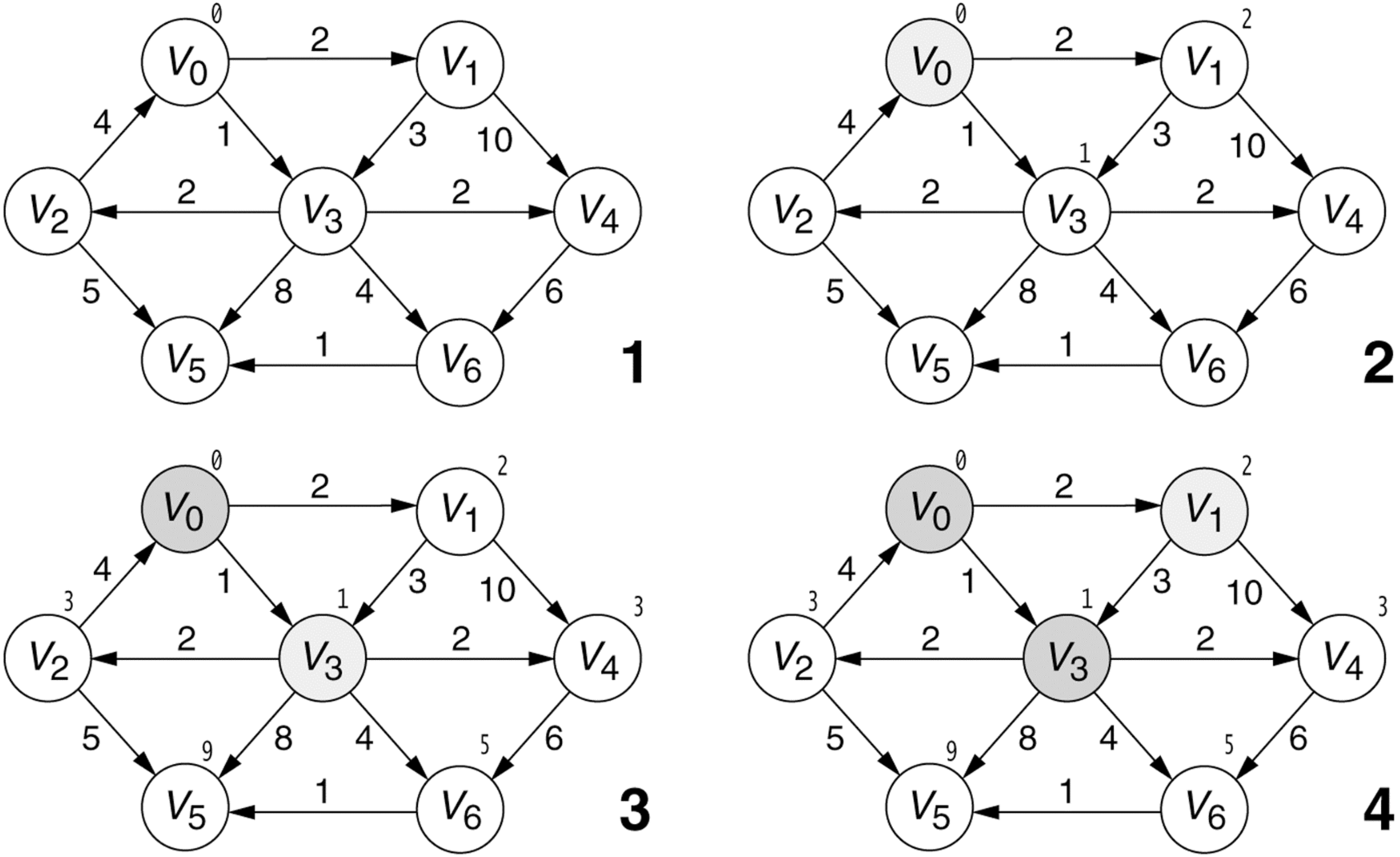
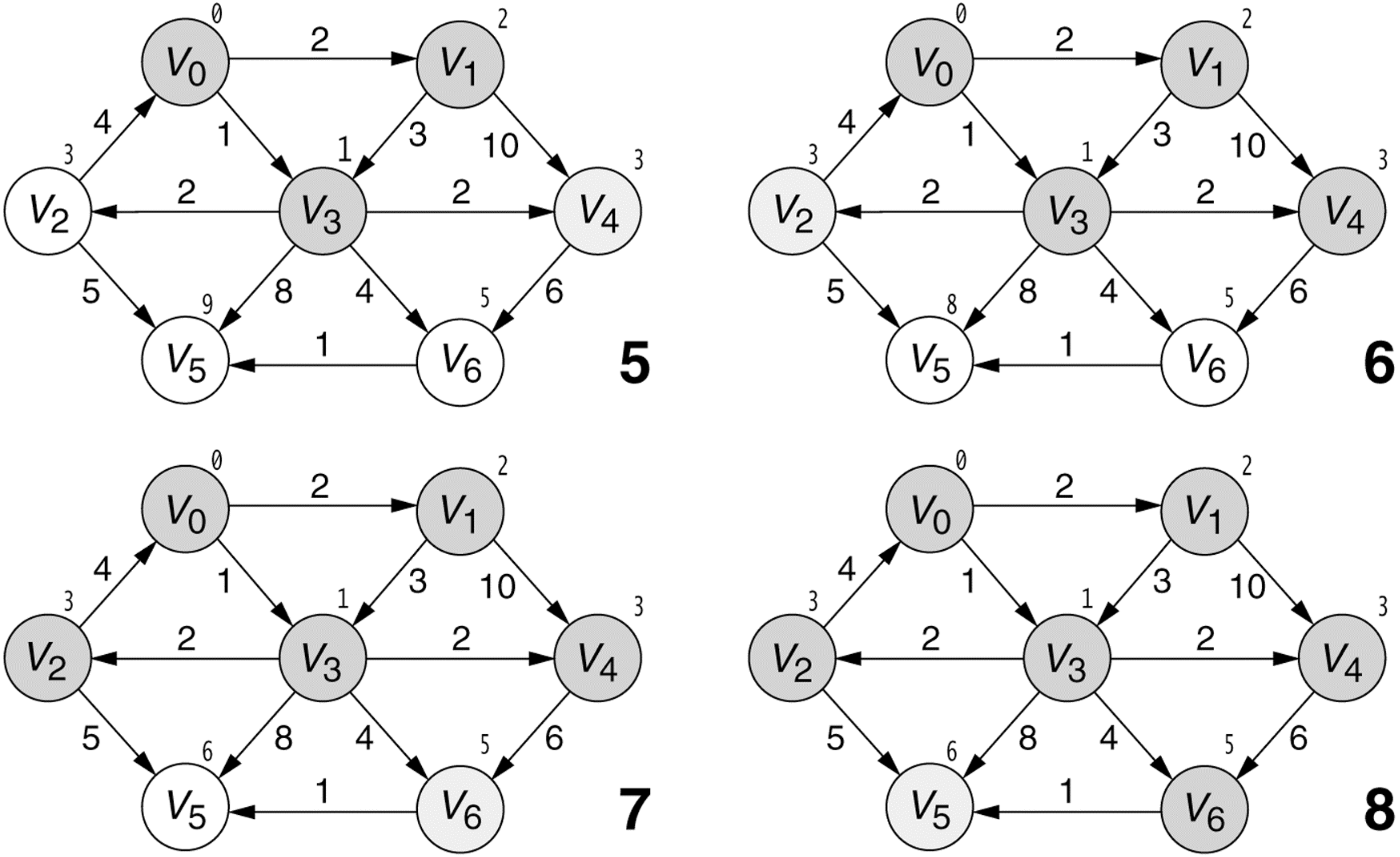


Figure 14.25B

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).

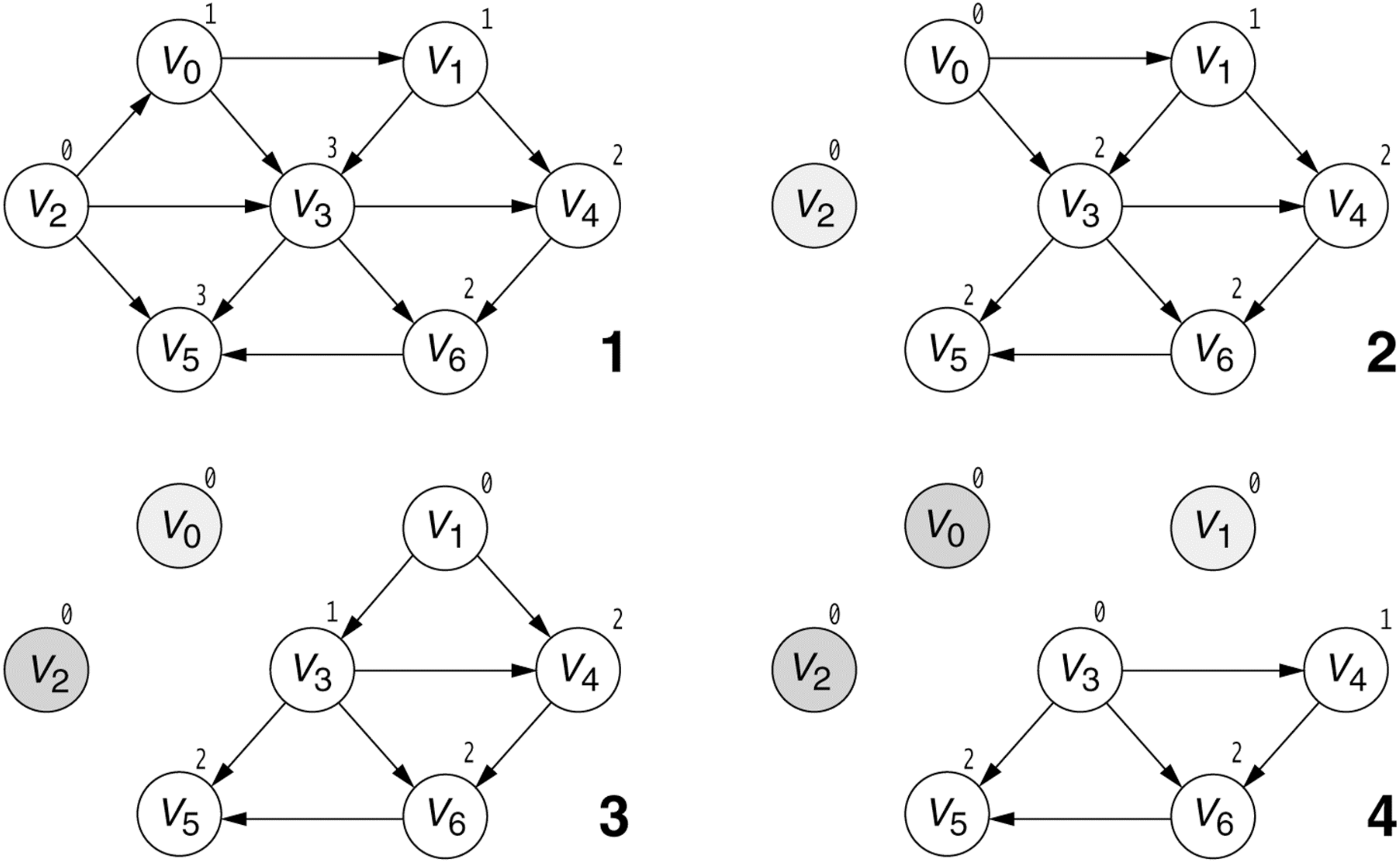


Figure 14.30B

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

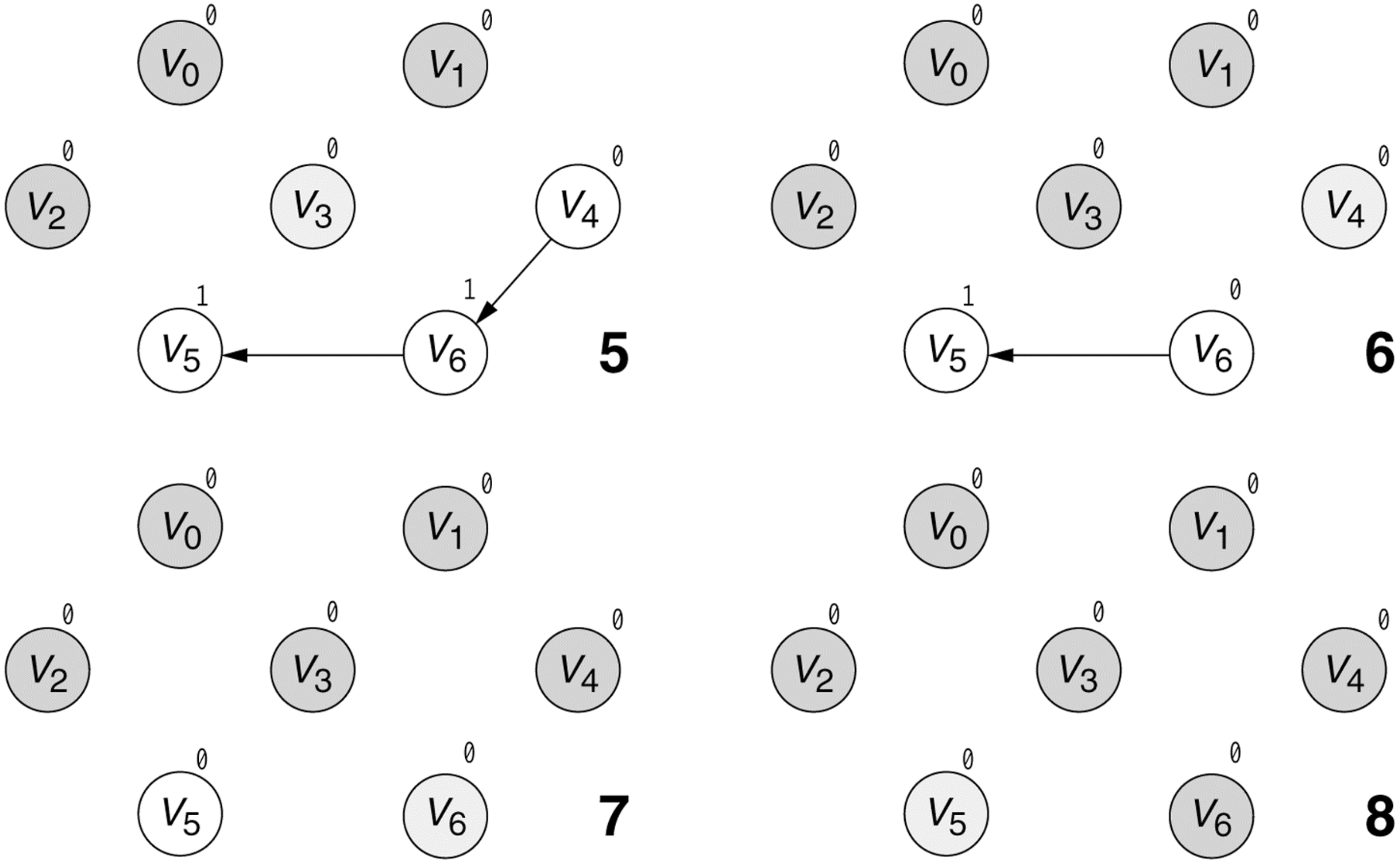


Figure 14.31A

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

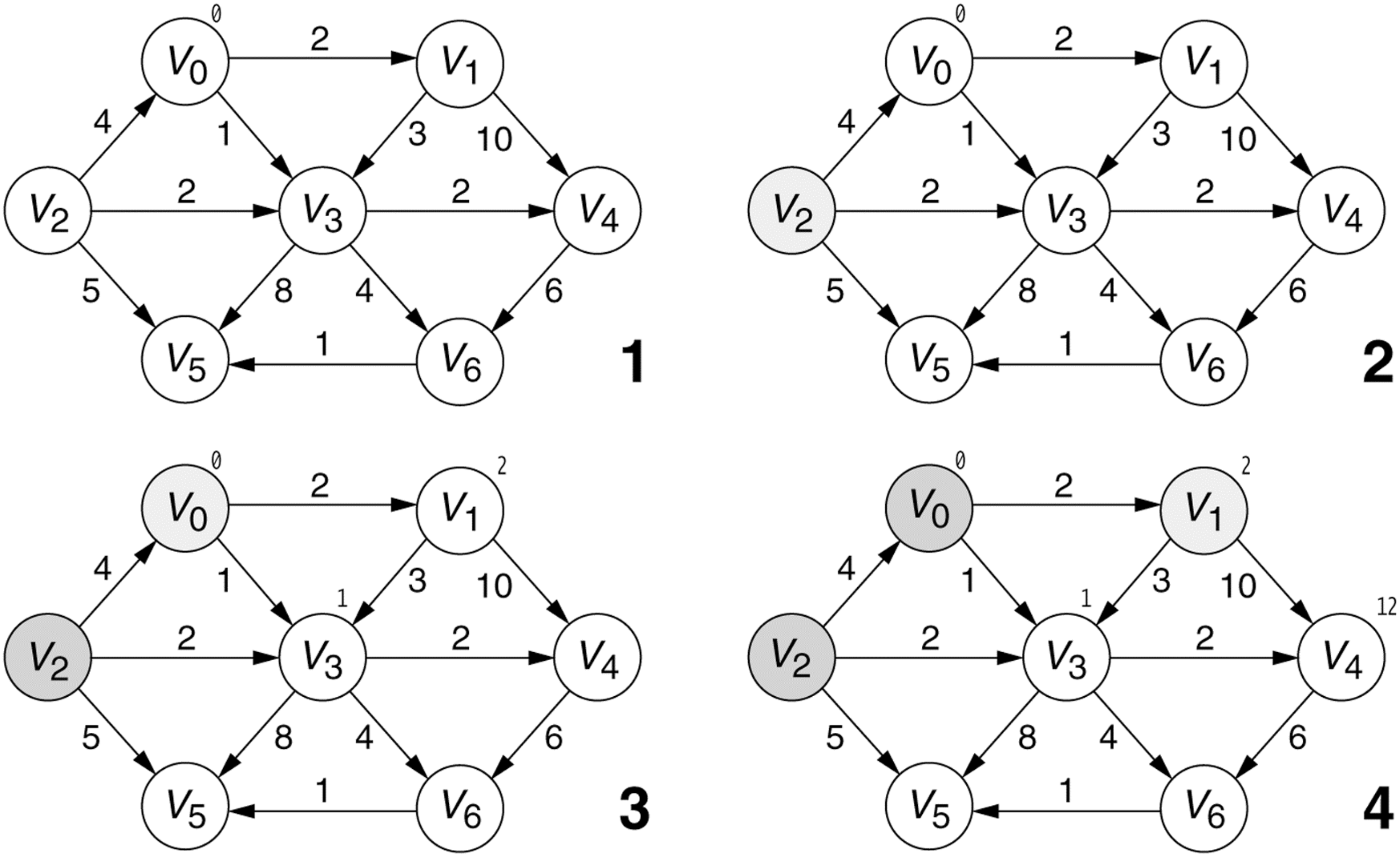


Figure 14.31B

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

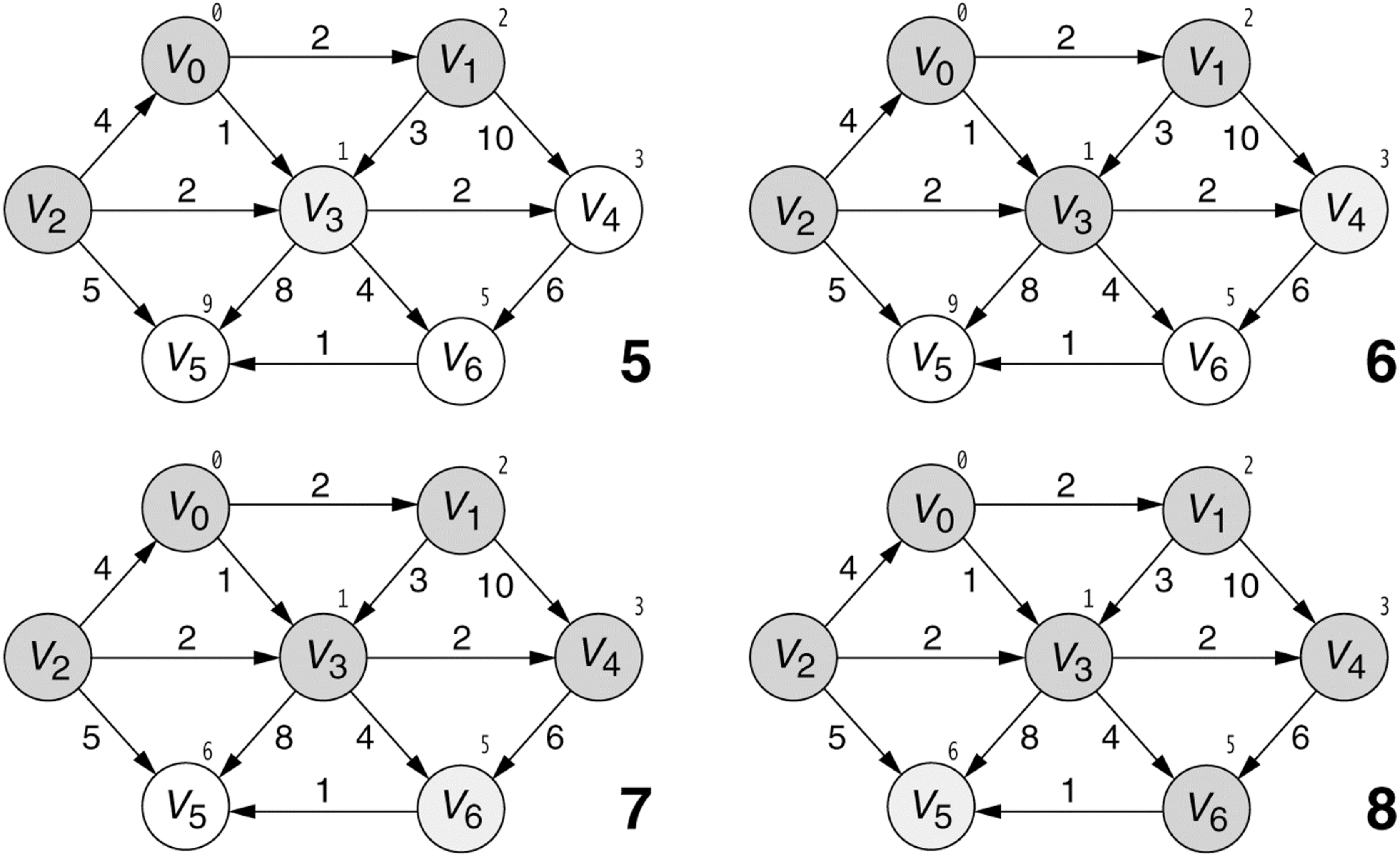


Figure 14.33

An activity-node graph

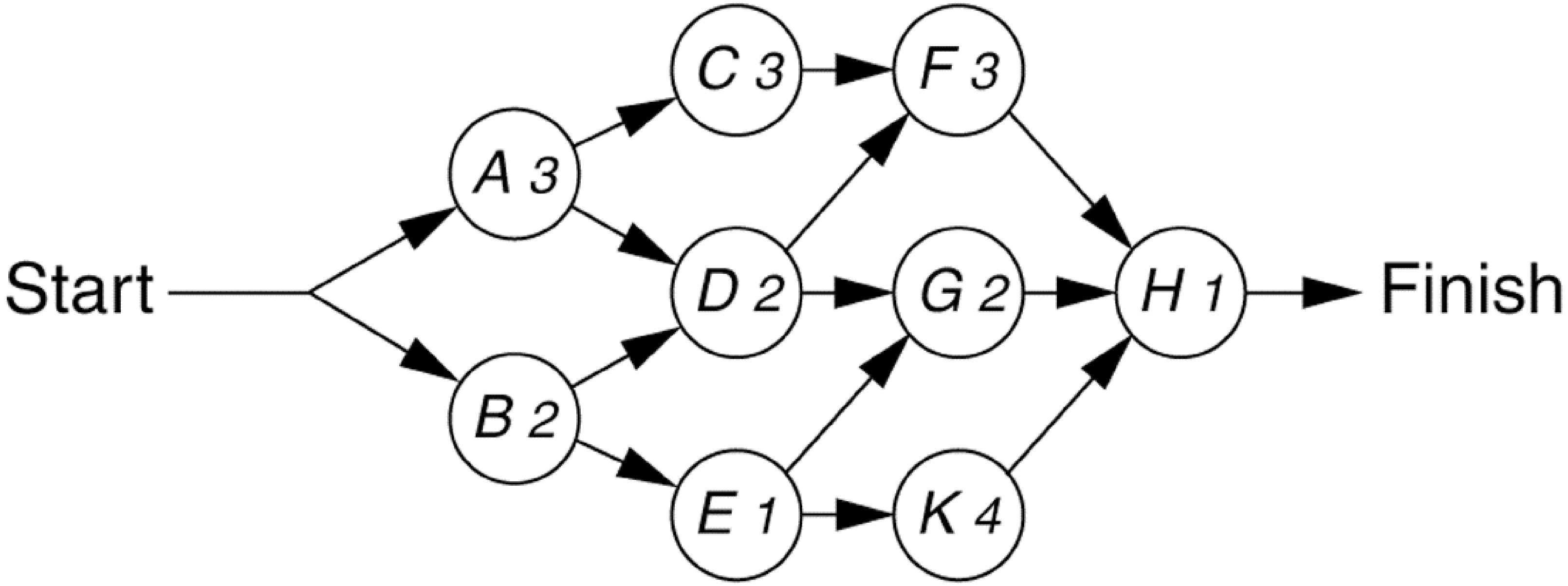


Figure 14.34

An event-node graph

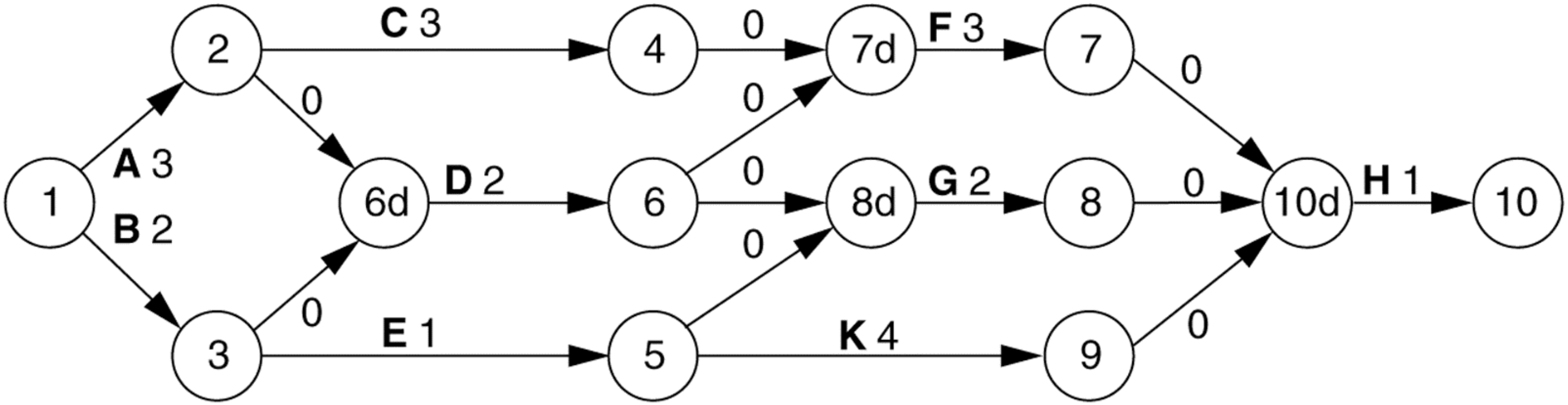


Figure 14.35

Earliest completion times

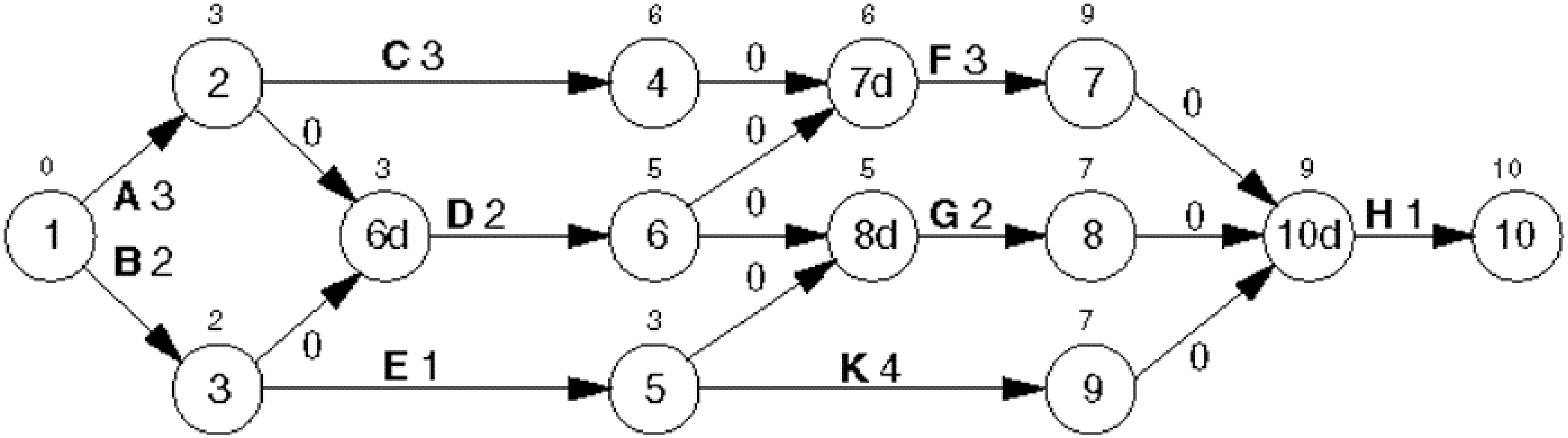


Figure 14.36

Latest completion times

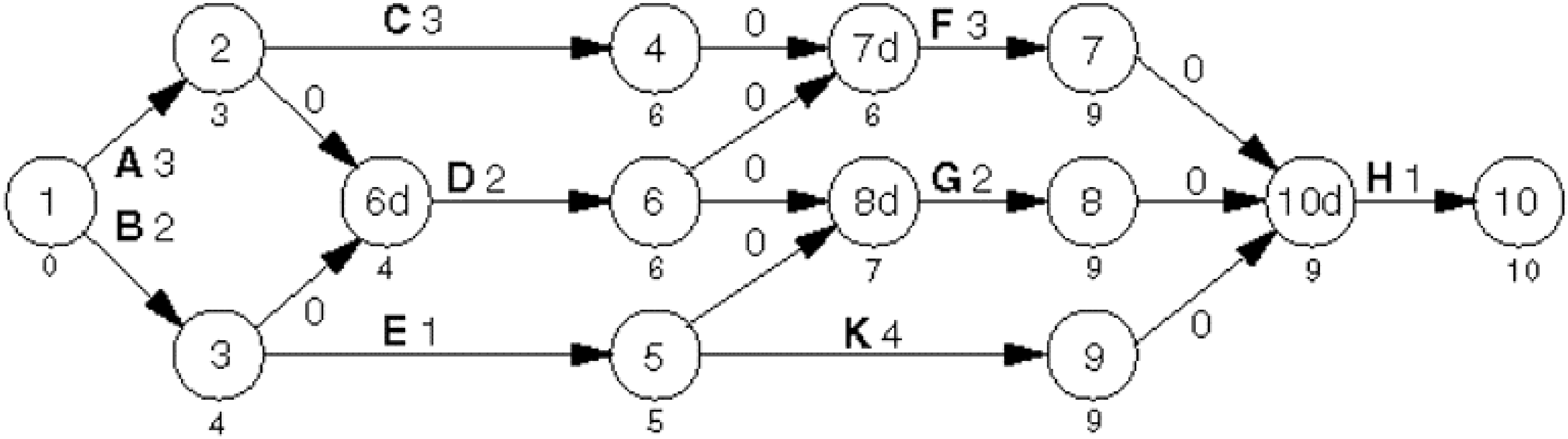


Figure 14.37

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

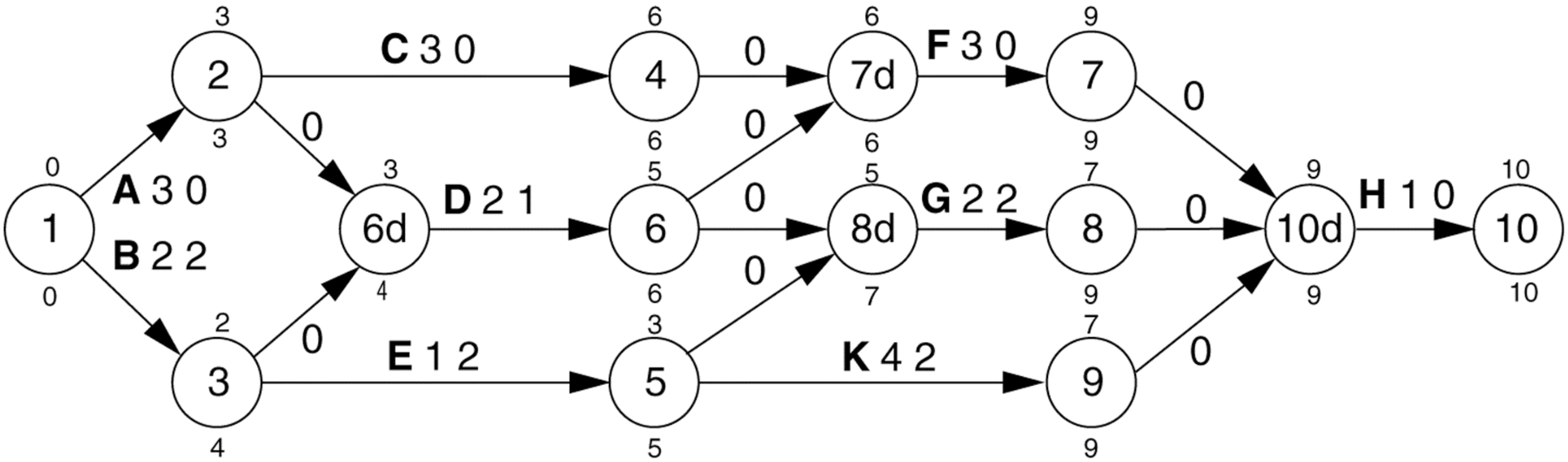
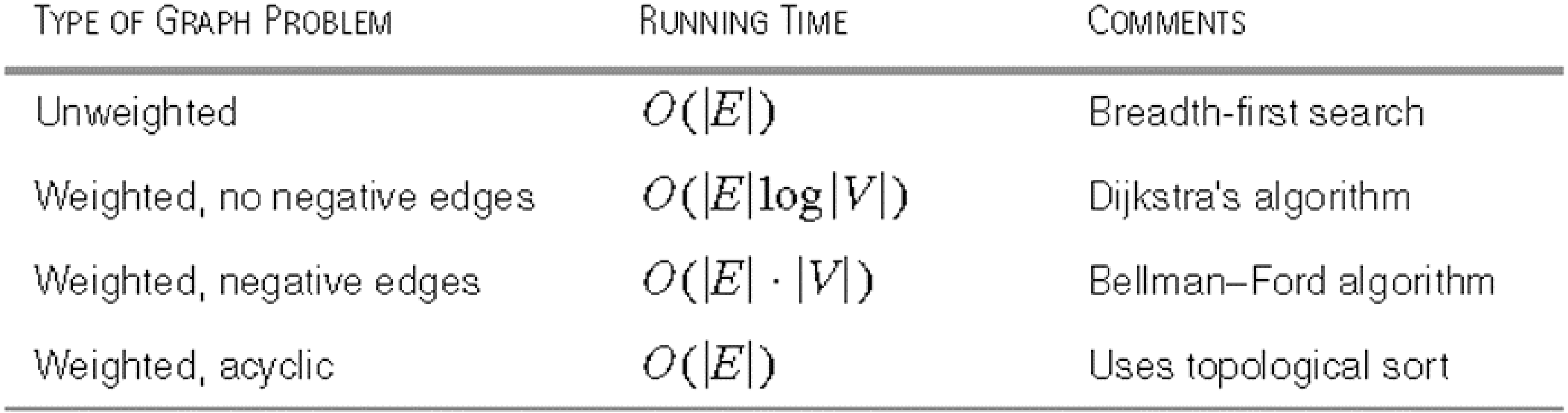


Figure 14.38

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

Random Number Generator

```
package weiss.util;
// Random class
// CONSTRUCTION: with (a) no initializer or (b) an integer
//     that specifies the initial state of the generator.
//     This random number generator is really only 31 bits,
//     so it is weaker than the one in java.util.
//
// int nextInt( )                          --> Uniform, [1 to 2^31-1]
```

Linear Congruential Generator

- Random numbers X_1, X_2, …

  $X_{i+1} = A * X_i \% M$
- A = 7, M = 11, Seed = 1

  7, 5, 2, 3, 10, 4, 6, 9, 8, 1, 7, 5, 2, …
- A = 5, M = 11, Seed = 1

  ??
- A = 48,271, M = 2,147,483,647 = $2^{31} - 1$

  Full-Period, except if seed = 179,424,105

nextInt()

- State = (A * state) % M;   // Easy!
- Problem is with overflow, if computations are done on 32 bit integers.

- Solution:

$$X_{i+1} = A * (X_i \% Q) - R \lfloor X_i / Q \rfloor + M\delta(X_i)$$

- 1st term never overflows
- 2nd term – no overflow if $R < Q$
- $\delta(X_i)$ is 0 if subtraction of first 2 terms is positive and 1 if it is negative.
- So, choose Q = 44,488 and R = 3,399.

nextInt() implementation

```
private static final int A = 48271;
private static final int M = 2147483647;
private static final int Q = M / A;
private static final int R = M % A;

public int nextInt( )
   {
        int tmpState = A * ( state % Q ) - R * ( state / Q );
        if( tmpState >= 0 )
            state = tmpState;
        else
            state = tmpState + M;

        return state;
   }
```

Variant of nextInt()

```
//  Return a random int in the closed range [low,high], and
public int nextInt( int low, int high )
   {
      double partitionSize = (double) M / ( high - low + 1 );
      return (int) ( nextInt( ) / partitionSize ) + low;
   }

// Return a pseudorandom double in the open range 0..1
public double nextDouble( )
   {
      return (double) nextInt( ) / M;
   }
```

Non-uniform Random Number Generators

```
// int nextDouble( )                          --> Uniform, (0 to 1)
// int nextInt( int high )                    --> Uniform [0..high)
// int nextInt( int low, int high )           --> Uniform [low..high]
// long nextLong( )                           --> Uniform [1 to 2^62-1]
// long nextLong( long low, long high )       --> Uniform [low..high]
// int nextPoisson( double expectedVal )      --> Poisson
// double nextNegExp( double expectedVal ) --> Negative exponential
// void permute( Object [ ] a )               --> Randomly permutate
```

Generating a Random Permutation

```
public static final void permute( Object [ ] a )
  {
      Random r = new Random( );

      for( int j = 1; j < a.length; j++ )
         swapReferences( a, j, r.nextInt( 0, j ) );
  }

private static final void swapReferences( Object [ ] a, int index1, int
    index2 )
  {
      Object tmp = a[ index1 ];
      a[ index1 ] = a[ index2 ];
      a[ index2 ] = tmp;
  }
```