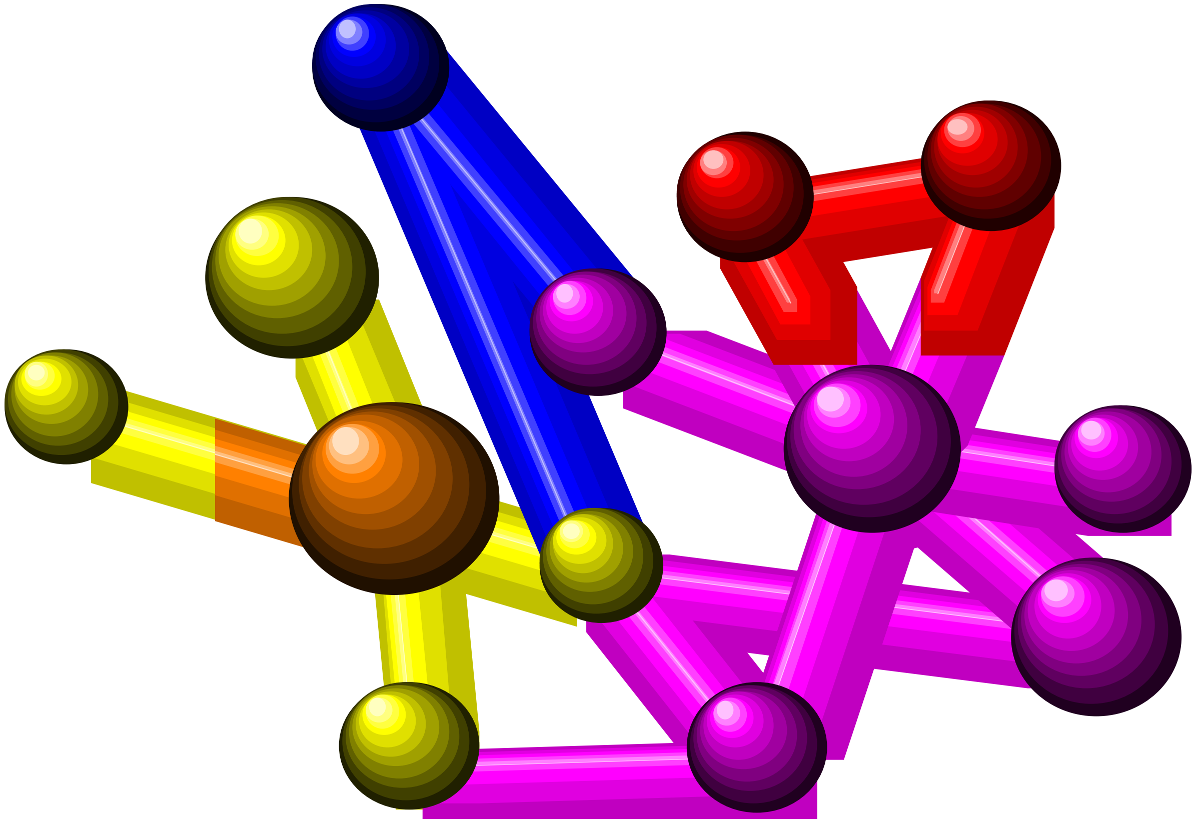
Applications III



Agenda

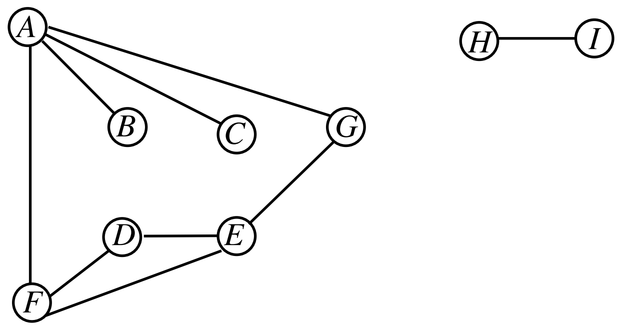
- **Graphs**
  - Terminology
  - Representation
  - Traversal
  - Shortest path
  - Topological sorting
- **Problem complexity**

Graphs

A **graph** is a useful abstract concept.

Intuitive definition: A graph is a set of *objects* and a set of *relations* between these objects.

Mathematical definition: A graph $G = (V, E)$ is a finite set of *vertices*, V, (or *nodes*) and a finite set of *edges*, E, where each edge connects two vertices $(E \subseteq V \times V)$.



$V = \{A, B, C, D, E, F, G, H, I\}$

$E = \{(A,B), (A,C), (A,F), (A,G), (D,E), (D,F), (E,F), (E,G), (H,I)\}$

Applications

Anything involving relationships among objects
can be modeled as a graph

Traffic networks:
Vertices: cities, crossroads
Edges: roads

Electric circuits:
Vertices: devices
Edges: wires

Organic molecules:
Vertices: atoms
Edges: bonds

Game graphs:
Vertices: board positions
Edges: moves

Applications

(continued)

Software systems:

Vertices: methods

Edges: method *A* calls method *B*

Object-oriented design (UML diagramming):

Vertices: classes/objects

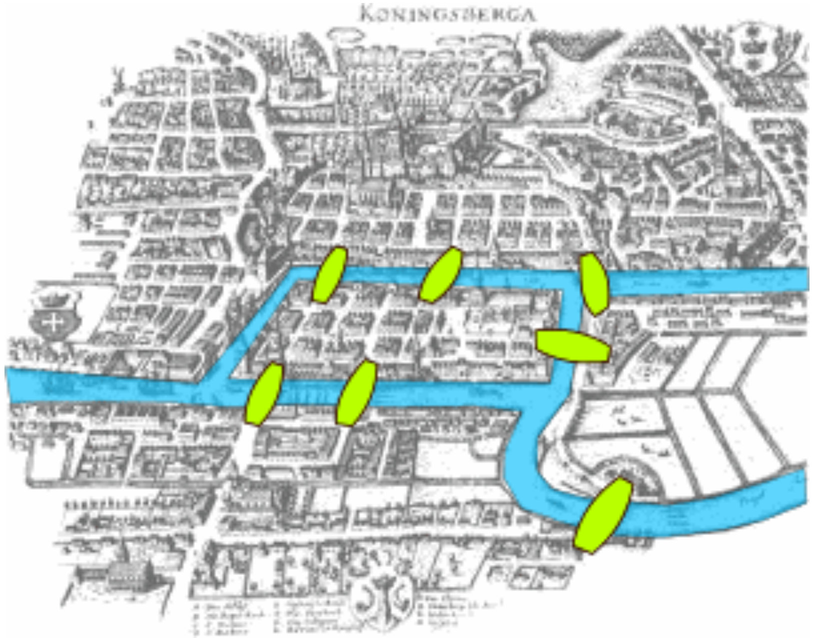
Edges: inheritance, aggregation, association

Project planning:

Vertices: subtasks

Edges: dependencies (subtask *A* must finish be before subtask *B* can start)

Historical foundation of graph theory



Map of Königsberg in Euler's time showing the actual layout of the seven bridges, highlighting the river Pregel and the bridges

The problem was to find a walk through the city that would cross each bridge once and only once. Euler proved in 1735 that this problem has no solution.

Euler’s analysis

L. Euler, 1707-83

During any walk in the graph, the number of times one enters a non-terminal vertex equals the number of times one leaves it.
Now if every bridge is traversed exactly once it follows that for each land mass (except possibly for the ones chosen for the start and finish), the number of bridges touching that land mass is **even** (half of them, in the particular traversal, will be traversed "toward" the landmass, the other half "away" from it).
However, all the four land masses are touched by an **odd** number of bridges.

Terminology

The two vertices of an edge is called its **end vertices**.

$$H - I$$

If an edge is a ordered pair of end vertices, then the edge is said to be **directed**. This is indicated on the visual representation by drawing the edge as an arrow.

$$H \rightarrow I$$

A **directed graph** (or **digraph)** is a graph in which all edges are directed.

A **undirected graph** is a graph in which no edges are directed.

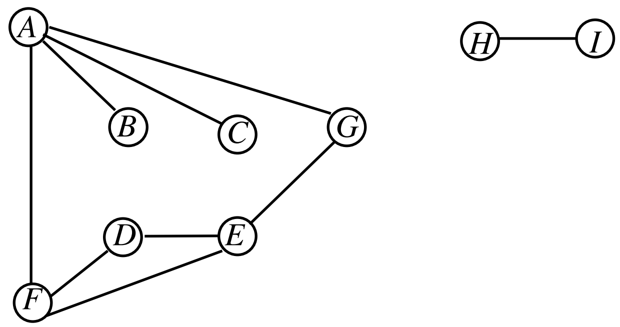
Terminology

(continued)

A **path** is a sequence of vertices connected by edges.

A **simple path** is a path in which all vertices are distinct.

A **cycle** is a path that is simple, except that the first and last vertex are the same.



Cycles: *FDEF*, *AFEGA*, and *AFDEGA*

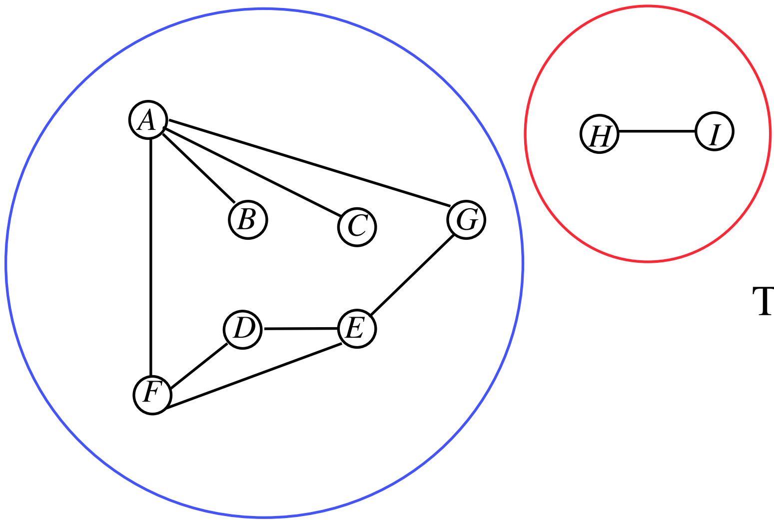
Terminology

(continued)

A graph $G' = (V', E')$ is a **subgraph** of a graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.

A graph is said to be **connected** if, for every two vertices u and v, there is a path from u to v or a path from v to u.

A graph, which is not strongly connected, consists of two or more connected subgraphs, called **components**.



Two components

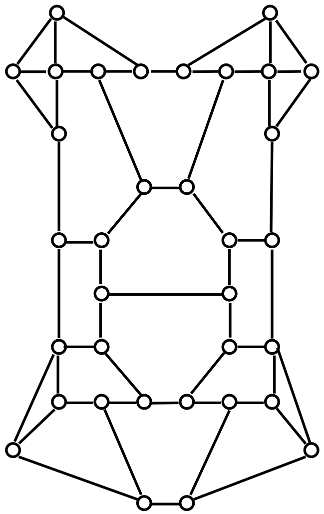
Terminology

(continued)

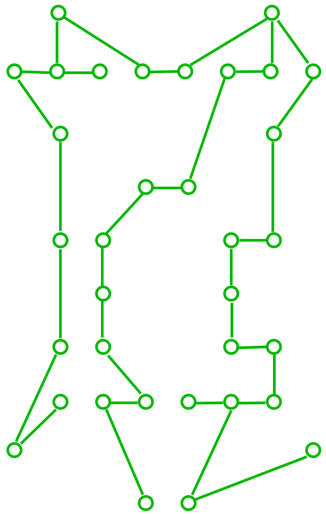
A **tree** is a connected graph without cycles.

A **forest** is a set of disjoint trees.

A **spanning tree** for a graph *G* is a tree composed of all vertices of *G* and some (or perhaps all) of its edges.



Graf *G*



Spanning tree for *G*

Terminology

(continued)

A graph in which every pair of vertices are connected by a unique edge is said to be **complete**.

[for an undirected complete graph: $|E| = |V|(|V|-1)/2$]

A **dense** graph is a graph in which the number of edges is close to the maximal number of edges.

A **sparse** graph is a graph with only a few edges.

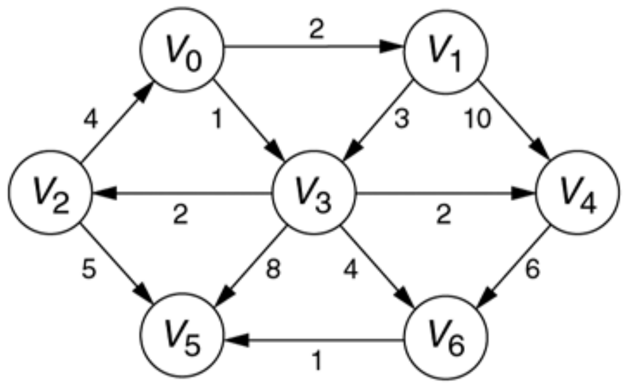
A graph is a **weighted graph** if a number (weight) is assigned to each edge.

[weights usually represent costs]

A directed weighted graph

figure 14.1

A directed graph



Basic graph problems

Paths:

Is there a path from *A* to *B*?

Cycles:

Does the graph contain a cycle?

Connectivity (spanning tree):

Is there a way to connect all vertices?

Biconnectivity:

Will the graph become disconnected if one vertex is removed?

Planarity:

Is there a way to draw the graph without edges crossing?

Basic graph problems

(continued)

Shortest path:
What is the shortest way from A to B?

Longest path:
What is the longest way from A to B?

Minimal spanning tree:
What is the cheapest way to connect all vertices?

Hamiltonian cycle:
Is there a way to visit all the vertices without visiting the same vertex twice?

Traveling salesman problem:
What is the shortest Hamiltonian cycle?

Representation of graphs

Graphs are *abstract* mathematical objects.
Algorithms have to work with *concrete* representations.

Many different representations are possible. The choice is decided by algorithms and graph types (sparse/dense, weighted/unweighted, directed/undirected).

Three data structures will be described:

(1) **edge set**
(2) **adjacency matrix**
(3) **adjacency lists**

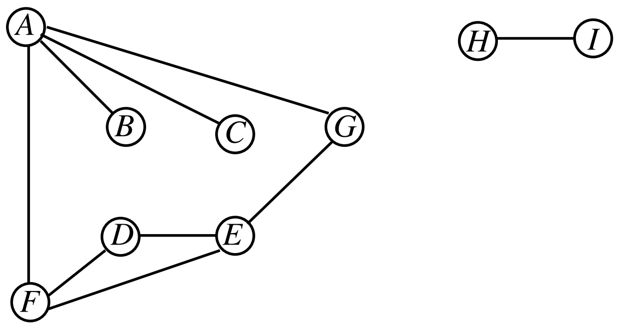
(1) Edge set

```
class Graph {
    Set<Edge> edges;
}
```

```
class Edge {
    Vertex source, dest;
    double cost;
}
```

```
class Vertex {
    String name;
}
```

(2) Adjacency matrix

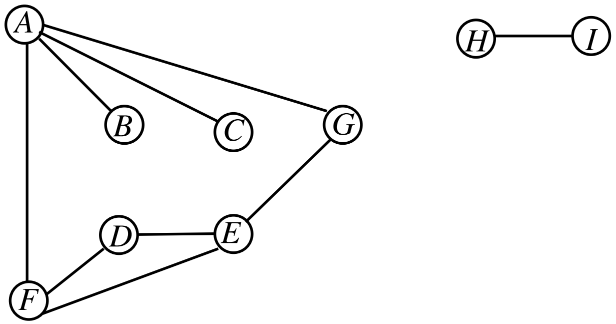


	A	B	C	D	E	F	G	H	I
A	0	1	1	0	0	1	1	0	0
B	1	0	0	0	0	0	0	0	0
C	1	0	0	0	0	0	0	0	0
D	0	0	0	0	1	1	0	0	0
E	0	0	0	1	0	1	1	0	0
F	1	0	0	1	1	0	0	0	0
G	1	0	0	0	1	0	0	0	0
H	0	0	0	0	0	0	0	0	1
I	0	0	0	0	0	0	0	1	0

```
class Graph { // unweighted
    boolean[][] adjMatrix;
}
```

```
class Graph { // weighted
    double[][] adjMatrix;
}
```

(3) Adjacency lists



A: $F \to C \to B \to G \to \square$

B: $A \to \square$

C: $A \to \square$

D: $F \to E \to \square$

E: $G \to F \to D \to \square$

F: $A \to E \to D \to \square$

G: $E \to A \to \square$

H: $I \to \square$

I: $H \to \square$

(3) Adjacency lists

```
class Graph {
    Map<String,Vertex> vertexMap;
}
```

```
class Vertex {
    String name;          // Vertex name
    List<Edge> adj;       // Adjacent vertices
}
```

```
class Edge {
    Vertex dest;          // Second vertex of edge
    double cost;          // Edge weight
}
```

figure 14.1

A directed graph

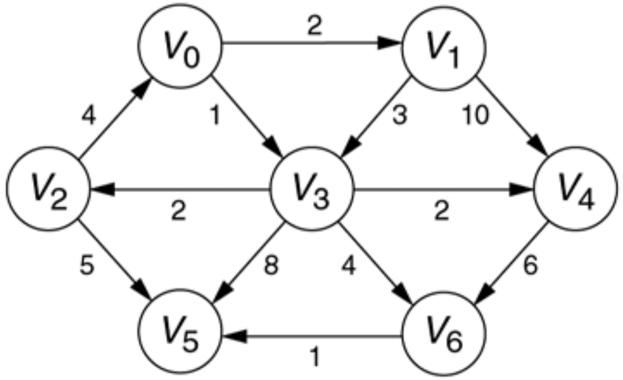
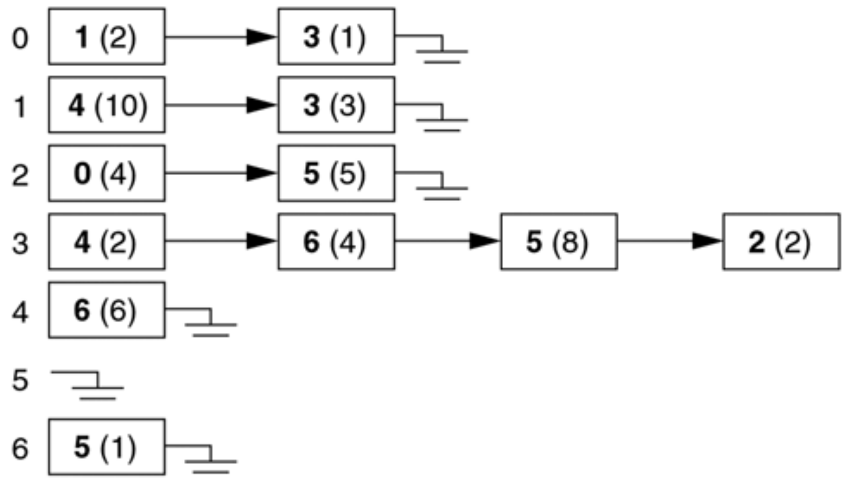


figure 14.2

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.



Comparison of representations

Space requirements:

Edge set: $O(|E|)$

Adjacency matrix: $O(|V|^2)$

Adjacency lists: $O(|V| + |E|)$

Choice of representation affects algorithm efficiency

Time complexity (worst case):

Is there an edge from A to B?

Edge set: $O(|E|)$

Adjacency matrix: $O(1)$

Adjacency lists: $O(|V|)$

Is there an edge from A to anywhere?

Edge set: $O(|E|)$

Adjacency matrix: $O(|V|)$

Adjacency lists: $O(1)$

Traversing graphs

Goal: "visit" every vertex of the graph.

Depth-first traversal (recursive):

* Mark all vertices as "unvisited"
* Visit vertex 1
* To visit a vertex *v*:
    * mark it
    * (recursively) visit all unmarked vertices connected to *v* by an edge

Solves some simple graph problems:
connectivity, cycles
Basis for solving difficult graph problems:
biconnectivity, planarity

Implementation of depth-first traversal

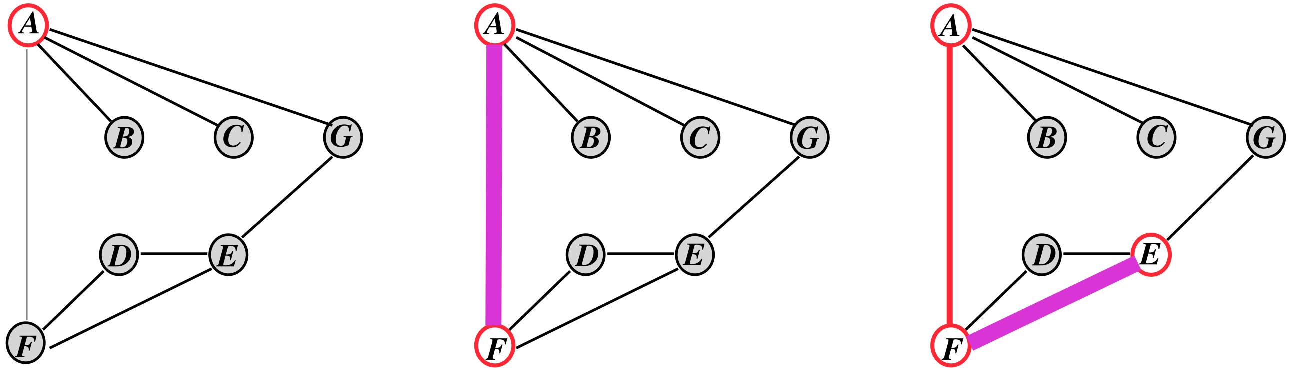
(adjacency lists)

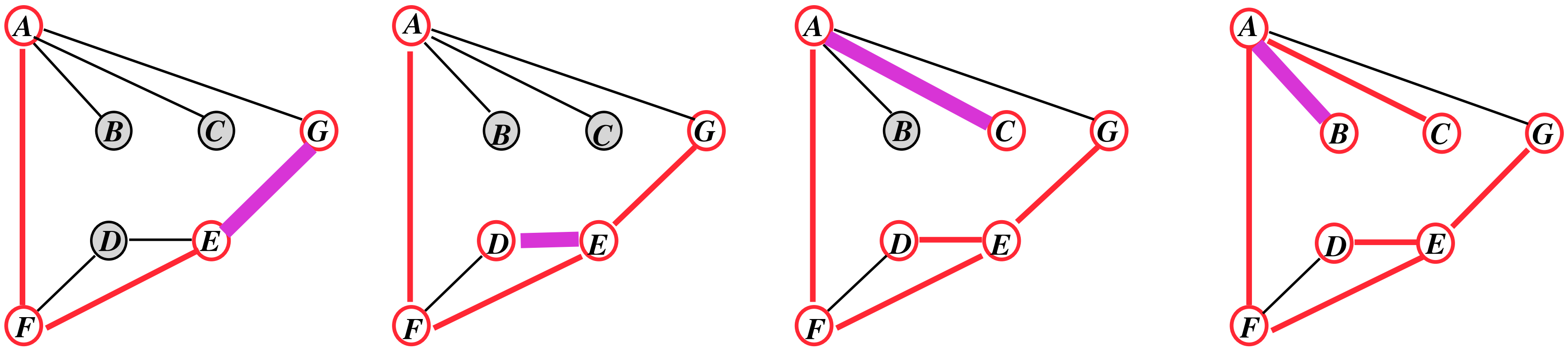
```
class Vertex {
    String name;
    List<Edge> adj;
    boolean visited;

    void visit() {
        visited = true;
        for (Edge e : adj) {
            Vertex w = e.dest;
            if (!w.visited)
                w.visit();
        }
    }
}
```

Time complexity: $O(|E|)$

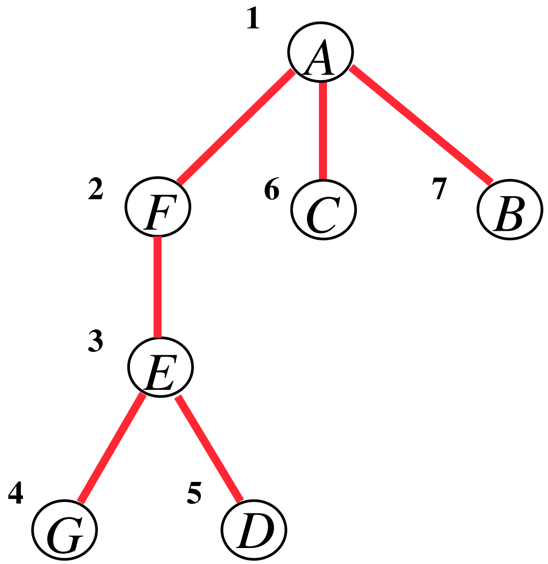
Depth-first traversal of a component

A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A





Depth-first traversal of a component results in a depth-first tree



A depth-first traversal of a connected graph represented by adjacency lists requires $O(|E|)$ time

Non-recursive depth-first traversal

Use an explicit stack of vertices.

```
void traverse(Vertex startVertex) {
    Stack<Vertex> stack = new Stack<Vertex>();
    stack.push(startVertex);
    startVertex.visited = true;
    while (!stack.empty()) {
        Vertex v = stack.pop();
        for (Edge e : v.adj) {
            Vertex w = e.dest;
            if (!w.visited) {
                stack.push(w);
                w.visited = true;
            }
        }
    }
}
```

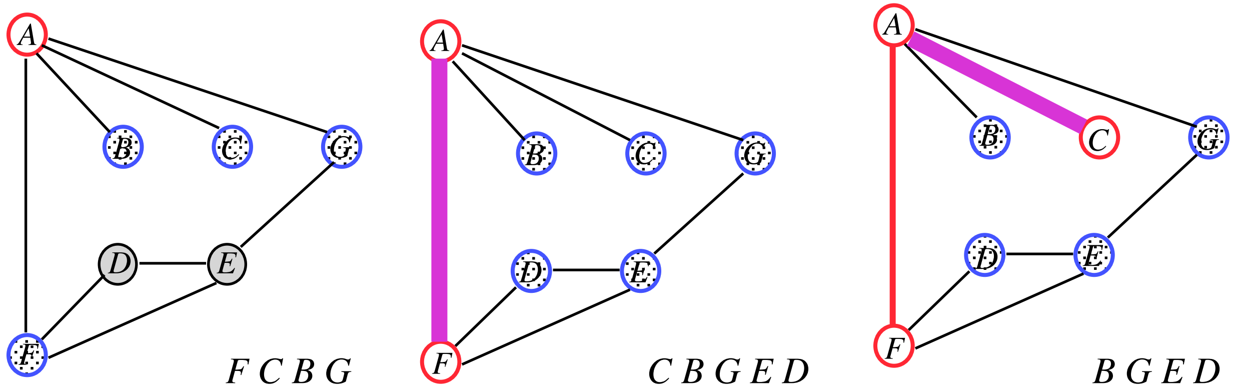
Breadth-first traversal

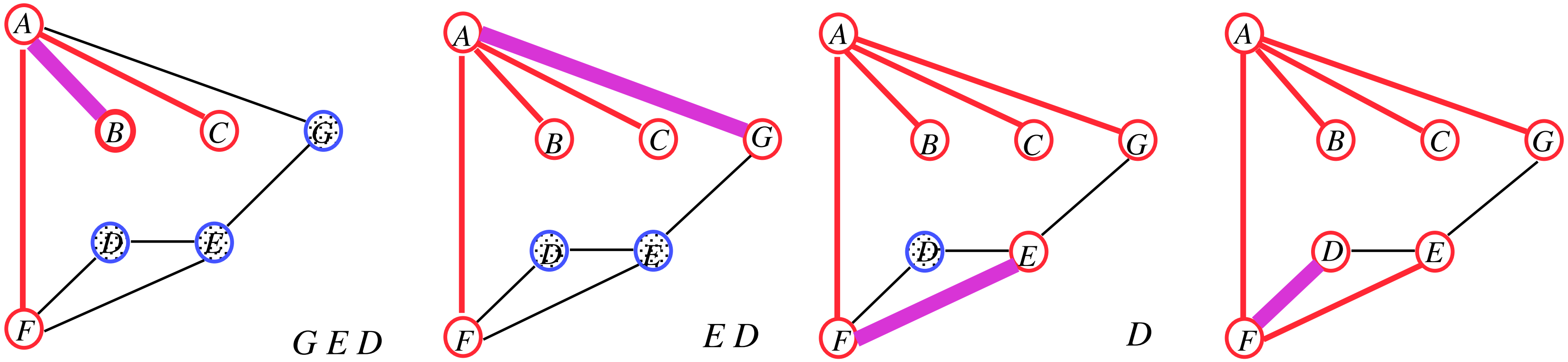
If the stack is replaced by a queue, the graph will be traversed in *breadth-first order* (level order).

```
void traverse(Vertex startVertex) {
    Queue<Vertex> queue = new LinkedList<>();
    queue.add(startVertex);
    startVertex.visited = true;
    while (!queue.isEmpty()) {
        Vertex v = queue.remove();
        for (Edge e : v.adj) {
            Vertex w = e.dest;
            if (!w.visited) {
                queue.add(w);
                w.visited = true;
            }
        }
    }
}
```

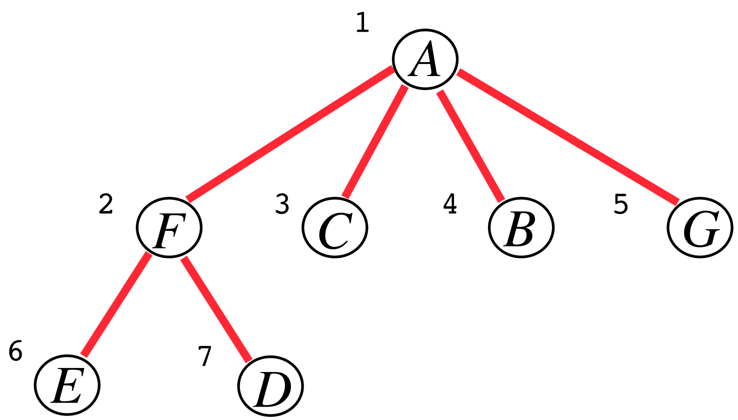
Breadth-first traversal of a component

A: *F C B G*
B: *A*
C: *A*
D: *F E*
E: *G F D*
F: *A E D*
G: *E A*



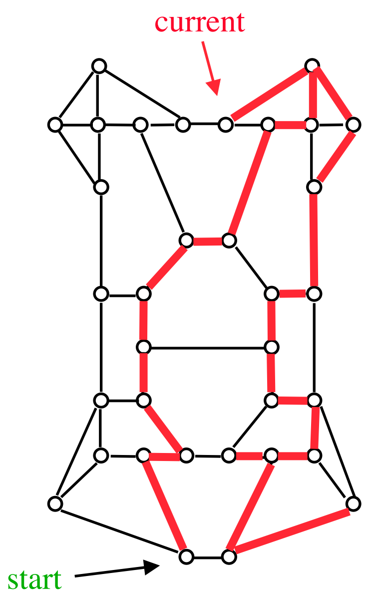


Breadth-first traversal of a component results in a breadth-first tree

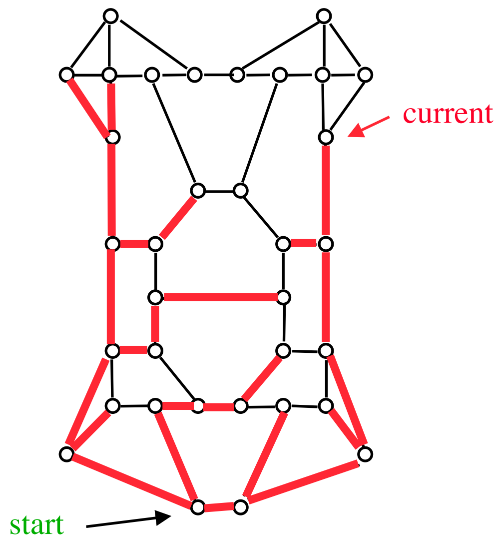


A breadth-first traversal of a connected graph represented by adjacency lists requires $O(|E|)$ time

Depth-first traversal versus breadth-first traversal



Depth-first



Breadth-first

Best-first traversal

If the queue is replaced by a priority queue, the graph will be traversed in *best-first order*.

```
Queue<Vertex> queue = new PriorityQueue<>();
```

Class `Vertex` should implement the `Comparable` interface, or the priority queue should rely on a supplied `Comparator` object.

$O(|E|)$ insertions and $O(|V|)$ removals; each takes $O(\log|V|)$ time for a heap-based priority queue.

Time complexity: $O((|V|+|E|)\log|V|)$

Shortest paths

The shortest path problem

Find the shortest path from vertex *A* to vertex *B*

Unweighted shortest path (minimize the number of edges):

Use **breadth-first** traversal.

Traverse the graph starting at *A*, using a *queue*.

Weighted shortest path (find the "cheapest" path):

Use **best-first** traversal (**Dijkstra's algorithm**):

Traverse the graph starting at *A*, using a *priority queue*.

The priority of each unvisited vertex is the cost of the currently cheapest path from *A* to that vertex.

Works only for graphs with non-negative weights.

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).

Result

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

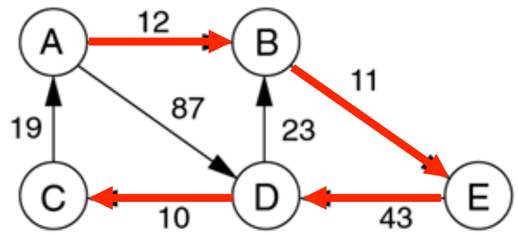
Input

	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

Starting vertex

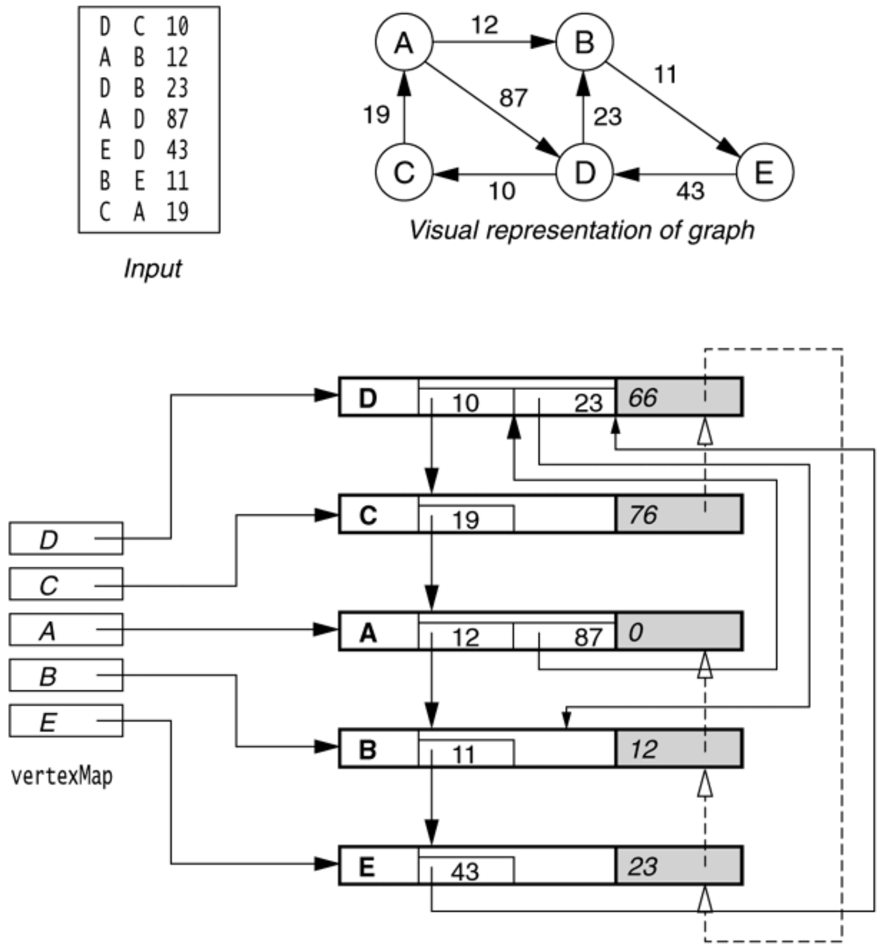
Goal vertex



Visual representation of graph

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

Dictionary



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.

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).

Class `Edge`

```
// 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;
    }
}
```

figure 14.6

The basic item stored in an adjacency list

Class `Vertex`

```
// Represents a vertex in the graph.
class Vertex
{
    public String     name;   // Vertex name
    public List<Edge> 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<Edge>( ); reset( ); }

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

figure 14.7

The `Vertex` class stores information for each vertex

```java
// 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
// ******************ERRORS*********************************
// Some error checking is performed to make sure that graph is ok
// and that graph satisfies properties needed by each
// algorithm.  Exceptions are thrown if errors are detected.

public class Graph
{
    public static final double INFINITY = Double.MAX_VALUE;

    public void addEdge( String sourceName, String destName, double cost )
      { /* Figure 14.10 */ }
    public void printPath( String destName )
      { /* Figure 14.13 */ }
    public void unweighted( String startName )
      { /* Figure 14.22 */ }
    public void dijkstra( String startName )
      { /* Figure 14.27 */ }
    public void negative( String startName )
      { /* Figure 14.29 */ }
    public void acyclic( String startName )
      { /* Figure 14.32 */ }

    private Vertex getVertex( String vertexName )
      { /* Figure 14.9 */ }
    private void printPath( Vertex dest )
      { /* Figure 14.12 */ }
    private void clearAll( )
      { /* Figure 14.11 */ }

    private Map<String,Vertex> vertexMap = new HashMap<String,Vertex>( );
}

// Used to signal violations of preconditions for
// various shortest path algorithms.
class GraphException extends RuntimeException
{
    public GraphException( String name )
      { super( name ); }
}
```

Shortest-path algorithms

figure 14.8

The Graph class skeleton

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

figure 14.9

The `getVertex` routine returns the `Vertex` object that represents `vertexName`, creating the object if it needs to do so

```java
/**
 * 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 ) );
}
```

figure 14.10

Add an edge to the graph

figure 14.11

Private routine for initializing the output members for use by the shortest-path algorithms

```java
/**
 * Initializes the vertex output info prior to running
 * any shortest path algorithm.
 */
private void clearAll( )
{
    for( Vertex v : vertexMap.values( ) )
        v.reset( );
}
```

figure 14.12

A recursive routine for printing the shortest path

```
/**
 * 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 );
}
```

figure 14.13

A routine for printing the shortest path by consulting the graph table (see Figure 14.5)

```
/**
 * Driver routine to handle unreachables and print total cost.
 * It calls recursive routine to print shortest path to
 * destNode after a shortest path algorithm has run.
 */
public void printPath( String destName )
{
    Vertex w = vertexMap.get( destName );
    if( w == null )
        throw new NoSuchElementException( );
    else if( w.dist == INFINITY )
        System.out.println( destName + " is unreachable" );
    else
    {
        System.out.print( "(Cost is: " + w.dist + ") " );
        printPath( w );
        System.out.println( );
    }
}
```

```
/**
 * A main routine that
 * 1. Reads a file (supplied as a command-line parameter)
 *    containing edges.
 * 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.
 */
public static void main( String [ ] args )
{
    Graph g = new Graph( );
    try
    {
        FileReader fin = new FileReader( args[0] );
        BufferedReader graphFile = new BufferedReader( fin );

        // Read the edges and insert
        String line;
        while( ( line = graphFile.readLine( ) ) != null )
        {
            StringTokenizer st = new StringTokenizer( line );

            try
            {
                if( st.countTokens( ) != 3 )
                {
                    System.err.println( "Skipping bad line " + line );
                    continue;
                }
                String source  = st.nextToken( );
                String dest    = st.nextToken( );
                int    cost    = Integer.parseInt( st.nextToken( ) );
                g.addEdge( source, dest, cost );
            }
            catch( NumberFormatException e )
              { System.err.println( "Skipping bad line " + line ); }
        }
     }
     catch( IOException e )
       { System.err.println( e ); }

     System.out.println( "File read..." );
     System.out.println( g.vertexMap.size( ) + " vertices" );

     BufferedReader in = new BufferedReader(
                         new InputStreamReader( System.in ) );
     while( processRequest( in, g ) )
         ;
}
```

figure 14.14

A simple `main`

Input format:
source_name dest_name cost

```java
/**
 * Process a request; return false if end of file.
 */
public static boolean processRequest( BufferedReader in, Graph g )
{
    String startName = null;
    String destName = null;
    String alg = null;

    try
    {
        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;
        System.out.print( " Enter algorithm (u, d, n, a ): " );
        if( ( alg = in.readLine( ) ) == null )
            return false;

        if( alg.equals( "u" ) )
            g.unweighted( startName );
        else if( alg.equals( "d" ) )
            g.dijkstra( startName );
        else if( alg.equals( "n" ) )
            g.negative( startName );
        else if( alg.equals( "a" ) )
            g.acyclic( startName );

        g.printPath( destName );
    }
    catch( IOException e )
      { System.err.println( e ); }
    catch( NoSuchElementException e )
      { System.err.println( e ); }
    catch( GraphException e )
      { System.err.println( e ); }
    return true;
}
```

figure 14.15

For testing purposes, `processRequest` calls one of the shortest-path algorithms

Unweighted shortest path

(breadth-first traversal)

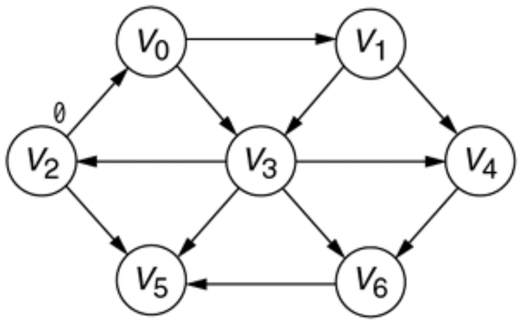


figure 14.16

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

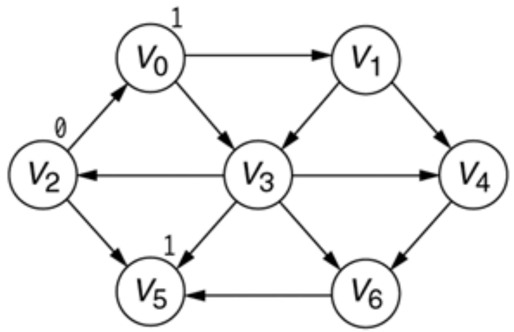


figure 14.18

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

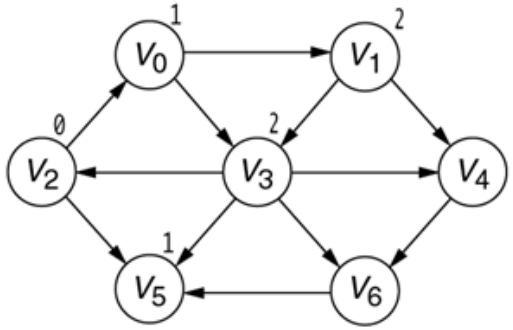
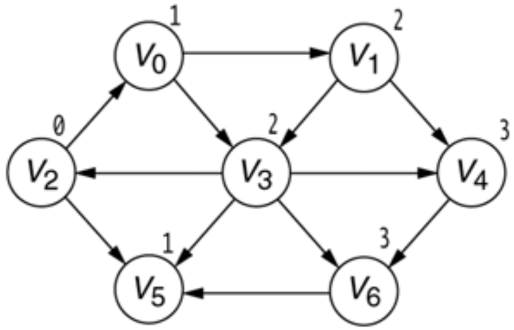


figure 14.19

The final shortest paths



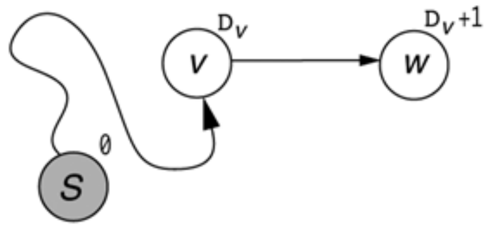


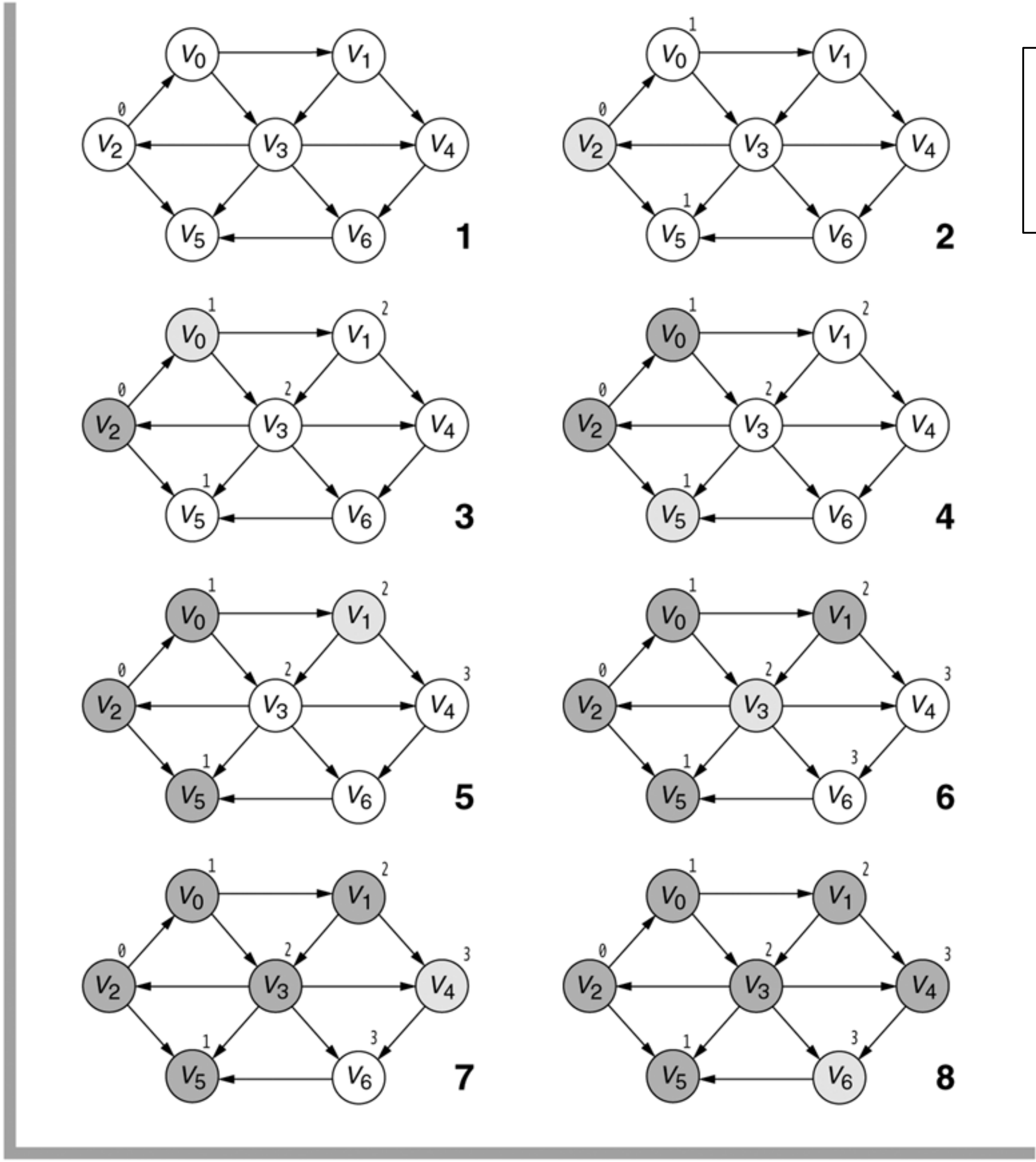
figure 14.20

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

(of cost $D_w = D_v+1$)

figure 14.21

Searching the graph in the unweighted shortest-path computation. The darkest-shaded vertices have already been completely processed, the lightest 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.



We maintain a *roving eyeball* that hops from vertex to vertex and is initially at V_2.

Roving eyeball
da. strejfende øjeæble

```java
/**
 * Single-source unweighted shortest-path algorithm.
 */
public void unweighted( String startName )
{
    clearAll( );

    Vertex start = vertexMap.get( startName );
    if( start == null )
        throw new NoSuchElementException( "Start vertex not found" );

    Queue<Vertex> q = new LinkedList<Vertex>( );
    q.add( start ); start.dist = 0;

    while( !q.isEmpty( ) )
    {
        Vertex v = q.remove( );

        for( Edge e : v.adj )
        {
            Vertex w = e.dest;

            if( w.dist == INFINITY )
            {
                w.dist = v.dist + 1;
                w.prev = v;
                q.add( w );
            }
        }
    }
}
```

figure 14.22

The unweighted shortest-path algorithm, using breadth-first search

Time complexity: $O(|E|)$

Positive weighted shortest path

(Dijkstra's algorithm, 1959)

E. W. Dijkstra, 1930-2002

For a given source vertex in the graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex.

It can also be used for finding costs of shortest paths from a single vertex to a single destination vertex by stopping the algorithm once the shortest path to the destination vertex has been determined.

Dijkstra’s algorithm

Let the node at which we are starting be called the *initial node*. Let the *distance of node Y* be the distance from the initial node to *Y*.

1. Assign to every node a distance value. Set it to zero for our initial node and to infinity for all other nodes.
2. Mark all nodes as unvisited. Set initial node as current.
3. For the current node, consider all its unvisited neighbors and calculate their *tentative* distance (from the initial node). If this distance is less than the previously recorded distance (infinity in the beginning, zero for the initial node), overwrite the distance.
4. When we are done considering all neighbors of the current node, mark it as visited. A visited node will not be checked ever again; its distance recorded now is final and minimal.
5. If all nodes have been visited, finish. Otherwise, set the unvisited node with the smallest distance (from the initial node) as the next “current node” and continue from step 3.

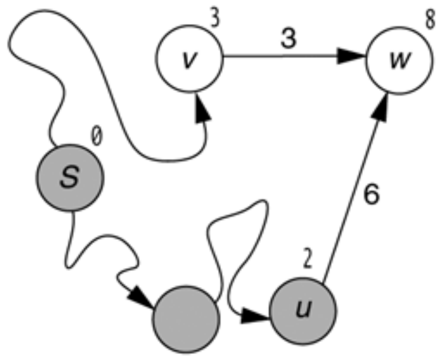
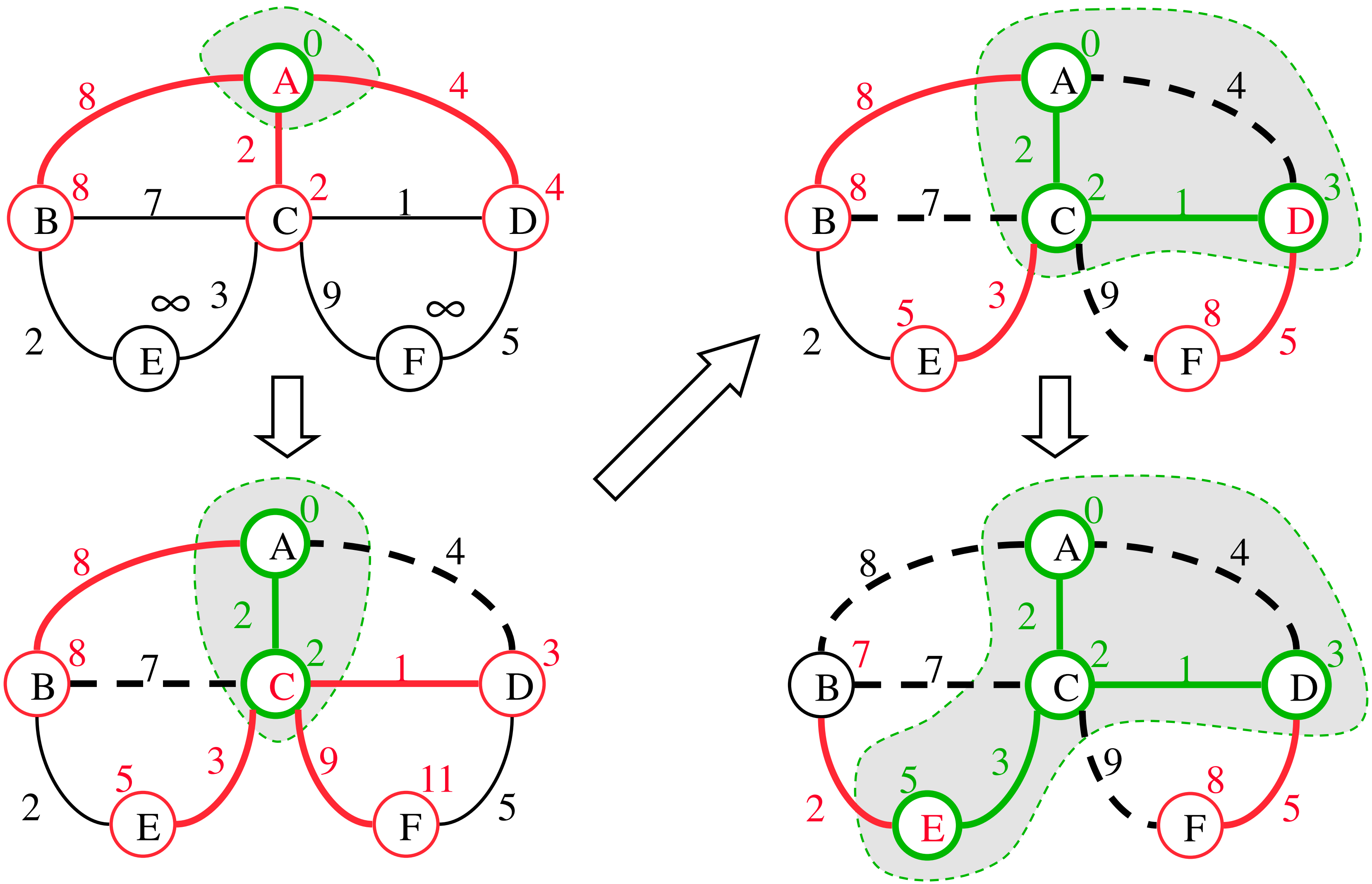


figure 14.23

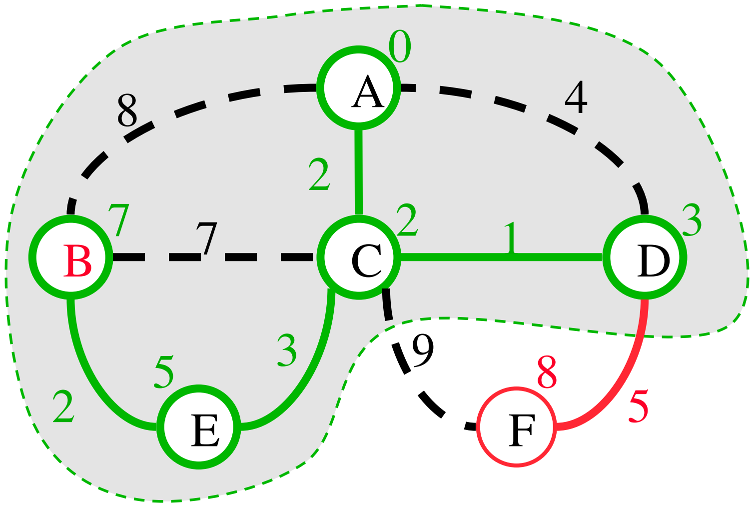
The eyeball is at v and w is adjacent, so D_w should be lowered to 6.

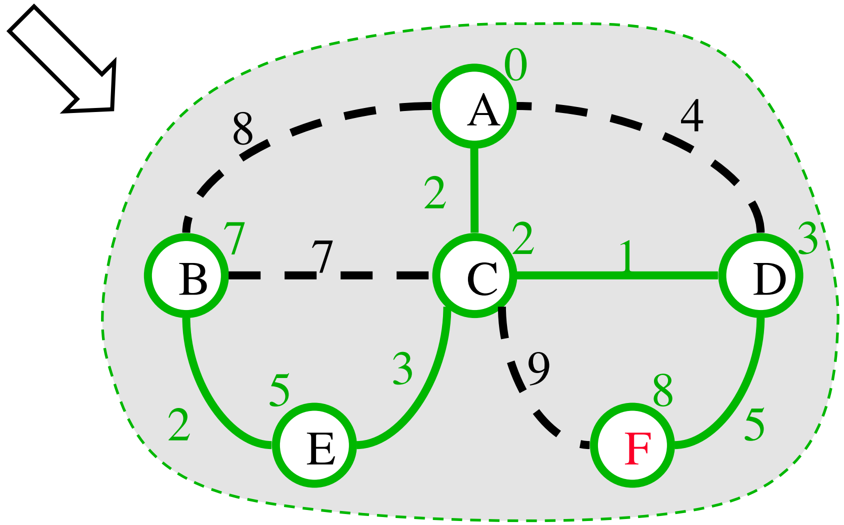
Example



0
A
8
4
2
8
7
2
1
4
B
C
D
∞
3
9
∞
2
5
E
F
0
A
8
4
2
8
7
2
1
3
B
C
D
5
3
9
11
2
5
E
F
0
A
8
4
2
8
7
2
1
3
B
C
D
5
3
9
8
2
5
E
F
0
A
8
4
2
7
7
2
1
3
B
C
D
5
3
9
8
2
5
E
F

Example continued





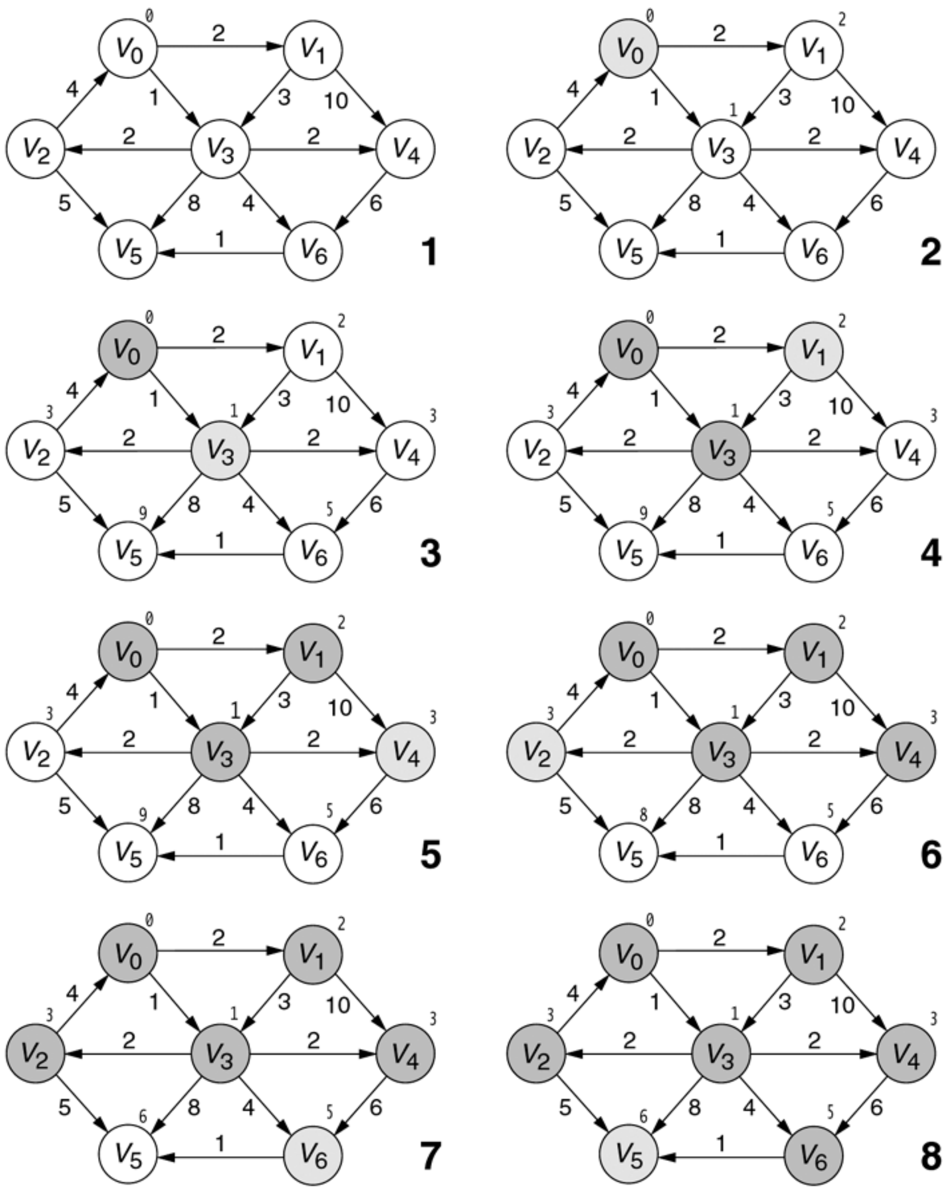
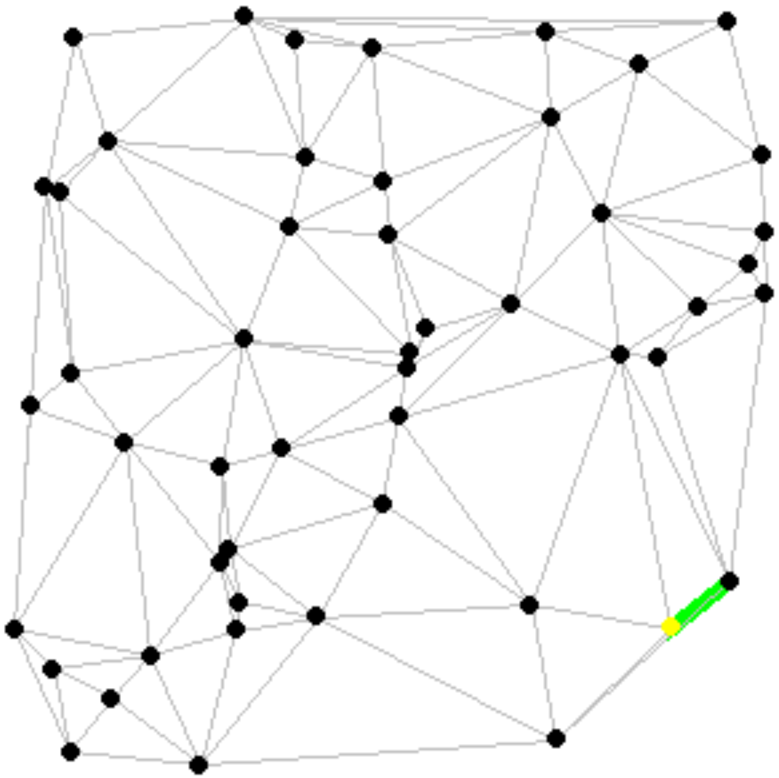


figure 14.25

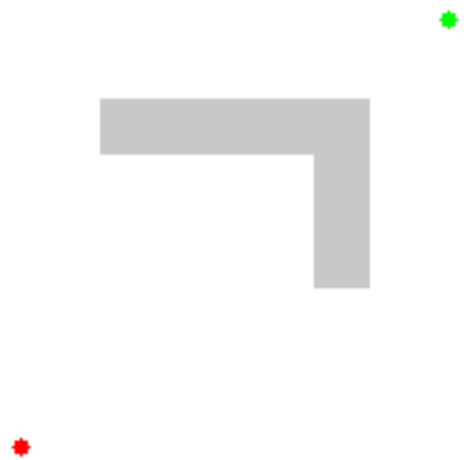
Stages of Dijkstra's algorithm. The conventions are the same as those in Figure 14.21.

Dijkstra's algorithm



www.combinatorica.com

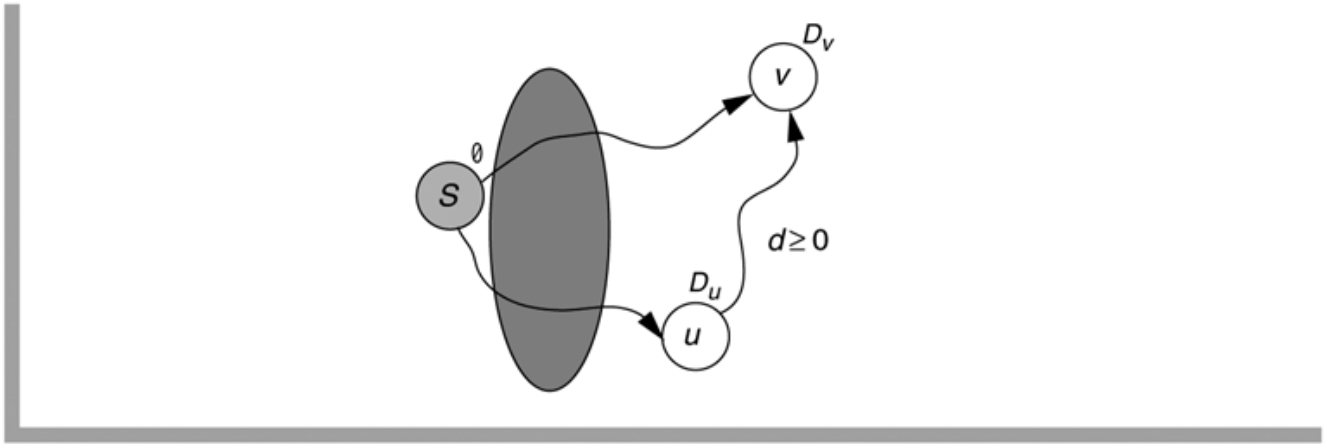
Dijkstra’s algorithm used for solving a robot planning problem



Proof of Dijkstra's algorithm

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.



Suppose there is a path from S to v of length less than D_v.
This path must go through a vertex u that has not yet been visited.
But since the length of the path from S to u, D_u, is less than D_v, we would have chosen u instead of v. Hence we have a contradiction.

Implementation of Dijkstra's algorithm

```
void dijkstra(Vertex startVertex) {
    clearAll();
    PriorityQueue<Vertex> pq = new PriorityQueue<>();
    pq.add(startVertex); startVertex.dist = 0;
    while (!pq.isEmpty()) {
        Vertex v = pq.remove();
        for (Edge e : v.adj) {
            Vertex w = e.dest;
            if (v.dist + e.cost < w.dist) {
                w.dist = v.dist + e.cost;
                w.prev = v;
                pq.update(w); // error: no such method!
            }
        }
    }
}
```

`pq.update(w)`: If `w` is not in `pq`, then add `w` to `pq`; otherwise, update `pq` by reestablishing its ordering property. Unfortunately, the `update` method is not available in Java's `PriorityQueue`.

Class `Path`

```java
// Represents an entry in the priority queue for Dijkstra's algorithm.
class Path implements Comparable<Path>
{
    public Vertex     dest;   // w
    public double     cost;   // d(w)

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

    public int compareTo( Path rhs )
    {
        double otherCost = rhs.cost;

        return cost < otherCost ? -1 : cost > otherCost ? 1 : 0;
    }
}
```

figure 14.26

Basic item stored in the priority queue

```java
/**
 * Single-source weighted shortest-path algorithm.
 */
public void dijkstra( String startName )
{
    PriorityQueue<Path> pq = new PriorityQueue<Path>( );

    Vertex start = vertexMap.get( startName );
    if( start == null )
        throw new NoSuchElementException( "Start vertex not found" );

    clearAll( );
    pq.add( new Path( start, 0 ) ); start.dist = 0;

    int nodesSeen = 0;
    while( !pq.isEmpty( ) && nodesSeen < vertexMap.size( ) )
    {
        Path vrec = pq.remove( );
        Vertex v = vrec.dest;
        if( v.scratch != 0 )  // already processed v
            continue;

        v.scratch = 1;
        nodesSeen++;

        for( Edge e : v.adj )
        {
            Vertex w = e.dest;
            double cvw = e.cost;

            if( cvw < 0 )
                throw new GraphException( "Graph has negative edges" );

            if( w.dist > v.dist + cvw )
            {
                w.dist = v.dist + cvw;
                w.prev = v;
                pq.add( new Path( w, w.dist ) );
            }
        }
    }
}
```

figure 14.27

A positive-weighted, shortest-path algorithm: Dijkstra's algorithm

Time complexity:
$O(|E| \cdot \log|V|)$

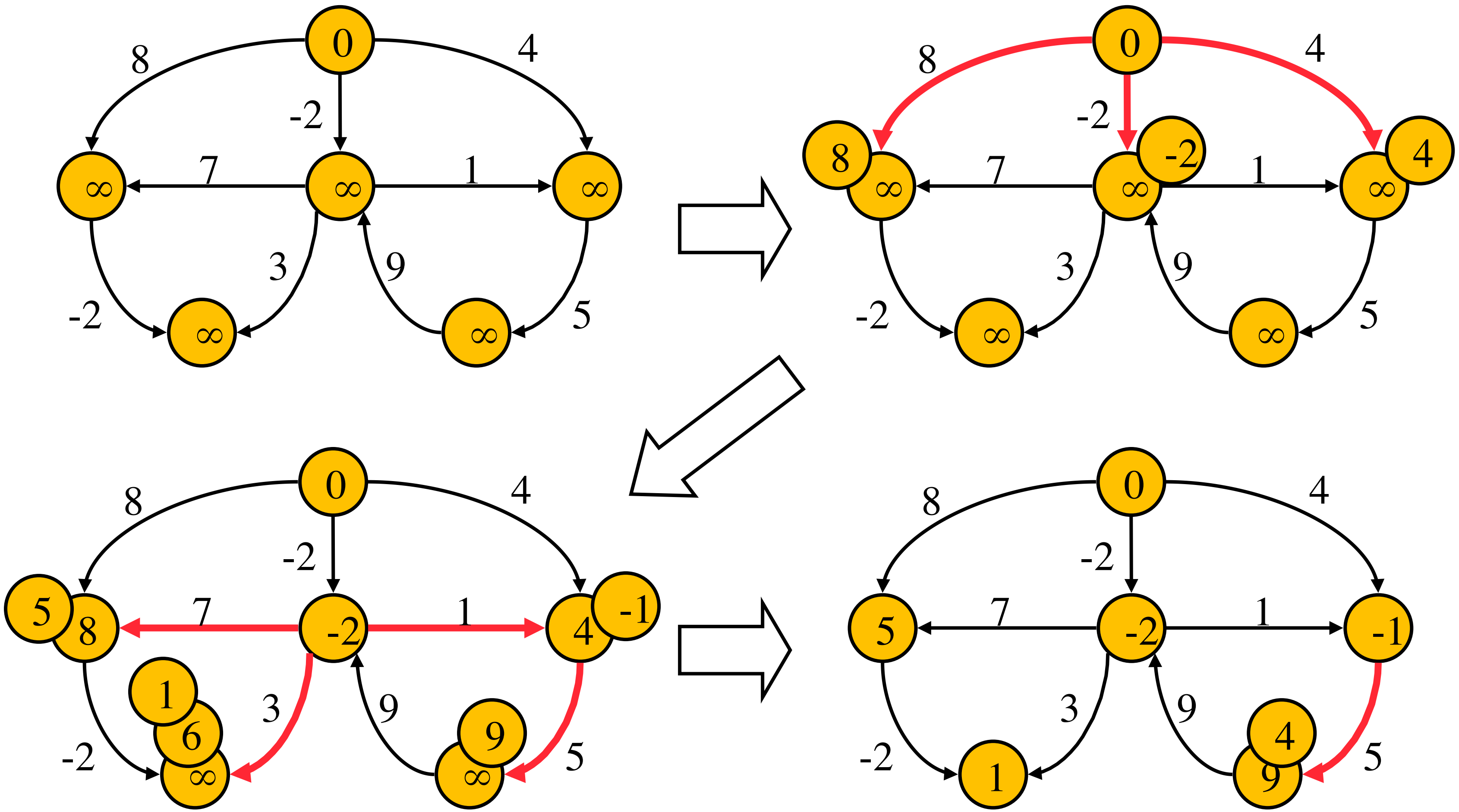
Negative-weighted shortest path

(The Bellman-Ford algorithm, 1958)

```
void bellmanFord(Vertex startVertex) {
     clearAll();
     startVertex.dist = 0;
     Collection<Vertex> vertices = vertexMap.values();
     for (int i = 1; i < vertices.size(); i++) {
         for (Vertex v : vertices) {
             for (Edge e : v.adj) {
                 Vertex w = e.dest;
                 if (v.dist + e.cost < w.dist) {
                     w.dist = v.dist + e.cost;
                     w.prev = v;
                 }
             }
         }
     }
}
```

Iteration i finds all shortest paths from `startVertex` that uses i or fewer edges.
Time complexity: $O(|E|\cdot|V|)$

Bellman-Ford example



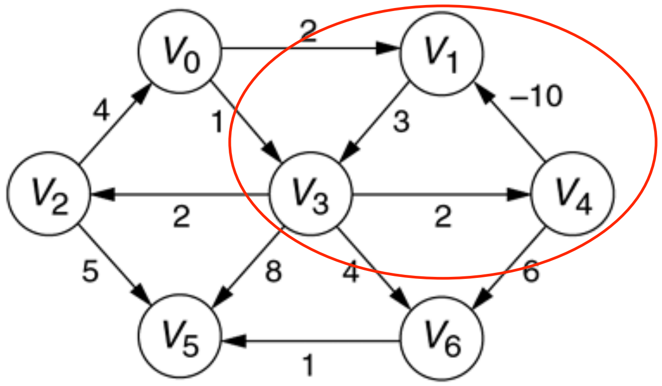


figure 14.28

A graph with a negative-cost cycle

Check for negative-cost cycles (add this code after the loop):

```
for (Vertex v : vertices) {
    for (Edge e : v.adj) {
        Vertex w = e.dest;
        if (v.dist + e.cost < w.dist)
            error("Negative-cost cycle detected");
    }
}
```

```
/**
 * Single-source negative-weighted shortest-path algorithm.
 */
public void negative( String startName )
{
    clearAll( );

    Vertex start = vertexMap.get( startName );
    if( start == null )
        throw new NoSuchElementException( "Start vertex not found" );

    Queue<Vertex> q = new LinkedList<Vertex>( );
    q.add( start ); start.dist = 0; start.scratch++;

    while( !q.isEmpty( ) )
    {
        Vertex v = q.removeFirst( );
        if( v.scratch++ > 2 * vertexMap.size( ) )
            throw new GraphException( "Negative cycle detected" );

        for( Edge e : v.adj )
        {
            Vertex w = e.dest;
            double cvw = e.cost;

            if( w.dist > v.dist + cvw )
            {
                w.dist = v.dist + cvw;
                w.prev = v;
                  // Enqueue only if not already on the queue
                if( w.scratch++ % 2 == 0 )
                    q.add( w );
                else
                    w.scratch--;  // undo the enqueue increment
            }
        }
    }
}
```

figure 14.29

A negative-weighted, shortest-path algorithm: Negative edges are allowed.

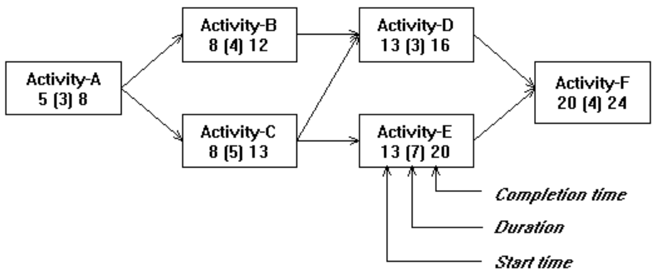
`v.scratch` is odd when vertex `v` is on the queue. `v.scratch/2` tells us how many times `v` has left the queue.

An edge can dequeue at most $O(|V|)$ times.
Time *complexity*: $O(|E|\cdot|V|)$

DAGs

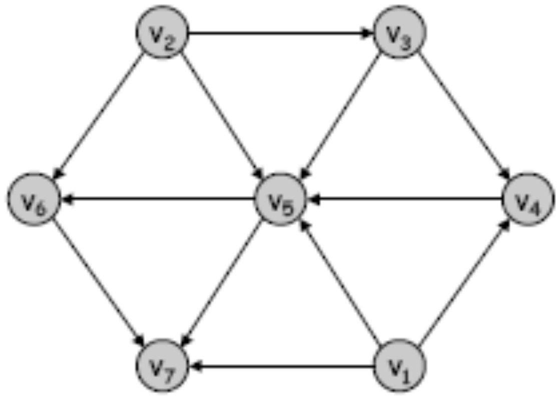
An oriented graph without cycles is called a **DAG** (**D**irected **A**cyclic **G**raph).

A DAG may, for instance, be used for modeling an activity network. Directed edges are used to specify that some activities must be finished before an activity can start.

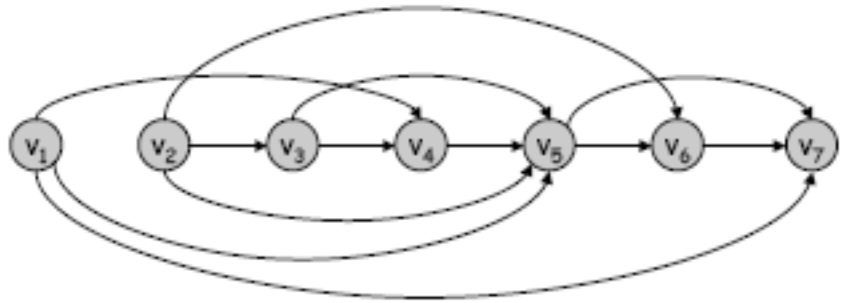


Topological sorting

The vertices of a DAG can be ordered so that if there is a path from *u* to *v*, then *v* appears after *u* in the ordering. This is called a **topological sort** of the graph.



a DAG



a topological ordering

Topological ordering: All directed edges point from left to right
[not necessarily unique]

A topological sorting algorithm

(1) Create an empty queue

(2) Choose a vertex without any ingoing edges

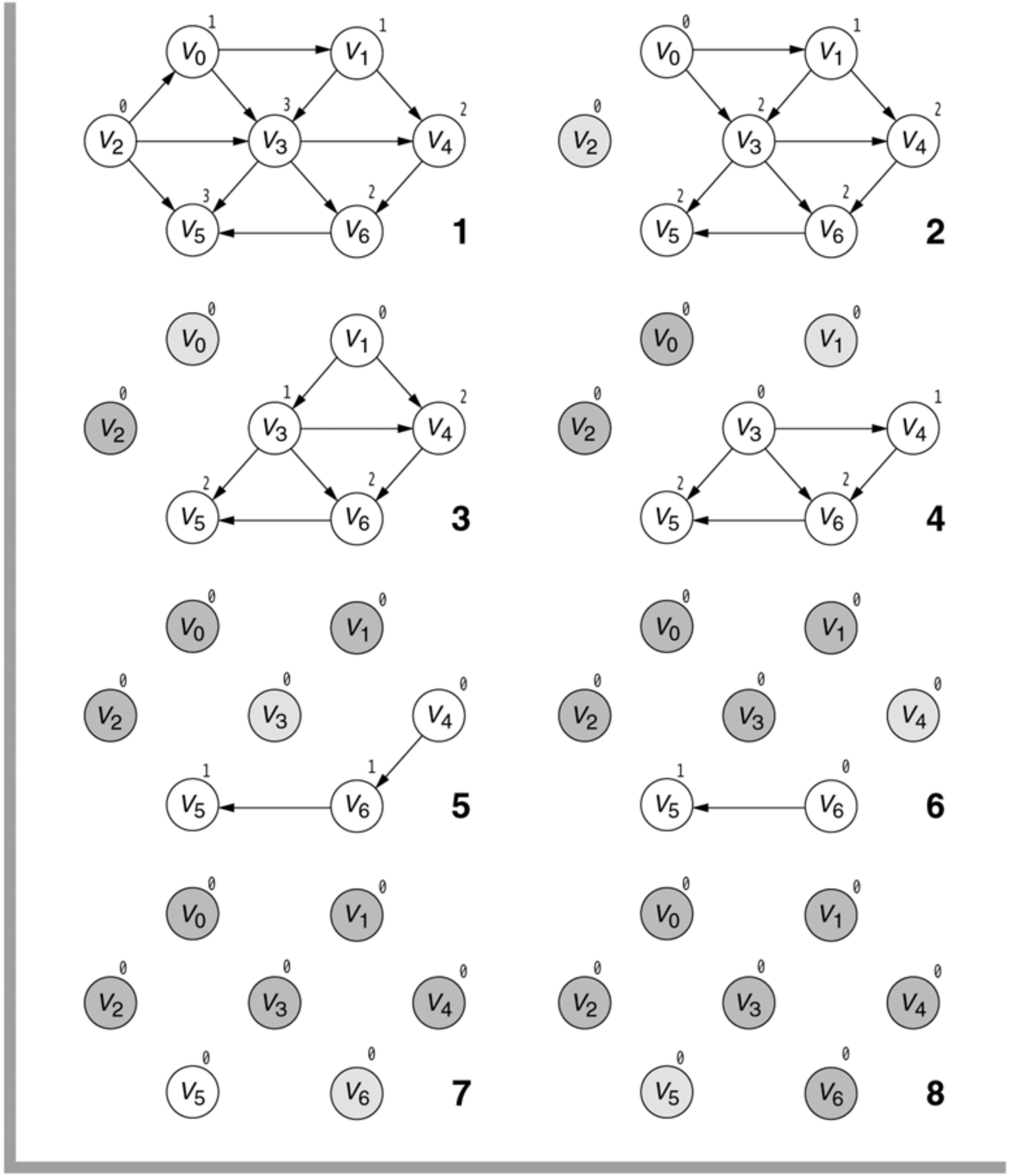
(3) Insert the vertex in the queue. Remove the vertex and all outgoing edges from the graph.

(4) Repeat (2) and (3) while the graph is not empty

Now the queue contains the vertices in topological order

figure 14.30

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



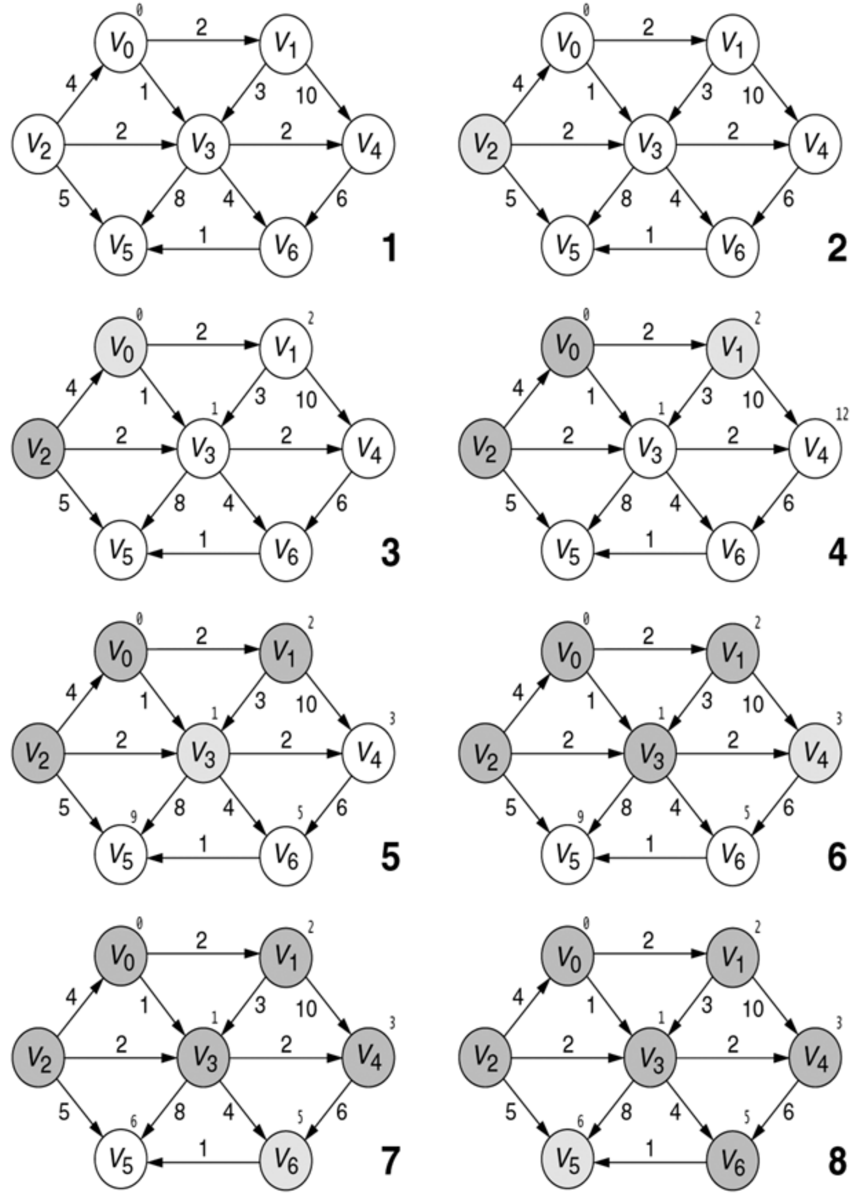
$V_2\ V_0\ V_1\ V_3\ V_4\ V_6\ V_5$

Java implementation

```
List<Vertex> tologicalOrder() {
    Collection<Vertex> vertices = vertexMap.values();
    for (Vertex v : vertices)
        v.scratch = 0;        // v's indegree = 0
    for (Vertex v : vertices)
        for (Edge e : v.adj)
            e.dest.scratch++;
    Queue<Vertex> q = new LinkedList<>();
    for (Vertex v : vertices)
        if (v.scratch == 0)
            q.add(v);
    List<Vertex> result = new ArrayList<>();
    int iterations = 0;
    while (!q.isEmpty() && ++iterations <= vertices.size()) {
        Vertex v = q.remove();
        result.add(v);
        for (Edge e : v.adj)
            if (--e.dest.scratch == 0)
                q.add(e.dest);
    }
    return iterations == vertices.size() ? result : null;
}
```

figure 14.31

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



Shortest path for a DAG

Visit order:
$V_2\ V_0\ V_1\ V_3\ V_4\ V_6\ V_5$

```
/**
 * Single-source negative-weighted acyclic-graph shortest-path algorithm.
 */
public void acyclic( String startName )
{
    Vertex start = vertexMap.get( startName );
    if( start == null )
        throw new NoSuchElementException( "Start vertex not found" );

    clearAll( );
    Queue<Vertex> q = new LinkedList<Vertex>( );
    start.dist = 0;

      // Compute the indegrees
    Collection<Vertex> vertexSet = vertexMap.values( );
    for( Vertex v : vertexSet )
        for( Edge e : v.adj )
            e.dest.scratch++;

      // Enqueue vertices of indegree zero
    for( Vertex v : vertexSet )
        if( v.scratch == 0 )
            q.add( v );

    int iterations;
    for( iterations = 0; !q.isEmpty( ); iterations++ )
    {
        Vertex v = q.remove( );

        for( Edge e : v.adj )
        {
            Vertex w = e.dest;
            double cvw = e.cost;

            if( --w.scratch == 0 )
                q.add( w );

            if( v.dist == INFINITY )
                continue;

            if( w.dist > v.dist + cvw )
            {
                w.dist = v.dist + cvw;
                w.prev = v;
            }
        }
    }

    if( iterations != vertexMap.size( ) )
        throw new GraphException( "Graph has a cycle!" );
}
```

figure 14.32

A shortest-path algorithm for acyclic graphs

Uses topological sort

Time complexity: $O(|E|)$

Complexity of shortest path algorithms

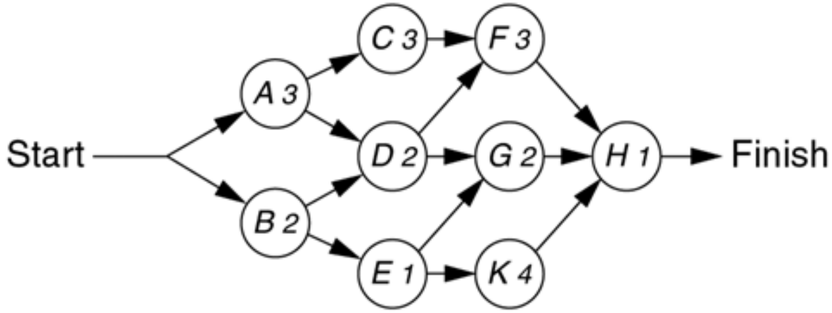
Type of Graph Problem	Running Time	Comments
Unweighted	$O(\|E\|)$	Breadth-first search
Weighted, no negative edges	$O(\|E\| \log \|V\|)$	Dijkstra's algorithm
Weighted, negative edges	$O(\|E\| \cdot \|V\|)$	Bellman–Ford algorithm
Weighted, acyclic	$O(\|E\|)$	Uses topological sort

figure 14.38

Worst-case running times of various graph algorithms

figure 14.33

An activity-node graph



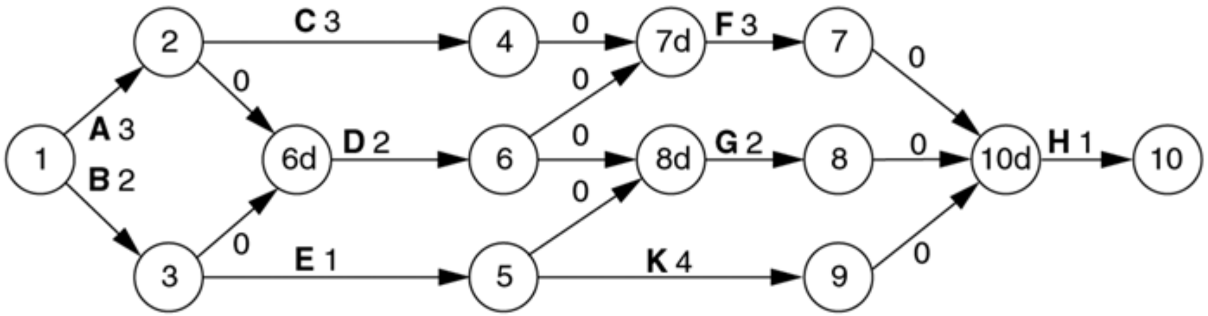


figure 14.34

An event-node graph

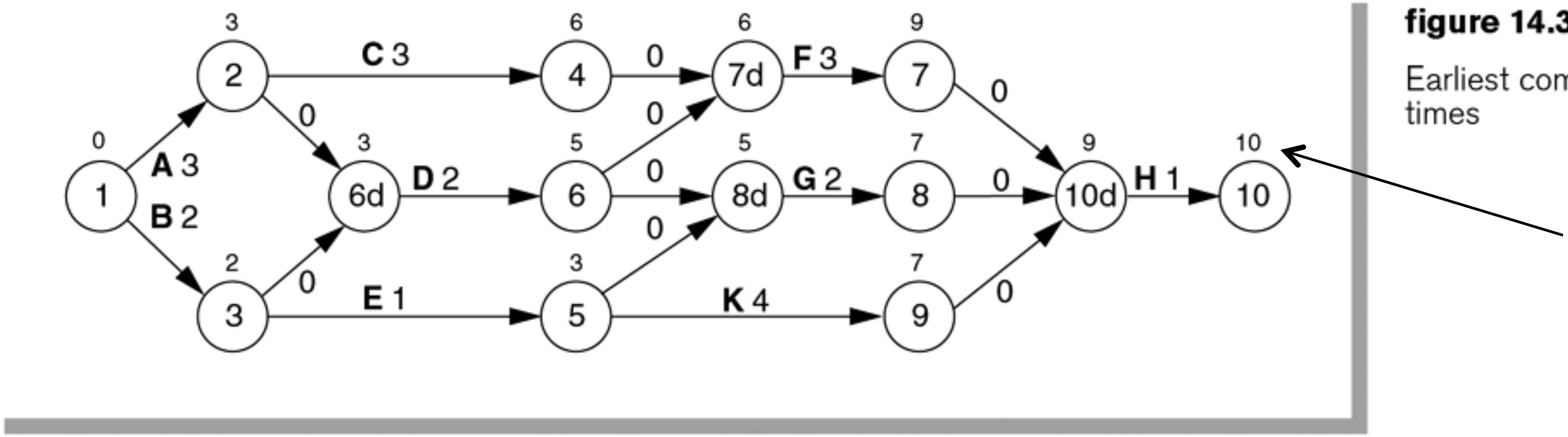
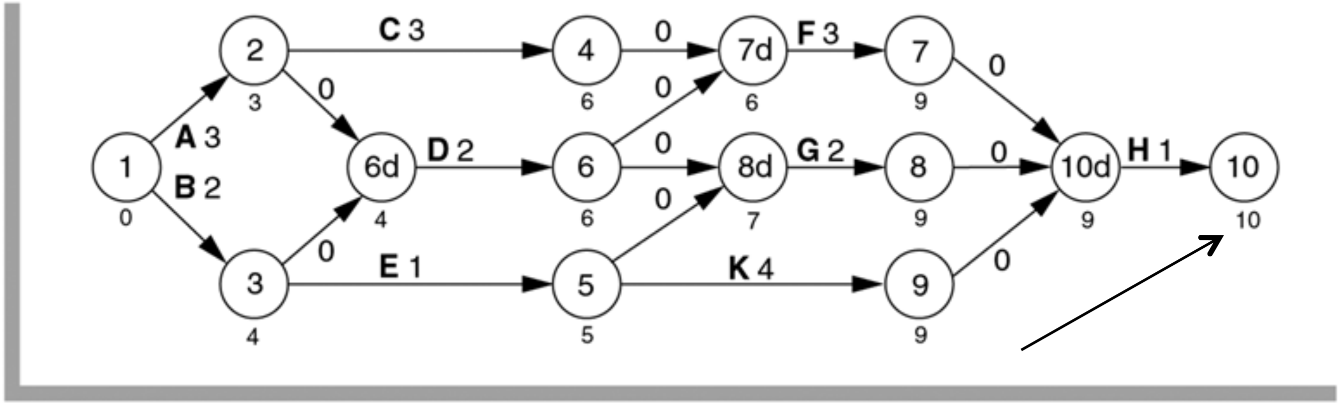


figure 14.35

Earliest completion times

figure 14.36

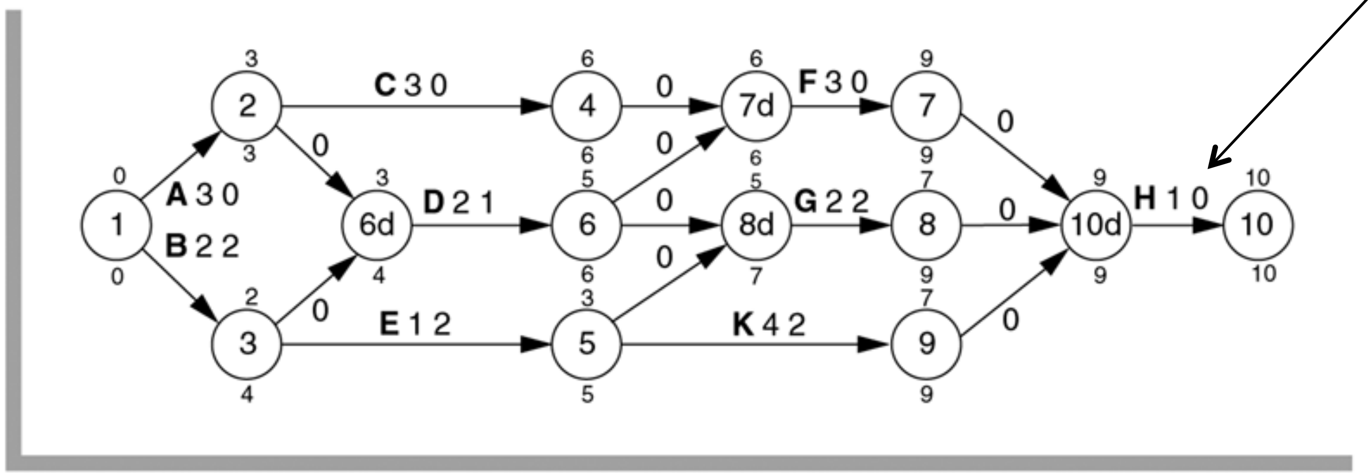
Latest completion times



slack

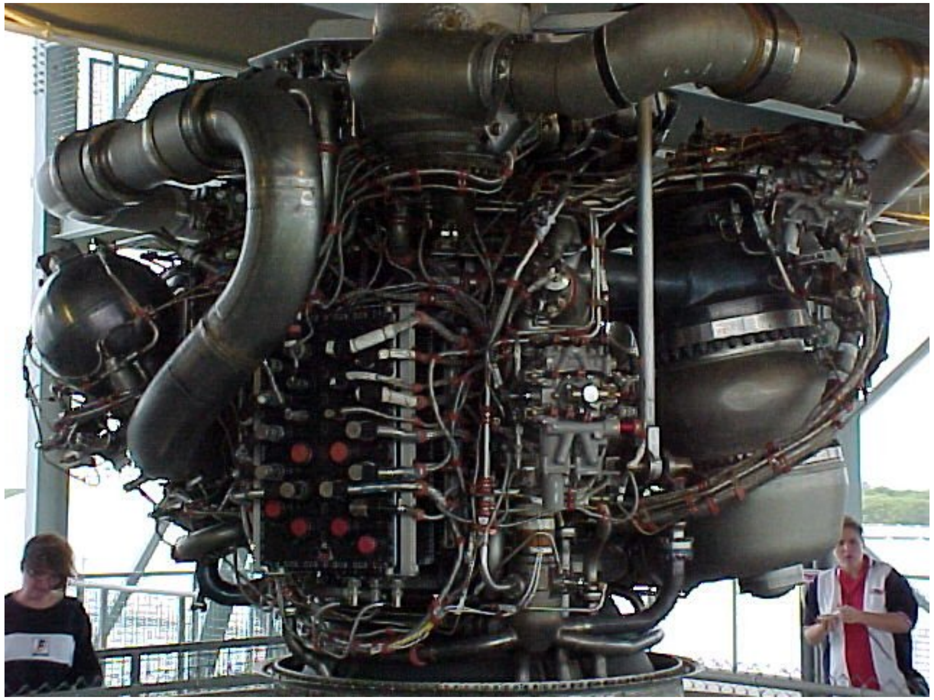
figure 14.37

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



Some activities have zero slack. These are critical activities that must be finished on schedule. A path consisting entirely of zero-slack edges is a **critical path**.

Problem complexity



Problem complexity

For a large class of important problems no fast solution algorithms are known.

An **efficient algorithm**: running time is limited by some polynomial
[$O(n^c)$]

A problem that can be solved by a efficient algorithm is said to be **easy**.

An **inefficient** algorithm: running time grows at least exponentially
[$\Omega(c^n)$]

A problem is said to be **hard** or intractable if there does not exist a polynomial-time algorithm for solving the problem.

Examples of hard problems

- **The traveling salesman problem**
  A salesman must visit N cities. Find a travel route that minimizes his costs.

- **Job scheduling**
  A number of jobs of varying duration are to be executed on two identical machines before a given deadline. Is it possible to meet the deadline?

- **Satisfiability**
  Is it possible to determine if the variables in a Boolean expression can be assigned in such a way as to make the expression evaluate to true?

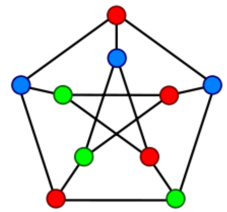
$$(a \vee b) \wedge (\neg a \vee b)$$

More examples of hard problems

- **Longest path**
  Find the longest simple path between two vertices of a graph.

- **Partitioning**
  Given at set of integers. Is it possible to partition the set into two subsets so that the sum of the elements in each of the two subsets is the same?

- **3-coloring**
  Is it possible to color the vertices of a graph by only three colors such that no two adjacent vertices have the same color?



NP-complete problems

For none of these problems do we know an algorithm that solves the problem in polynomial time.

All experts are convinced that such algorithms do not exist. However, this has not yet been proved.

The problems belong to the class of problems called **NP-complete** problems.

NP-completeness

An NP-complete problem is a problem that can be solved in *polynomial* time on a **nondeterministic** machine.

A nondeterministic machine has the wonderful ability to make the correct choice in any situation where a choice is to be made.

A usual deterministic machine may be used to simulate correct choices in exponential time by trying each possible choice.

If only one NP-complete problem can be solved in polynomial time, every NP-complete problem can be solved in polynomial time.

Decidability

Undecidable problems are decision problems which no algorithm can decide.

Examples:

- Prove that an algorithm always terminates (the stop problem)
- Decide if a formula in the predicate logic is valid
- Decide if two syntax descriptions define the same language

Termination?

(a)

```
while (x != 1)
    x = x - 2;
```

(b)

```
while (x != 1)
    if (x % 2 == 0)
        x = x / 2;
    else
        x = 3 * x + 1;
```

Collatz sequences:

12, 6, 3, 10, 5, 16, 8, 4, 2, 1

9,28,14,7,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1

Collatz conjecture (1937): No matter what number you start with, you will always eventually reach 1.
The conjecture has still not been proven!

Termination?
(continued)

(c)

```
for (int x = 3; ; x++)
for (int a = 1; a <= x; a++)
for (int b = 1; b <= x; b++)
for (int c = 1; c <= x; c++)
for (int n = 3; n <= x; n++)
    if (Math.pow(a,n) + Math.pow(b,n) == Math.pow(c,n))
        System.exit(0);
```

The program terminates, if and only if Fermat's last theorem is false.

For $n \geq 3$, no three positive integers a, b, and c can satisfy $a^n + b^n = c^n$.

P. de Fermat (1601-65)

The theorem was proven in 1995

The halting problem

It is impossible to design an algorithm that for any algorithm can decide if it terminates.

Proof (by contradiction):

Assume there exists a method `terminates(p)`, which for any method `p` returns `true` if `p` terminates; otherwise, `false`.

Now define:

```
void p() {
    while (terminates(p)) /* do nothing */;
}
```

What is the result of the call `terminates(p)`?