CHAPTER 9

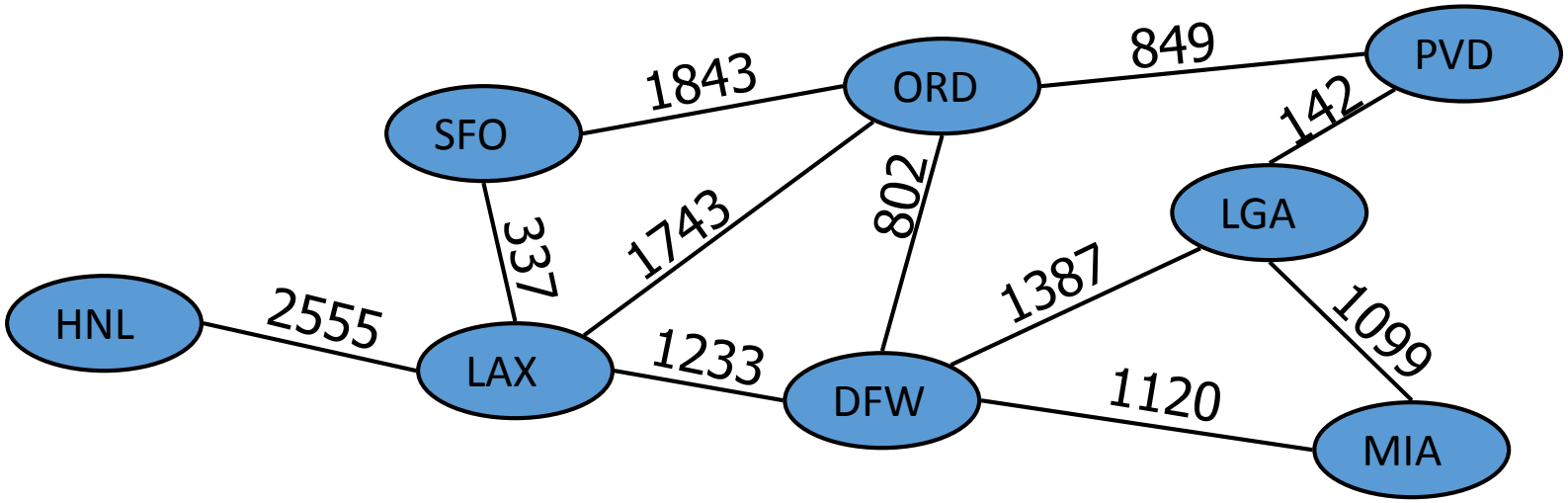
Data Structures and Algorithm Analysis in Java 3rd Edition by Mark Allen Weiss

GRAPH ALGORITHMS

- A graph G = (V,E) consists of set of vertices, V , and a set of edges, E. Each edge is a pair (v, w) where v, w ϵ V.
- A path in a graph is a sequence of vertices w_1, w_2, w_3, … w_N such that (w_i, w_{i+1},) ϵ E for 1 <= i < N. A simple path is path such that all vertices are distinct.
- A cycle is path of length at least one such that $w_1 = w_N$.
- An undirected graph is connected if there is path from every vertex to every other vertex.
- A complete graph is a graph in which there is an edge between every pair of vertices.

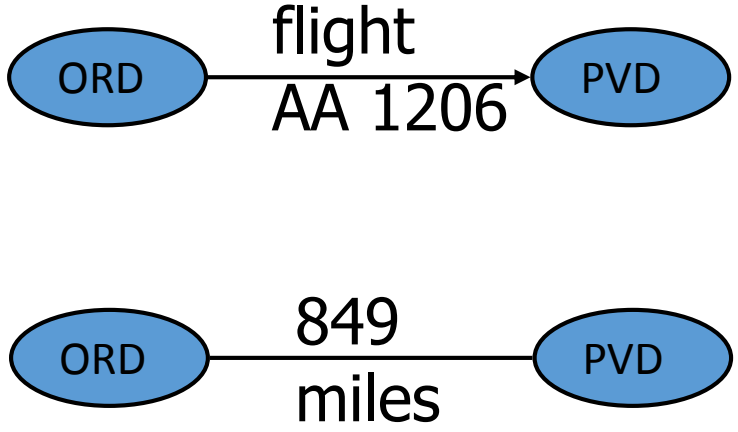
Graphs

- A graph is a pair (V, E), where
  - V is a set of nodes, called vertices
  - E is a collection of pairs of vertices, called edges
  - Vertices and edges are positions and store elements
- Example:
  - A vertex represents an airport and stores the three-letter airport code
  - An edge represents a flight route between two airports and stores the mileage of the route



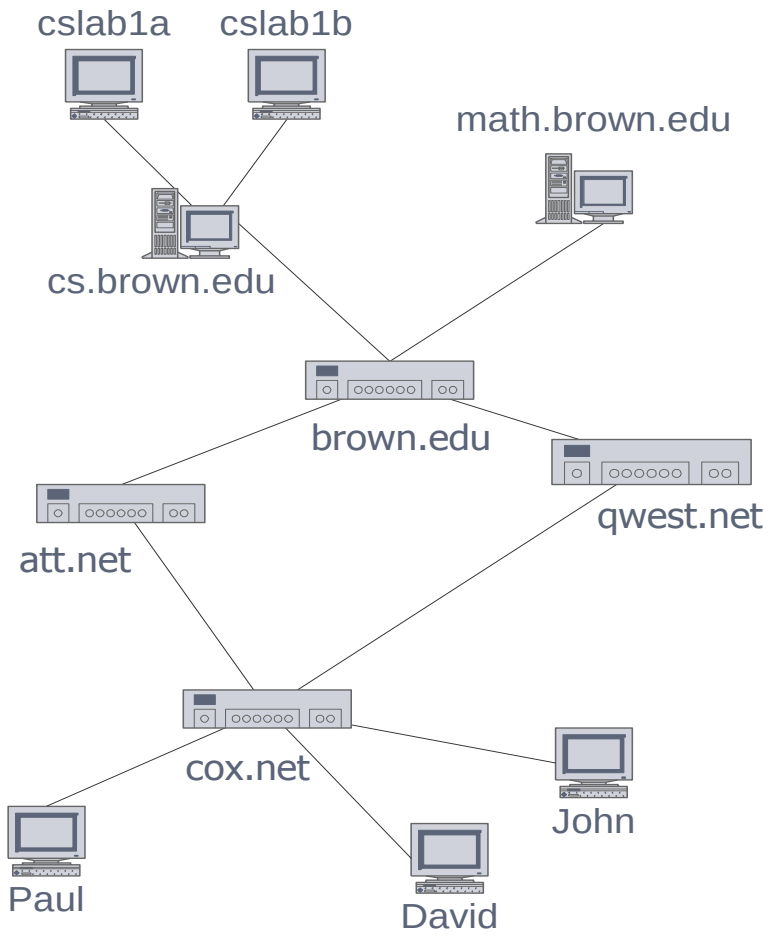
Edge Types

- Directed edge
  - ordered pair of vertices (u,v)
  - first vertex u is the origin
  - second vertex v is the destination
  - e.g., a flight
- Undirected edge
  - unordered pair of vertices (u,v)
  - e.g., a flight route
- Directed graph
  - all the edges are directed
  - e.g., route network
- Undirected graph
  - all the edges are undirected
  - e.g., flight network



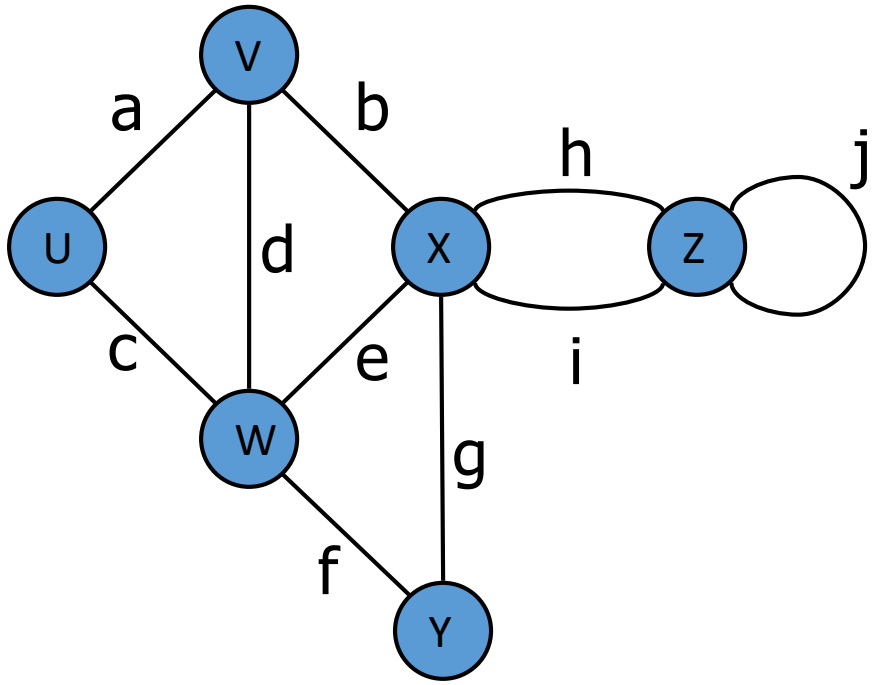
Applications

- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - Web
- Databases
  - Entity-relationship diagram



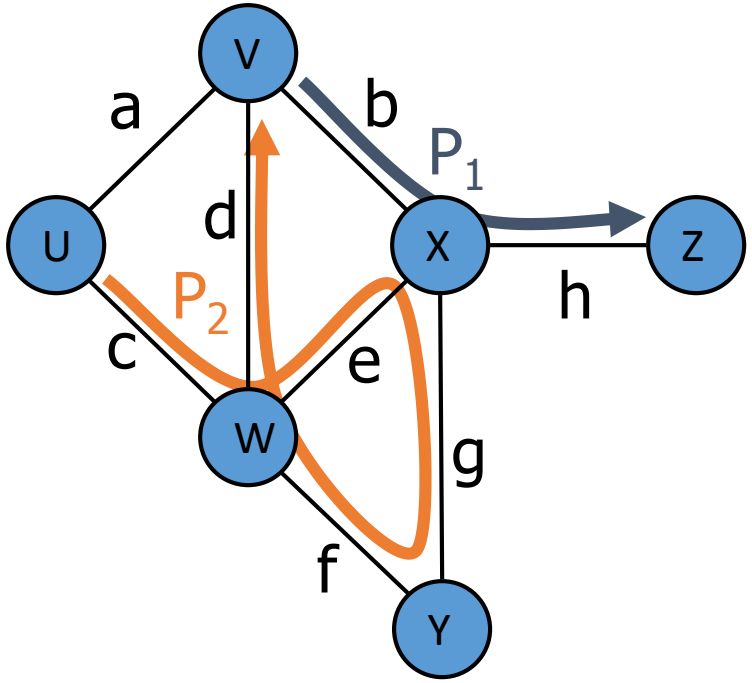
Terminology

- End vertices (or endpoints) of an edge
  - U and V are the endpoints of a
- Edges incident on a vertex
  - a, d, and b are incident on V
- Adjacent vertices
  - U and V are adjacent
- Degree of a vertex
  - X has degree 5
- Parallel edges
  - h and i are parallel edges
- Self-loop
  - j is a self-loop



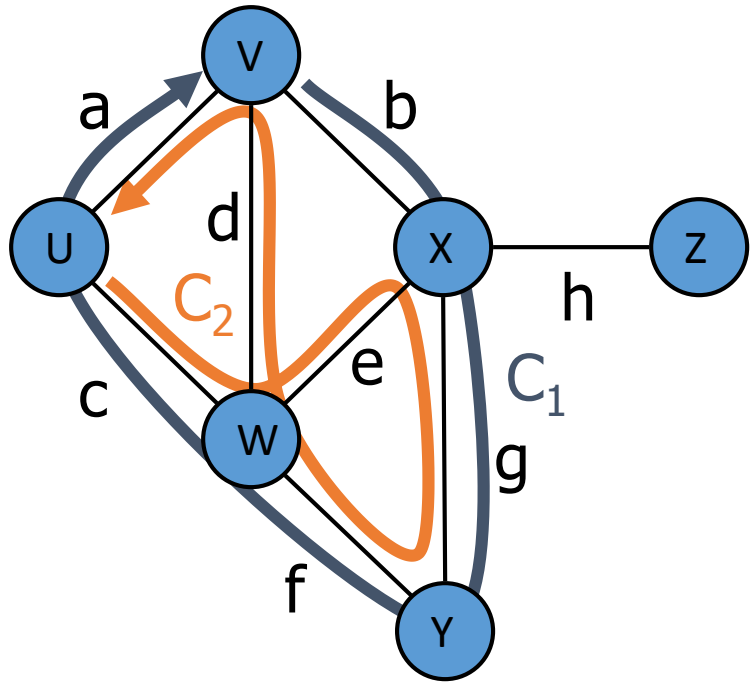
Terminology (cont.)

- Path
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
- Simple path
  - path such that all its vertices and edges are distinct
- Examples
  - P_1=(V,b,X,h,Z) is a simple path
  - P_2=(U,c,W,e,X,g,Y,f,W,d,V) is a path that is not simple



Terminology (cont.)

- Cycle
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints
- Simple cycle
  - cycle such that all its vertices and edges are distinct
- Examples
  - C_1=(V,b,X,g,Y,f,W,c,U,a,↵) is a simple cycle
  - C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,↵) is a cycle that is not simple



Properties

Property 1

$\sum_v \deg(v) = 2m$

Proof: each edge is counted twice

Property 2

In an undirected graph with no self-loops and no multiple edges

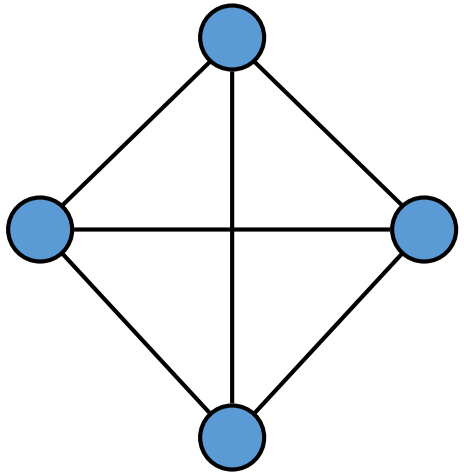
$m \leq n\,(n-1)/2$

Proof: each vertex has degree at most $(n-1)$

What is the bound for a directed graph?

Notation

n	number of vertices
m	number of edges
$\deg(v)$	degree of vertex v



Example

- $n = 4$
- $m = 6$
- $\deg(v) = 3$

REPRESENTATION OF GRAPHS

- One simple way to represent graph is to use a two dimensional array. This is known as an adjacency matrix representation.
- If the graph is not dense a better solution is an adjacency list representation.

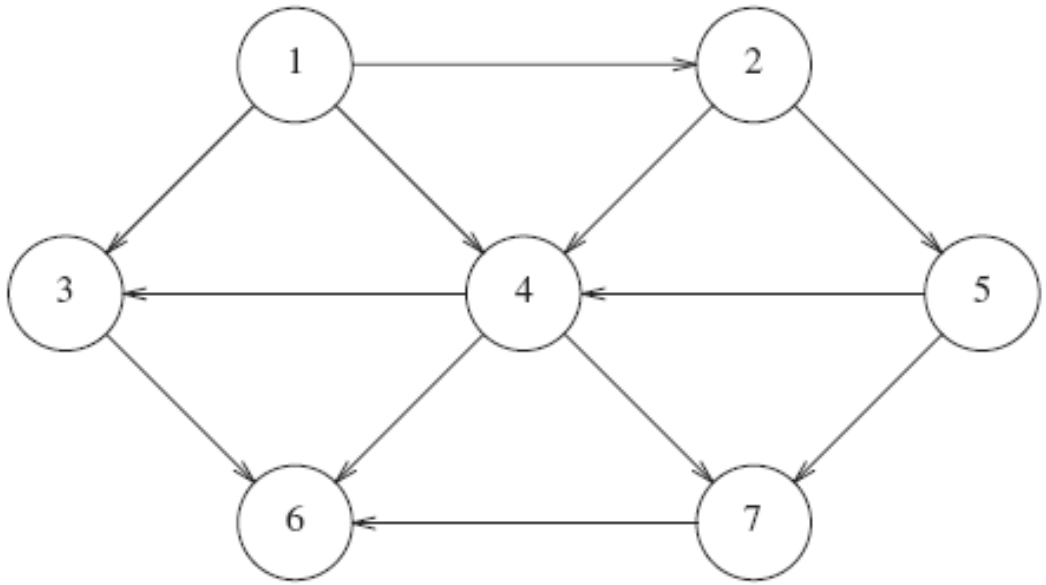


Figure 9.1 A directed graph

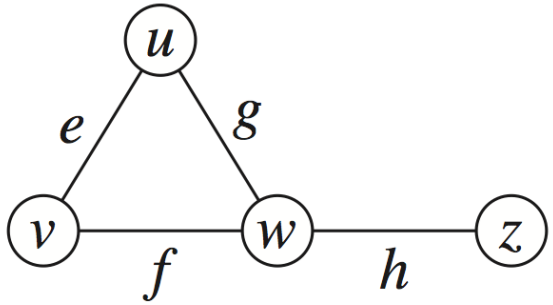
ADJACENCY LIST

1	2, 4, 3
2	4, 5
3	6
4	6, 7, 3
5	4, 7
6	(empty)
7	6

Figure 9.2 An adjacency list representation of a graph

Adjacency Matrix Structure

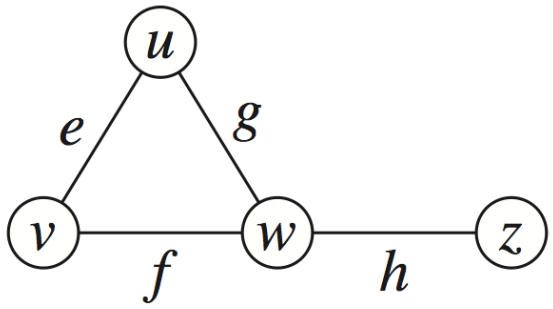
- Edge list structure
- Augmented vertex objects
  - Integer key (index) associated with vertex
- 2D-array adjacency array
  - Reference to edge object for adjacent vertices
  - Null for non nonadjacent vertices
- The “old fashioned” version just has 0 for no edge and 1 for edge

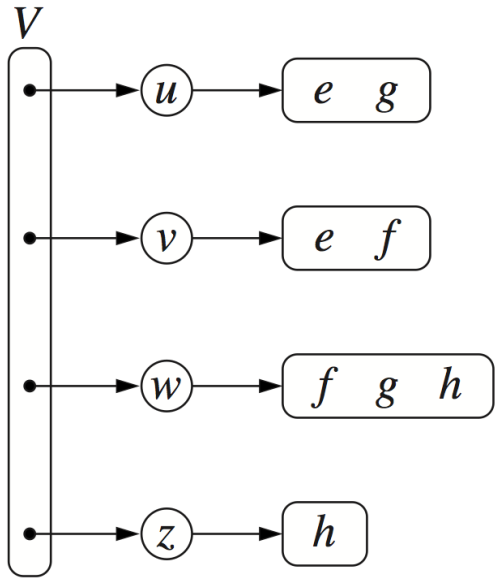


		0	1	2	3
u	0		e	g	
v	1	e		f	
w	2	g	f		h
z	3			h	

Adjacency List Structure

- Incidence sequence for each vertex
  - sequence of references to edge objects of incident edges
- Augmented edge objects
  - references to associated positions in incidence sequences of end vertices





TOPOLOGICAL SORT

- A topological sort is an ordering of vertices in a directed acyclic graph, such that if there is a path from v_i to v_j, then v_j appears after v_i in that ordering

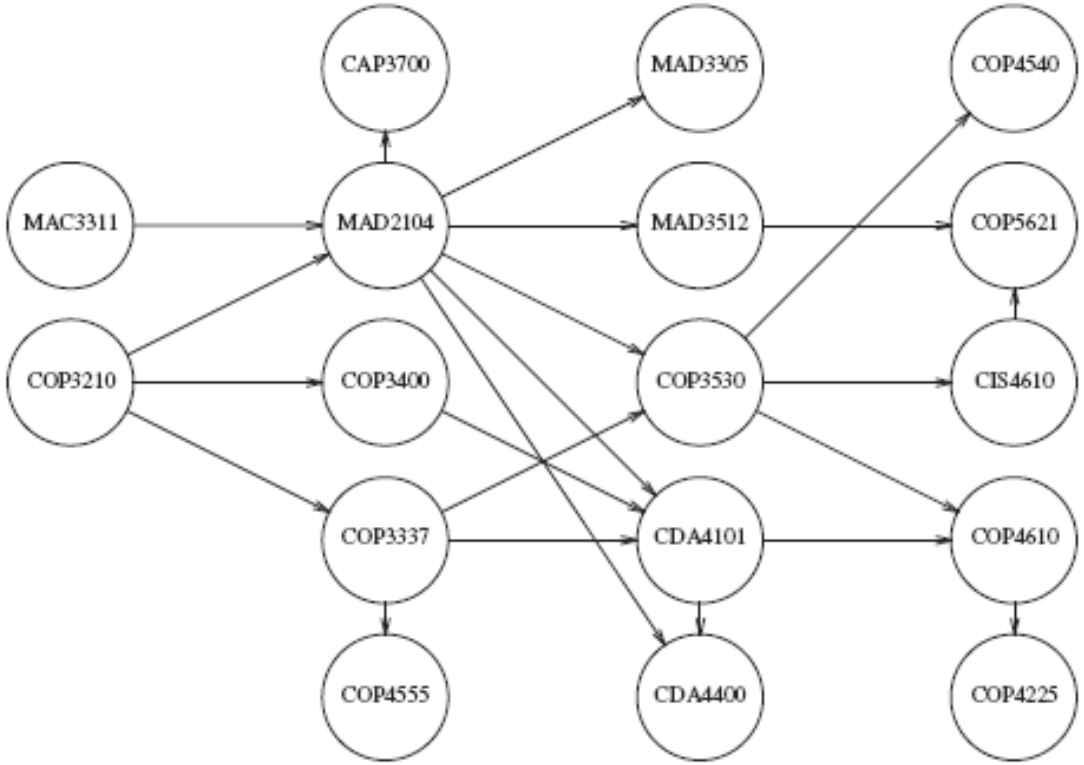


Figure 9.3 An acyclic graph representing course prerequisite structure

The ordering is not necessarily unique; any legal ordering will do.
For example : $v_1, v_2, v_5, v_4, v_3, v_7, v_6$ and $v_1, v_2, v_5, v_4, v_7, v_3, v_6$,

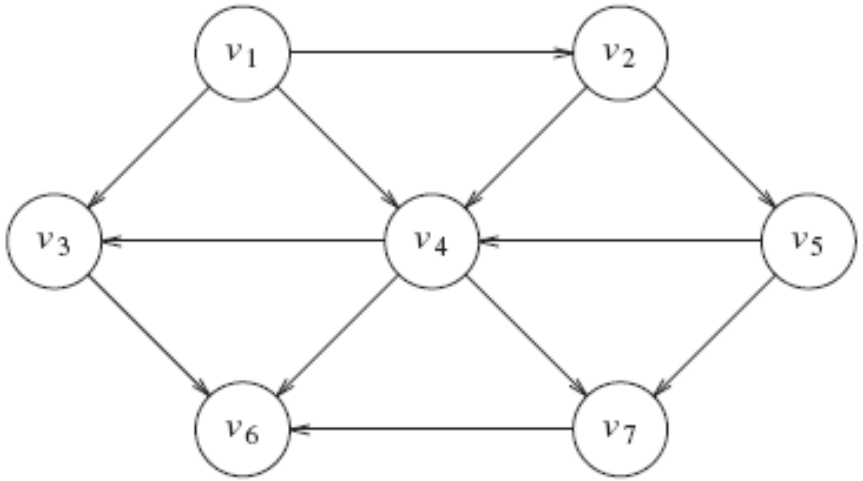


Figure 9.4 An acyclic graph

ALGORITHM FOR TOPOLOGICAL SORT

1. Label ("mark") each vertex with its in-degree

2. While there are vertices not yet output:
   a) Choose a vertex **v** with labeled with in-degree of 0
   b) Output **v** and *conceptually* remove it from the graph
   c) For each vertex **u** adjacent to **v** (i.e. **u** such that (**v**,**u**) in **E**), decrement the in-degree of **u**

ALGORITHM FOR TOPOLOGICAL SORT

```
void topsort( ) throws CycleFoundException
{
    for( int counter = 0; counter < NUM_VERTICES; counter++ )
    {
        Vertex v = findNewVertexOfIndegreeZero( );
        if( v == null )
            throw new CycleFoundException( );
        v.topNum = counter;
        for each Vertex w adjacent to v
            w.indegree--;
    }
}
```

Figure 9.5 Simple topological sort pseudocode

The method findNewVertexOfIndgreeZero() scans the array looking for vertex with indegree 0 that has not already been assigned a topological number. Each call to it takes O(|V|) time. Since there are |V| such calls, running time is $O(|V|^2)$.

BETTER ALGORITHM

The trick is to avoid searching for a zero-degree node every time!

- Keep the "pending" zero-degree nodes in a list, stack, queue, bag, table, or something
- Order we process them affects output but not correctness or efficiency provided add/remove are both $O(1)$

Using a queue:

1. Label each vertex with its in-degree, enqueue 0-degree nodes
2. While queue is not empty
   a) **v** = dequeue()
   b) Output **v** and remove it from the graph
   c) For each vertex **u** adjacent to **v** (i.e. **u** such that (**v**,**u**) in $\mathbf{E}$), decrement the in-degree of **u**, if new degree is 0, enqueue it

BETTER ALGORITHM

```
void topsort( ) throws CycleFoundException
{
    Queue<Vertex> q = new Queue<Vertex>( );
    int counter = 0;

    for each Vertex v
        if( v.indegree == 0 )
            q.enqueue( v );

    while( !q.isEmpty( ) )
    {
        Vertex v = q.dequeue( );
        v.topNum = ++counter;  // Assign next number

        for each Vertex w adjacent to v
            if( --w.indegree == 0 )
                q.enqueue( w );
    }
    if( counter != NUM_VERTICES )
        throw new CycleFoundException( );
}
```

Figure 9.7 Pseudocode to perform topological sort

	Indegree Before Dequeue #						
Vertex	1	2	3	4	5	6	7
v_1	0	0	0	0	0	0	0
v_2	1	0	0	0	0	0	0
v_3	2	1	1	1	0	0	0
v_4	3	2	1	0	0	0	0
v_5	1	1	0	0	0	0	0
v_6	3	3	3	3	2	1	0
v_7	2	2	2	1	0	0	0
Enqueue	v_1	v_2	v_5	v_4	v_3, v_7		v_6
Dequeue	v_1	v_2	v_5	v_4	v_3	v_7	v_6

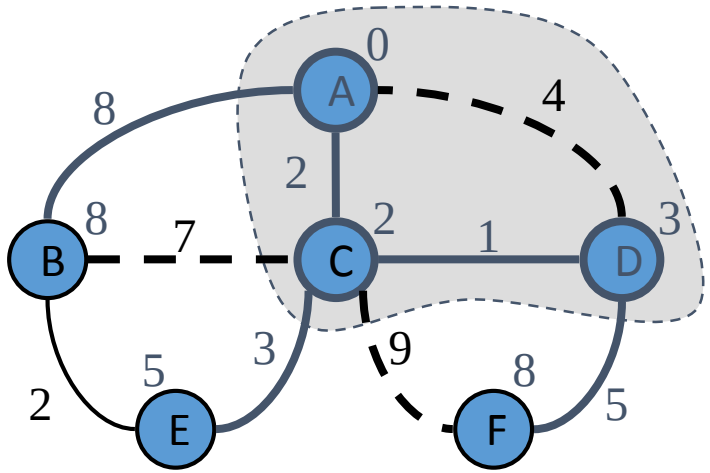
Figure 9.6 Result of applying topological sort to the graph in Figure 9.4

BETTER ALGORITHM

- The time to perform this algorithm is O(|E|+|V|). body of the for loop is executed at most one per edge.
- Indegree computation

```
for each Vertex v
   v.indegree = 0;
for each vertex v
   for each vertex w adjacent to v
         w.indegree++;
```

Shortest Paths



SHORTEST-PATH ALGORITHM

- Given an input a weighted graph G + (V,E), and distinguished vertex, S, find the shortest weighted path from s to every other vertex in G.
- The input is a weighted graph: associated with each edge (v_i , v_j) is a cost c_{ij} to traverse the edge.

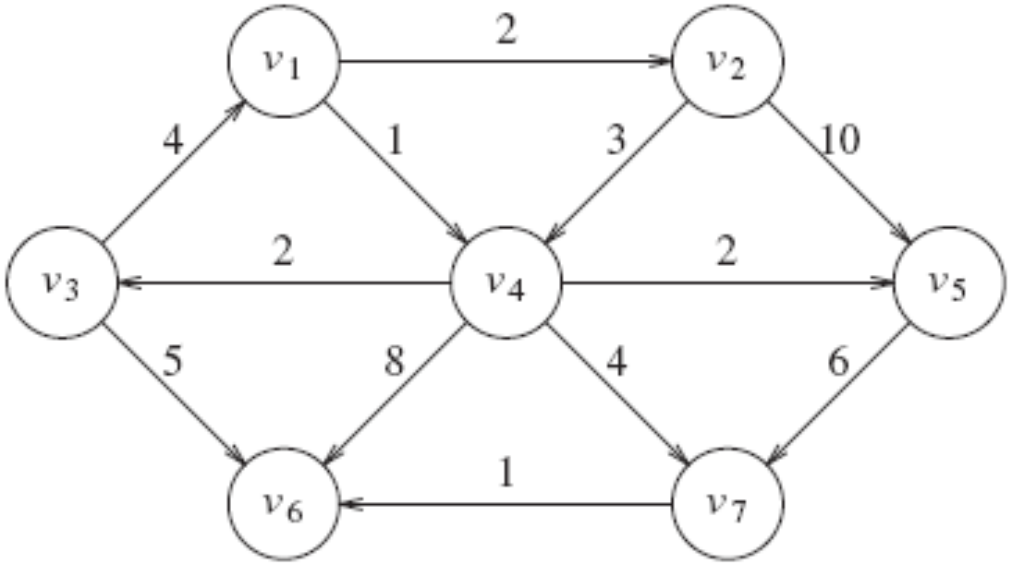
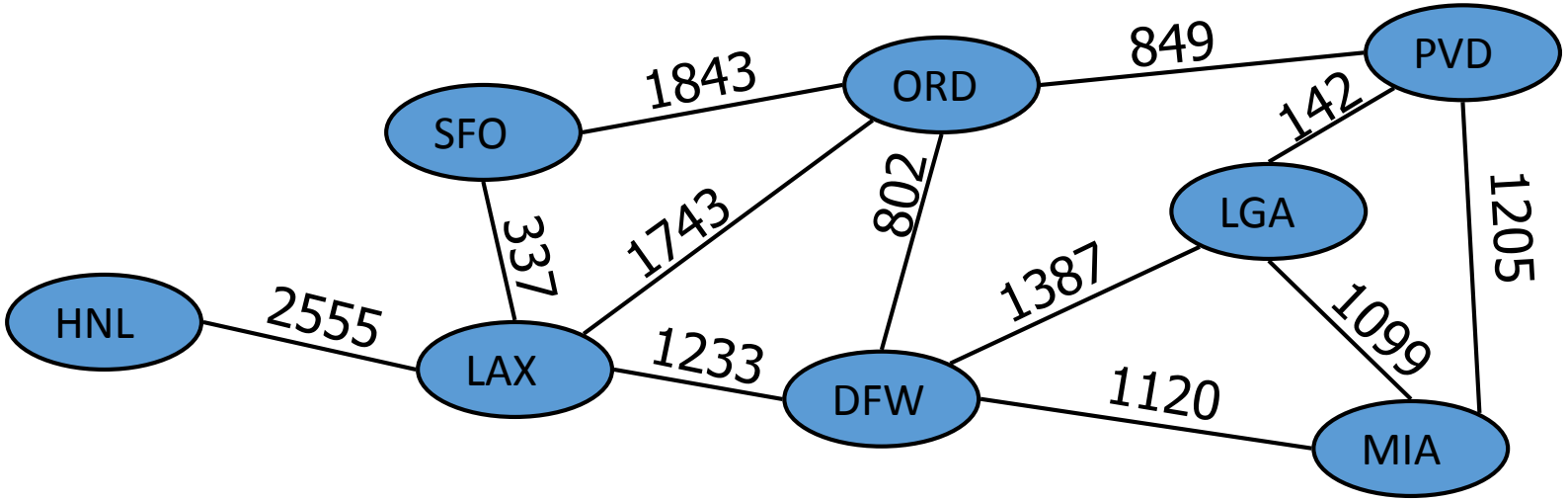


Figure 9.8 A directed graph *G*

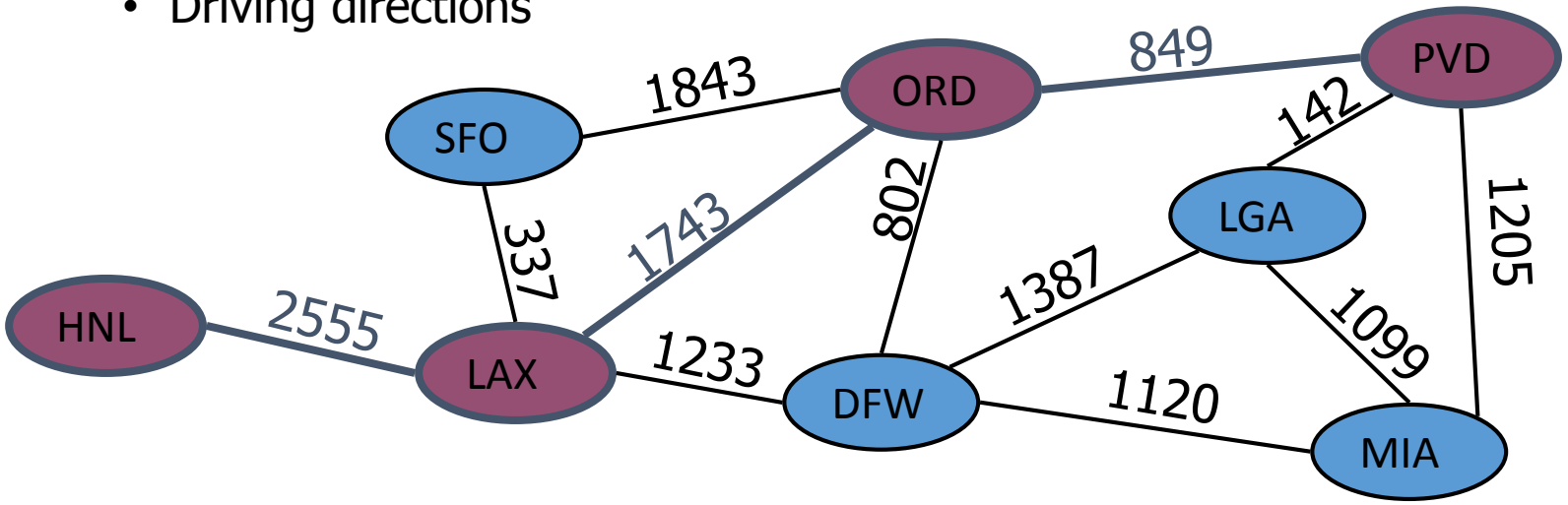
Weighted Graphs

- In a weighted graph, each edge has an associated numerical value, called the weight of the edge
- Edge weights may represent, distances, costs, etc.
- Example:
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



Shortest Paths

- Given a weighted graph and two vertices u and v, we want to find a path of minimum total weight between u and v.
  - Length of a path is the sum of the weights of its edges.
- Example:
  - Shortest path between Providence and Honolulu
- Applications
  - Internet packet routing
  - Flight reservations
  - Driving directions



Shortest Path Properties

Property 1:

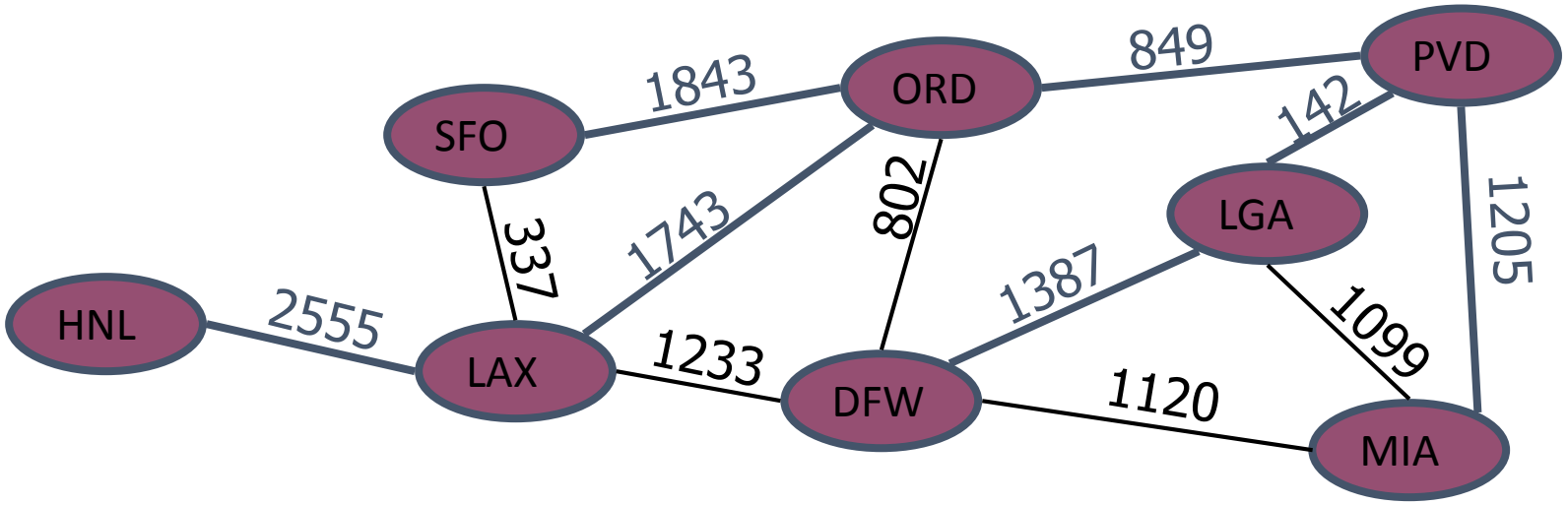
A subpath of a shortest path is itself a shortest path

Property 2:

There is a tree of shortest paths from a start vertex to all the other vertices

Example:

Tree of shortest paths from Providence



GRAPH WITH NEGATIVE EDGES

- The shortest path is not defines if the graph has negative –cost cycle.
- Example

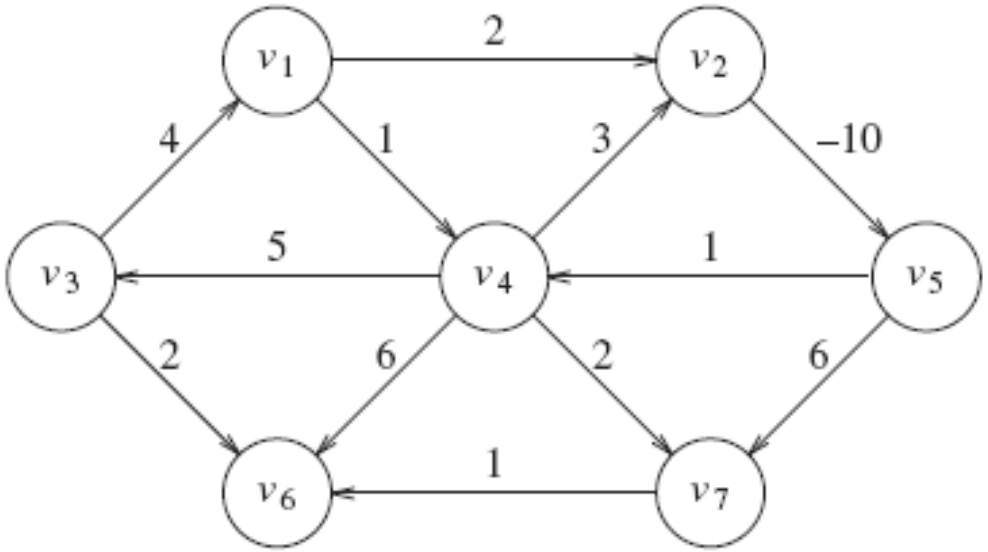


Figure 9.9 A graph with a negative-cost cycle

UNWEIGHTED SHORTEST PATH

- There are no weights on the edges so number of edges on the path is the solution.

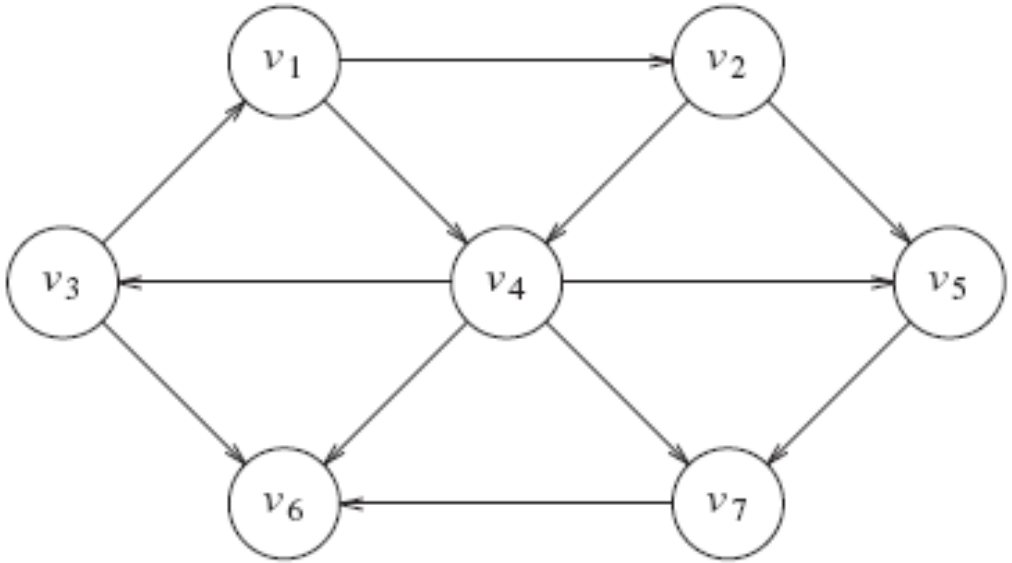


Figure 9.10 An unweighted directed graph G

Let the source s be v_3

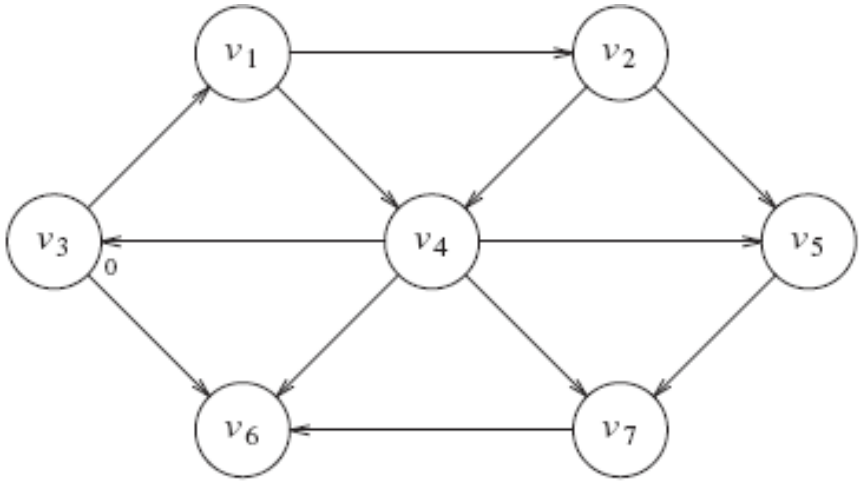


Figure 9.11 Graph after marking the start node as reachable in zero edges

To look for vertices that are a distance 1 away from s ; look at the vertices adjacent to s.

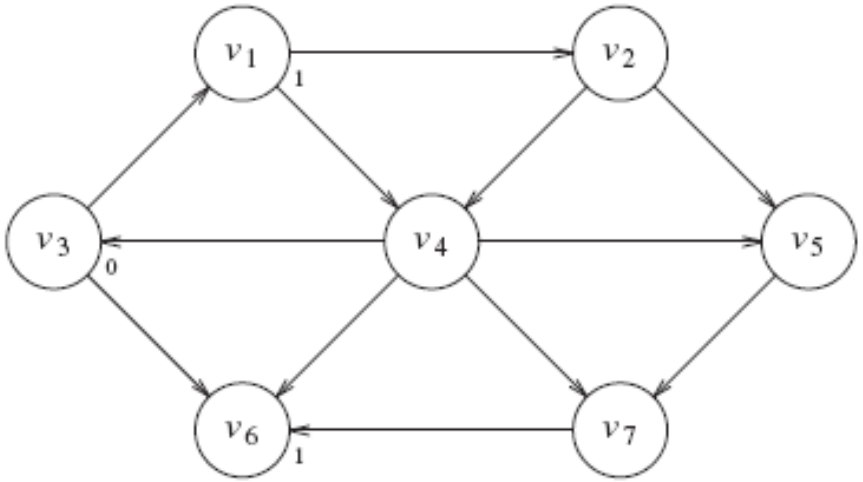


Figure 9.12 Graph after finding all vertices whose path length from s is 1

Find the vertices whose shortest path from s is exactly 2 by finding all the vertices adjacent to v_1 and v_6

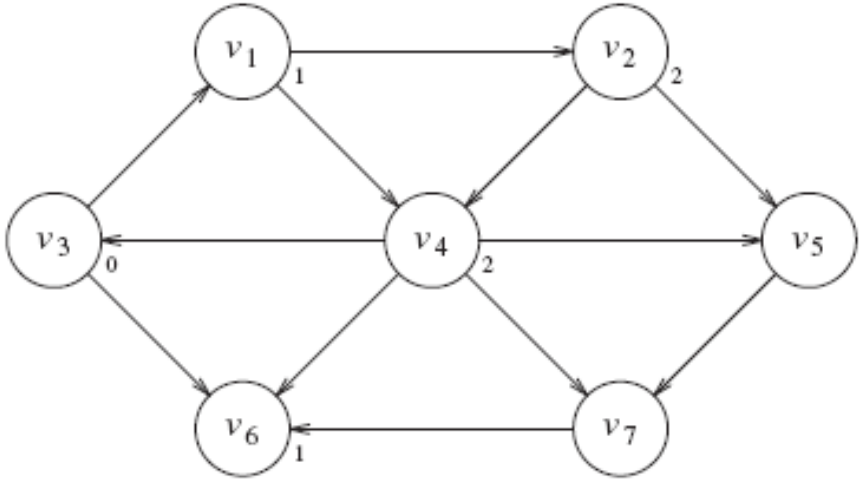


Figure 9.13 Graph after finding all vertices whose shortest path is 2

Finally by examining vertices adjacent to the recently evaluated v_2 and v_4, find that v_5 and v_7 have shortest path of three edges.

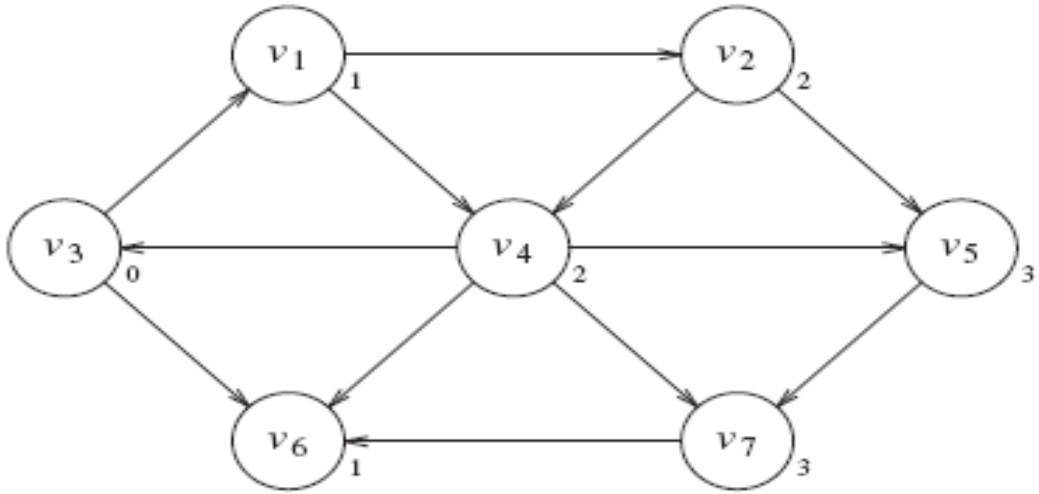


Figure 9.14 Final shortest paths

UNWEIGHTED SHORTEST PATH

- This strategy for searching a graph is know as breadth-first search. It operates by processing vertices in layers. Similar to level order traversal for trees.

v	*known*	d_v	p_v
v_1	F	∞	0
v_2	F	∞	0
v_3	F	0	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

Figure 9.15 Initial configuration of table used in unweighted shortest-path computation

Algorithm I

```
void unweighted( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    for( int currDist = 0; currDist < NUM_VERTICES; currDist++ )
        for each Vertex v
            if( !v.known && v.dist == currDist )
            {
                v.known = true;
                for each Vertex w adjacent to v
                    if( w.dist == INFINITY )
                    {
                        w.dist = currDist + 1;
                        w.path = v;
                    }
            }
}
```

Figure 9.16 Pseudocode for unweighted shortest-path algorithm

- The running time of the algorithm is $O(|V|^2)$.
- This can be improved by using a queue for the next list of vertices to check. The running time is $O(|E| + |V|)$.

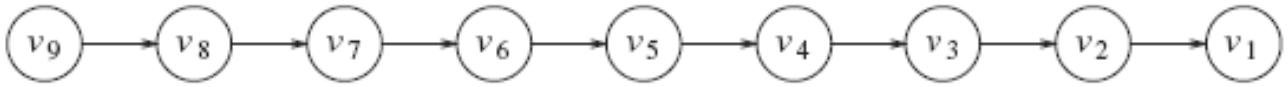


Figure 9.17 A bad case for unweighted shortest-path algorithm using Figure 9.16

Algorithm II

```
void unweighted( Vertex s )
{
    Queue<Vertex> q = new Queue<Vertex>( );

    for each Vertex v
        v.dist = INFINITY;

    s.dist = 0;
    q.enqueue( s );

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

        for each Vertex w adjacent to v
            if( w.dist == INFINITY )
            {
                w.dist = v.dist + 1;
                w.path = v;
                q.enqueue( w );
            }
    }
}
```

Figure 9.18 Pseudocode for unweighted shortest-path algorithm

	Initial State			v_3 Dequeued			v_1 Dequeued			v_6 Dequeued		
v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	F	∞	0	F	1	v_3	T	1	v_3	T	1	v_3
v_2	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_3	F	0	0	T	0	0	T	0	0	T	0	0
v_4	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_5	F	∞	0	F	∞	0	F	∞	0	F	∞	0
v_6	F	∞	0	F	1	v_3	F	1	v_3	T	1	v_3
v_7	F	∞	0	F	∞	0	F	∞	0	F	∞	0
Q:	v_3			v_1, v_6			v_6, v_2, v_4			v_2, v_4		

	v_2 Dequeued			v_4 Dequeued			v_5 Dequeued			v_7 Dequeued		
v	known	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	T	1	v_3	T	1	v_3	T	1	v_3	T	1	v_3
v_2	T	2	v_1	T	2	v_1	T	2	v_1	T	2	v_1
v_3	T	0	0	T	0	0	T	0	0	T	0	0
v_4	F	2	v_1	T	2	v_1	T	2	v_1	T	2	v_1
v_5	F	3	v_2	F	3	v_2	T	3	v_2	T	3	v_2
v_6	T	1	v_3	T	1	v_3	T	1	v_3	T	1	v_3
v_7	F	∞	0	F	3	v_4	F	3	v_4	T	3	v_4
Q:	v_4, v_5			v_5, v_7			v_7			empty		

Figure 9.19 How the data change during the unweighted shortest-path algorithm

Dijkstra's Algorithm

- The distance of a vertex v from a vertex s is the length of a shortest path between s and v
- Dijkstra's algorithm computes the distances of all the vertices from a given start vertex s
- Assumptions:
  - the graph is connected
  - the edges are undirected
  - the edge weights are nonnegative

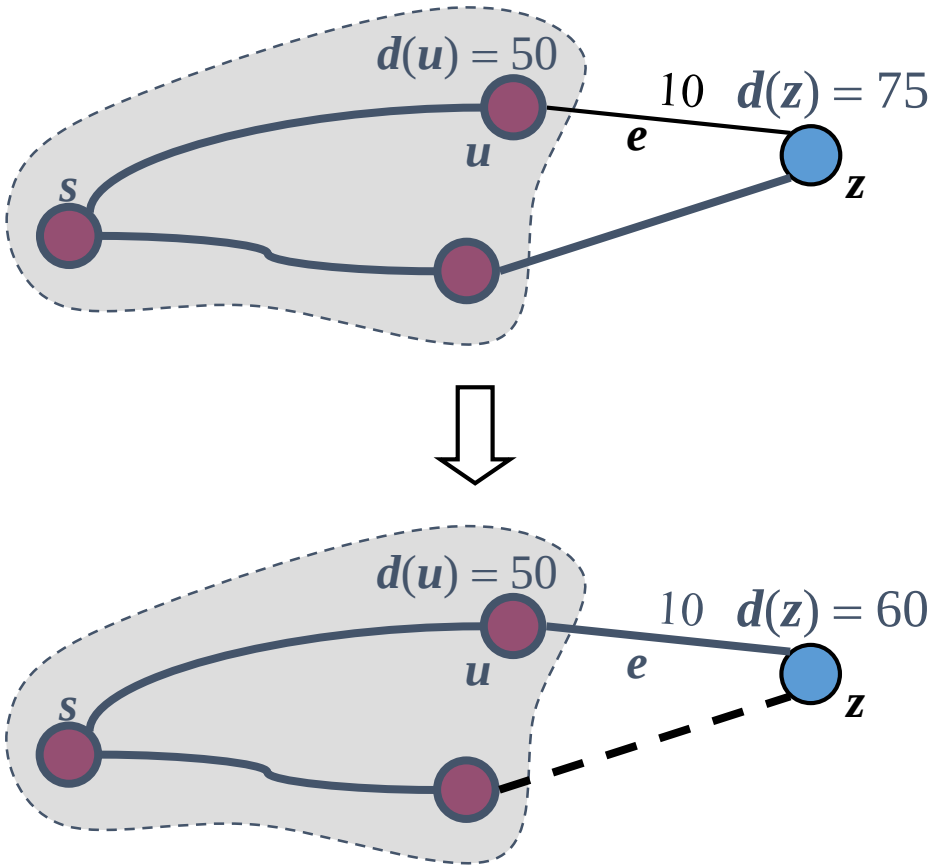
- We grow a "cloud" of vertices, beginning with s and eventually covering all the vertices
- We store with each vertex v a label $d(v)$ representing the distance of v from s in the subgraph consisting of the cloud and its adjacent vertices
- At each step
  - We add to the cloud the vertex u outside the cloud with the smallest distance label, $d(u)$
  - We update the labels of the vertices adjacent to u

Edge Relaxation

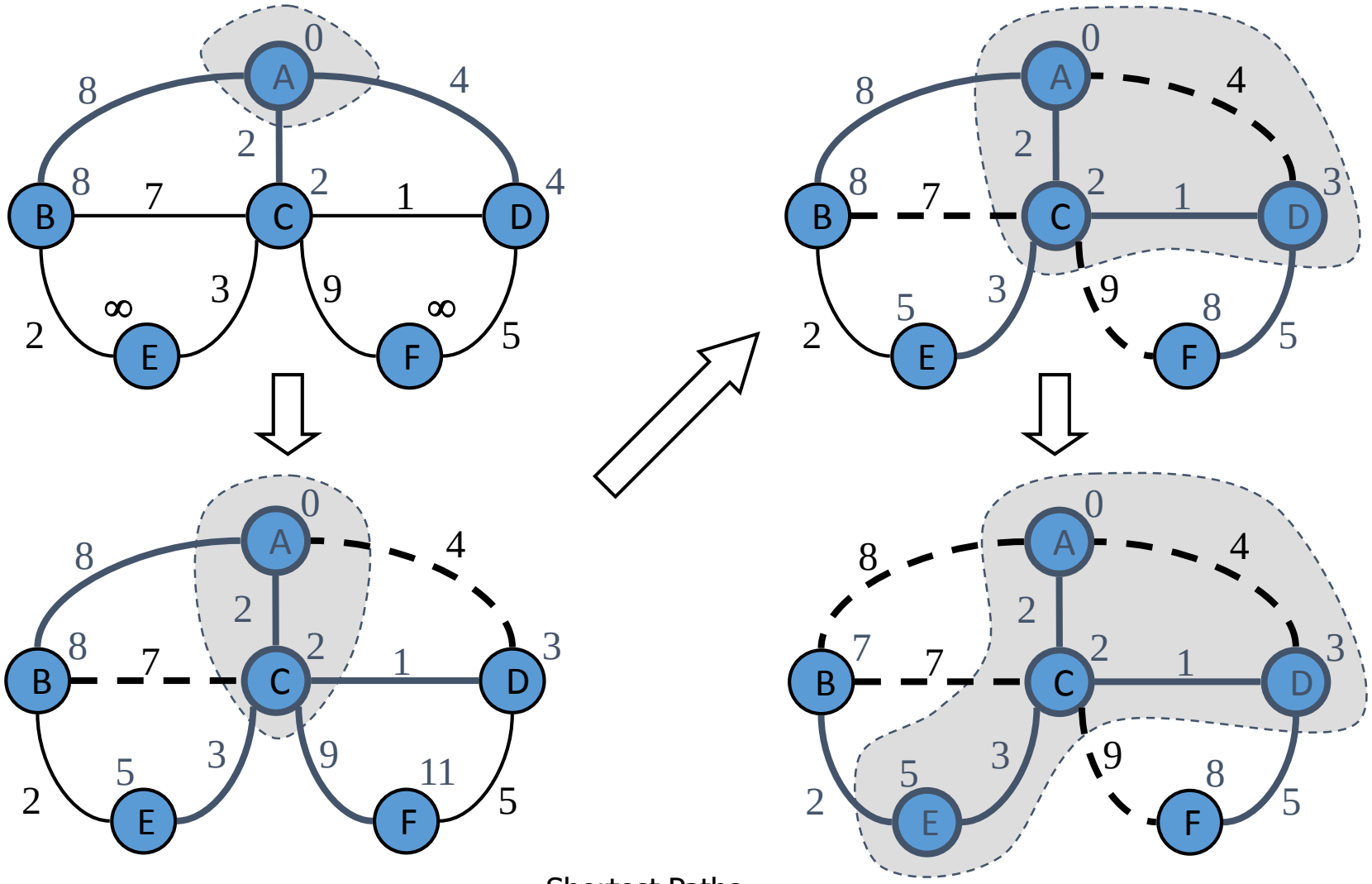
- Consider an edge $e = (u,z)$ such that
  - u is the vertex most recently added to the cloud
  - z is not in the cloud

- The relaxation of edge e updates distance $d(z)$ as follows:

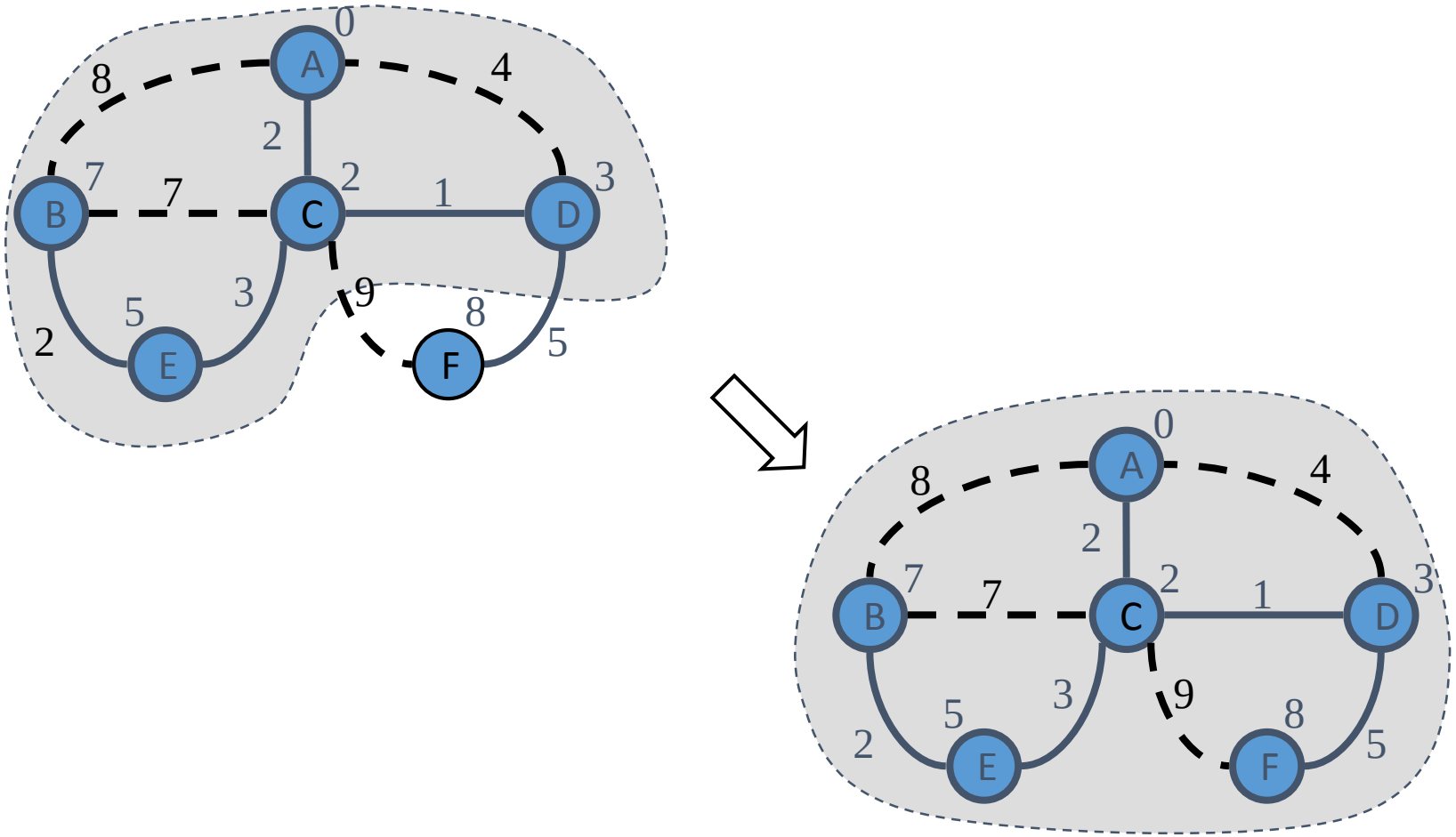
$$d(z) \leftarrow \min\{d(z), d(u) + weight(e)\}$$



Example


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

Example (cont.)



DIJKSTRA'S ALGORITHM

- This thirty-year old solution is a prime example of a greedy algorithm.
- The algorithm proceeds in stages, at each stage selects a vertex v, which has the smallest d_v among all the unknown vertices.

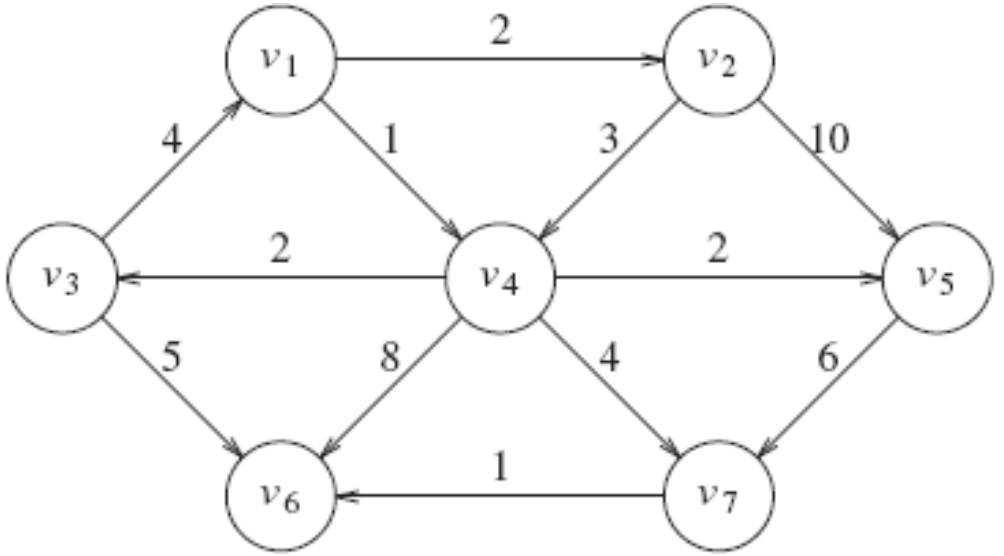


Figure 9.20 The directed graph G (again)

v	*known*	d_v	p_v
v_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

Figure 9.21 Initial configuration of table used in Dijkstra's algorithm

v	*known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	∞	0
v_4	F	1	v_1
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

Figure 9.22 After v_1 is declared *known*

v	*known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

Figure 9.23 After v_4 is declared *known*

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

Figure 9.24 After v_2 is declared *known*

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	F	8	v_3
v_7	F	5	v_4

Figure 9.25 After v_5 and then v_3 are declared *known*

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	F	6	v_7
v_7	T	5	v_4

Figure 9.26 After v_7 is declared *known*

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	T	6	v_7
v_7	T	5	v_4

Figure 9.27 After v_6 is declared *known* and algorithm terminates

To print out the actual path from start vertex to some vertex v, we can write a recursive routine to follow the trail left in the p variable.

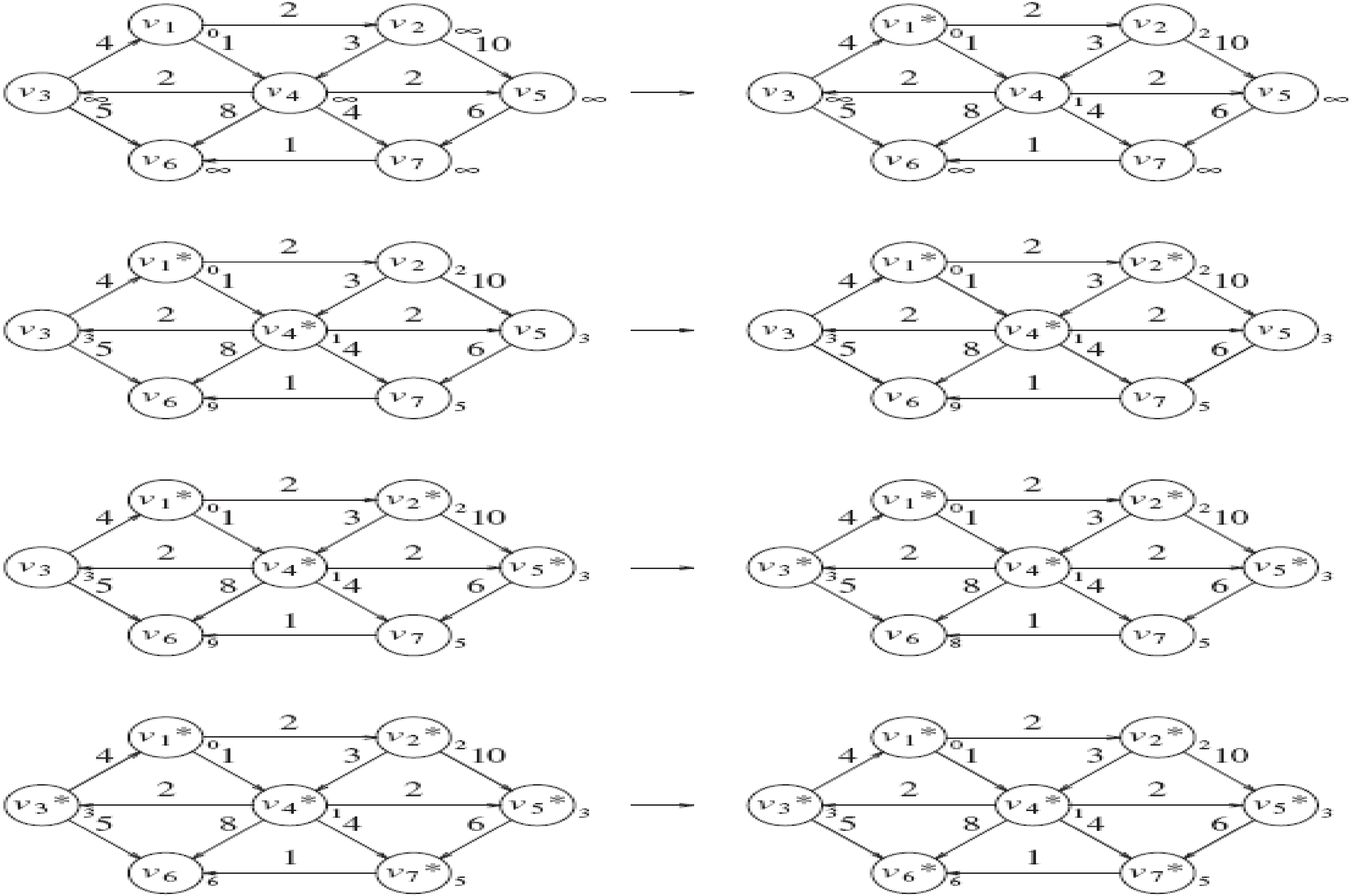
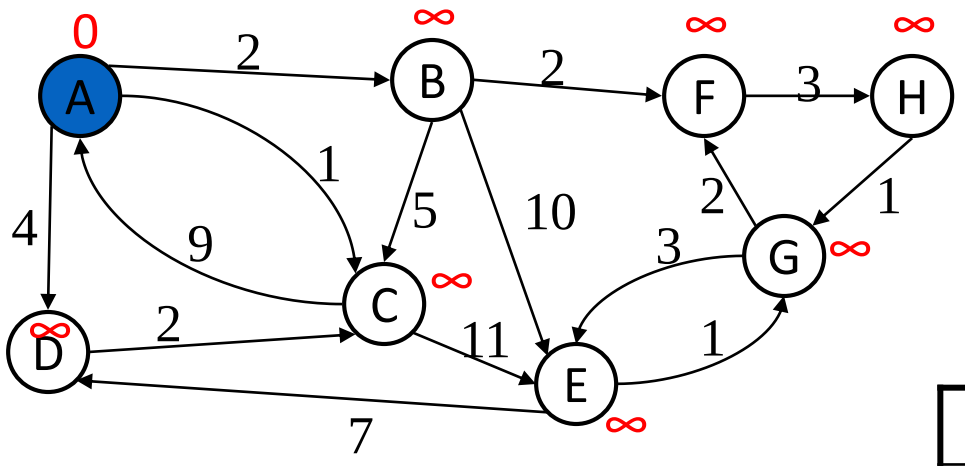


Figure 9.28 Stages of Dijkstra's algorithm

Example #1



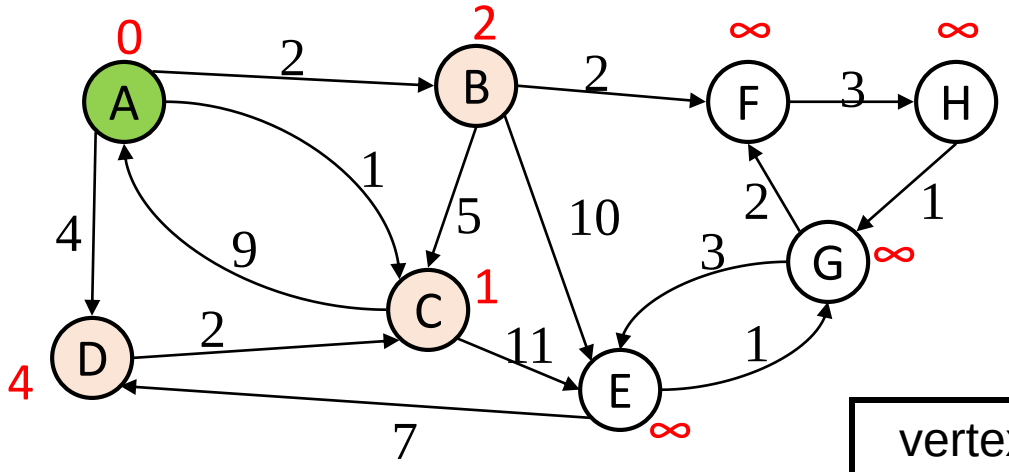
1. While there are unknown nodes in the graph
   a) Select the unknown node `v` with lowest cost
   b) Mark `v` as known
   c) For each edge `(v,u)` with weight `w`,

```
   c1 = v.cost + w // cost of best path through v to u
   c2 = u.cost // cost of best path to u previously known
   if(c1 < c2){ // if the path through v is better
      u.cost = c1
      u.path = v // for computing actual paths
   }
```

vertex	known?	cost	path
A		0	
B		??	
C		??	
D		??	
E		??	
F		??	
G		??	
H		??	

Order Added to Known Set:

Example #1

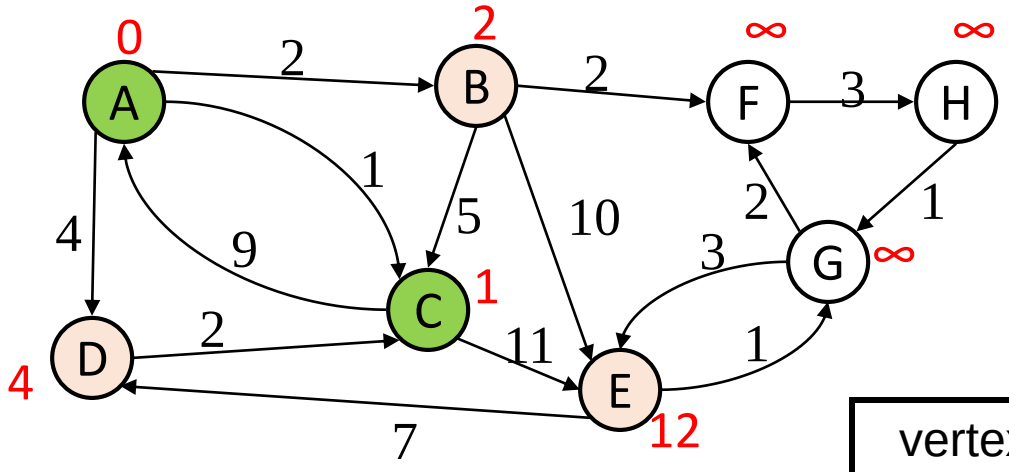


Order Added to Known Set:

A

vertex	known?	cost	path
A	Y	0	
B		≤ 2	A
C		≤ 1	A
D		≤ 4	A
E		??	
F		??	
G		??	
H		??	

Example #1

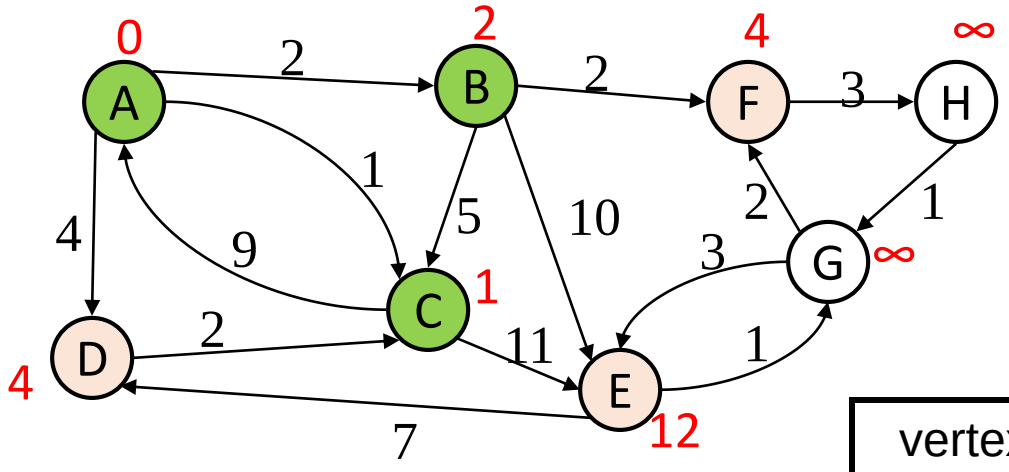


Order Added to Known Set:

A, C

vertex	known?	cost	path
A	Y	0	
B		≤ 2	A
C	Y	1	A
D		≤ 4	A
E		≤ 12	C
F		??	
G		??	
H		??	

Example #1

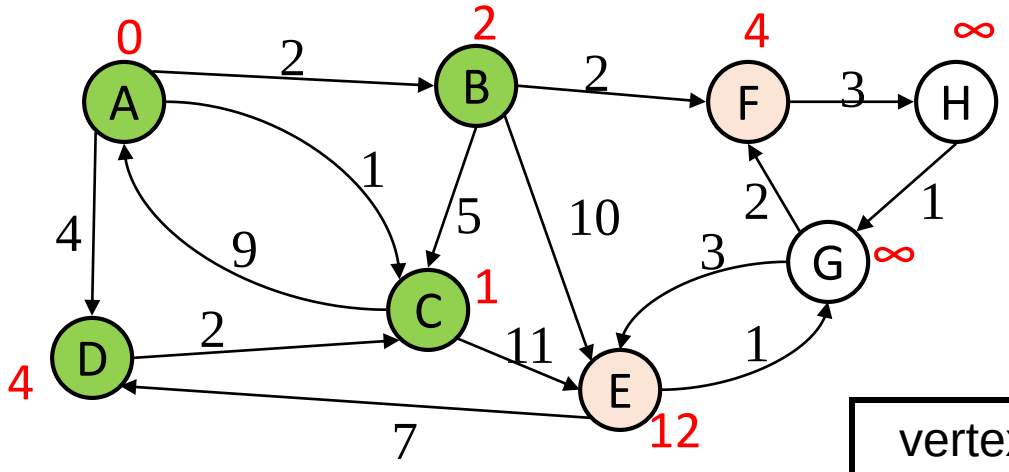


Order Added to Known Set:

A, C, B

vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D		≤ 4	A
E		≤ 12	C
F		≤ 4	B
G		??	
H		??	

Example #1

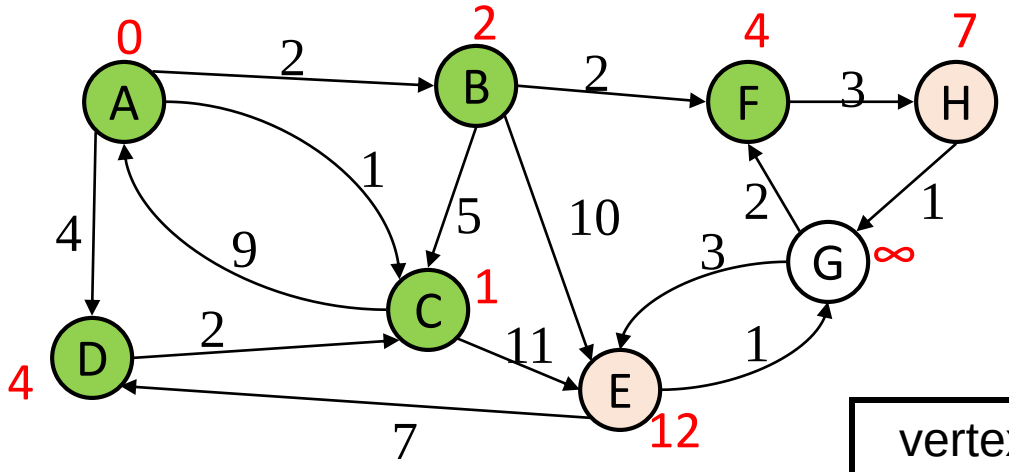


Order Added to Known Set:

A, C, B, D

vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 12	C
F		≤ 4	B
G		??	
H		??	

Example #1

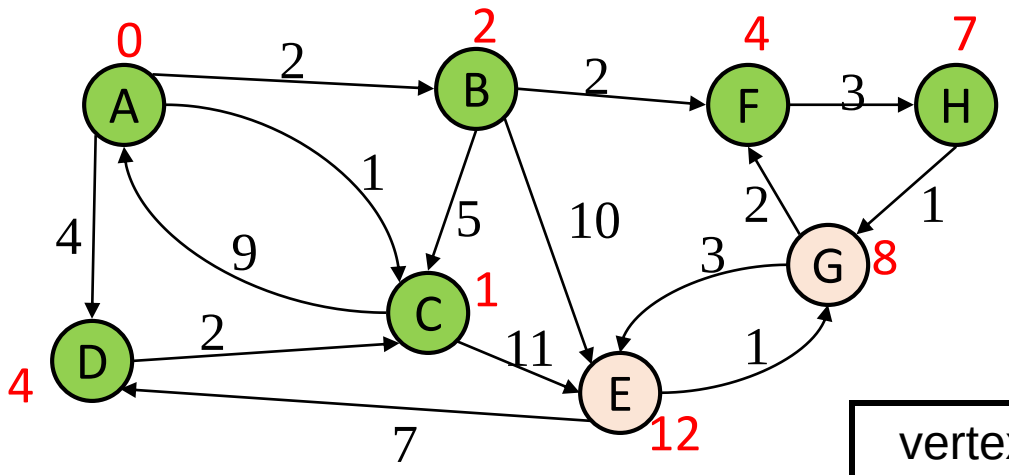


Order Added to Known Set:

A, C, B, D, F

vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 12	C
F	Y	4	B
G		??	
H		≤ 7	F

Example #1

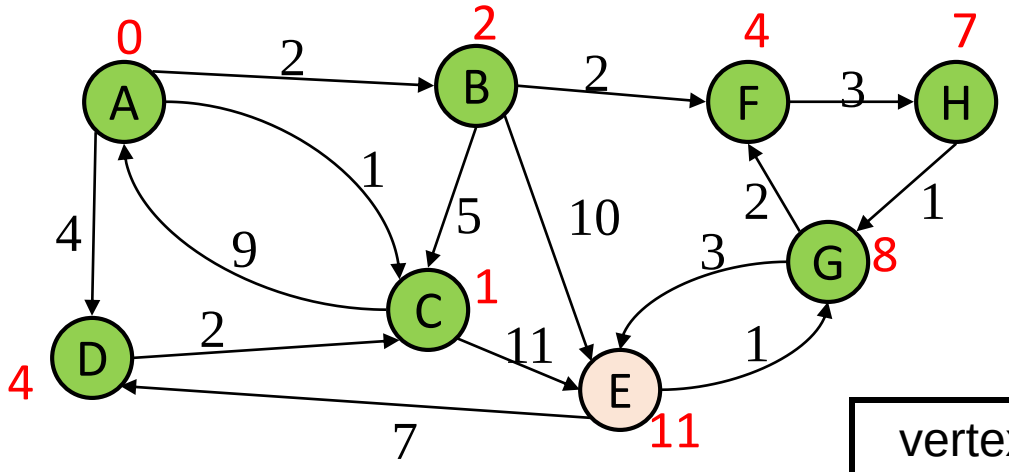


Order Added to Known Set:

A, C, B, D, F, H

vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 12	C
F	Y	4	B
G		≤ 8	H
H	Y	7	F

Example #1

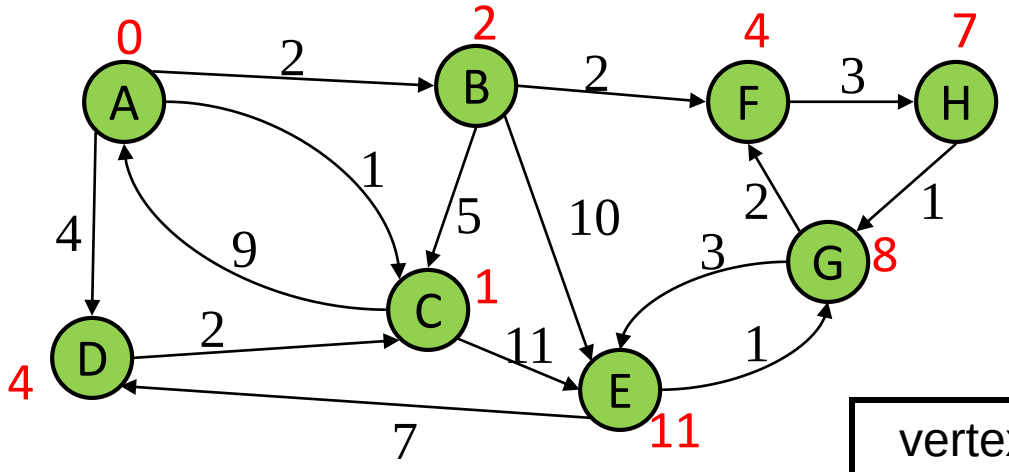


vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

Order Added to Known Set:

A, C, B, D, F, H, G

Example #1



Order Added to Known Set:

A, C, B, D, F, H, G, E

vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	Y	11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

VERTEX CLASS

```
class Vertex
{
    public List     adj;     // Adjacency list
    public boolean  known;
    public DistType dist;    // DistType is probably int
    public Vertex   path;
        // Other fields and methods as needed
}
```

Figure 9.29 Vertex class for Dijkstra's algorithm

SHORTEST PATH ROUTINE

```
/*
 * Print shortest path to v after dijkstra has run.
 * Assume that the path exists.
 */
void printPath( Vertex v )
{
    if( v.path != null )
    {
        printPath( v.path );
        System.out.print( " to " );
    }
    System.out.print( v );
}
```

Figure 9.30 Routine to print the actual shortest path

```
void dijkstra( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    while( there is an unknown distance vertex )
    {
        Vertex v = smallest unknown distance vertex;

        v.known = true;

        for each Vertex w adjacent to v
            if( !w.known )
            {
                DistType cvw = cost of edge from v to w;

                if( v.dist + cvw < w.dist )
                {
                    // Update w
                    decrease( w.dist to v.dist + cvw );
                    w.path = v;
                }
            }
    }
}
```

Figure 9.31 Pseudocode for Dijkstra's algorithm

- The routine to find the minimum d_v will take $O(|V|)$ time and thus $O(|V|^2)$ time over the course of algorithm.
- The time for updating d_w is constant per update, and there is at most one update per edge for a total of $O(|E|)$.
- Thus the total running time is $O(|E| + |V|^2)$
- If the graph is dense then this is fine otherwise the algorithm is too slow.
- We can improve by using a priority queue. This gives a running time of $O(|E|\log|V| + |V|\log|V|)$

Dijkstra's Algorithm

Algorithm ShortestPath(G, s):

Input: A weighted graph G with nonnegative edge weights, and a distinguished vertex s of G.

Output: The length of a shortest path from s to v for each vertex v of G.

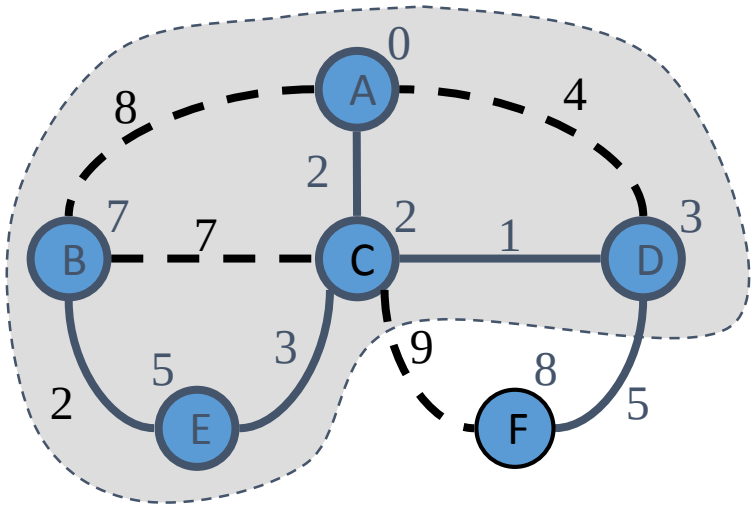
Initialize $D[s] = 0$ and $D[v] = \infty$ for each vertex $v \neq s$.
Let a priority queue Q contain all the vertices of G using the D labels as keys.
while Q is not empty **do**
  {pull a new vertex u into the cloud}
  u = value returned by Q.remove_min()
  for each vertex v adjacent to u such that v is in Q **do**
    {perform the ***relaxation*** procedure on edge (u, v)}
    if $D[u] + w(u, v) < D[v]$ **then**
      $D[v] = D[u] + w(u, v)$
      Change to $D[v]$ the key of vertex v in Q.
return the label $D[v]$ of each vertex v

Analysis of Dijkstra's Algorithm

- Graph operations
  - We find all the incident edges once for each vertex
- Label operations
  - We set/get the distance and locator labels of vertex z $O(\deg(z))$ times
  - Setting/getting a label takes $O(1)$ time
- Priority queue operations
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $O(\log n)$ time
  - The key of a vertex in the priority queue is modified at most $\deg(w)$ times, where each key change takes $O(\log n)$ time
- Dijkstra's algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list/map structure
  - Recall that $\sum_v \deg(v) = 2m$
- The running time can also be expressed as $O(m \log n)$ since the graph is connected

Why Dijkstra's Algorithm Works

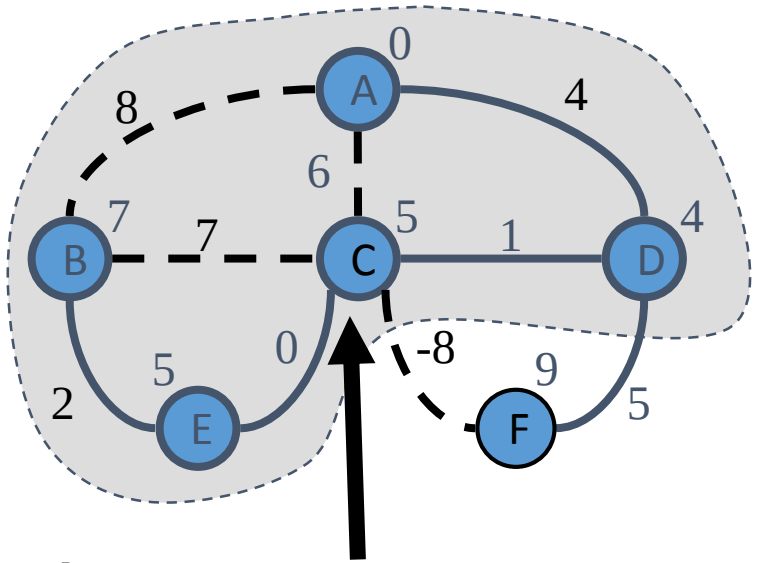
- Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.
  - Suppose it didn't find all shortest distances. Let F be the first wrong vertex the algorithm processed.
  - When the previous node, D, on the true shortest path was considered, its distance was correct
  - But the edge (D,F) was relaxed at that time!
  - Thus, so long as d(F)≥d(D), F's distance cannot be wrong. That is, there is no wrong vertex



Why It Doesn't Work for Negative-Weight Edges

- Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.

- If a node with a negative incident edge were to be added late to the cloud, it could mess up distances for vertices already in the cloud.



C's true distance is 1, but it is already in the cloud with d(C)=5!

GRAPHS WITH NEGATIVE EDGES

```
void weightedNegative( Vertex s )
{
    Queue<Vertex> q = new Queue<Vertex>( );

    for each Vertex v
        v.dist = INFINITY;

    s.dist = 0;
    q.enqueue( s );

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

        for each Vertex w adjacent to v
            if( v.dist + cvw < w.dist )
            {
                // Update w
                w.dist = v.dist + cvw;
                w.path = v;
                if( w is not already in q )
                    q.enqueue( w );
            }
    }
}
```

Figure 9.32 Pseudocode for weighted shortest-path algorithm with negative edge costs

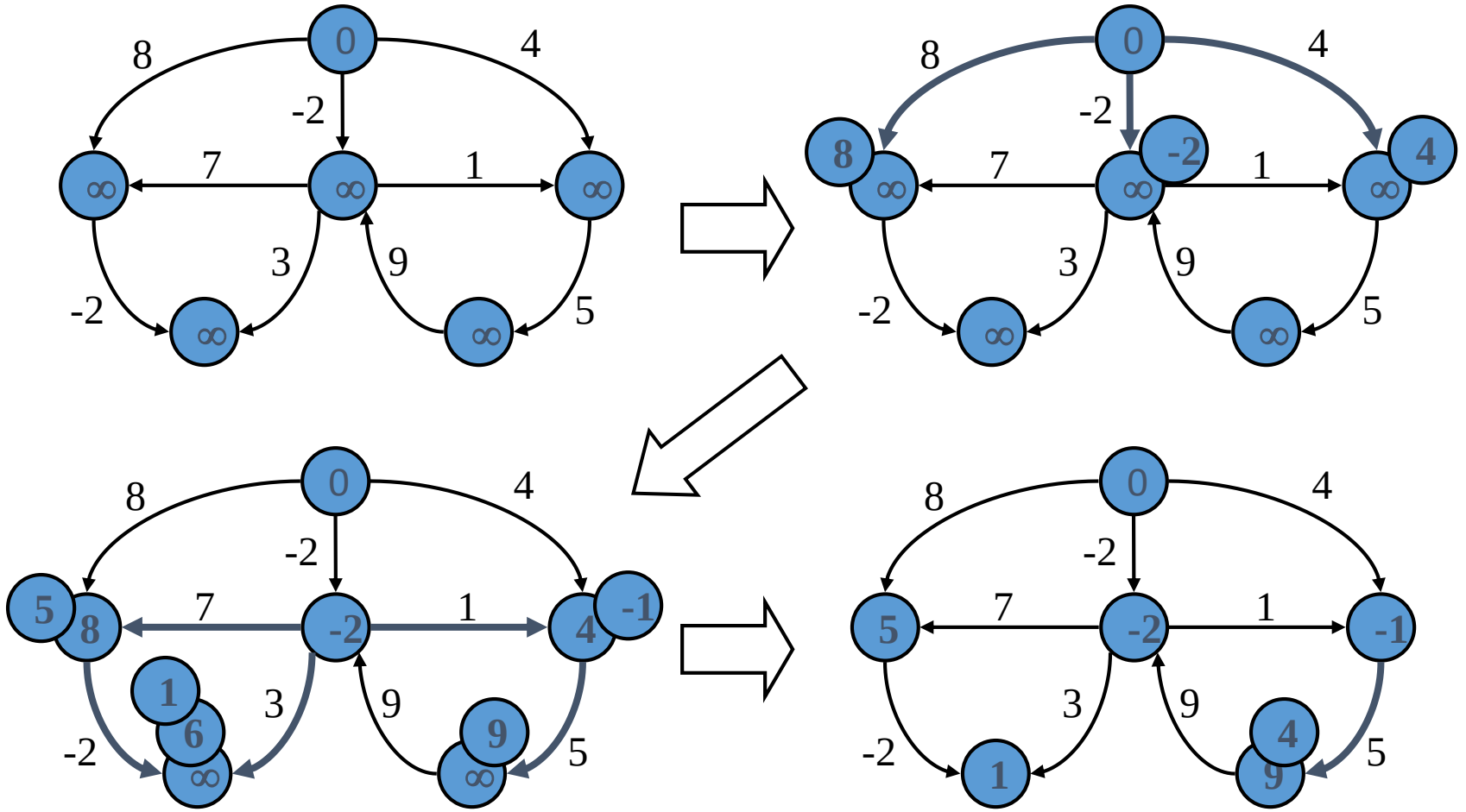
Bellman-Ford Algorithm

- Works even with negative-weight edges
- Must assume directed edges (for otherwise we would have negative-weight cycles)
- Iteration i finds all shortest paths that use i edges.
- Running time: O(nm).
- Can be extended to detect a negative-weight cycle if it exists
  - How?

```
Algorithm BellmanFord(G, s)
    for all v ∈ G.vertices()
        if v = s
            setDistance(v, 0)
        else
            setDistance(v, ∞)
    for i ← 1 to n − 1 do
        for each e ∈ G.edges()
            { relax edge e }
            u ← G.origin(e)
            z ← G.opposite(u,e)
            r ← getDistance(u) + weight(e)
            if r < getDistance(z)
                setDistance(z,r)
```

Bellman-Ford Example

Nodes are labeled with their d(v) values



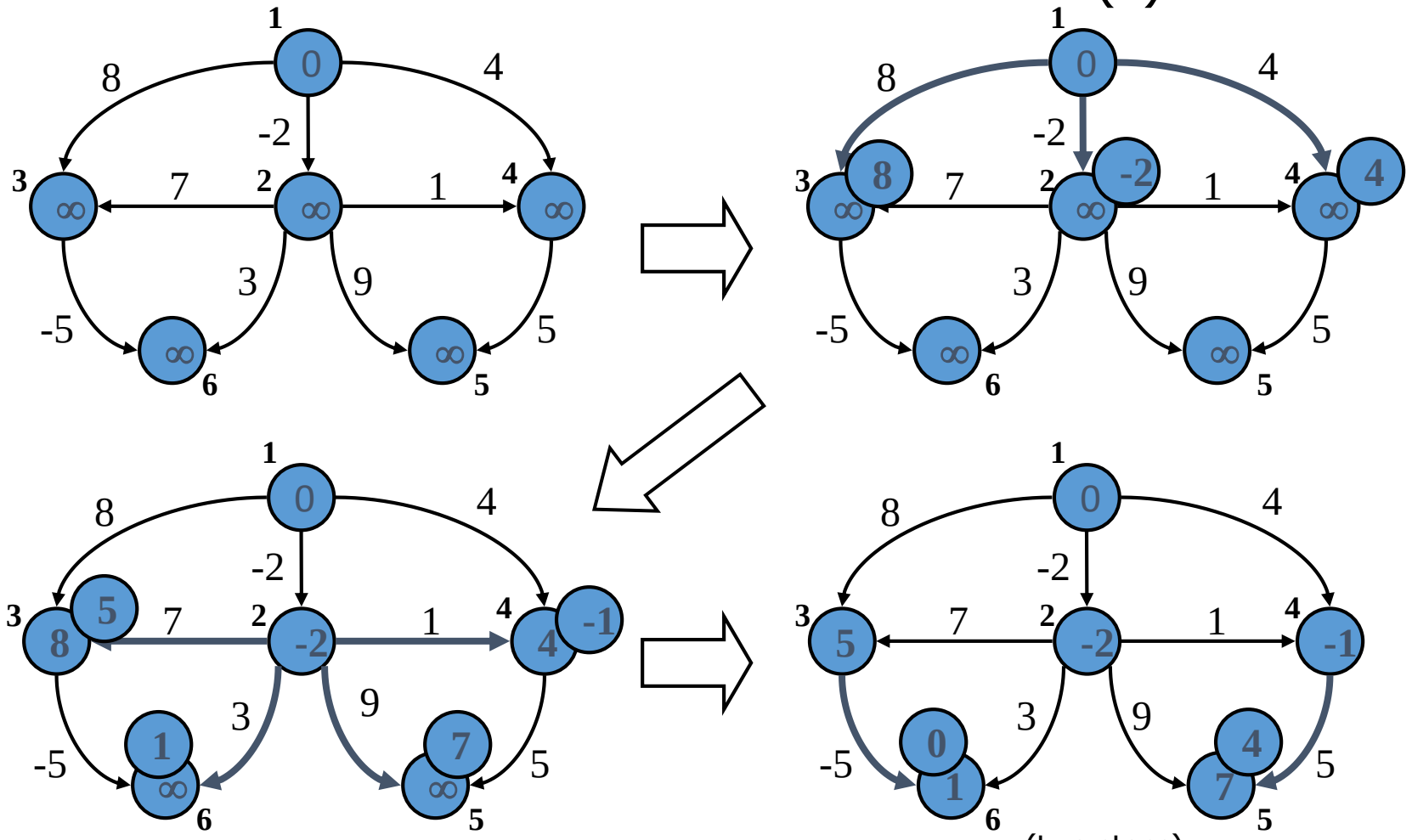
DAG-based Algorithm

- Works even with negative-weight edges
- Uses topological order
- Doesn't use any fancy data structures
- Is much faster than Dijkstra's algorithm
- Running time: O(n+m).

```
Algorithm DagDistances(G, s)
    for all v ∈ G.vertices()
        if v = s
            setDistance(v, 0)
        else
            setDistance(v, ∞)
    { Perform a topological sort of the vertices }
    for u ← 1 to n do    {in topological order}
        for each e ∈ G.outEdges(u)
            { relax edge e }
            z ← G.opposite(u,e)
            r ← getDistance(u) + weight(e)
            if r < getDistance(z)
                setDistance(z,r)
```

DAG Example

Nodes are labeled with their d(v) values



ACYCLIC GRAPHS

- The DAG algorithm can be done in one pass, selections and update can take place as the topological sort is being performed.
- Important use of acyclic graphs is critical path analysis.

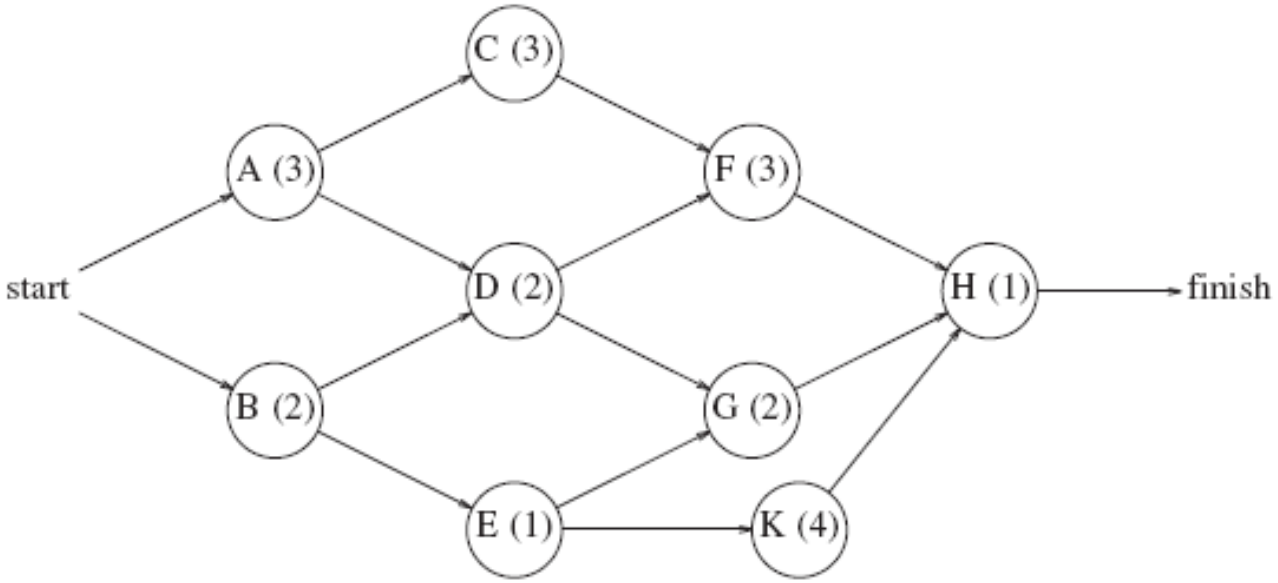


Figure 9.33 Activity-node graph

ACYCLIC GRAPHS

- This type of graph could be used to model construction projects.
- Convert the activity-node graph to event node graph, to calculate the earliest completion time, activities that can be delayed without affecting the minimum completion time.

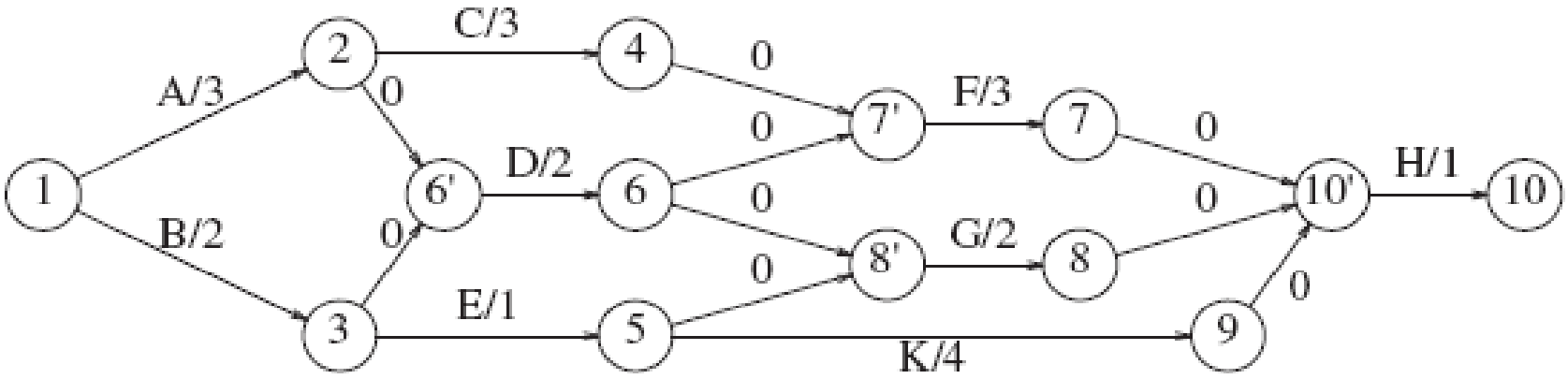


Figure 9.34 Event-node graph

ACYCLIC GRAPHS

- To find the earliest completion time of the project, find the longest path from the first event to last event.
- if EC_i is the earliest completion time for node i, then use rule

  $EC_1 = 0$

  $EC_w = \max(EC_v + c_{v,w})$ for every edge $(v,w) \in E$
- Compute the latest time LC_i

  $LC_n = EC_n$

  $LC_v = \min(LC_w - c_{v,w})$ for every edge $(v,w) \in E$

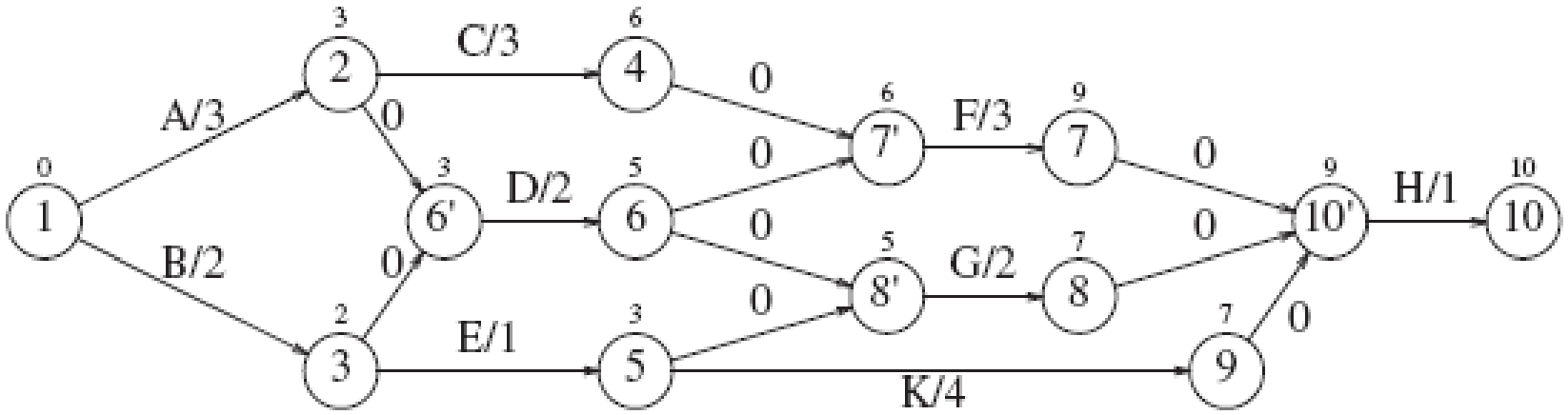


Figure 9.35 Earliest completion times

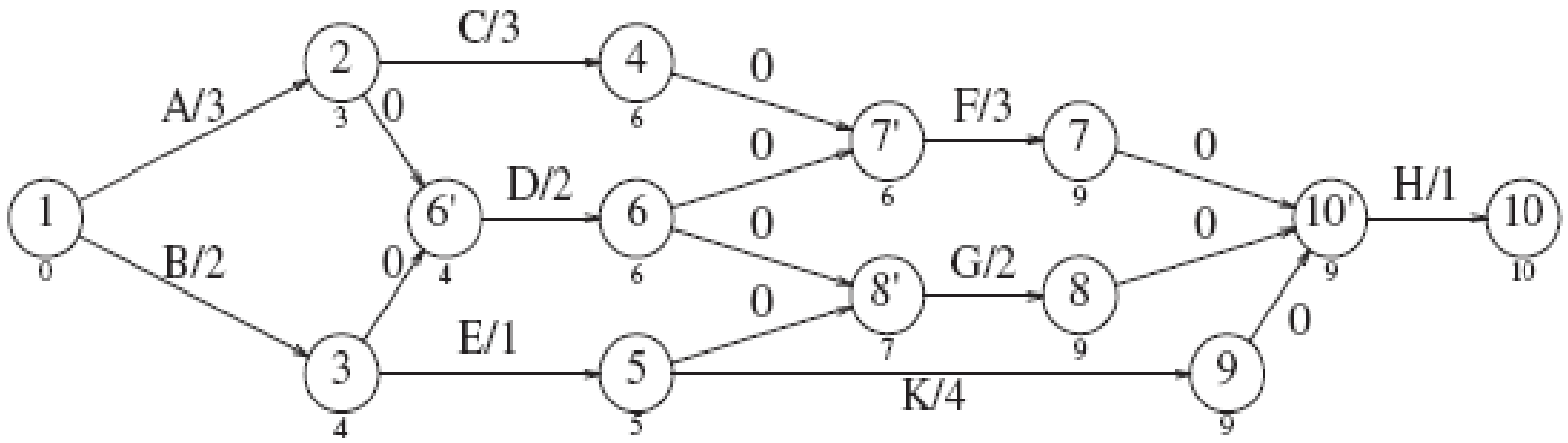


Figure 9.36 Latest completion times

ACYCLIC GRAPHS

- The earliest completion times are computed for vertices by their topological order and the latest completion times are computed by reverse topological order.

- The slack time for each edge in the event-node graph represents the amount of time that the completion of the corresponding activity can be delayed without delaying the overall completion.

$$Slack_{(v,w)} = LC_w - EC_v - c_{v,w}$$

ACYCLIC GRAPHS

- The critical activities are the ones with zero slack, which must be finished on schedule.
- There is at least one path consisting entirely of zero-slack edges, such a path is a critical path.

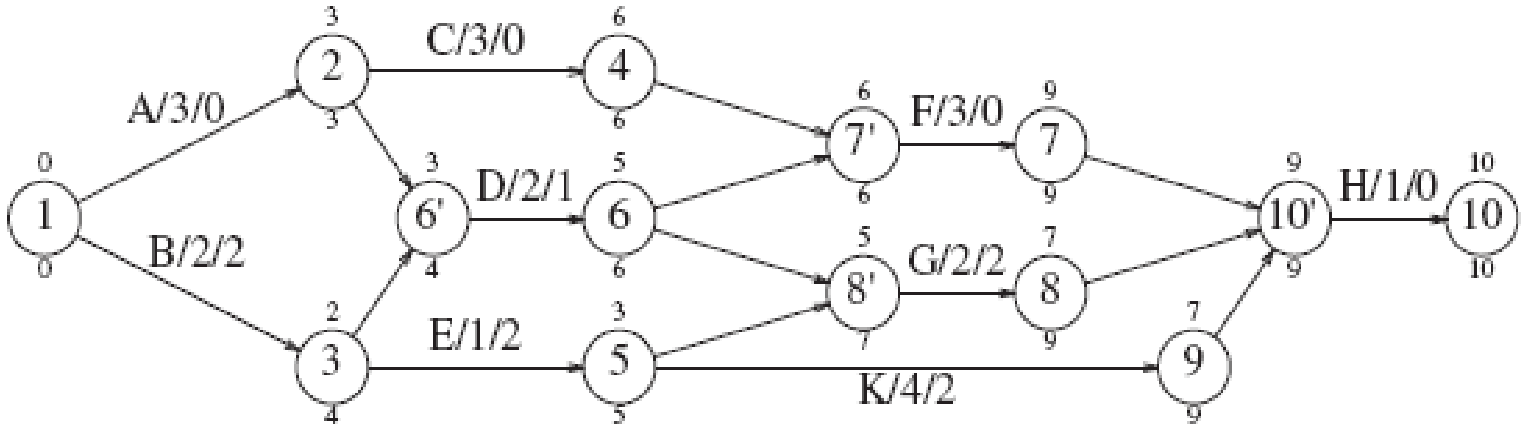


Figure 9.37 Earliest completion time, latest completion time, and slack

ALL-PAIRS SHORTEST PATH PROBLEM

Suppose we are given a directed graph G=(V,E) and a weight function w: E->R.

We assume that G does not contain cycles of weight 0 or less.

The All-Pairs Shortest Path Problem asks to find the length of the shortest path between any pair of vertices in G.

FLOYD-WARSHALL (not in book)

We will now investigate a dynamic programming solution that solved the problem in $O(n^3)$ time for a graph with n vertices.

This algorithm is known as the Floyd-Warshall algorithm, but it was apparently described earlier by Roy.

REPRESENTATION OF THE INPUT

We assume that the input is represented by a weight matrix $W= (w_{ij})_{i,j}$ that is defined by

$w_{ij}= 0$ if $i=j$

$w_{ij}= w(i,j)$ if $i \neq j$ and (i,j) in E

$w_{ij}= \infty$ if $i \neq j$ and (i,j) not in E

If the graph has n vertices, we return a distance matrix (d_{ij}), where d_{ij} the length of the path from i to j.

INTERMEDIATE VERTICES

Without loss of generality, we will assume that V={1,2,…,n}, i.e., that the vertices of the graph are numbered from 1 to n.

Given a path $p=(v_1, v_2,\ldots, v_m)$ in the graph, we will call the vertices v_k with index k in {2,…,m-1} the intermediate vertices of p.

FLOYD'S ALGORITHM

- For any pair of vertices v_i, $v_j \in V$, consider all paths from v_i to v_j whose intermediate vertices belong to the set $\{v_1, v_2, \ldots, v_k\}$. Let $p_{i,j}^{(k)}$ (of weight $d_{i,j}^{(k)}$) be the minimum-weight path among them.
- If vertex v_k is not in the shortest path from v_i to v_j, then $p_{i,j}^{(k)}$ is the same as $p_{i,j}^{(k-1)}$.
- If v_k is in $p_{i,j}^{(k)}$, then we can break $p_{i,j}^{(k)}$ into two paths - one from v_i to v_k and one from v_k to v_j . Each of these paths uses vertices from $\{v_1, v_2, \ldots, v_{k-1}\}$.

FLOYD'S ALGORITHM

From our observations, the following recurrence relation follows:

$$d_{i,j}^{(k)} = \begin{cases} w(v_i, v_j) & \text{if } k = 0 \\ \min\left\{d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\right\} & \text{if } k \geq 1 \end{cases}$$

This equation must be computed for each pair of nodes and for $k = 1, n$. The serial complexity is $O(n^3)$.

FLOYD'S ALGORITHM

```
1.    procedure FLOYD_ALL_PAIRS_SP(A)
2.    begin
3.        D^(0) = A;
4.        for k := 1 to n do
5.            for i := 1 to n do
6.                for j := 1 to n do
7.                    d_{i,j}^(k) := min(d_{i,j}^(k-1), d_{i,k}^(k-1) + d_{k,j}^(k-1));
8.    end FLOYD_ALL_PAIRS_SP
```

Floyd's all-pairs shortest paths algorithm. This program computes the all-pairs shortest paths of the graph *G = (V,E)* with adjacency matrix *A*.

REMARKS

- A shortest path does not contain any vertex twice, as this would imply that the path contains a cycle. By assumption, cycles in the graph have a positive weight, so removing the cycle would result in a shorter path, which is impossible.
- Consider a shortest path p from i to j such that the intermediate vertices are from the set {1,…,k}.
  - If the vertex k is not an intermediate vertex on p, then $d_{ij}^{(k)} = d_{ij}^{(k-1)}$
  - If the vertex k is an intermediate vertex on p, then $d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$

Interestingly, in either case, the subpaths contain merely nodes from {1,…,k-1}.

- Therefore, we can conclude that

  $d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)} , d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$

RECURSIVE FORMULATION

If we do not use intermediate nodes, i.e., when k=0, then

$$d_{ij}^{(0)} = w_{ij}$$

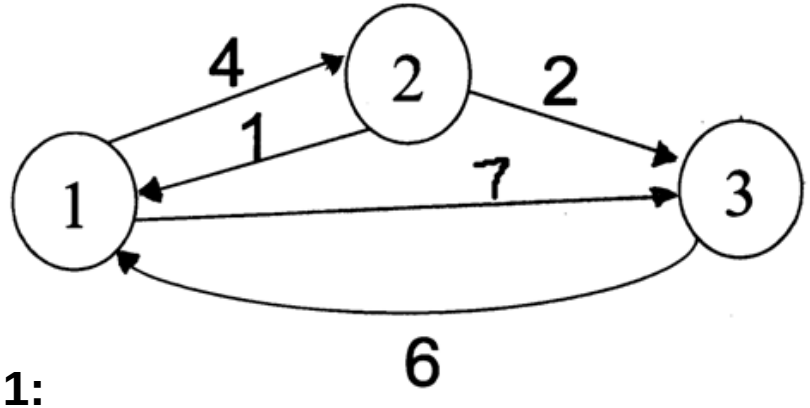
If k>0, then

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)} , d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$$

Floyd Warshall Algorithm - Example

$$D^{(0)} = \begin{bmatrix} 0 & 4 & 7 \\ 1 & 0 & 2 \\ 6 & \infty & 0 \end{bmatrix}$$

Original weights.



$$D^{(1)} = \begin{bmatrix} 0 & 4 & 7 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$

Consider Vertex 1:

D(3,2) = D(3,1) + D(1,2)

$$D^{(2)} = \begin{bmatrix} 0 & 4 & 6 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$

Consider Vertex 2:

D(1,3) = D(1,2) + D(2,3)

$$D^{(3)} = \begin{bmatrix} 0 & 4 & 6 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$

Consider Vertex 3:

Nothing changes.

FLOYD WARSHALL – PATH RECONSTRUCTION

- The path matrix will store the last vertex visited on the path from i to j.
  - So path[i][j] = k means that in the shortest path from vertex i to vertex j, the LAST vertex on that path before you get to vertex j is k.

- Based on this definition, we must initialize the path matrix as follows:
  - path[i][j] = i if i!=j and there exists an edge from i to j
  - = NIL otherwise

- The reasoning is as follows:
  - If you want to reconstruct the path at this point of the algorithm when you aren't allowed to visit intermediate vertices, the previous vertex visited MUST be the source vertex i.
  - NIL is used to indicate the absence of a path.

FLOYD WARSHALL – PATH RECONSTRUCTION

- Before you run Floyd's, you initialize your distance matrix ***D*** and path matrix ***P*** to indicate the use of no immediate vertices.
  - (Thus, you are only allowed to traverse direct paths between vertices.)

- Then, at each step of Floyd's, you essentially find out whether or not using vertex k will *improve* an estimate between the distances between vertex i and vertex j.

- <u>If it ***does improve*** the estimate here's what you need to record:</u>

  1) record the new shortest path weight between i and j
  2) record the fact that the shortest path between i and j goes through k

FLOYD WARSHALL – PATH RECONSTRUCTION

- If it ***does improve*** the estimate here's what you need to record:
  1) record the new shortest path weight between i and j
     - **We don't need to change our path and we do not update the path matrix**
  2) record the fact that the shortest path between i and j goes through k
     - **We want to store the last vertex from the shortest path from vertex k to vertex j.** ***This will NOT necessarily be k, but rather, it will be path[k][j].***

This gives us the following update to our algorithm:

```
if (D[i][k]+D[k][j] < D[i][j]) { // Update is necessary to use k as intermediate vertex
    D[i][j] = D[i][k]+D[k][j];
    path[i][j] = path[k][j];
}
```

PATH RECONSTRUCTION

- Now, the once this path matrix is computed, we have all the information necessary to reconstruct the path.
  - Consider the following path matrix (indexed from 1 to 5 instead of 0 to 4):

NIL	**3**	**4**	**5**	**1**
4	**NIL**	**4**	**2**	**1**
4	**3**	**NIL**	**2**	**1**
4	**3**	**4**	**NIL**	**1**
4	**3**	**4**	**5**	**NIL**

- Reconstruct the path from vertex1 to vertex 2:
  - First look at path[1][2] = 3. This signifies that on the path from 1 to 2, 3 is the last vertex visited before 2.
    - Thus, the path is now, 1…3->2.
  - Now, look at path[1][3], this stores a 4. Thus, we find the last vertex visited on the path from 1 to 3 is 4.
  - So, our path now looks like 1…4->3->2. So, we must now look at path[1][4]. This stores a 5,
  - thus, we know our path is 1…5->4->3->2. When we finally look at path[1][5], we find 1,
  - which means our path really is 1->5->4->3->2.

TRANSITIVE CLOSURE

- Computing a transitive closure of a graph gives you complete information about which vertices are connected to which other vertices.

- Input:
  - Un-weighted graph G: $W[i][j] = 1$, if $(i,j) \in E$, $W[i][j] = 0$ otherwise.
- Output:
  - $T[i][j] = 1$, if there is a path from i to j in G, $T[i][j] = 0$ otherwise.
- Algorithm:
  - Just run Floyd-Warshall with weights 1, and make $T[i][j] = 1$, whenever $D[i][j] < \infty$.
  - More efficient: use only Boolean operators

TRANSITIVE CLOSURE

```
Transitive-Closure(W[1..n][1..n])
01 T ← W     // T(0)
02 for k ←1 to n do // compute T(k)
03    for i ←1 to n do
04       for j ←1 to n do
05          T[i][j] ← T[i][j] ∨ (T[i][k] ∧ T[k][j])
06 return T
```

- This is the SAME as the other Floyd-Warshall Algorithm, except for when we find a non-infinity estimate, we simply add an edge to the transitive closure graph.

- Every round we build off the previous paths reached.
  - After iterating through all vertices being intermediate vertices, we have tried to connect all pairs of vertices i and j through all intermediate vertices k.

Applications

- **All-pairs shortest path** - Computing shortest paths between every pair of vertices in a directed graph.
- **Transitive closure**. Basically for determining reachability of nodes. Transitive closure has many uses in determining relationships between things.
- **Detecting negative-weight cycles in graphs**. (If there is a negative weight cycle, the distance from a the start node to itself will be negative after running the algorithm)
- **Bipartiteness** - Basically, you can detect whether a graph is bipartite via parity. Set all edges to a weight of 1, all vertices at an odd shortest-distance away from some arbitrary root V are in one subset, all vertices at an even distance away are in another subset.

MINIMAX

- Minimax in graph problems involves finding a path between two nodes that minimizes the maximum cost along the path.
- Example problems include finding feasible paths to take by car with a limited fuel tank and rest stations at every node.
- A little tweak in the Floyd-Warshall algorithm enables finding minimax. You can also add in an update to a predecessor matrix in order to keep track of the actual path found.

MINIMAX

- *// Minimax variant of Floyd-Warshall example*
  // input: d is an distance matrix for n nodes e.g. d[i][j] is the direct distance from i to j.
  // the distance from a node to itself e.g. d[i][i] should be initialized to 0.
  // output: d[i][j] will contain the length of the minimax path from i to j.

```
for (k=0; k<n; k++)
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      d[i][j] = min(d[i][j], max(d[i][k], d[k][j]));
```

MAXIMIN

- The other way around from Minimax - here you have problems where you need to find the path the maximizes the minimum cost along a path.
- Example problems include trying to maximize the load a cargo truck can take when roads along a path may have a weight limit, or trying to find a network routing path that meets a minimum bandwidth requirement for some application. See below for the tweak for Floyd-Warshall that enables finding maximin.
-

```
// Maximin variant of Floyd-Warshall example
// input: d is an distance matrix for n nodes e.g. d[i][j] is the direct distance from i to j.
//     the distance from a node to itself e.g. d[i][i] should be initialized to 0.
// output: d[i][j] will contain the length of the maximin path from i to j.
for (k=0; k<n; k++)
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      d[i][j] = max(d[i][j], min(d[i][k], d[k][j]));
```

SHORTEST-PATH EXAMPLE

- Java routine to compute word ladder: each word is formed by changing one character in the ladder's previous word.
- For instance to convert zero to five :

  zero -> hero -> here -> hire -> fire -> five

- Objective is to write a single source unweighted shortest path algorithm with a Map representation of the graph.

findChian routine

- *findChian* routine takes the Map representing the adjacency lists and the two words to be connected.
- it returns another Map in which keys are words and the corresponding value is the word prior to the key on the shortest path ladder. This map is called the previousWord
- For instance in our example, the value for the key five is fire , the value for key fire is hire and for hire is here and so on.

The routine is an direct implementation of the pseudo code for unweighted shortest-path graph using queues.

getChainFromPreviousMap

- This routine uses the map constructed by findChain routine to return the words used to form the word ladder, by working its way backwards.
- By using a LinkedList and inserting at the front, we obtain the word ladder in the correct order.

```
1       // Runs the shortest path calculation from the adjacency map, returns a List
2       // that contains the sequence of word changes to get from first to second.
3       // Returns null if no sequence can be found for any reason.
4       public static List<String>
5       findChain( Map<String,List<String>> adjacentWords, String first, String second )
6       {
7           Map<String,String> previousWord = new HashMap<String,String>( );
8           LinkedList<String> q = new LinkedList<String>( );
9
10          q.addLast( first );
11          while( !q.isEmpty( ) )
12          {
13              String current = q.removeFirst( );
14              List<String> adj = adjacentWords.get( current );
15
16              if( adj != null )
17                  for( String adjWord : adj )
18                      if( previousWord.get( adjWord ) == null )
19                      {
20                          previousWord.put( adjWord, current );
21                          q.addLast( adjWord );
22                      }
23          }
24
25          previousWord.put( first, null );
26
27          return getChainFromPreviousMap( previousWord, first, second );
28      }
29
30      // After the shortest path calculation has run, computes the List that
31      // contains the sequence of word changes to get from first to second.
32      // Returns null if there is no path.
33      public static List<String> getChainFromPreviousMap( Map<String,String> prev,
34                                                          String first, String second )
35      {
36          LinkedList<String> result = null;
37
38          if( prev.get( second ) != null )
39          {
40              result = new LinkedList<String>( );
41              for( String str = second; str != null; str = prev.get( str ) )
42                  result.addFirst( str );
43          }
44
45          return result;
46      }
```

Figure 9.38 Java code to find word ladders

MINIMUM SPANNING TREE

- A minimum spanning tree of an undirected graph G is a tree formed from graph edges that connects all the vertices of G at lowest cost.
- Such a tree exists if and only if the graph is connected.

- Properties
  - Number of edges in the minimum spanning tree is |V| - 1.
  - For any spanning tree T, if an edge e that is not in T is added, a cycle is created.

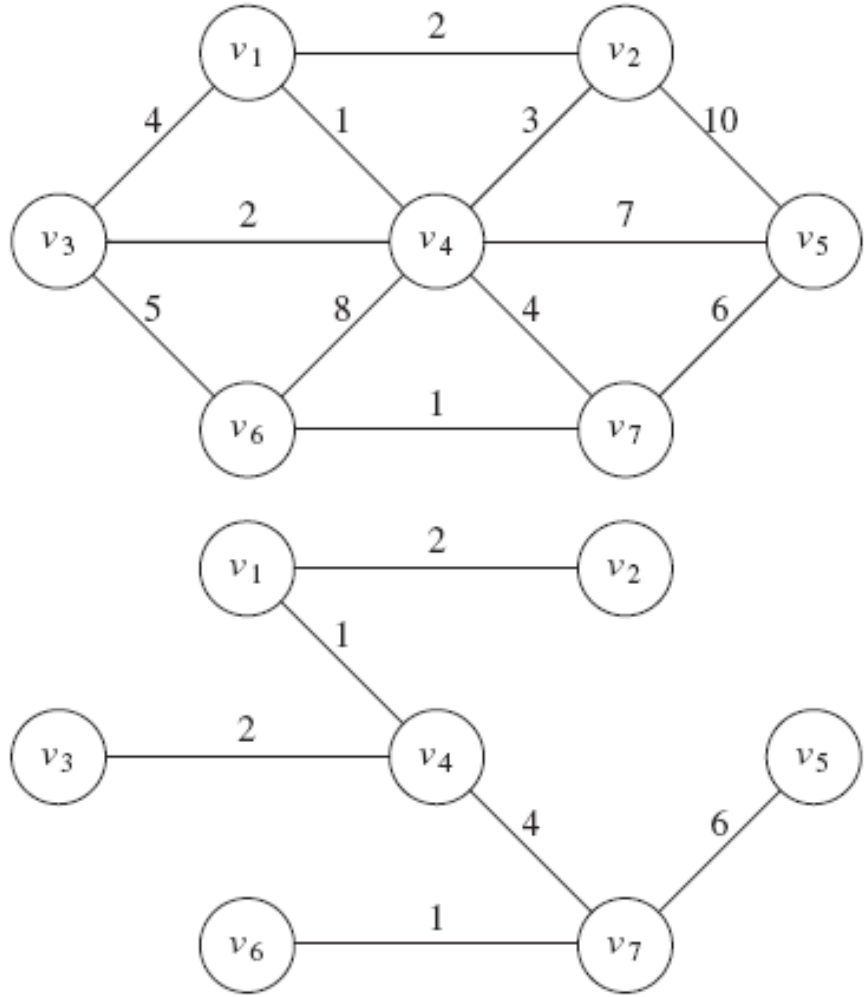


Figure 9.50 A graph G and its minimum spanning tree

PRIM'S ALGORITHM

- In each stage of the algorithm one node is picked as the root, an edge is added, and thus an associated vertex is also added to the tree.
- The algorithm finds, at each stage, a new vertex to add to the tree by choosing the edge (u,v) such that the cost of (u,v) is minimum among all edges where u is in the tree and v is not.
- This algorithm is similar to Dijkstra's algorithm for shortest paths.
- The definition of d_v is different, so is the update rule.
  - After a vertex v is selected, for each unknown vertex w adjacent to v, $d_w = \min(d_w, c_{w,v})$

PRIM'S ALGORITHM

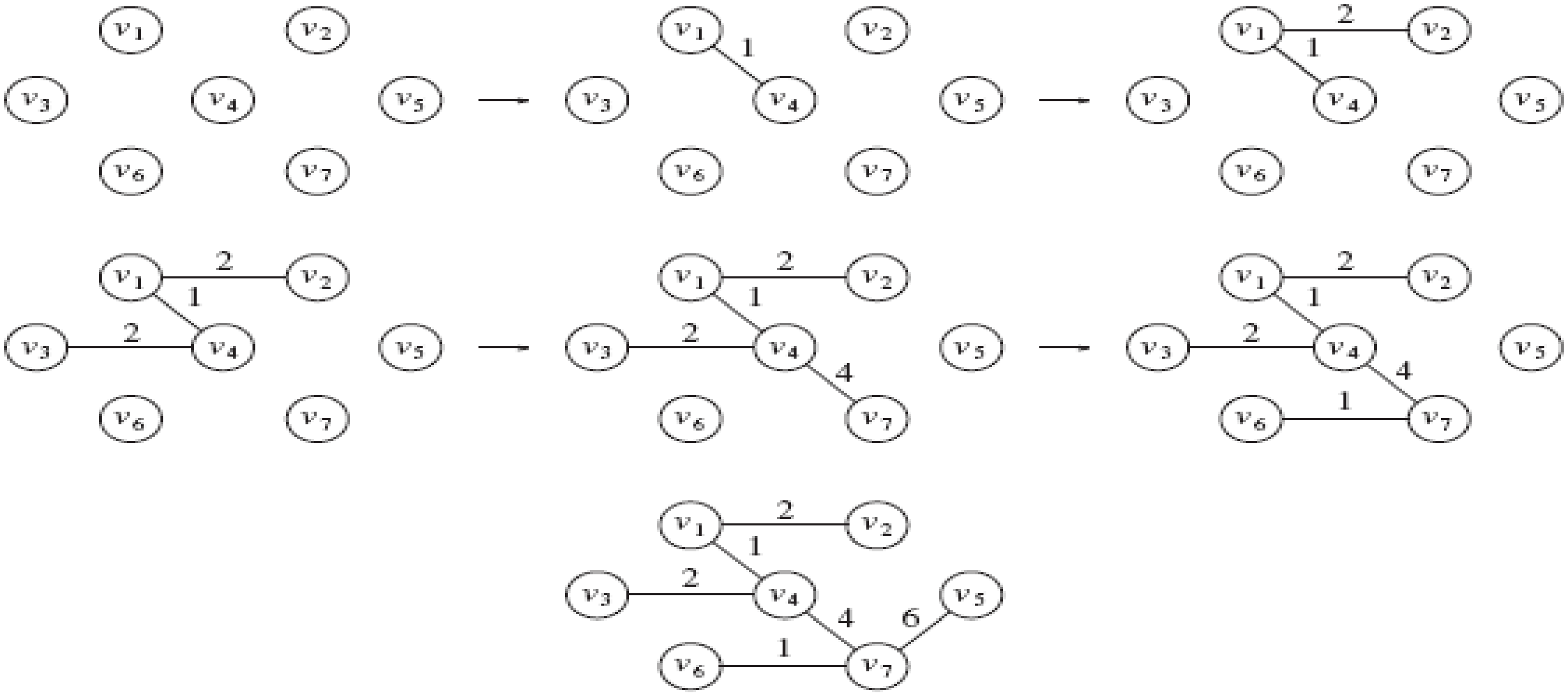
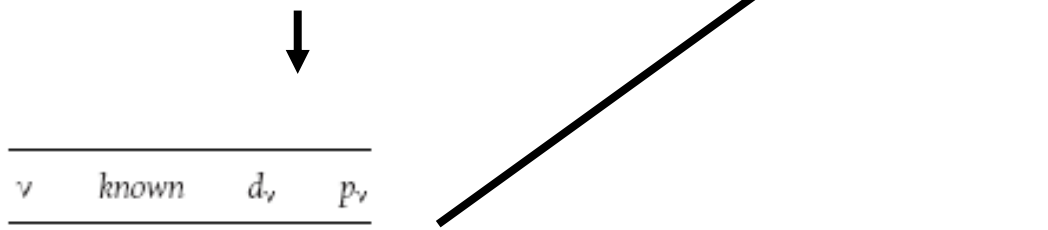


Figure 9.51 Prim's algorithm after each stage

v	*known*	d_v	p_v
v_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

Figure 9.52 Initial configuration of table used in Prim's algorithm



v	*known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	4	v_1
v_4	F	1	v_1
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

Figure 9.53 The table after v_1 is declared *known*

v	*known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	2	v_4
v_4	T	1	v_1
v_5	F	7	v_4
v_6	F	8	v_4
v_7	F	4	v_4

Figure 9.54 The table after v_4 is declared *known*

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	2	v_4
v_4	T	1	v_1
v_5	F	7	v_4
v_6	F	5	v_3
v_7	F	4	v_4

Figure 9.55 The table after v_2 and then v_3 are declared *known*

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	2	v_4
v_4	T	1	v_1
v_5	F	6	v_7
v_6	F	1	v_7
v_7	T	4	v_4

Figure 9.56 The table after v_7 is declared *known*

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	2	v_4
v_4	T	1	v_1
v_5	T	6	v_7
v_6	T	1	v_7
v_7	T	4	v_4

Figure 9.57 The table after v_6 and v_5 are selected (Prim's algorithm terminates)

KRUSKAL'S ALGORITHM

- A second greedy strategy is continually to select the edge in order of smallest weight and accept an edge if it does not cause a cycle.
- Formally, Kruskal,s algorithm maintains a forest - a collection of trees.
- Initially, there are |V| single-node trees.
- Adding an edge merges the two trees into one, the algorithm terminates when there is only one tree.
- This tree is the minimum spanning tree

KRUSKAL'S ALGORITHM

- The algorithm terminates when enough edges are accepted. The appropriate data structure is the union/find algorithm.
- Each vertex is initially in its own set. If u and v are in the same set, the edge is rejected because adding (u,v) would form a cycle.
- The edges could be sorted to facilitate the selection , but building a heap in linear time is a much better idea.
- The worst case running time of this algorithm is O(|E| log|E|), which is dominated by the heap operations

KRUSKAL'S ALGORITHM

Edge	Weight	Action
(v_1, v_4)	1	Accepted
(v_6, v_7)	1	Accepted
(v_1, v_2)	2	Accepted
(v_3, v_4)	2	Accepted
(v_2, v_4)	3	Rejected
(v_1, v_3)	4	Rejected
(v_4, v_7)	4	Accepted
(v_3, v_6)	5	Rejected
(v_5, v_7)	6	Accepted

Figure 9.58 Action of Kruskal's algorithm on G

KRUSKAL'S ALGORITHM

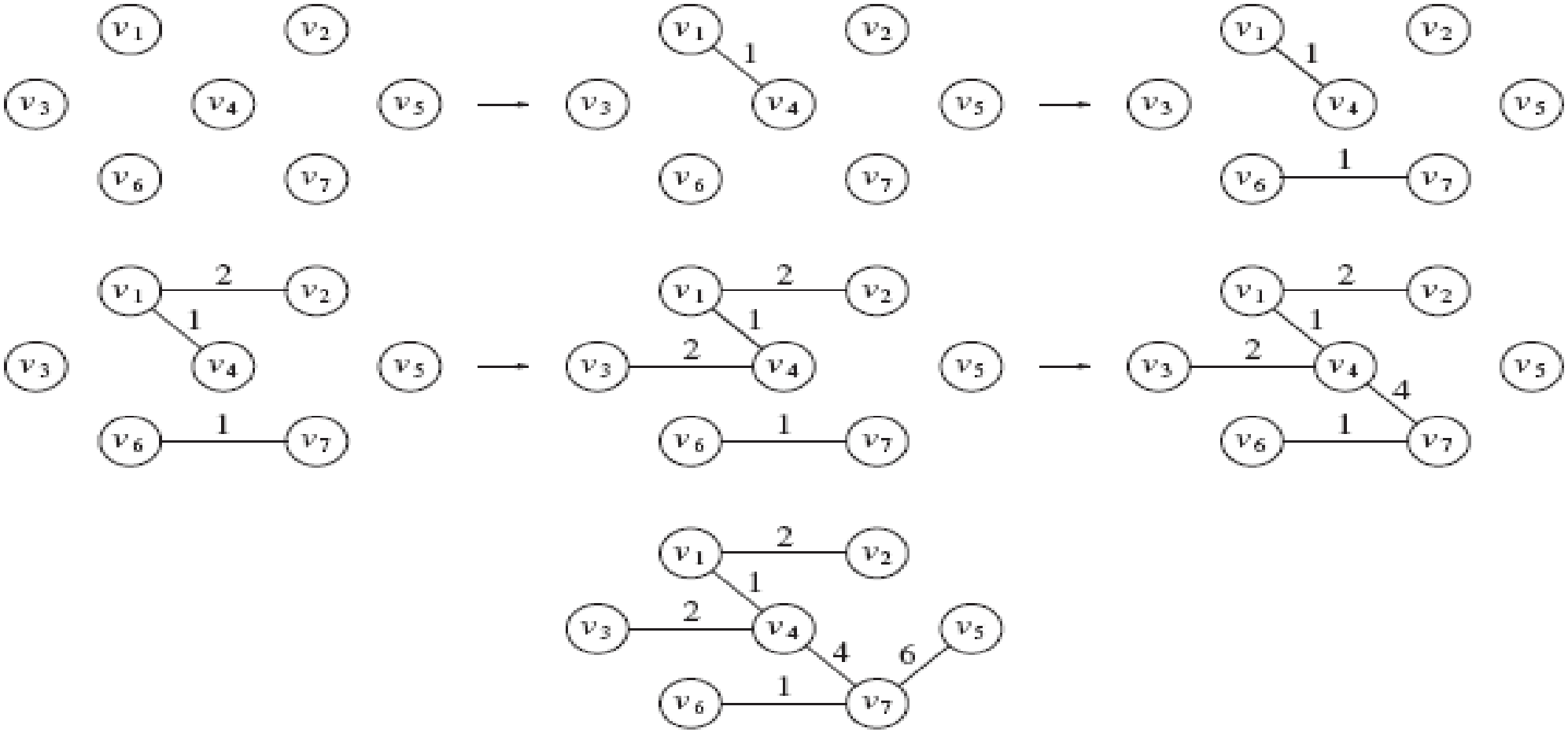


Figure 9.59 Kruskal's algorithm after each stage

KRUSKAL'S ALGORITHM

```
ArrayList<Edge> kruskal( List<Edge> edges, int numVertices )
{
    DisjSets ds = new DisjSets( numVertices );
    PriorityQueue<Edge> pq = new PriorityQueue<>( edges );
    List<Edge> mst = new ArrayList<>( );

    while( mst.size( ) != numVertices - 1 )
    {
        Edge e = pq.deleteMin( );       // Edge e = (u, v)
        SetType uset = ds.find( e.getu( ) );
        SetType vset = ds.find( e.getv( ) );

        if( uset != vset )
        {
            // Accept the edge
            mst.add( e );
            ds.union( uset, vset );
        }
    }
    return mst;
}
```

Figure 9.60 Pseudocode for Kruskal's algorithm

APPLICATION OF DEPTH –FIRST SEACH

- Depth-first search is a generalization of preorder traversal. Starting at some vertex, v, process v first and then recursively traverse all vertices adjacent to v.
- If the graph is not connected the strategy might fail to visit some nodes. We then search for an unmarked node and apply depth-first traversal there.

```
void dfs( Vertex v )
{
    v.visited = true;
    for each Vertex w adjacent to v
        if( !w.visited )
            dfs( w );
}
```

Figure 9.61 Template for depth-first search (pseudocode)

Undirected Graphs

- An undirected graph is connected if and only if a depth-first search from any node visits every node.
- A preorder numbering of the tree, using only tree edges, tells us the order in which the vertices were marked.

- If the graph is not connected, then processing all nodes (and edges) requires several calls to dfs, and each generates a tree. The entire collection is s depth-first forest.

Undirected Graphs

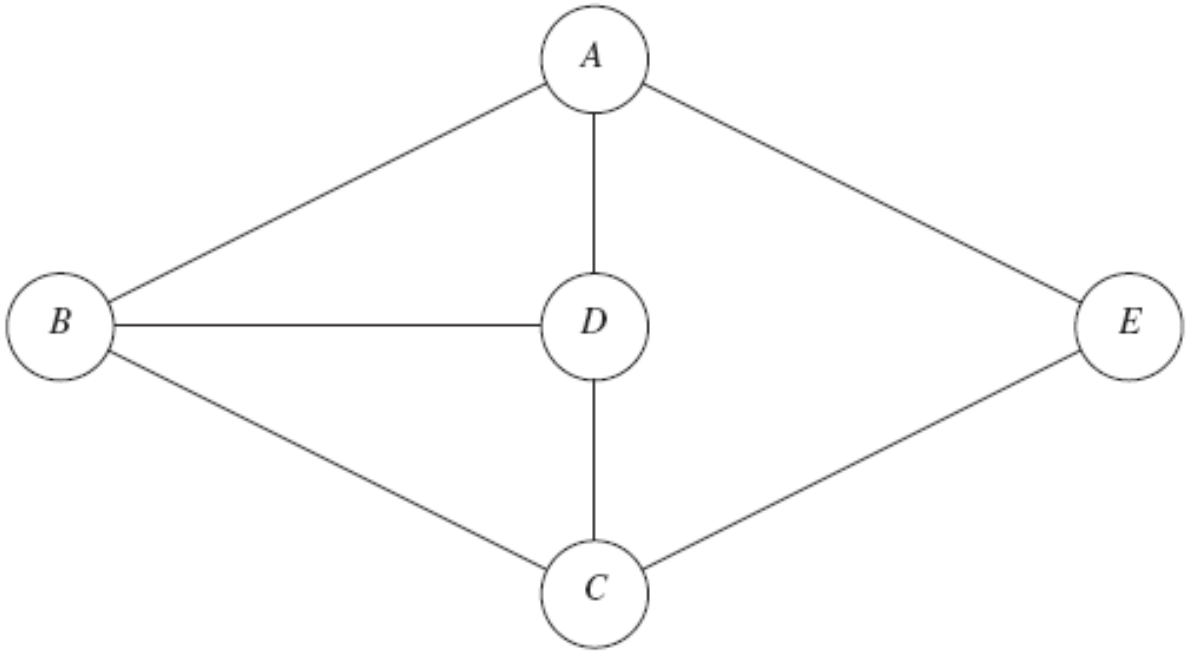


Figure 9.62 An undirected graph

DEPTH-FIRST SEARCH

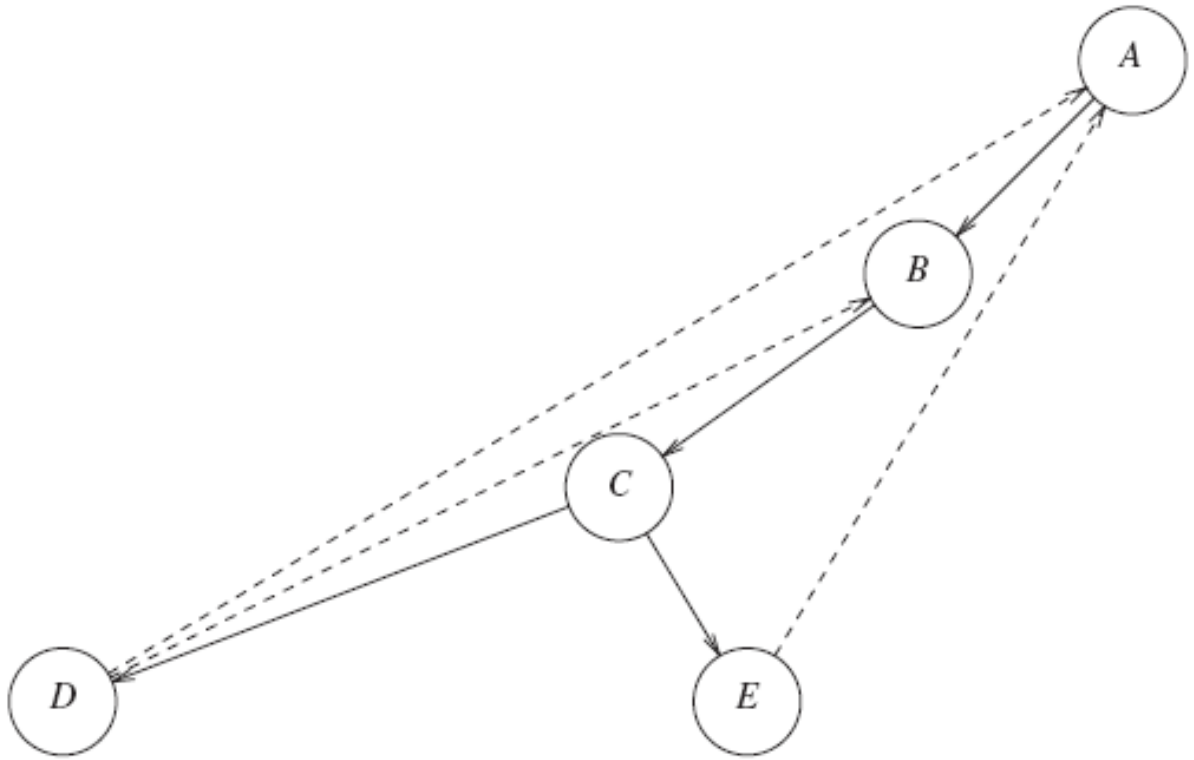


Figure 9.63 Depth-first search of previous graph

BICONNECTIVITY

- A connected undirected graph is biconnected if there are no vertices whose removal disconnects the rest of the graph.

- If a graph is not biconnected, the vertices whose removal would disconnect the graph are known as articulation points

- Depth-first search provides a linear – time algorithm to find all articulation points in a connected graph.

C and D are articulation points the graph. The removal of C disconnects G and the removal of D disconnects E and F, from the rest of the graph

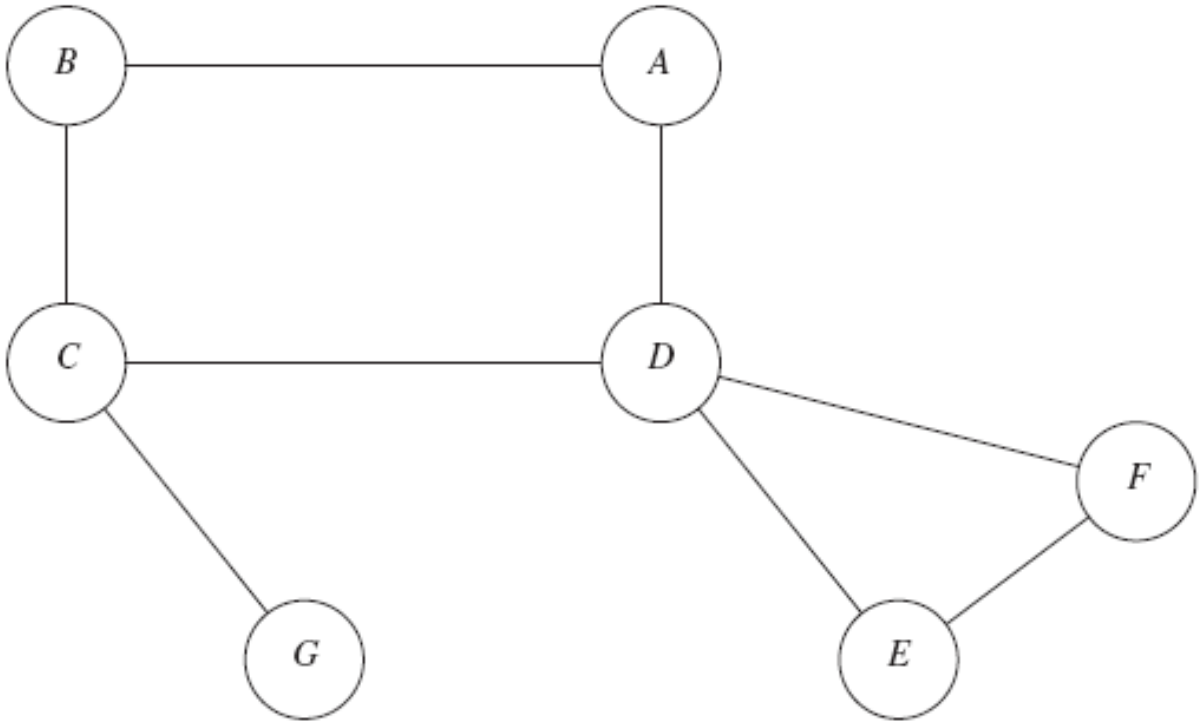


Figure 9.64 A graph with articulation points C and D

ALGORITHM

- First starting at any vertex, perform a depth-first search and number the nodes as they are visited. This number is called Num(v) for vertex v.
- For each vertex v in depth-first scanning tree, compute the lowest-numbered vertex, that is reachable from v by taking zero or more tree edges and one back edge. This number is called Low(v).

Num and Low for every vertex is shown.

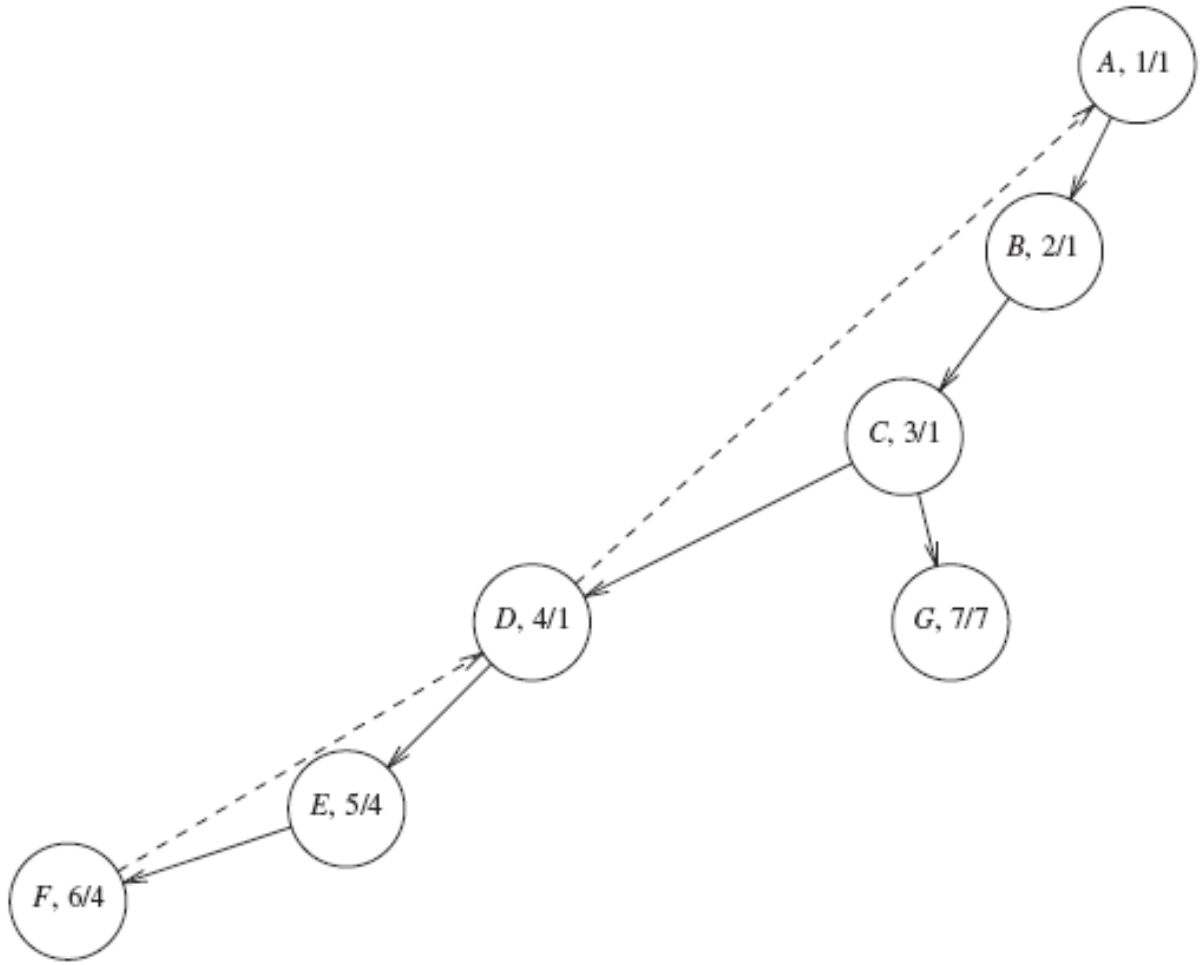


Figure 9.65 Depth-first tree for previous graph, with *Num* and *Low*

ALGORITHM

- The lowest numbered vertex reachable by A, B and C is vertex 1 (A), because they can all take tree edges to D and then one back to A.
- To compute Low perform post order traversal of the depth-first spanning tree. By definition of Low, Low(v) is the minimum of
  - Num(v)
  - the lowest Num(w) among all back edges (v,w)
  - the lowest Low(w), among all tree edges(v,w)

ALGORITHM

- For any edge (v,w) we can tell whether it is a tree edge or a back edge merely by checking Num(v) and Num(w)
- To compute Low(v): scan down v's adjacency list, apply the proper rule, and keep track of the minimum.
- The running time is O(|E| + |V|)
- To find the articulation point apply the following rules:
  - The root is an articulation point if and only if it has more than one child.
  - Any other vertex v is an articulation point if and only if v had some child w such that Low(w) > = Num(v)

C is chosen as root for the depth-first search

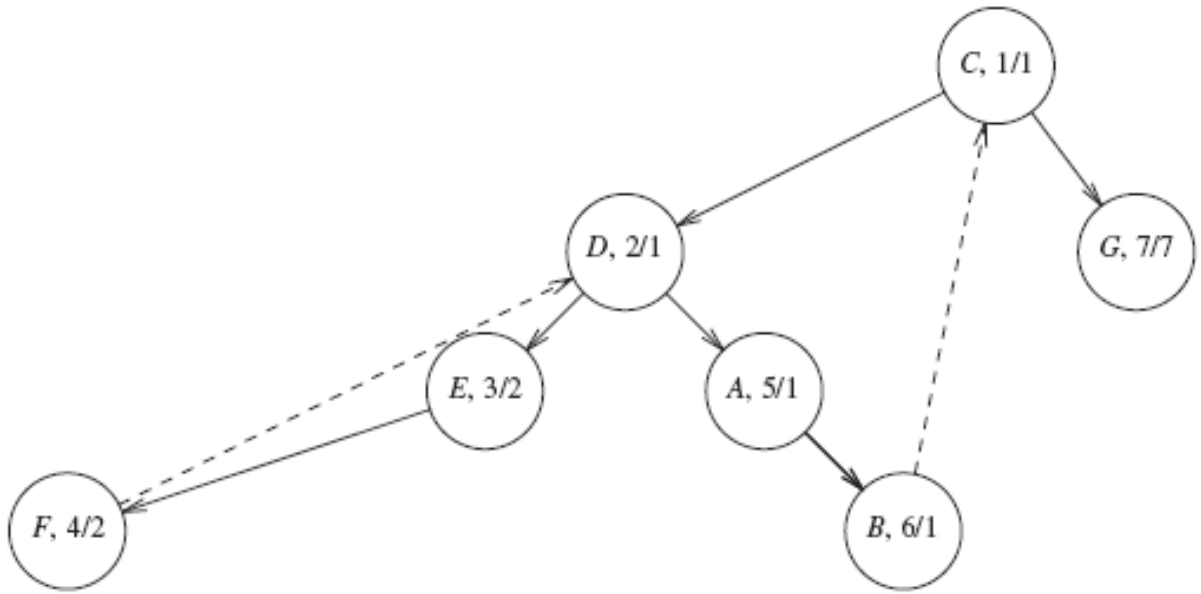


Figure 9.66 Depth-first tree that results if depth-first search starts at *C*

Pseudocode

- Vertex contains the data fields visited, num, low and parent.
- The algorithm can be implemented by doing a preorder traversal to compute Num and a postorder traversal to compute Low and the articulation points.
- The last if statement handles a special case; if w is adjacent to v, then recursive call to w will find v is adjacent to w. This is not a back edge, only an edge that has already been considered and needs to be ignored.

Pseudocode For assigning Num

```
// Assign Num and compute parents
void assignNum( Vertex v )
{
    v.num = counter++;
    v.visited = true;
    for each Vertex w adjacent to v
        if( !w.visited )
        {
            w.parent = v;
            assignNum( w );
        }
}
```

Figure 9.67 Routine to assign *Num* to vertices (pseudocode)

Pseudocode For assigning Low

```
// Assign low; also check for articulation points.
void assignLow( Vertex v )
{
    v.low = v.num;  // Rule 1
    for each Vertex w adjacent to v
    {
        if( w.num > v.num )  // Forward edge
        {
            assignLow( w );
            if( w.low >= v.num )
                System.out.println( v + " is an articulation point" );
            v.low = min( v.low, w.low );  // Rule 3
        }
        else
        if( v.parent != w )  // Back edge
            v.low = min( v.low, w.num );  // Rule 2
    }
}
```

Figure 9.68 Pseudocode to compute *Low* and to test for articulation points (test for the root is omitted)

Combined

```
void findArt( Vertex v )
{
    v.visited = true;
    v.low = v.num = counter++;  // Rule 1
    for each Vertex w adjacent to v
    {
        if( !w.visited )  // Forward edge
        {
            w.parent = v;
            findArt( w );
            if( w.low >= v.num )
                System.out.println( v + " is an articulation point" );
            v.low = min( v.low, w.low );  // Rule 3
        }
        else
        if( v.parent != w )  // Back edge
            v.low = min( v.low, w.num );  // Rule 2
    }
}
```

Figure 9.69 Testing for articulation points in one depth-first search (test for the root is omitted) (pseudocode)

EULER CIRCUIT

- A popular puzzle is to reconstruct these figures using a pen, drawing each line exactly once.
- The pen may not lift from the paper while drawing is performed.
- As an extra challenge make the pen finish at the same point at which it started.

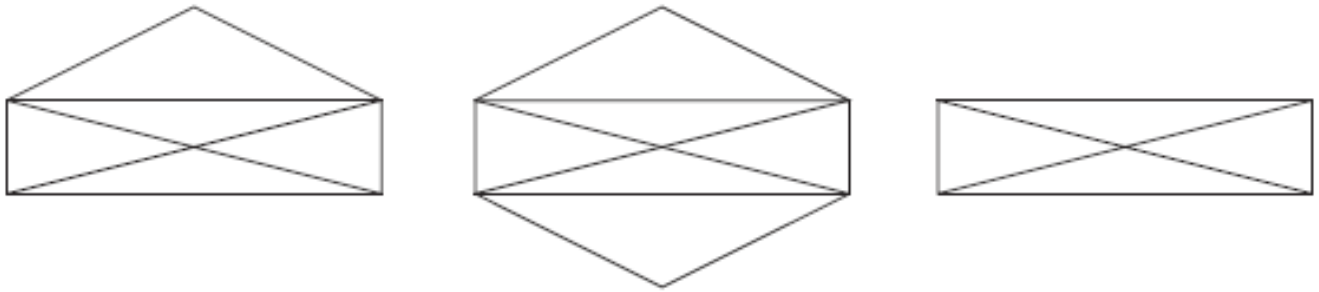


Figure 9.70 Three drawings

- Convert this problem to a graph theory problem by assigning a vertex to each intersection.
- After this conversion , find a path in the graph that visit every edge exactly once, or a cycle that visit every edge exactly once.

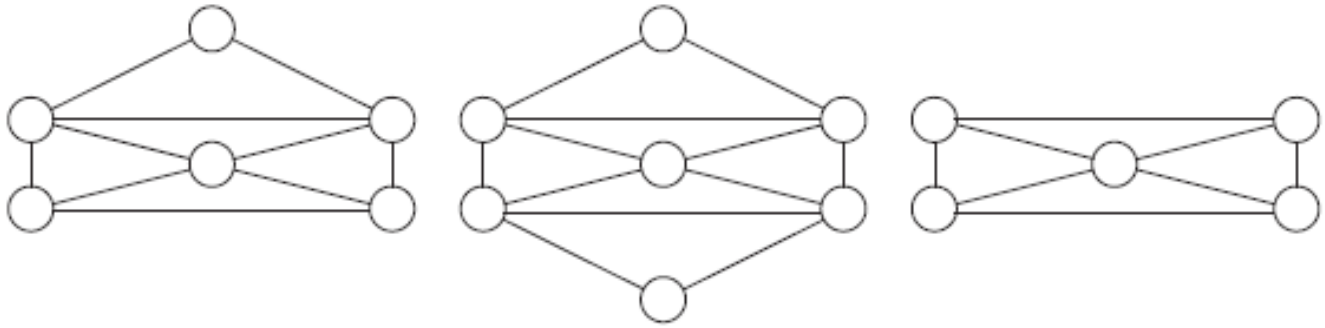


Figure 9.71 Conversion of puzzle to graph

- The first observation that can be made is that an Euler circuit, which must end on its starting vertex, is possible only if the graph is connected and each vertex has an even number degree (number of edges).
- Euler tour is possible if exactly two vertices have odd degree.
- This necessary condition is also sufficient condition. Any connected graph, all of whose vertices have even degree, must have a Euler circuit.
- This circuit can be found in linear time

METHOD

- Test the necessary and sufficient condition first.
- The main problem is that we might visit a portion of the graph and return to the starting point prematurely.
- If all the edges coming out of the start vertex have been used, then part of the graph is untraversed.
- The easiest way to fix this is to find the first vertex on this path that has an untraversed edge and perform another depth-first search.
- This will give another circuit, which can spliced into the original.
- This is continued until all edges have been traversed.

For Example: Suppose we start at vertex 5 and traverse 5,4,10,5. Then we are stuck and most of the graph is untraversed

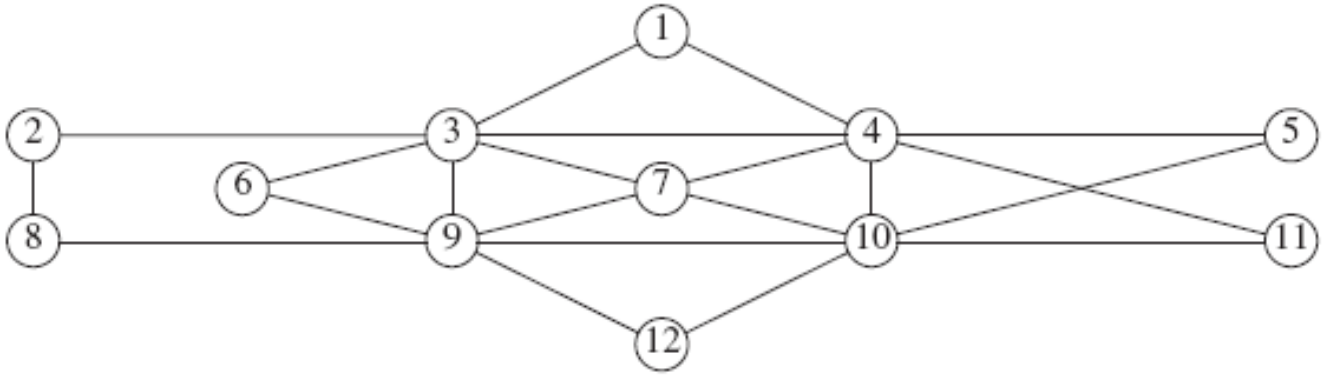


Figure 9.72 Graph for Euler circuit problem

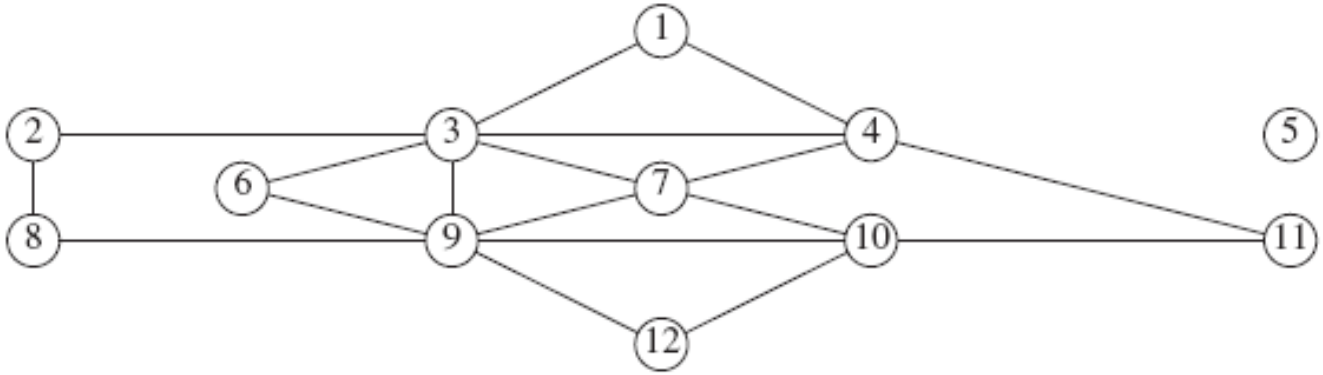


Figure 9.73 Graph remaining after 5, 4, 10, 5

- Then continue from vertex 4, which still has untraversed edges. A depth-first search might come up with path of 4, 1, 3, 7, 4, 11, 10, 7, 9, 3, 4.

- If splice this path into the previous path of 5, 4, 10, 5 to get 5, 4, 1, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5

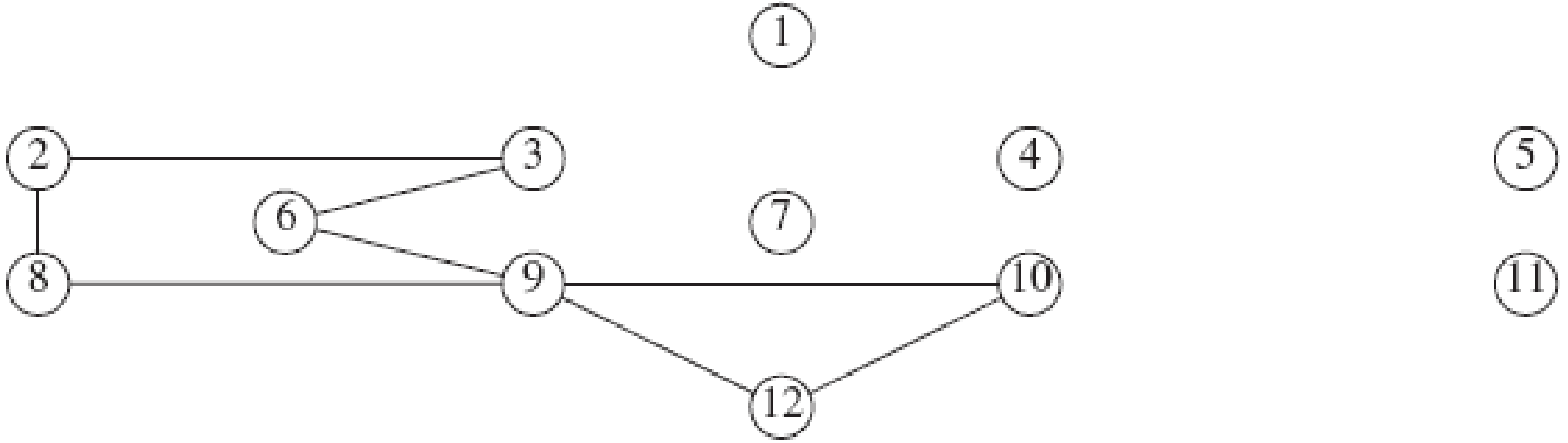


Figure 9.74 Graph after the path 5, 4, 1, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5

- The next vertex on the path that has untraversed edge is vertex 3. A possible circuit 3, 2, 8, 9, 6, 3.
- When spliced in it gives the path 5, 4, 10, 5 to get 5, 4, 1, 3, 2, 8, 9, 6, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5.
- The next vertex that has untraversed edge is 9, and the algorithm finds the circuit 9, 12, 10, 9. When this added to current path, a circuit of 5, 4, 1, 3, 2, 8, 9, 12, 10, 9, 6, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5 is obtained

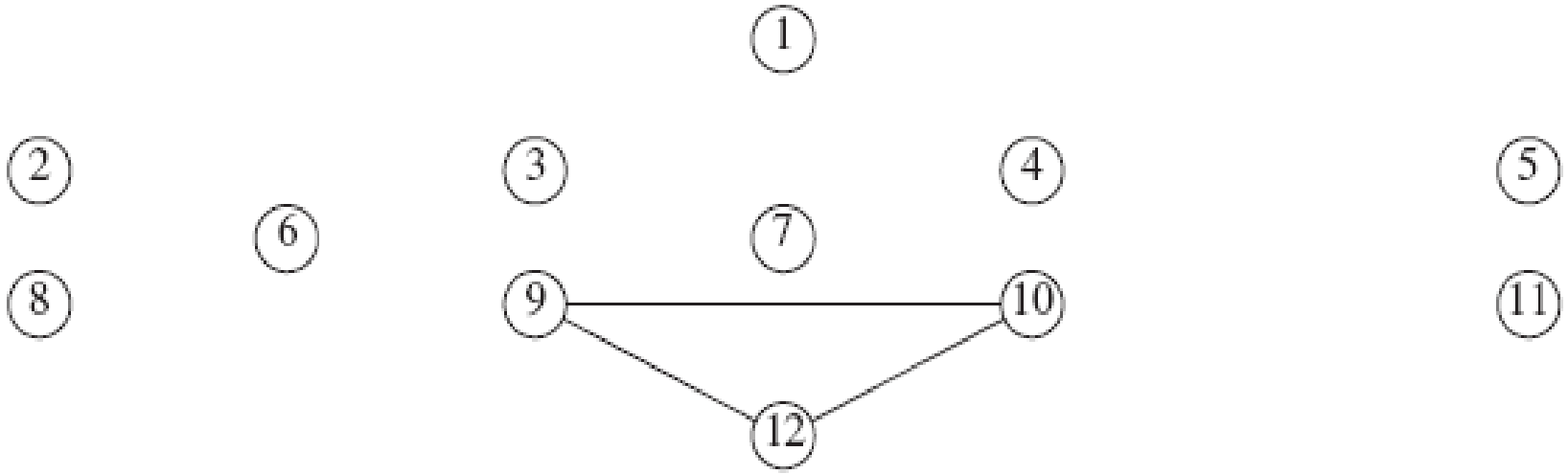


Figure 9.75 Graph remaining after the path 5, 4, 1, 3, 2, 8, 9, 6, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5

- To make this algorithm efficient, appropriate data structures must be used.
- To make splicing simple, the path should be maintained as a linked list.
- To avoid repetitious scanning of adjacency list, for each adjacency list store the last edge scanned.
- The total work performed on the vertex search phase is $O(|E|)$ during the entire life of the algorithm. The running time of the algorithm is $O(|E| + |V|)$

DIRECTED GRAPHS

- Directed graphs can be traversed in linear time using depth-first search.
- If the graph is not strongly connected a depth-search search starting at some node might not visit all nodes.
- In this case perform depth-first searches starting at some unmarked node, until all vertices are visited.

Start DFS at vertex B. This visits B, C, A, D, E and F. Restart at some unvisited vertices, like H, which visits J and I. Finally start at G, which is the last vertex to be visited.

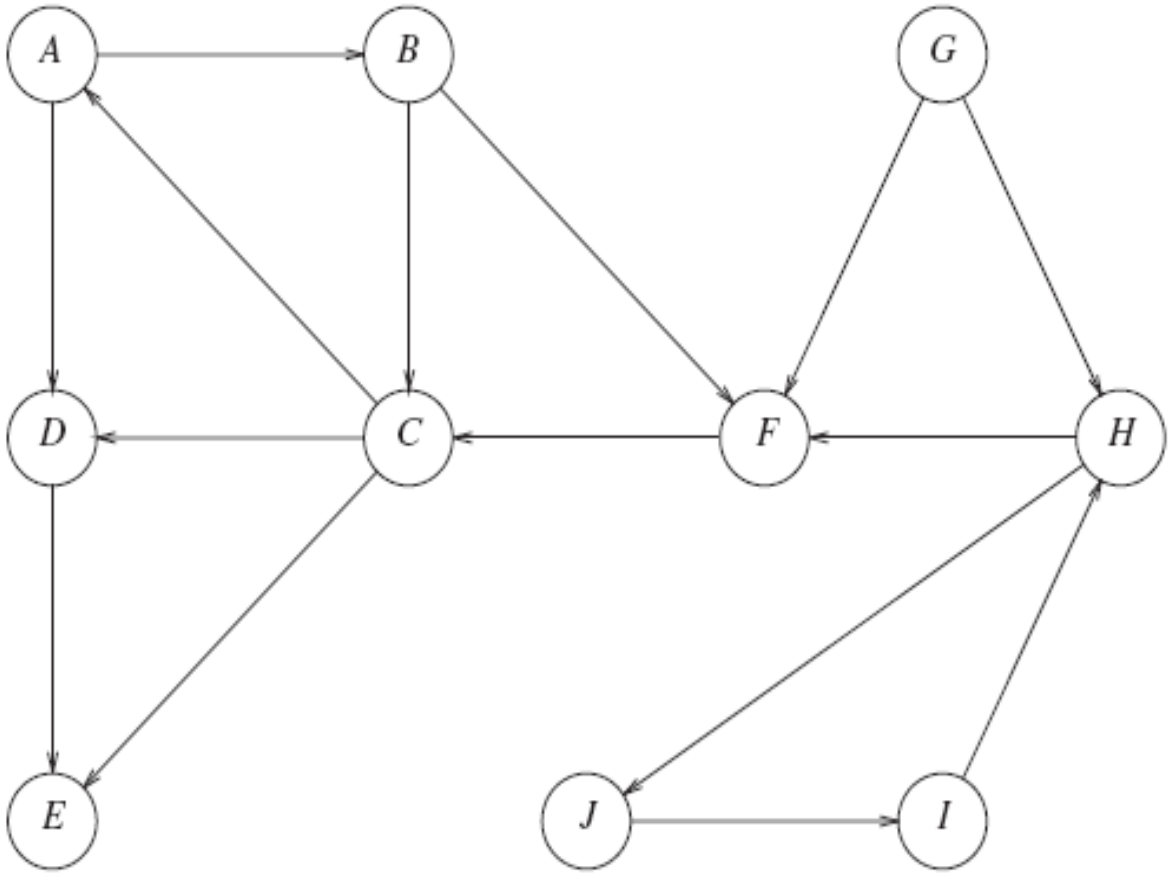


Figure 9.76 A directed graph

- The dashed arrows in the depth-first spanning forest are edges (v,w) for which w was already marked at the time of consideration.
- In undirected graphs, these are always back edges, but here there are three types of edges that do not lead to new vertices.
  - The are **back edges** like (A,B) and (I,H)
  - There are also **forward edges**, such as (C,D) and (C,E)
  - There are **cross edges** such as (F,C) and (G,F) (they always go from right to left)

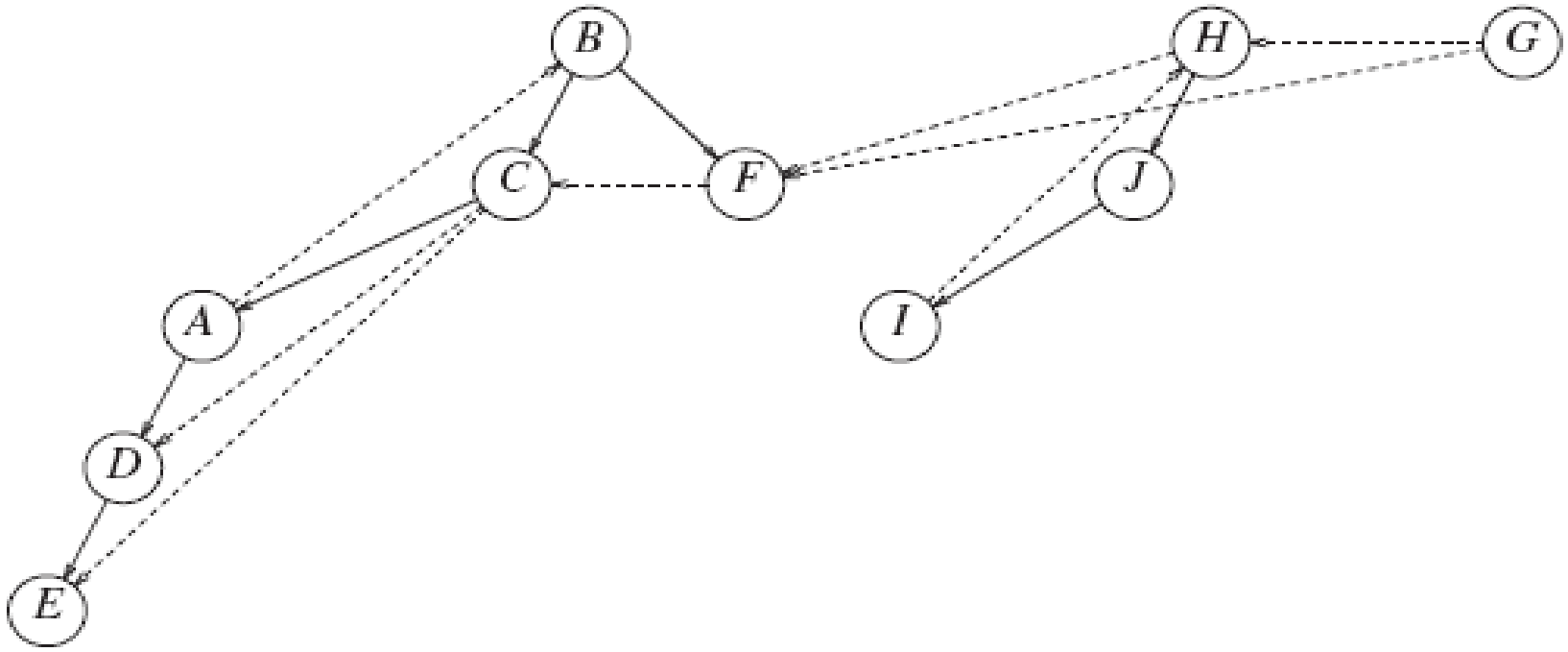


Figure 9.77 Depth-first search of previous graph

DIRECTED GRAPHS

- To test whether a directed graph is acyclic use depth-first search. The graph is acyclic if and only if it has no back edges.

- Another way to perform topological sort is to assign the vertices topological number N, N-1, ….1 by postorder traversal of the depth-first spanning forest.

- As long as the graph is acyclic the ordering will be consistent.

FINDING STRONG COMPONENTS

- To test whether a directed graph is strongly connected perform two depth-first searches.

- if it is not strongly connected this will produce the subset of vertices that are strongly connected.

TECHNIQUES

- Perform a depth-first search on the input graph G.
- The vertices of G are numbered by a postorder traversal of the depth-first spanning forest.
- Reverse all the edges in G forming a new graph G_r .
- Perform depth-first search of the graph G_r , always starting a new dfs at the highest numbered vertex.

The DFS is started at vertex G, because that is highest numbered vertex.
This leads us nowhere so the next search is started at H, This call visits I and J.
The next call starts at B and visits A, C and F. The next call is dfs(D) and finally dfs(E)

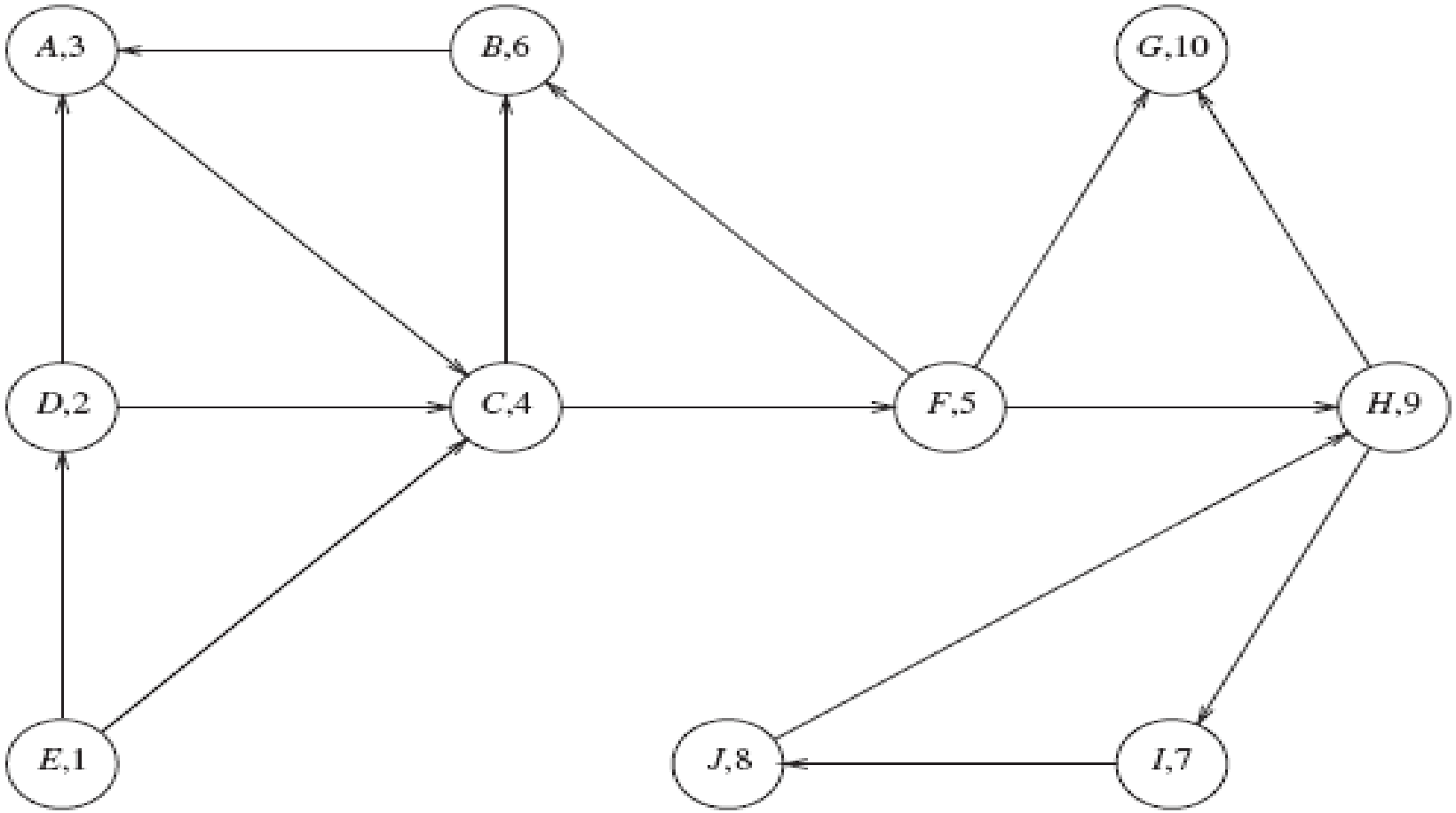


Figure 9.78 G_r numbered by postorder traversal of G (from Figure 9.76)

Each of the tree in spanning forest forms a strongly connected component.

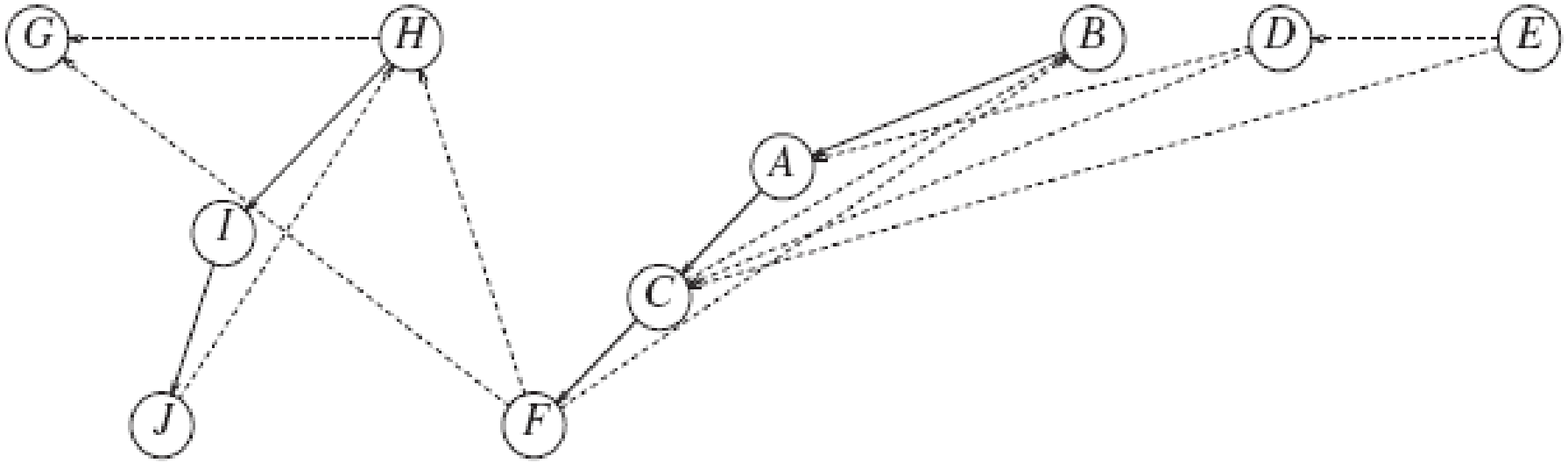
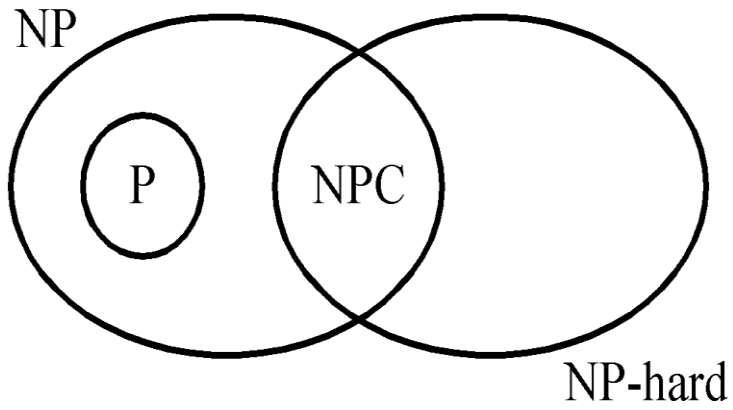


Figure 9.79 Depth-first search of G_r—strong components are $\{G\}$, $\{H, I, J\}$, $\{B, A, C, F\}$, $\{D\}$, $\{E\}$

Introduction to NP-Completeness



P: the class of problems which can be solved by a deterministic polynomial algorithm.
NP : the class of decision problem which can be solved by a non-deterministic polynomial algorithm.
NP-hard: the class of problems to which every NP problem reduces.
NP-complete (NPC): the class of problems which are NP-hard and belong to NP.

Halting Problem

- In computability theory, the **halting problem** is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running or continue to run forever.
- The intuitive reason that this problem is undecidable is that such a program might have a hard time checking itself.

Traveling Salesman Problem

Given a complete graph G = (V,E) with edge costs and an integer, is there a simple cycle that visits all vertices and has total cost <= K.

The traveling salesman problem is NP-Complete. Its solution can be checked in polynomial time, but it cannot be computed in polynomial time. To show it is NP-Complete, just reduce some NP complete problem (like Hamilton Cycle) to ***TSP.***

TSP to Hamiltonian Cycle

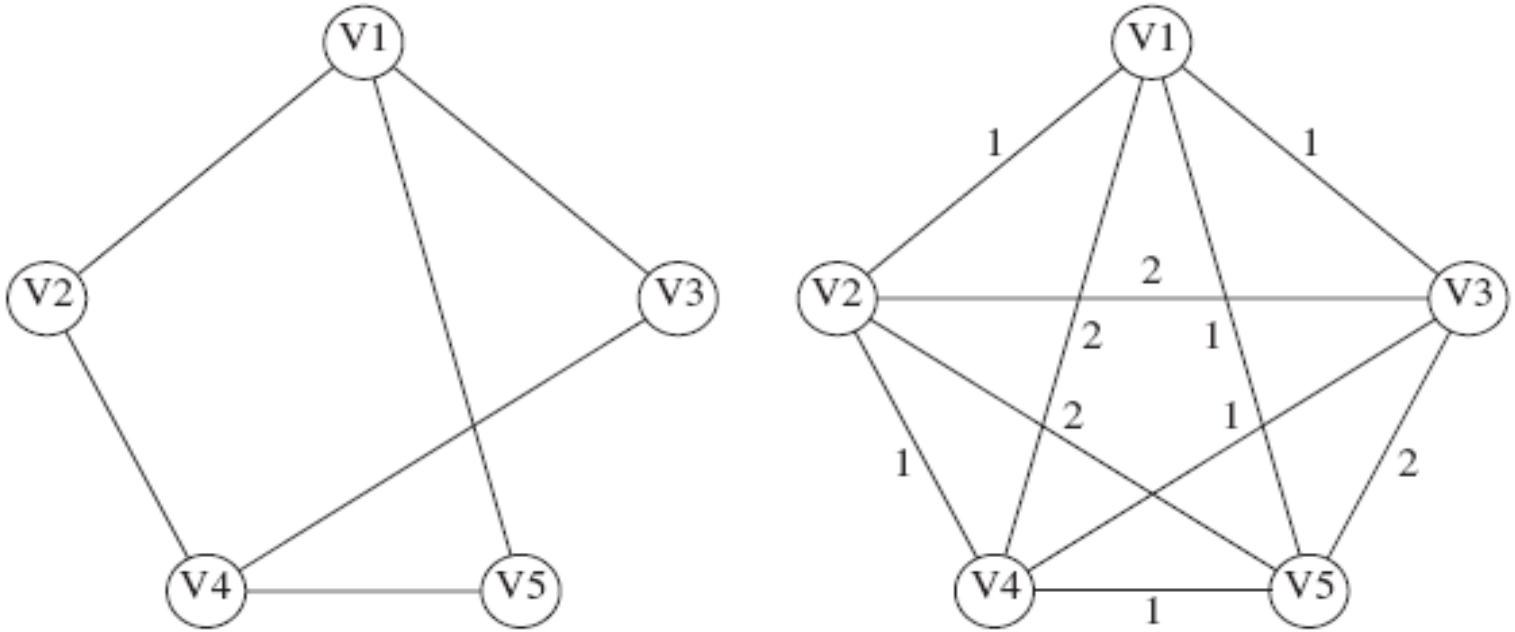


Figure 9.80 Hamiltonian cycle problem transformed to traveling salesman problem

Satisfiability Problem

- In computer science, the **Boolean Satisfiability Problem** is the problem of determining if there exists an interpretation that satisfies a given Boolean formula.
- SAT is one of the first problems that was proven to be NP-complete.
- There is no known algorithm that efficiently solves SAT, and it is generally believed that no such algorithm exists.
- Resolving the question whether SAT has an efficient algorithm is equivalent to the P versus NP problem, which is the most famous open problem in the theory of computing.