Data Structures and Algorithms

- Graph 2 -

School of Electrical Engineering
Korea University

Graphs with negative costs

- In case of a graph with negative edge costs, Dijkstra's algorithm does not work
- A tempting solution is to add a constant **Δ** to each edge cost, thus removing negative edges

  => Paths with more edges become more weighty than paths with fewer edges

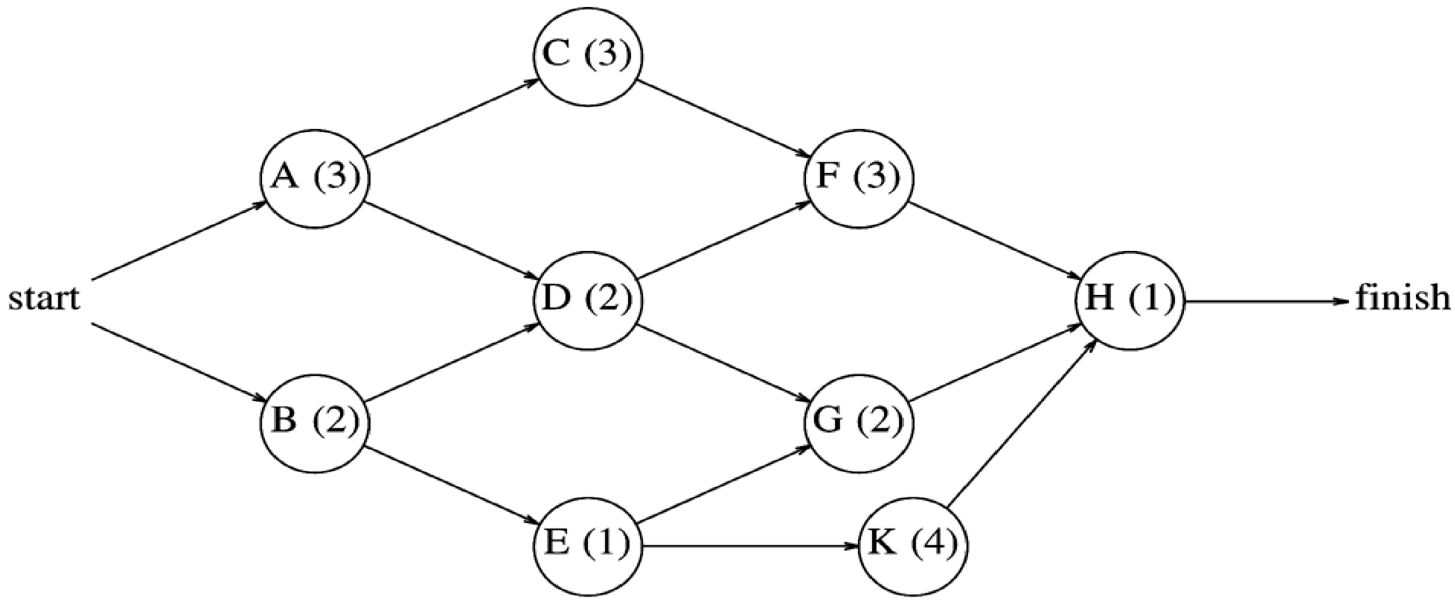
Dijkstra's algorithm

```
        WeightedNegative( Table T )
        {
            Queue Q;
            Vertex V, W;

/* 1*/      Q = CreateQueue( NumVertex ); MakeEmpty( Q );
/* 2*/      Enqueue( S, Q ); /* Enqueue the start vertex S */

/* 3*/      while( !IsEmpty( Q ) )
            {
/* 4*/          V = Dequeue( Q );
/* 5*/          for each W adjacent to V
/* 6*/              if( T[ V ].Dist + Cvw < T[ W ].Dist )
                    {
                        /* Update W */
/* 7*/                  T[ W ].Dist = T[ V ].Dist + Cvw;
/* 8*/                  T[ W ].Path = V;
/* 9*/                  if( W is not already in Q )
/*10*/                      Enqueue( W, Q );
                    }
            }
/*11*/      DisposeQueue( Q );
        }
```

Acyclic Graphs

- In case of acyclic graph, it is possible to improve Dijkstra's algorithm by selecting vertices in topological order
- The algorithm can be done in one pass
- Usage
  - Modeling downhill skiing problem
  - Modeling nonreversible chemical reactions
  - Critical path analysis using *activity node graph*

Activity Node Graph



- Each node represents an activity that must be performed, along with the time it takes to complete the activity
- The edge represents precedence relationships

Application

- Construction projects
  - Earliest completion time of the project

    (Ex) 10 time units for the path A, C, F, H

  - Which activities can be delayed, by how long without affecting the minimum completion time

    (Ex) B can be delayed 2 time units

Activity node graph

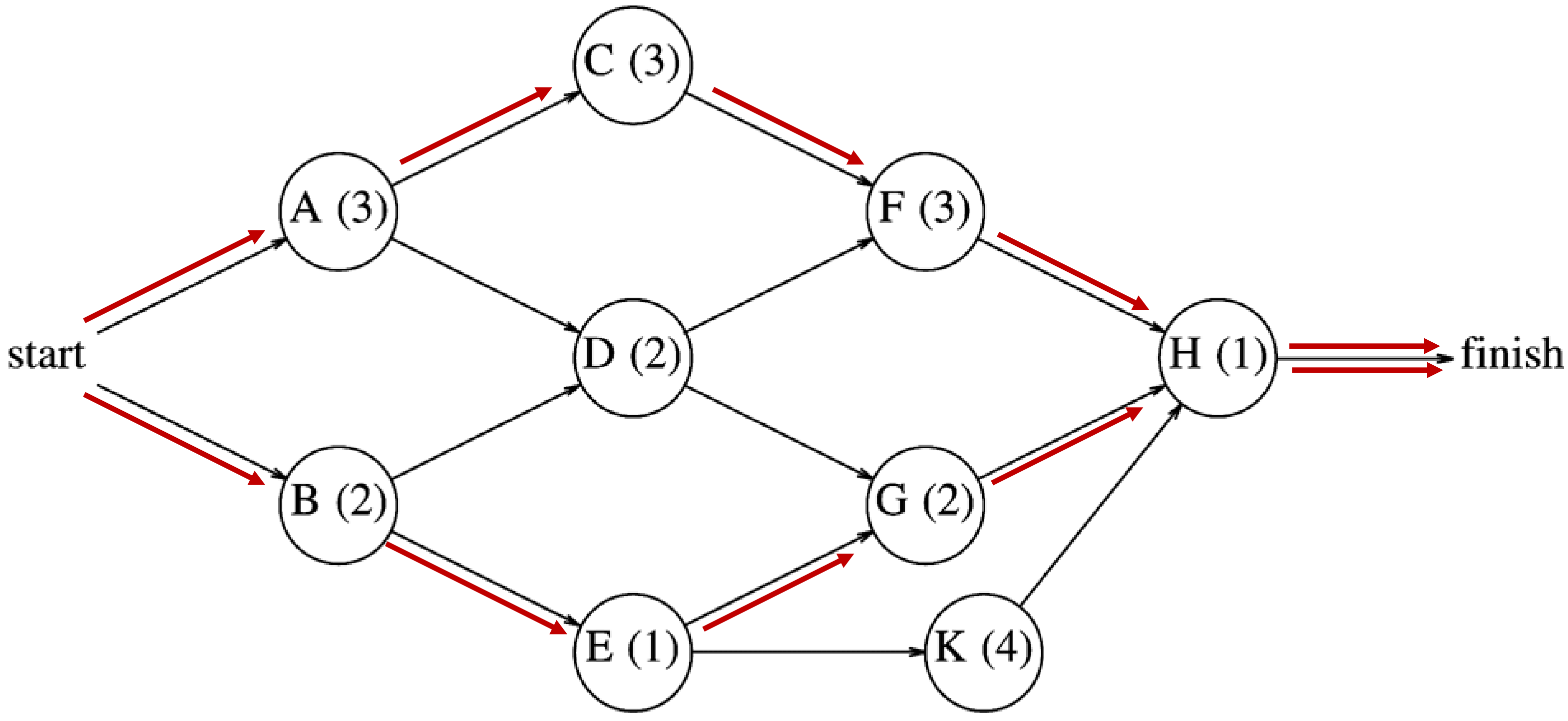


Figure 9.34 Activity-node graph

Event node graph

To perform these calculations, convert the *activity-node graph* to an *event -node graph*

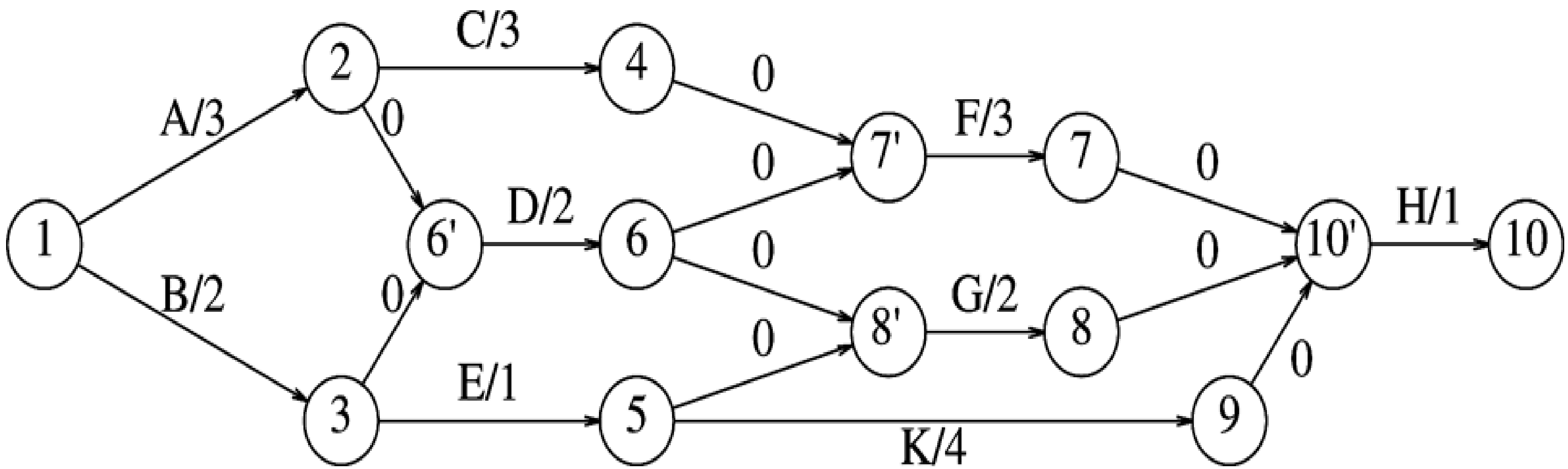
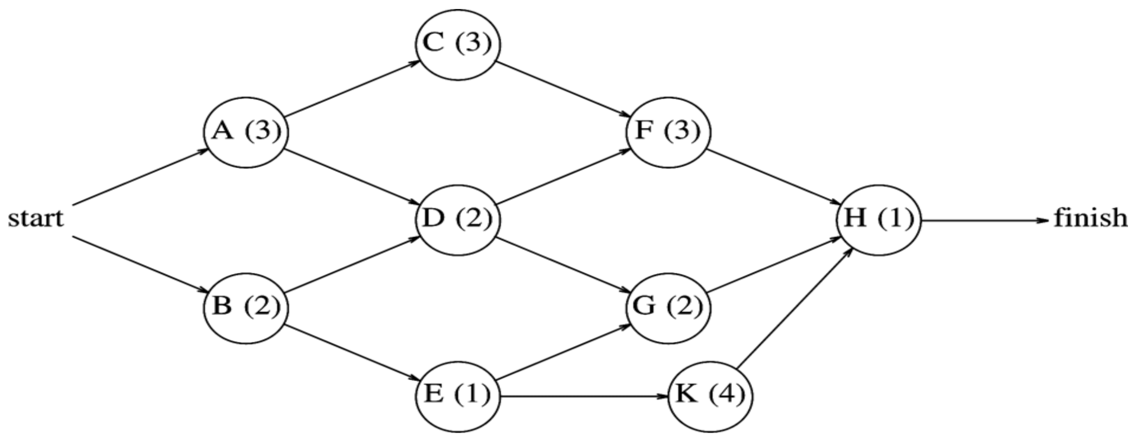
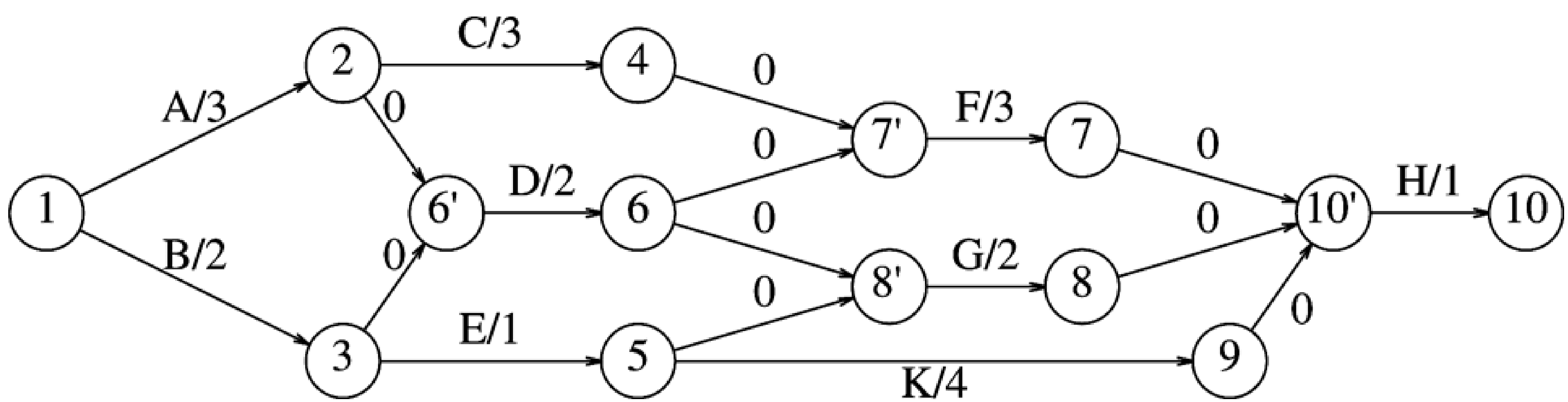


Figure 9.35 Event-node graph

Graph conversion





Earliest(Latest) Completion Time

- Let EC_i is the earliest completion time for node i, then

$$EC_1 = 0$$

$$EC_w = \max_{(v,w) \in E} (EC_v + c_{v,w})$$

- Let LC_i is the latest completion time for node i, then

$$LC_n = EC_n$$

$$LC_v = \min_{(v,w) \in E} (LC_w - c_{v,w})$$

Earliest Completion Time EC_i

- $EC_1 = 0$

  $EC_w = \max_{(v,w) \in E} (EC_v + c_{v,w})$

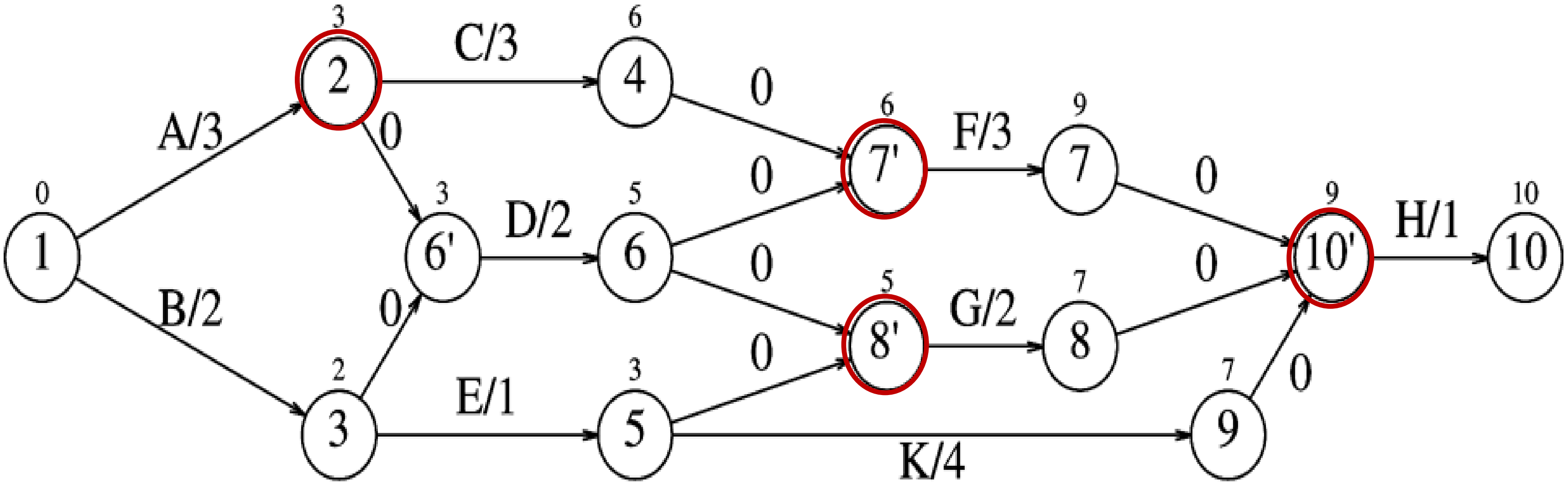


Figure 9.36 Earliest completion times

Latest Completion Time LC_i

- $LC_n = EC_n$

  $LC_v = \min_{(v,w)\in E} (LC_w - c_{v,w})$

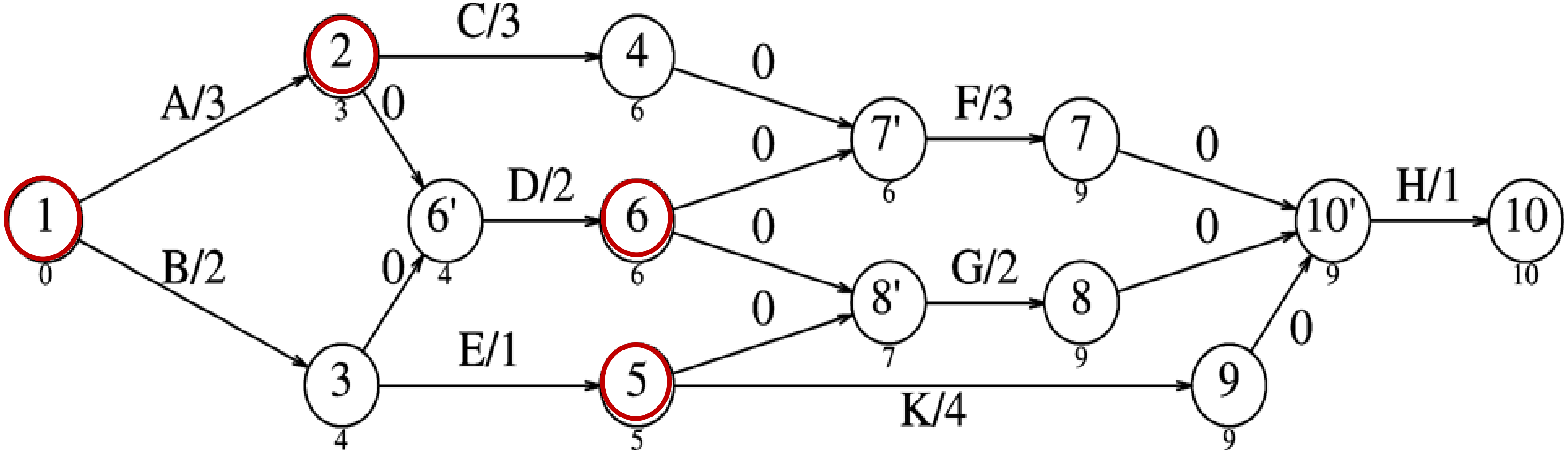


Figure 9.37 Latest completion times

Slack Time

- ***Slack time*** for each edge represents the amount of time that the completion of the corresponding activity can be delayed without delaying the overall completion.

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

Slack Time

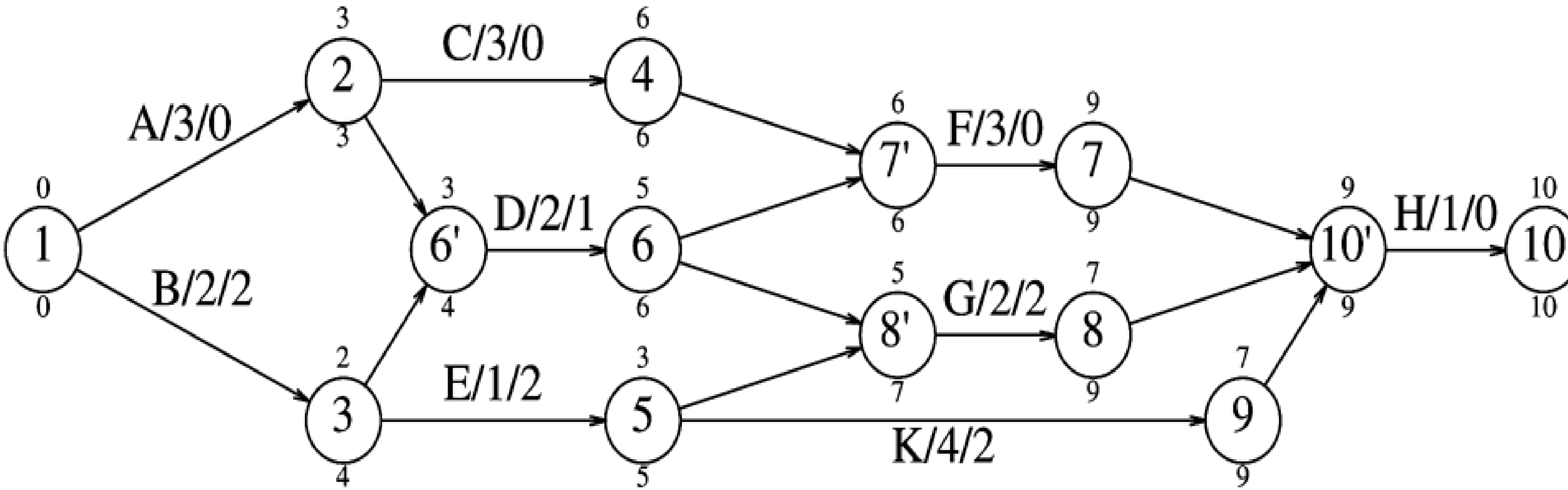


Figure 9.38 Earliest completion time, latest completion time, and slack

- There is at least ***one critical path*** consisting entirely of zero-slack edges, which must finish on schedule

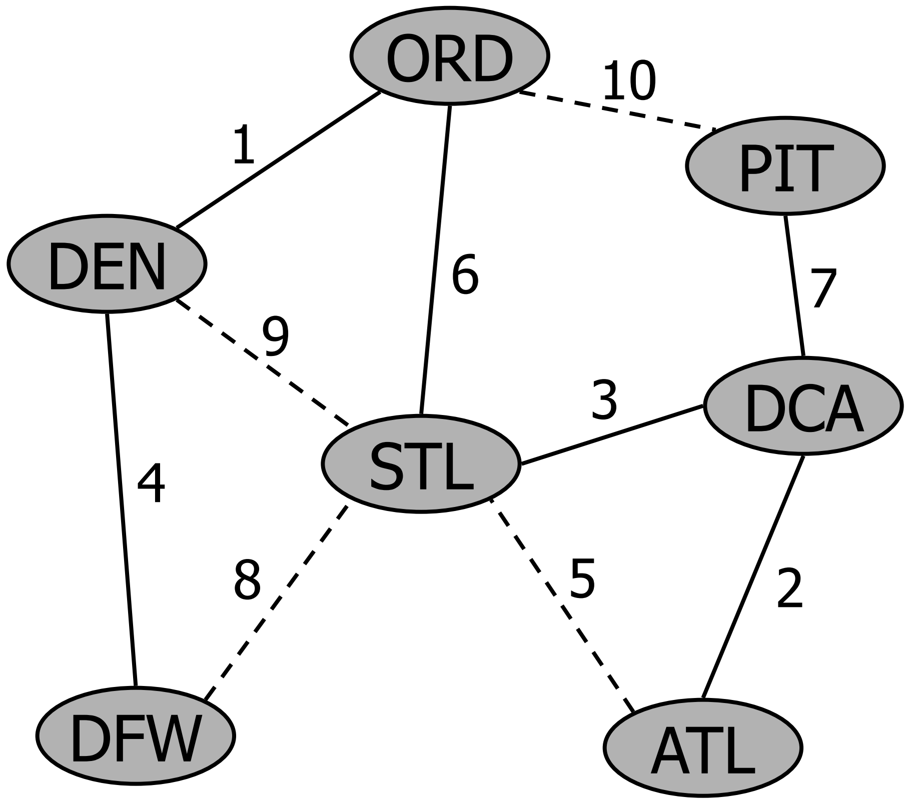
Minimum Spanning Tree

- Assumes undirected and connected graph
- A tree formed from graph edges that connects all the vertices of G at lowest total cost
- Example in Fig. 9.48
- No. of edges in the MST = |v| – 1

Minimum Spanning Tree

- Spanning subgraph
  – Subgraph of G containing all the vertices of G
- Spanning tree
  – Spanning subgraph that is itself a tree
- Minimum spanning tree (MST)
  – Spanning tree of a weighted graph with minimum total edge weight
- Applications
  – Communications networks
  – Transportation networks

Minimum Spanning Tree



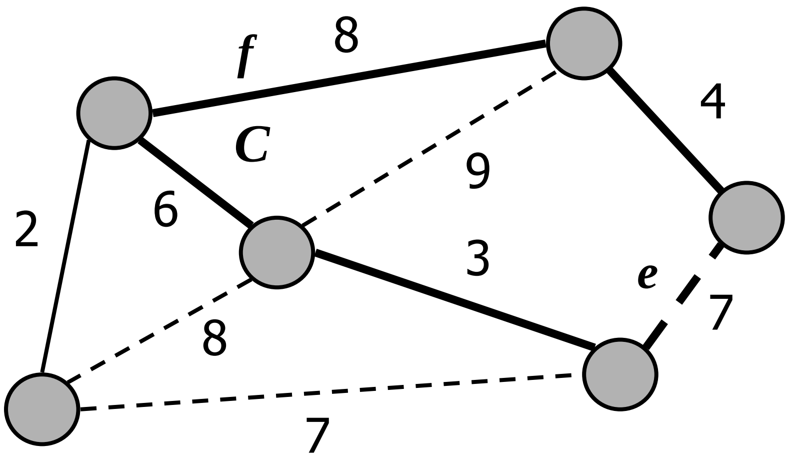
Cycle Property

- Let T be a minimum spanning tree of a weighted graph G
- Let e be an edge of G that is not in T and let C be the cycle formed by e with T
- For every edge f of C, $weight(f) \leq weight(e)$

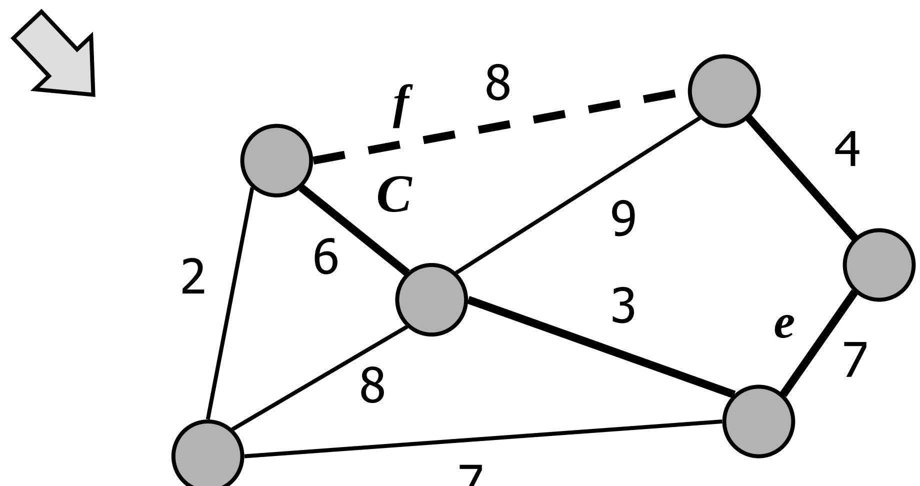
<u>Proof by contradiction</u>

If $weight(f) > weight(e)$, we can get a spanning tree of smaller weight by replacing e with f

Cycle Property



Replacing ***f*** with ***e*** yields
a better spanning tree



Partition Property

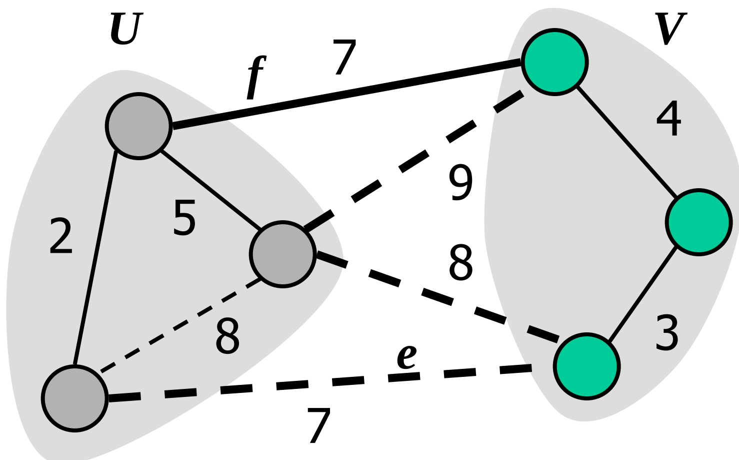
- Partition the vertices of G into subsets U and V
- Let e be an edge of minimum weight across U and V
- There is a MST of G containing edge e

  Proof:
  - Let T be an MST of G
  - If T does not contain e, consider the cycle C formed by e with T and let f be an edge of C across the partition
  - By the cycle property, $weight(f) \le weight(e)$

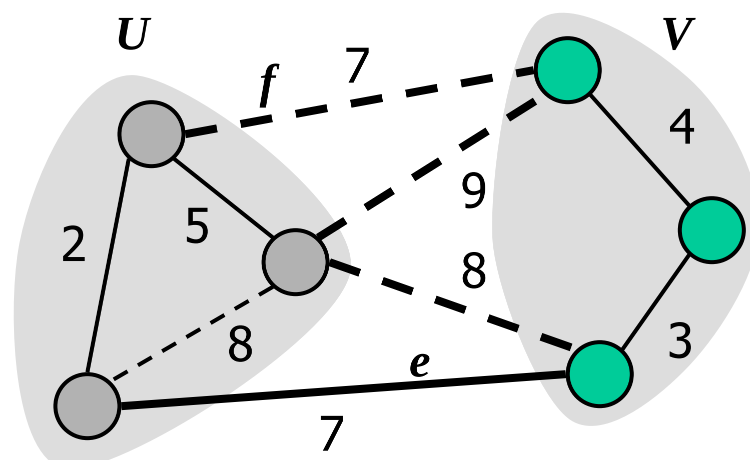
    Thus, $weight(f) = weight(e)$
  - We obtain another MST by replacing f with e

7

Partition Property



Replacing *f* with *e* yields another MST



Minimum spanning tree

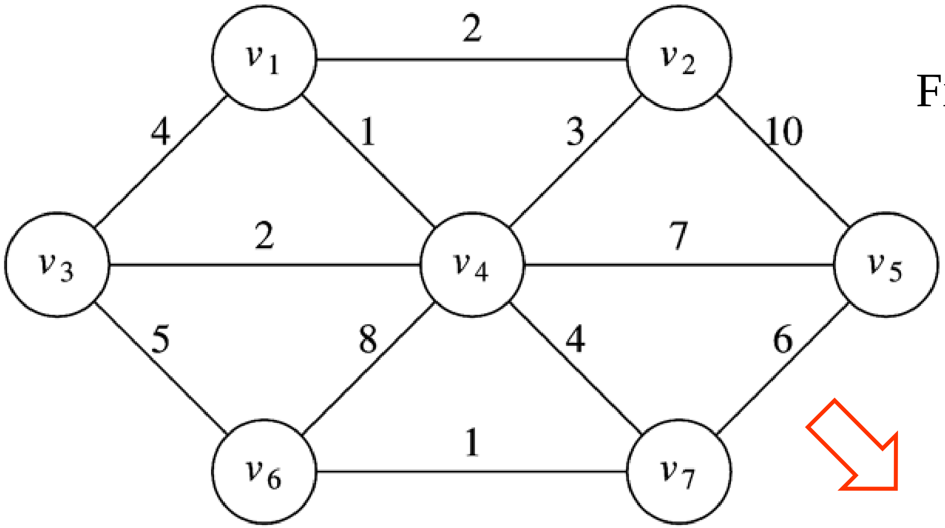
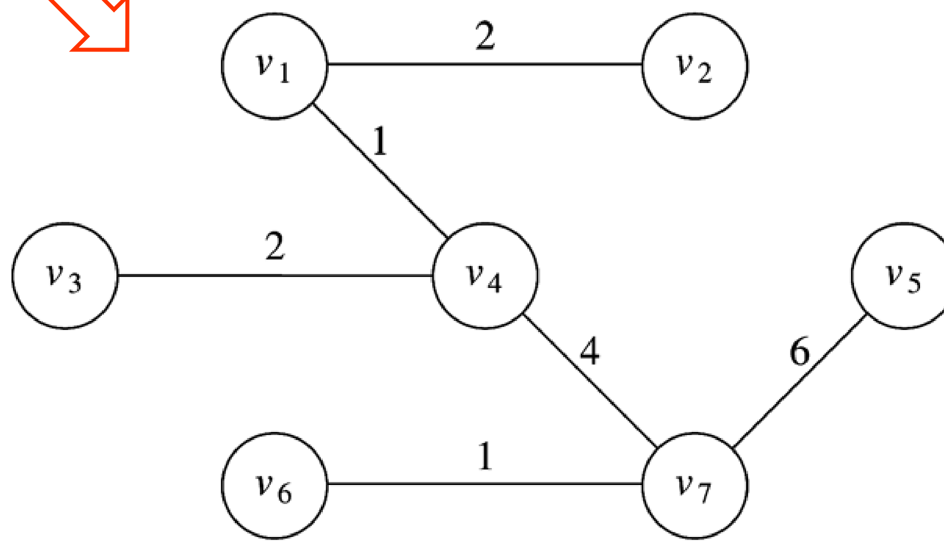


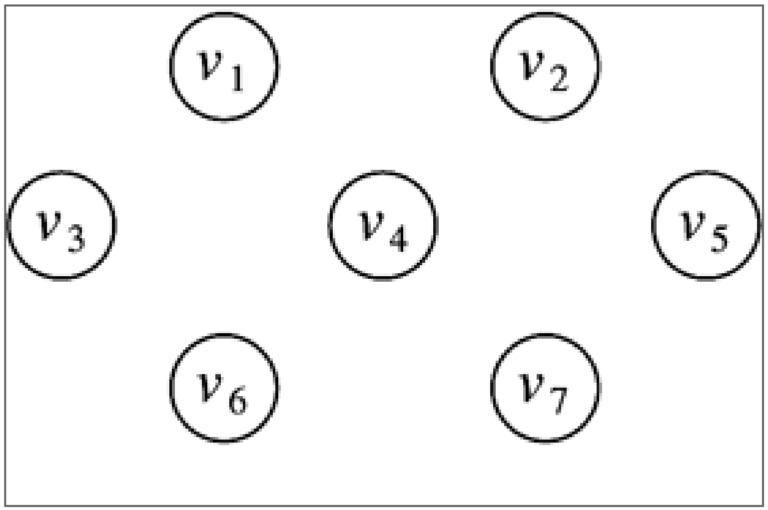
Figure 9.48 A graph G



Minimum spanning tree

Algorithms for MST

- Prim's Algorithm
  - Very similar to Dijkstra's Algorithm
- Kruskal's Algorithm
  - Continually select the edges in order of smallest weight and accept an edge if it does not cause a cycle



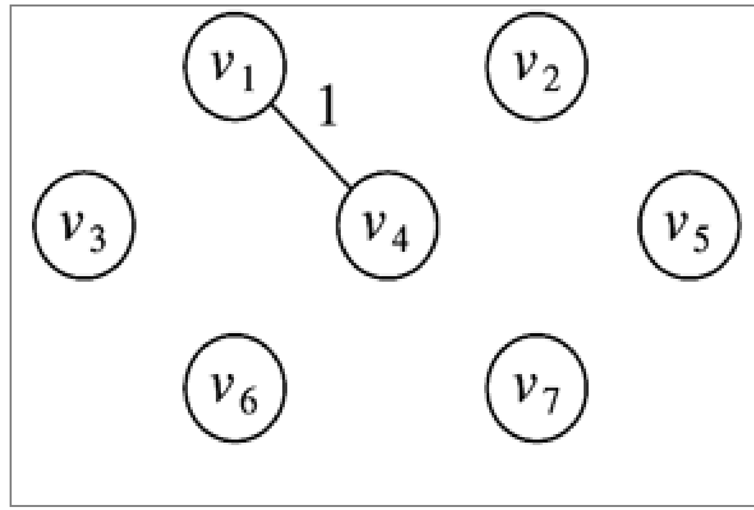


Figure 9.49 Prim's algorithm after each stage

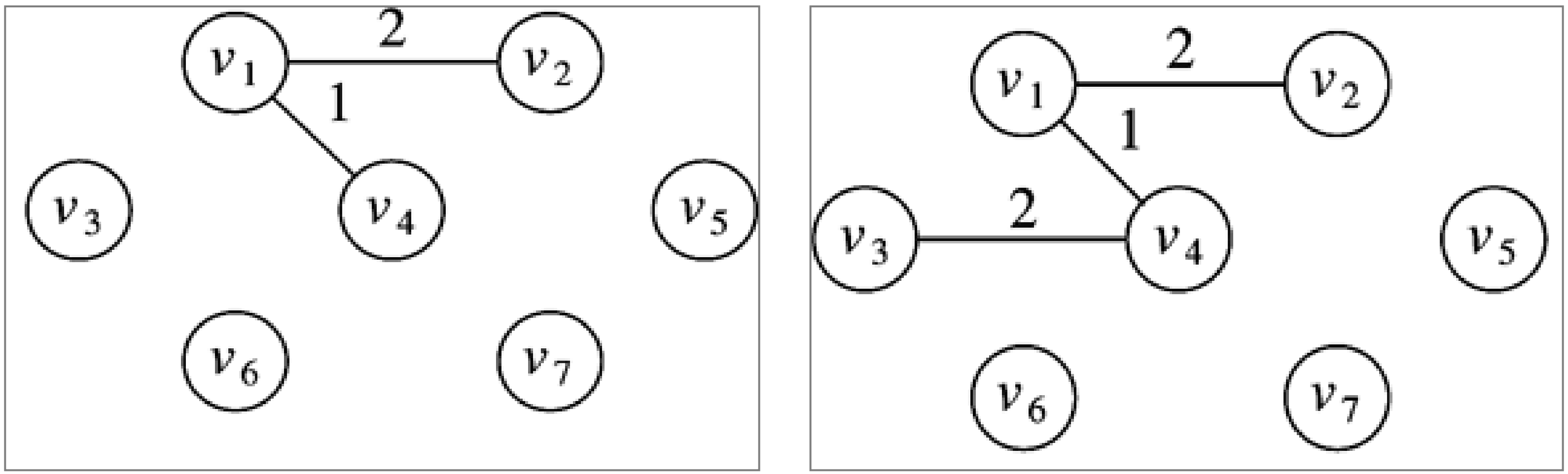


Figure 9.49 Prim's algorithm after each stage

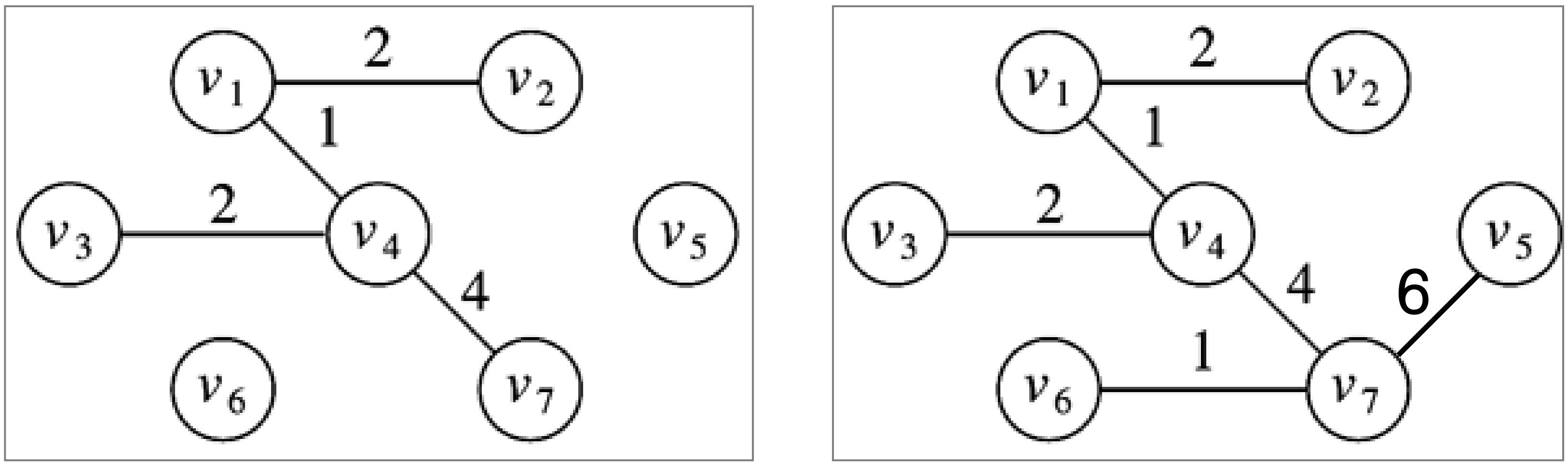


Figure 9.49 Prim’s algorithm after each stage

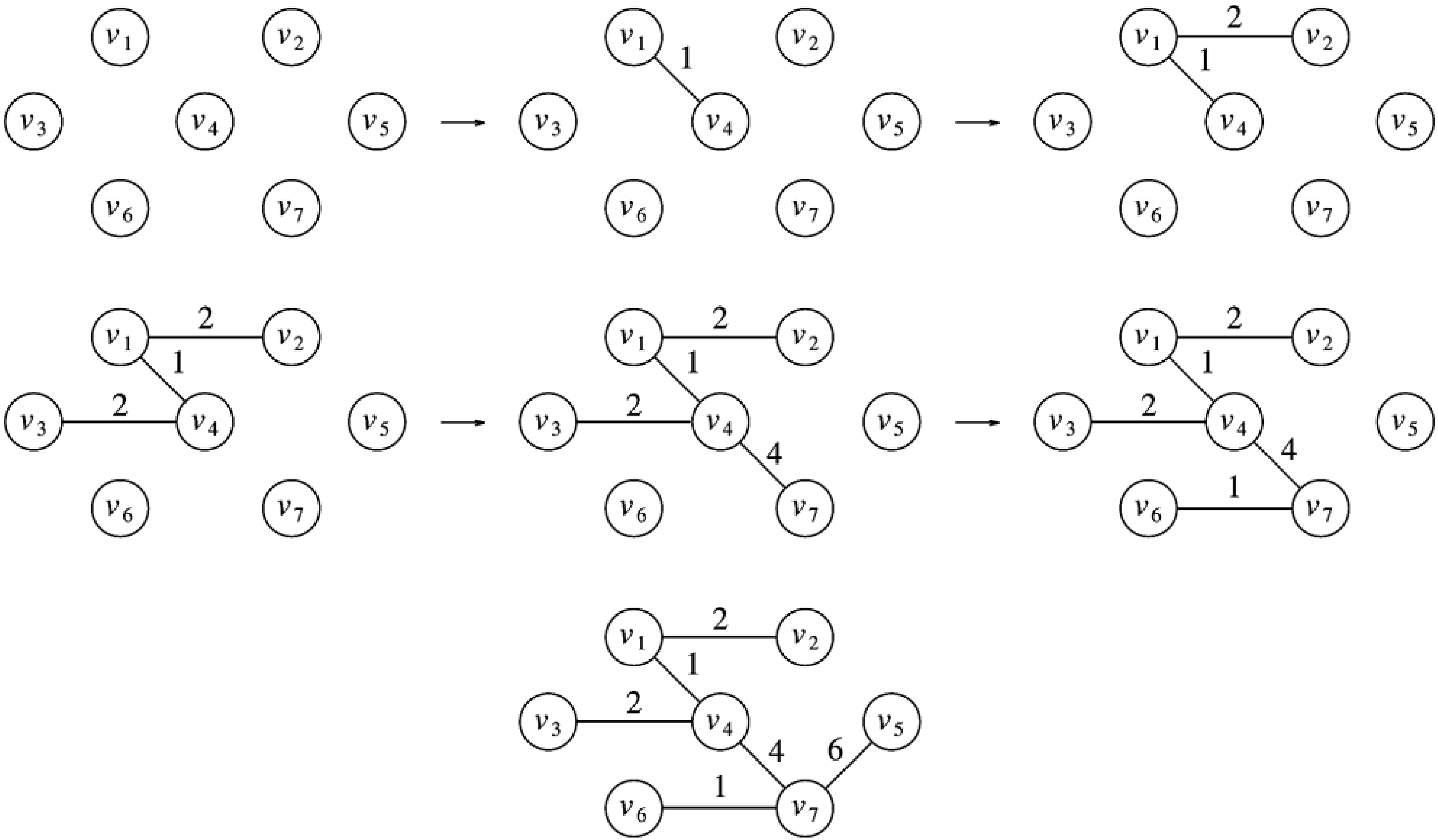


Figure 9.49 Prim's algorithm after each stage

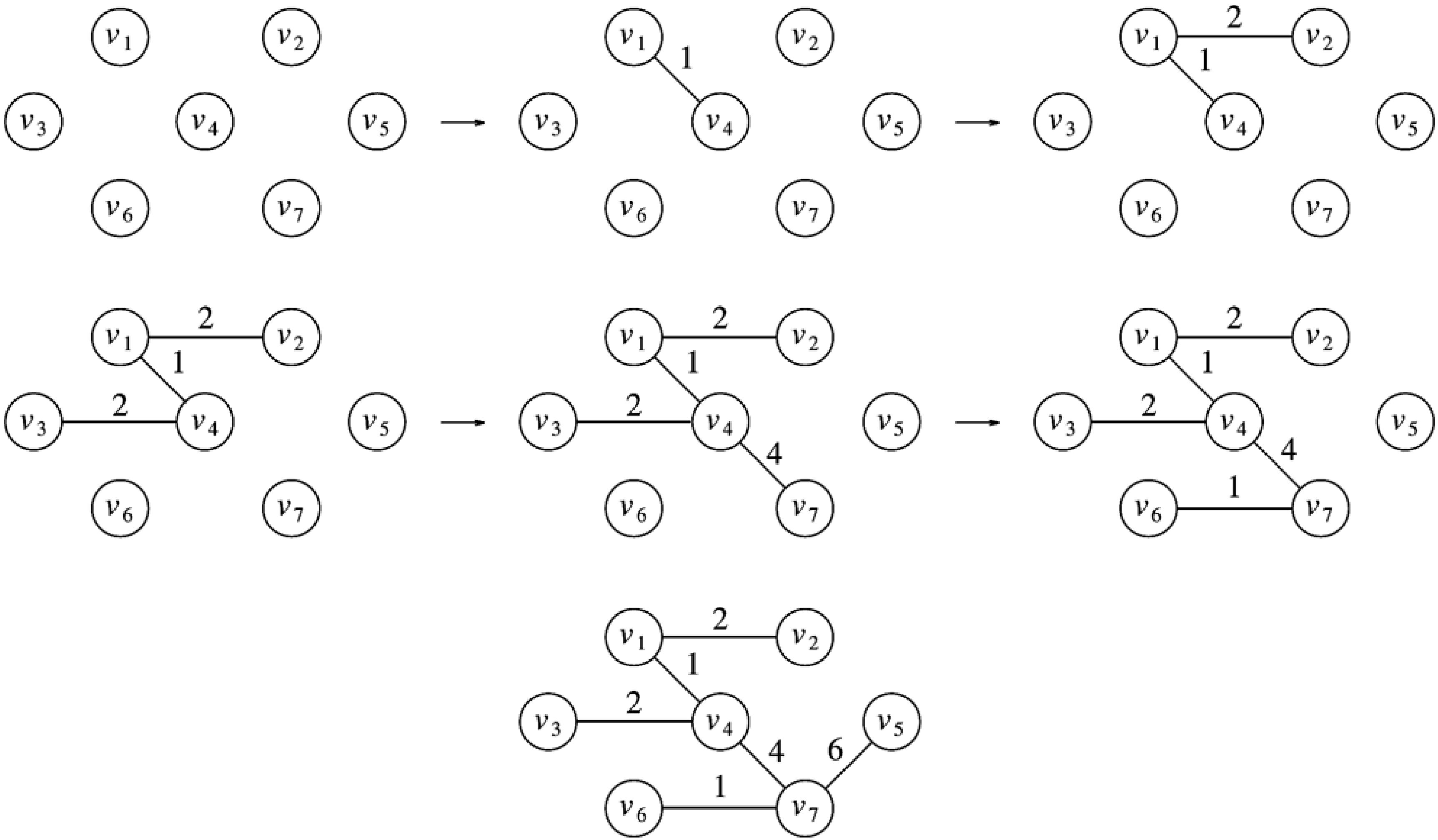


Figure 9.49 Prim's algorithm after each stage

MST by Prim's Algorithm

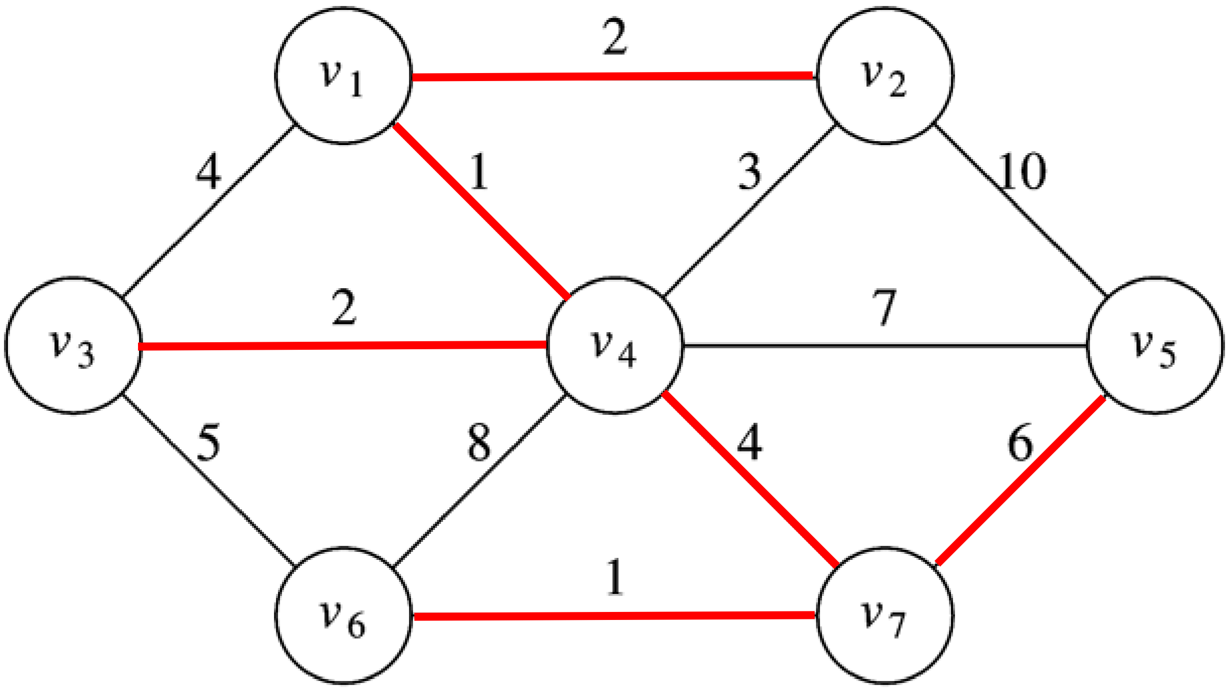


Table in Prim's algorithm

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

Figure 9.50 Initial configuration of table used in Prim's algorithm

Table in Prim's algorithm

Figure 9.51 The table After v_1 is declared known

v	Known	d_v	p_v
v_1	1	0	0
v_2	0	2	v_1
v_3	0	4	v_1
v_4	0	1	v_1
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0

v	Known	d_v	p_v
v_1	1	0	0
v_2	0	2	v_1
v_3	0	2	v_4
v_4	1	1	v_1
v_5	0	7	v_4
v_6	0	8	v_4
v_7	0	4	v_4

Figure 9.52 The table After v_4 is declared known

Table in Prim's algorithm

Figure 9.53 The table After v_2 and then v_3 are declared known

v	Known	d_v	p_v
v_1	1	0	0
v_2	1	2	v_1
v_3	1	2	v_4
v_4	1	1	v_1
v_5	0	7	v_4
v_6	0	5	v_3
v_7	0	4	v_4

v	Known	d_v	p_v
v_1	1	0	0
v_2	1	2	v_1
v_3	1	2	v_4
v_4	1	1	v_1
v_5	0	6	v_7
v_6	0	1	v_7
v_7	1	4	v_4

Figure 9.54 The table After v_7 is declared known

Table in Prim's algorithm

Figure 9.55 The table After v_6 and v_5 are selected (Prim's algorithm terminates)

v	Known	d_v	p_v
v_1	1	0	0
v_2	1	2	v_1
v_3	1	2	v_4
v_4	1	1	v_1
v_5	1	6	v_7
v_6	1	1	v_7
v_7	1	4	v_4

Prim-Jarnik's Algorithm

- We assume that the graph is connected
- We pick an arbitrary vertex s and we grow the MST as a cloud of vertices, starting from s
- We store with each vertex v a label $d(v)$ representing the smallest weight of an edge connecting v to a vertex in the cloud

- At each step
  - We add to the cloud the vertex u outside the cloud with the smallest distance label
  - We update the labels of the vertices adjacent to u

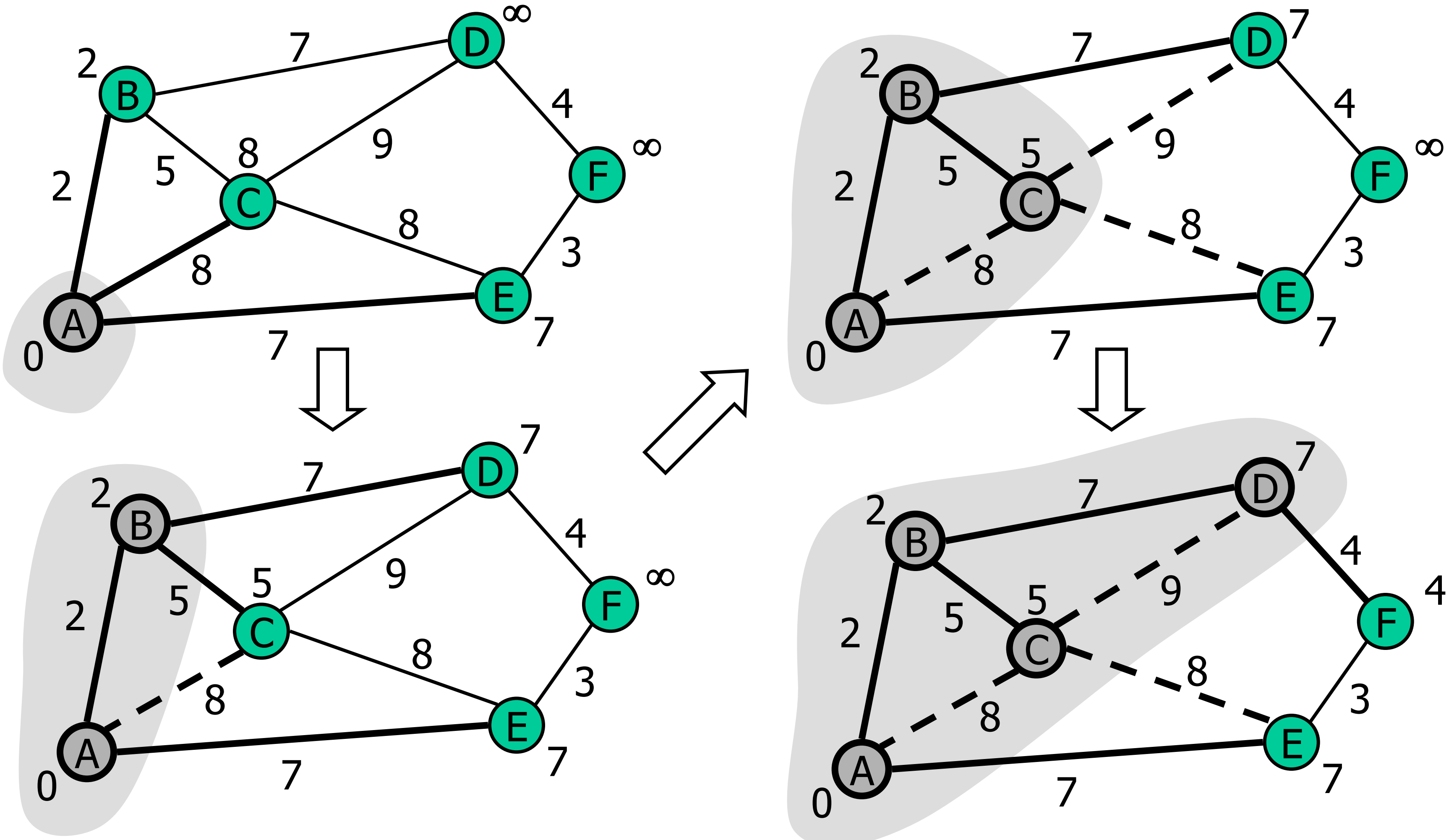
Prim-Jarnik's Algorithm (cont.)

- A priority queue stores the vertices outside the cloud
  - Key: distance
  - Element: vertex
- Locator-based methods
  - ***insert(k,e)*** returns a locator
  - ***replaceKey(l,k)*** changes the key of an item
- We store three labels with each vertex:
  - Distance
  - Parent edge in MST
  - Locator in priority queue

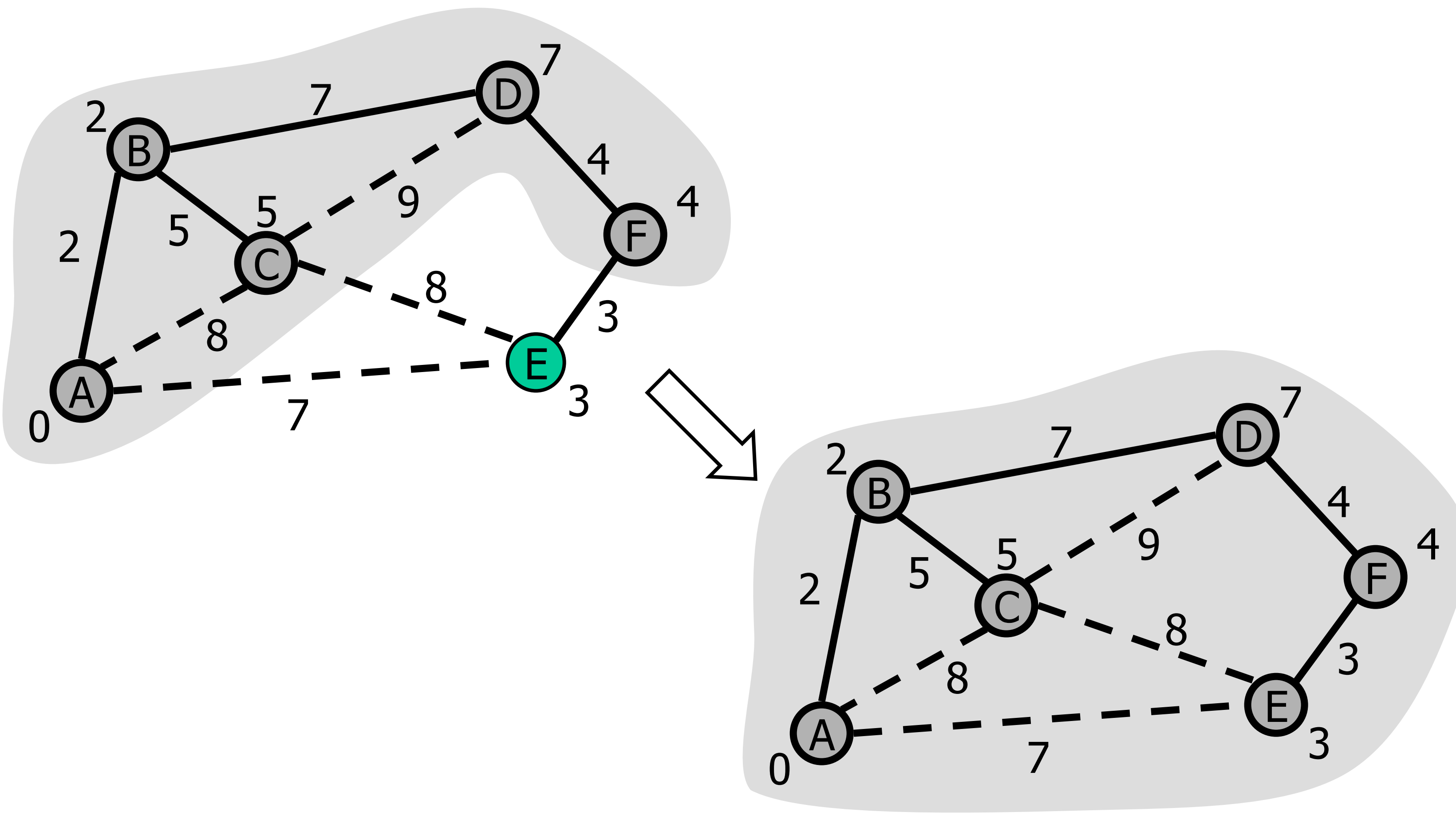
Prim-Jarnik's Algorithm (cont.)

```
Algorithm PrimJarnikMST(G)
   Q ← new heap-based priority queue
   s ← a vertex of G
   for all v ∈ G.vertices()
        if v = s
             setDistance(v, 0)
        else
             setDistance(v, ∞)
        setParent(v, ∅)
        l ← Q.insert(getDistance(v), v)
        setLocator(v,l)
   while ¬Q.isEmpty()
        u ← Q.removeMin()
        for all e ∈ G.incidentEdges(u)
             z ← G.opposite(u,e)
             r ← weight(e)
             if r < getDistance(z)
                  setDistance(z,r)
                  setParent(z,e)
                  Q.replaceKey(getLocator(z),r)
```

Example



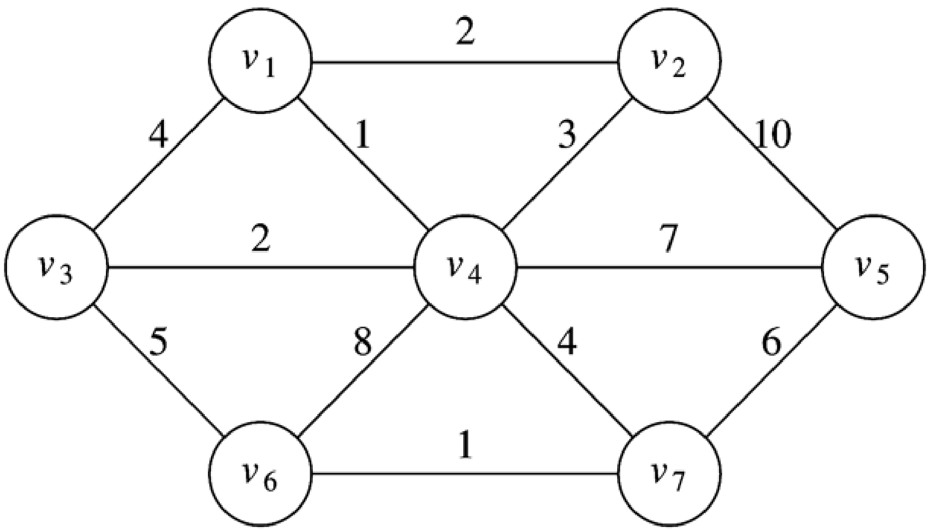
Example (contd.)



Kruskal's Algorithm

- Continually select the edges in order of smallest weight and accept an edge if it does not cause a cycle

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 9.56 Action of Kruskal's algorithm on G

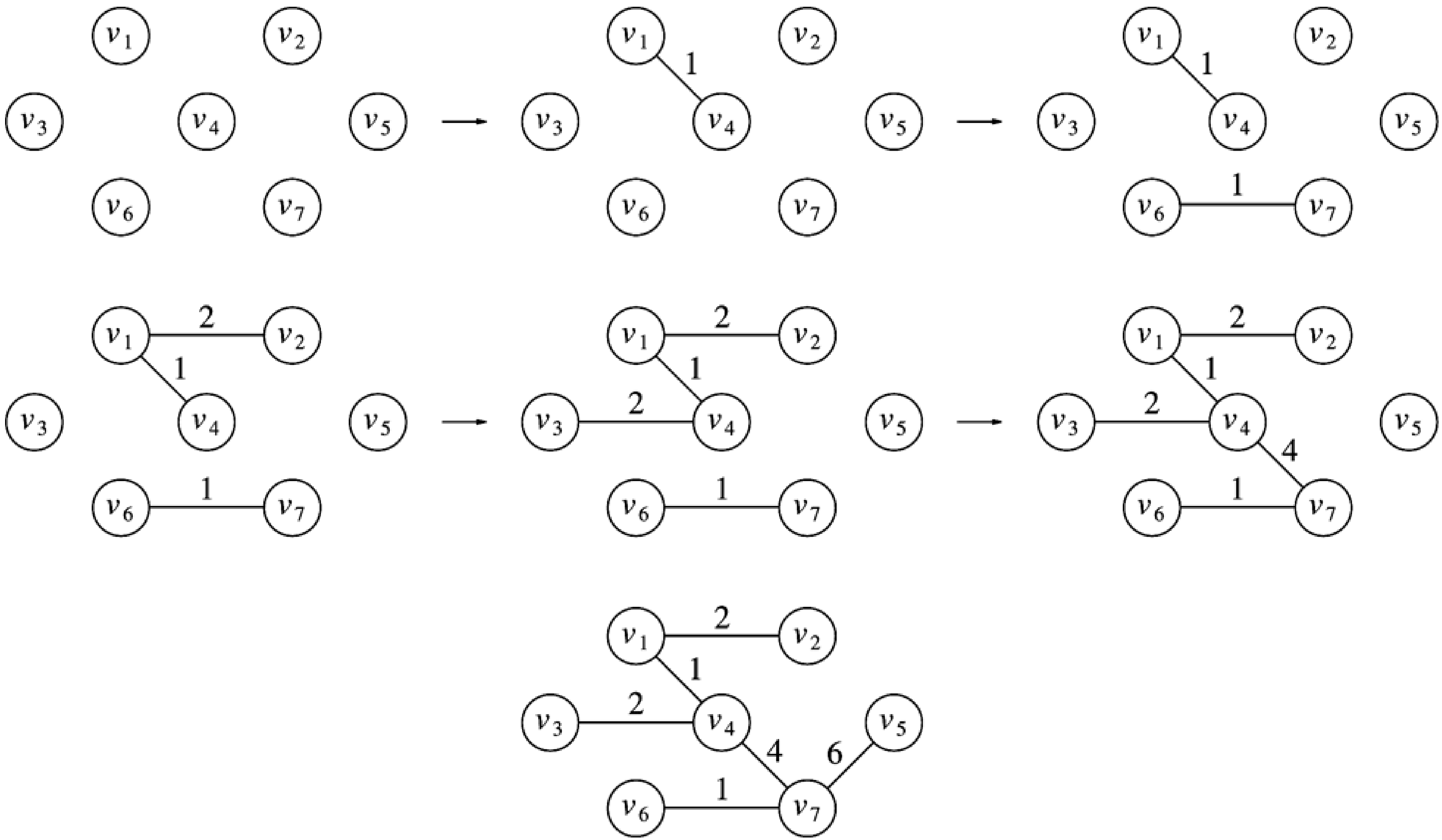
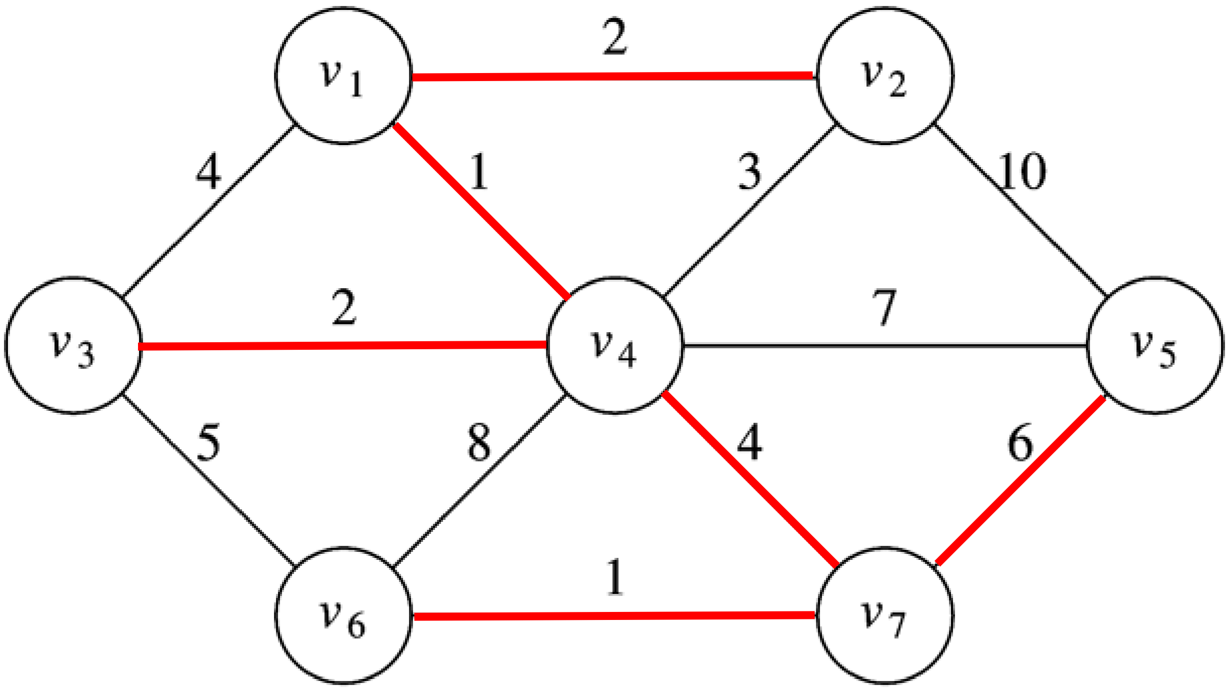


Figure 9.57 Kruskal's algorithm after each stage

MST by Kruskal Algorithm



Kruskal algorithm

```
        Kruskal( Graph G )
        {
                int EdgesAccepted;
                DisjSet S;
                PriorityQueue H;
                Vertex U, V;
                SetType Uset, Vset;
                Edge E;

/* 1*/          Initialize( S );
/* 2*/          ReadGraphIntoHeapArray( G, H );
/* 3*/          BuildHeap( H );
```

Kruskal algorithm

```
/* 4*/        EdgesAccepted = 0;
/* 5*/        while( EdgesAccepted < NumVertex - 1 )
              {
/* 6*/            E = DeleteMin( H );  /* E = (U,V) */
/* 7*/            Uset = Find( U, S );
/* 8*/            Vset = Find( V, S );
/* 9*/            if( Uset != Vset )
                  {
                      /* Accept the edge */
/*10*/                EdgesAccepted++;
/*11*/                SetUnion( S, USet, VSet );
                  }
              }
          }
```

Kruskal algorithm

- A priority queue stores the edges outside the cloud
  - Key: weight
  - Element: edge

- At the end of the algorithm
  - We are left with one cloud that encompasses the MST
  - A tree *T* which is our MST

Kruskal algorithm II

Algorithm ***KruskalMST***(***G***)
for each vertex ***V*** in ***G*** **do**
define a ***Cloud(v)*** of ← {***v***}
let ***Q*** be a priority queue.
Insert all edges into ***Q*** using their weights as the key
T ← ∅
while ***T*** has fewer than ***n***-1 edges **do**
edge ***e*** = ***T.removeMin()***
Let ***u, v*** be the endpoints of *e*
if ***Cloud(v) ≠ Cloud(u)*** **then**
Add edge ***e*** to ***T***
Merge ***Cloud(v)*** and ***Cloud(u)***
return ***T***

Dijkstra vs. Prim-Jarnik

```
Algorithm DijkstraShortestPaths(G, s)
   Q ← new heap-based priority queue

   for all v ∈ G.vertices()
      if v = s
         setDistance(v, 0)
      else
         setDistance(v, ∞)
      setParent(v, ∅)
      l ← Q.insert(getDistance(v), v)
      setLocator(v,l)
   while ¬Q.isEmpty()
      u ← Q.removeMin()
      for all e ∈ G.incidentEdges(u)
         z ← G.opposite(u,e)
         r ← getDistance(u) + weight(e)
         if r < getDistance(z)
            setDistance(z,r)
            setParent(z,e)
            Q.replaceKey(getLocator(z),r)
```

```
Algorithm PrimJarnikMST(G)
   Q ← new heap-based priority queue
   s ← a vertex of G
   for all v ∈ G.vertices()
      if v = s
         setDistance(v, 0)
      else
         setDistance(v, ∞)
      setParent(v, ∅)
      l ← Q.insert(getDistance(v), v)
      setLocator(v,l)
   while ¬Q.isEmpty()
      u ← Q.removeMin()
      for all e ∈ G.incidentEdges(u)
         z ← G.opposite(u,e)
         r ← weight(e)
         if r < getDistance(z)
            setDistance(z,r)
            setParent(z,e)
            Q.replaceKey(getLocator(z),r)
```

Depth-First Search

- Is a generalization of preorder traversal
- Starting at some vertex v, process v and then recursively traverse all vertices adjacent to v
- For graph, be careful to avoid cycles.
  => use ***Visited[ ]*** flag to mark it visited.
- Give DFS Spanning Tree of the graph

```
void
Dfs( Vertex V )
{
    Visited[ V ] = True;
    for each W adjacent to V
        if( !Visited[ W ] )
            Dfs( W );
}
```

Figure 9.59 Template for depth-first search

Depth-First Search

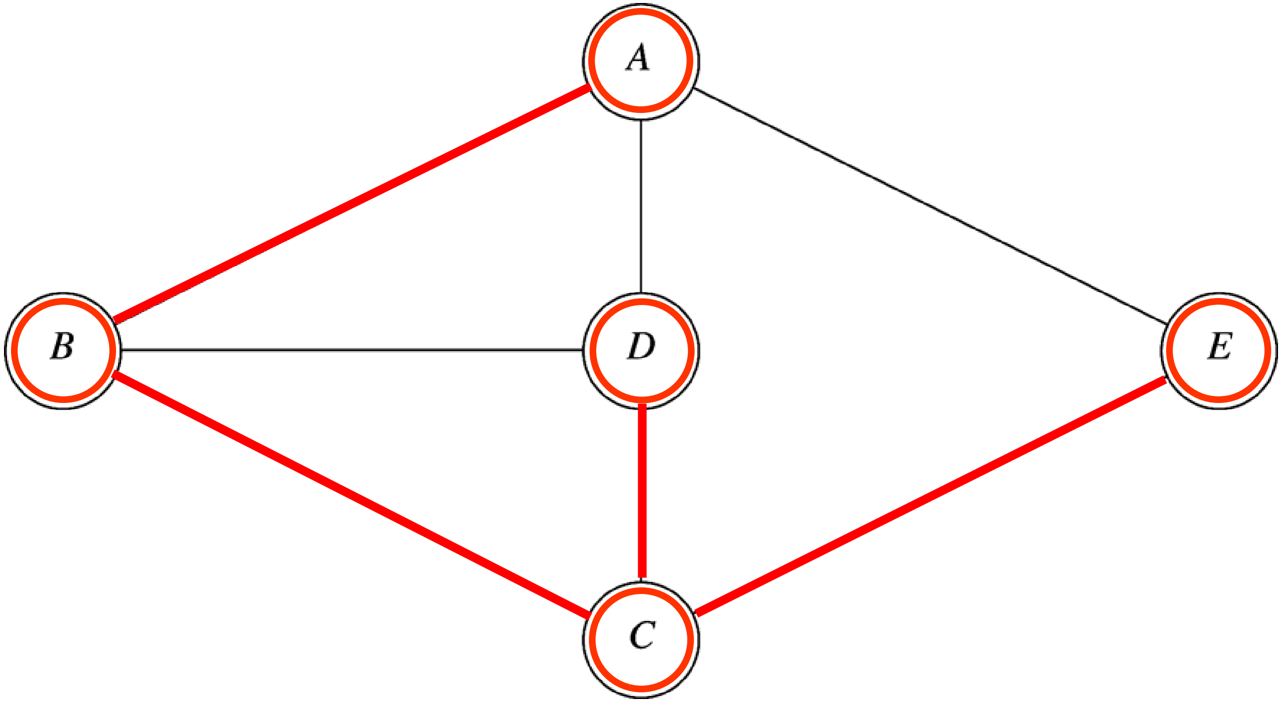


Figure 9.60 An undirected graph G

Depth-First Search

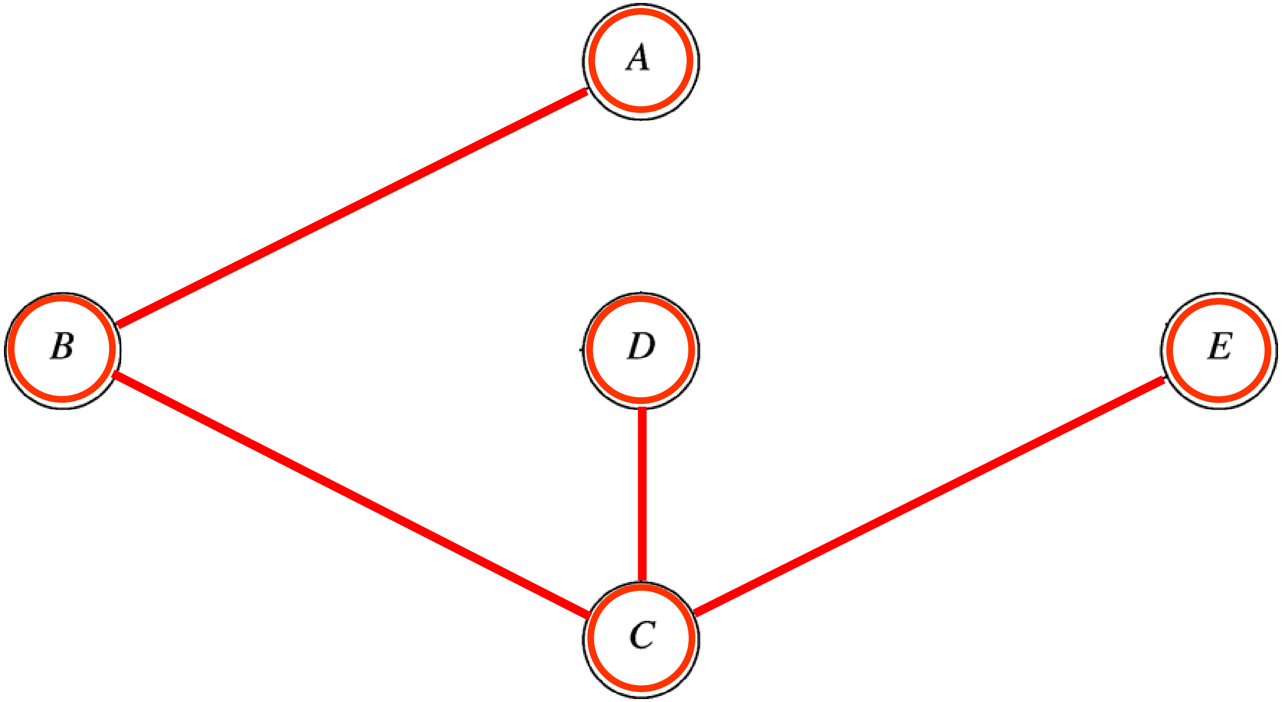


Figure 9.60 DFS Spanning Tree of G

Depth-First Search

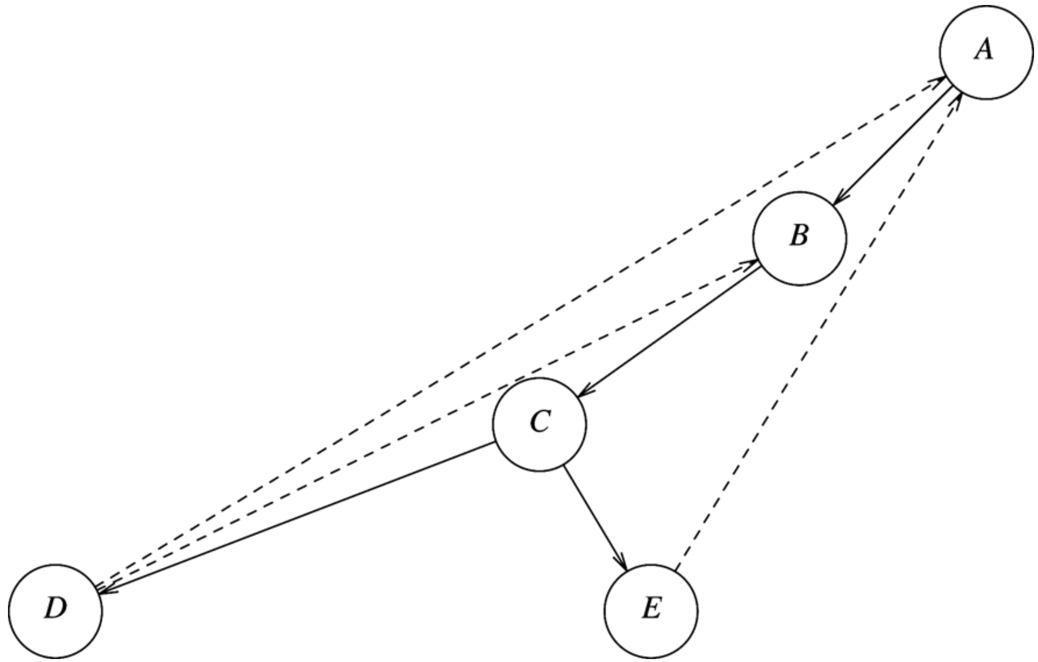


Figure 9.61 Depth-first search of previous graph

Breadth-First Search

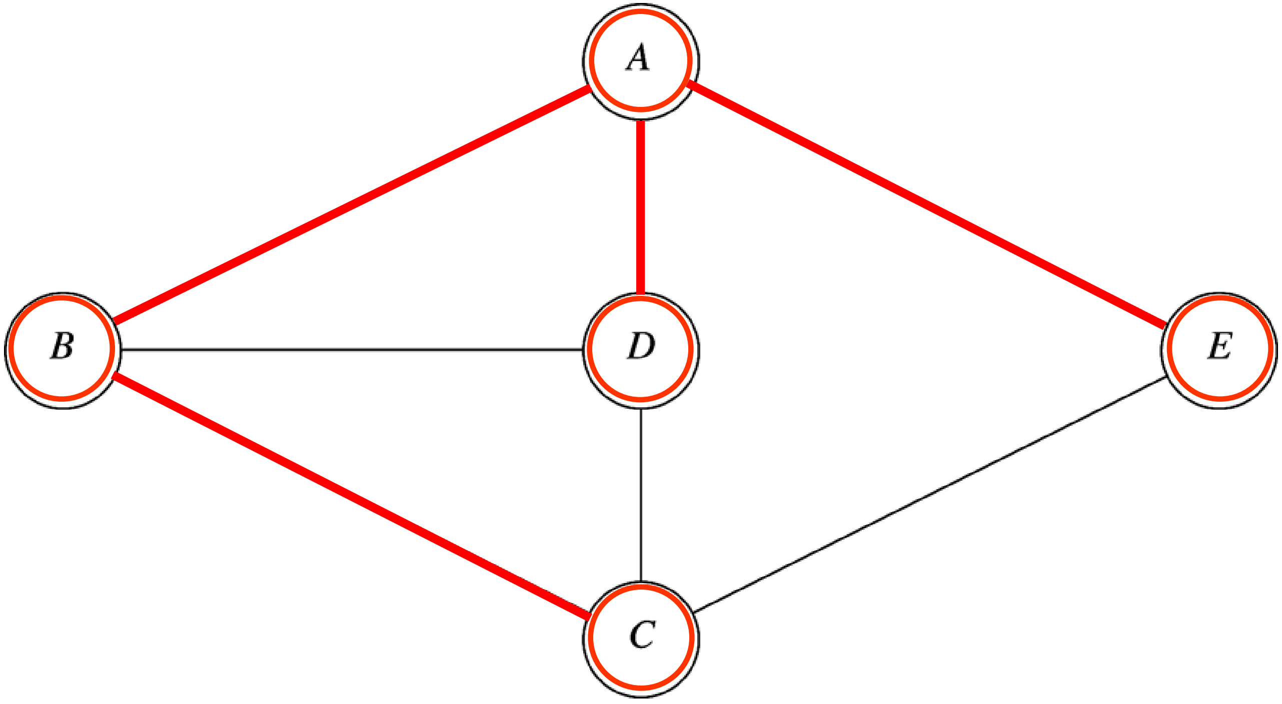


Figure 9.60 An undirected graph G

Breadth-First Search

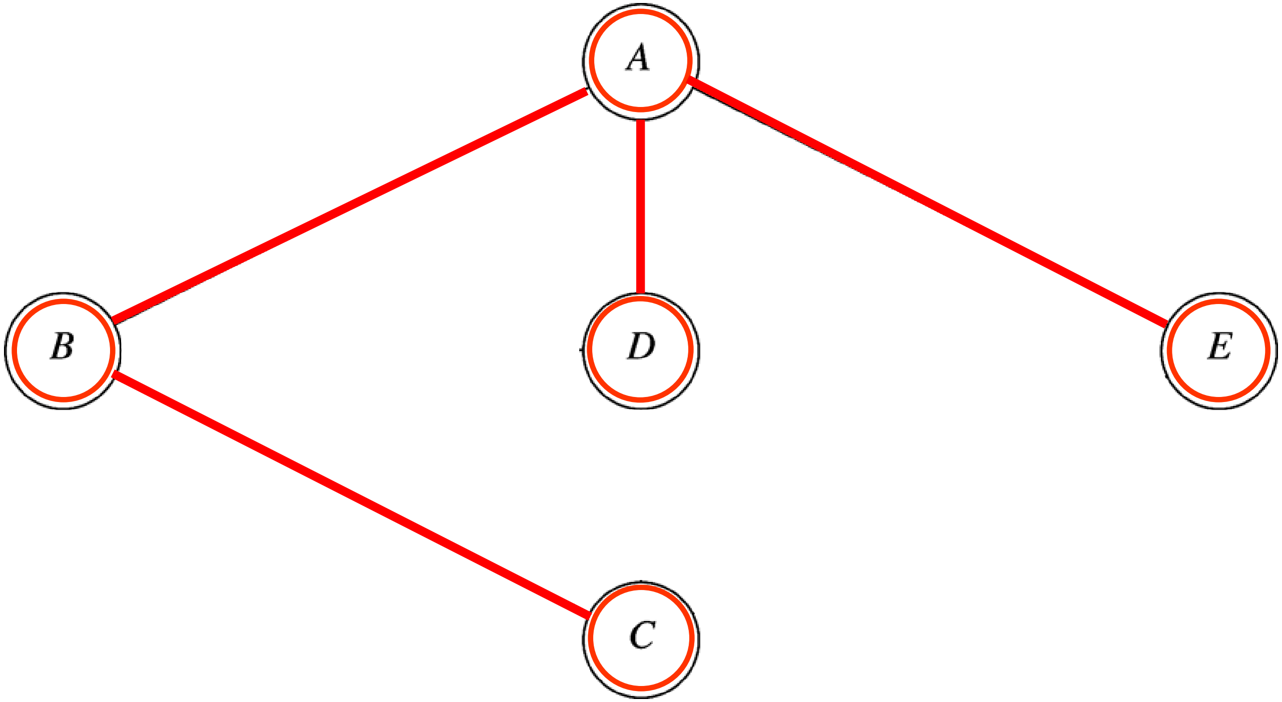
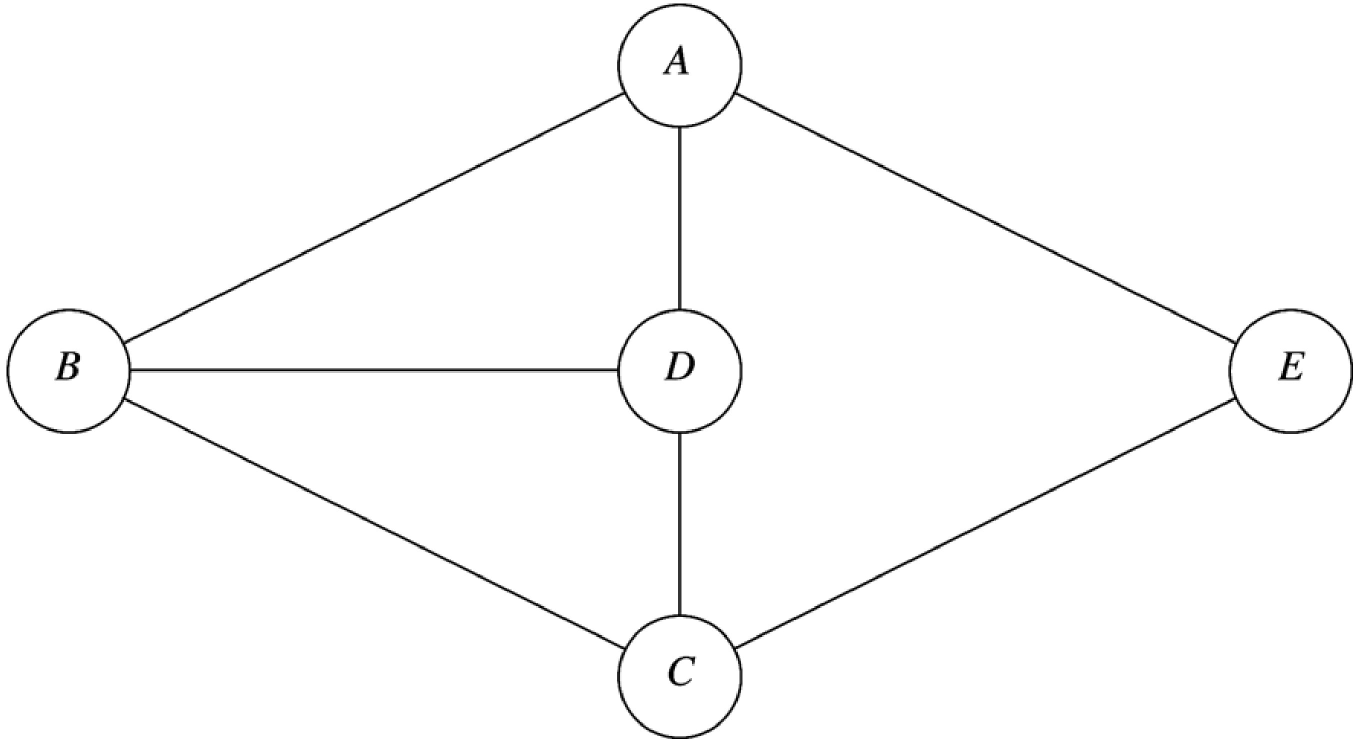


Figure 9.60 DFS Spanning Tree of G

Biconnectivity

- A connected undirected graph is biconnected if there are no vertices whose removal disconnects the rest of the graph
- Application domains: mail delivery on the computer network, alternate route on a mass transit system
- If a graph is not biconnected, the vertices whose removal would disconnect the graph are known as ***articulation points***.

Biconnected Graph



Articulation Points

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

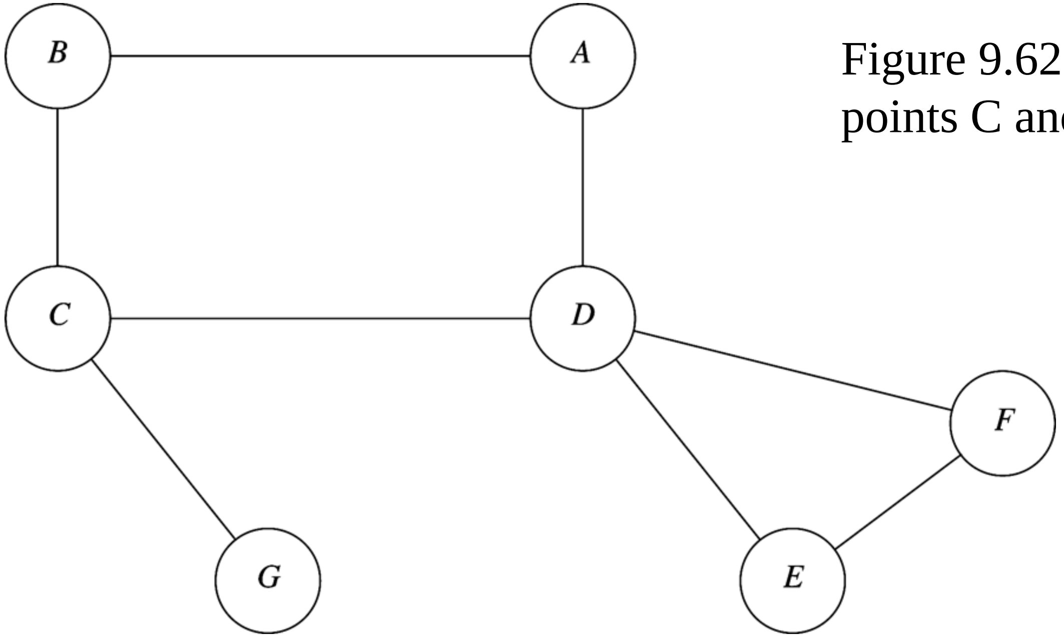
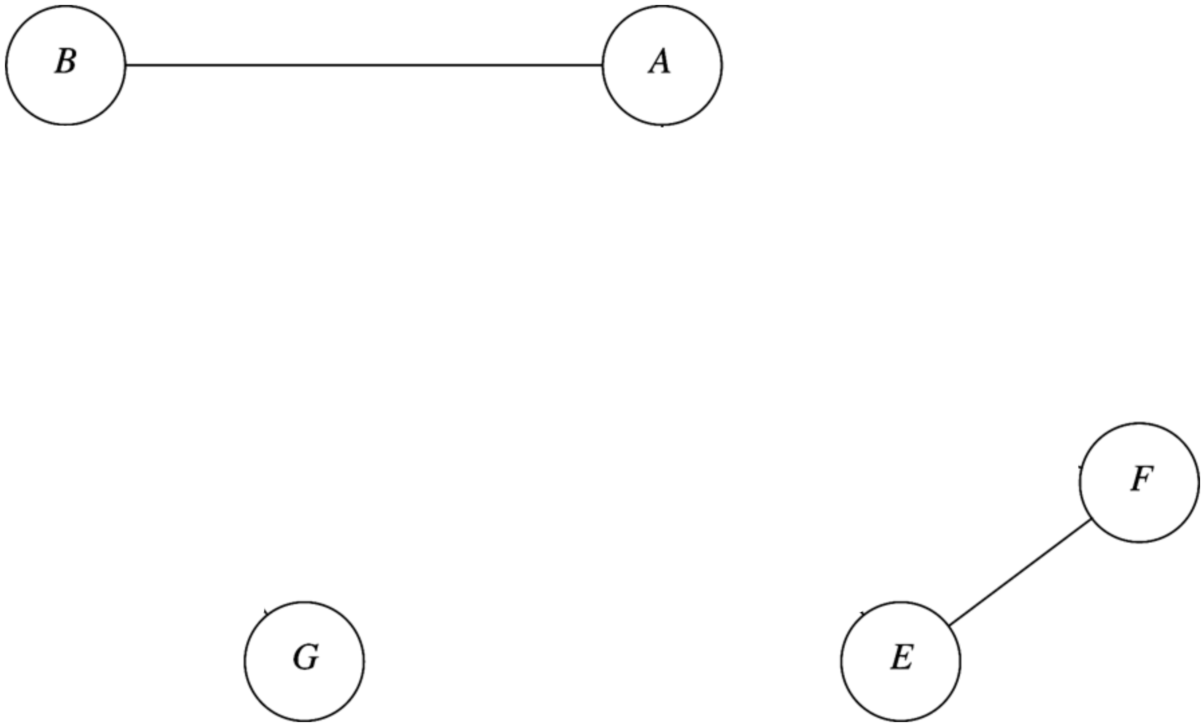


Figure 9.62 A graph with articulation points C and D

Not Biconnected Graph



Removal of C and D in Figure 9.62

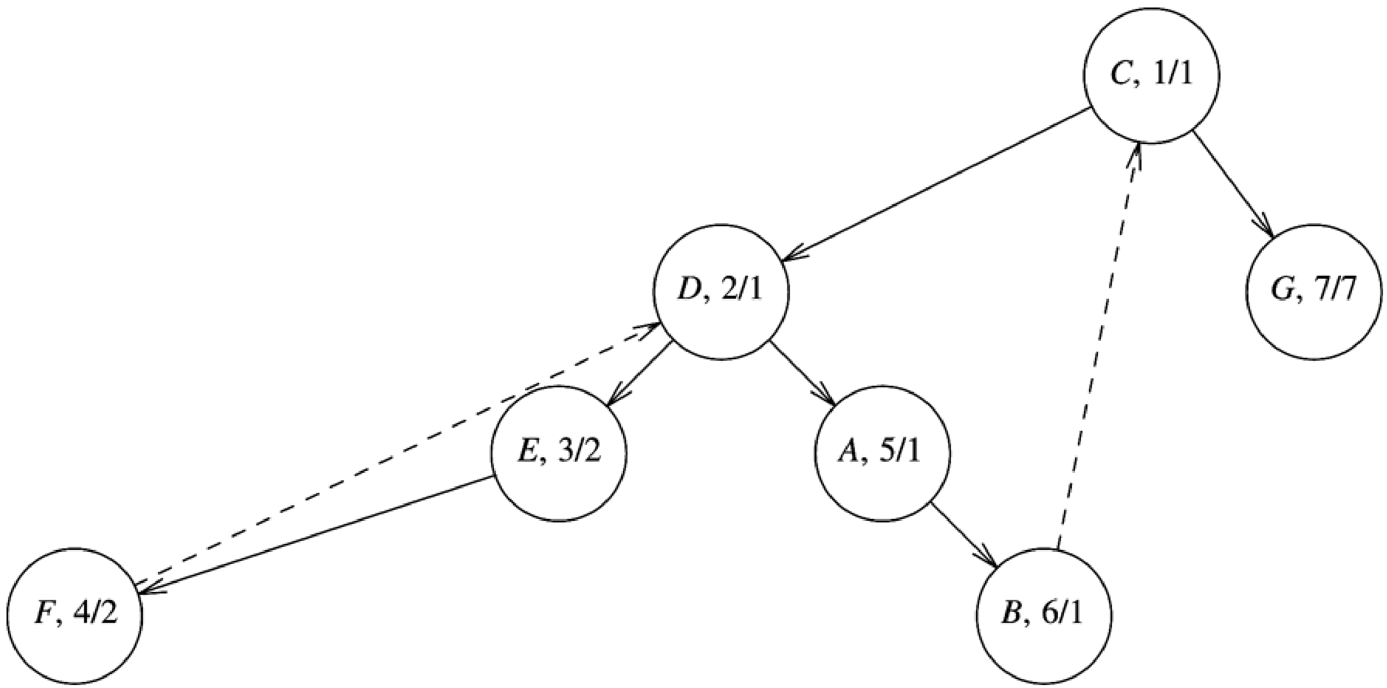
Finding Articulating points

- Depth-first search provides all articulation points in a connected graph in linear time.

1. Starting at any vertex, perform DPS ant number the nodes as they are visited. For each vertex v, we call this preorder number $Num(v)$.
2. For every vertex v in the DPS spanning tree, compute the lowest-numbered vertex, $Low(v)$ that is reachable from v by taking zero or more tree edges and them possibly one back edge (in that order)

Depth-first tree with Num & Low

Depth-first tree starting at C



```
        void
        AssignNum( Vertex V )
        {
            Vertex W;

/* 1*/      Num[ V ] = Counter++;
/* 2*/      Visited[ V ] = True;
/* 3*/      for each W adjacent to V
/* 4*/          if( !Visited[ W ] )
                {
/* 5*/              Parent[ W ] = V;
/* 6*/              AssignNum( W );
                }
        }
```

Figure 9.65 Routine to assign *Num* to vertices

```
          AssignLow( Vertex V )
          {
              Vertex W;

/* 1*/        Low[ V ] = Num[ V ];  /* Rule 1 */
/* 2*/        for each W adjacent to V
              {
/* 3*/            if( Num[ W ] > Num[ V ] )  /* Forward edge */
                  {
/* 4*/                AssignLow( W );
/* 5*/                if( Low[ W ] >= Num[ V ] )
/* 6*/                    printf( "%v is an articulation point\n", v );
/* 7*/                Low[ V ] = Min( Low[ V ], Low[ W ] ); /* Rule 3 */
                  }
                  else
/* 8*/            if( Parent[ V ] != W )  /* Back edge */
/* 9*/                Low[ V ] = Min( Low[ V ], Num[ W ] ); /* Rule 2 */
              }
          }
```

Figure 9.66 Pseudocode to compute *Low* and to test for articulation points (test for the root is omitted)

```
        FindArt( Vertex V )
        {
            Vertex W;

/* 1*/      Visited[ V ] = True;
/* 2*/      Low[ V ] = Num[ V ] = Counter++;  /* Rule 1 */
/* 3*/      for each W adjacent to V
            {
/* 4*/          if( !Visited[ W ] )  /* Forward edge */
                {
/* 5*/              Parent[ W ] = V;
/* 6*/              FindArt( W );
/* 7*/              if( Low[ W ] >= Num[ V ] )
/* 8*/                  printf( "%v is an articulation point\n", v );
/* 9*/              Low[ V ] = Min( Low[ V ], Low[ W ] ); /* Rule 3 */
                }
                else
/*10*/          if( Parent[ V ] != W )  /* Back edge */
/*11*/              Low[ V ] = Min( Low[ V ], Num[ W ] ); /* Rule 2 */
            }
        }
```

Figure 9.67 Testing for articulation points in one depth-first search (test for the root is