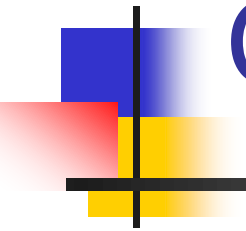


Graphs

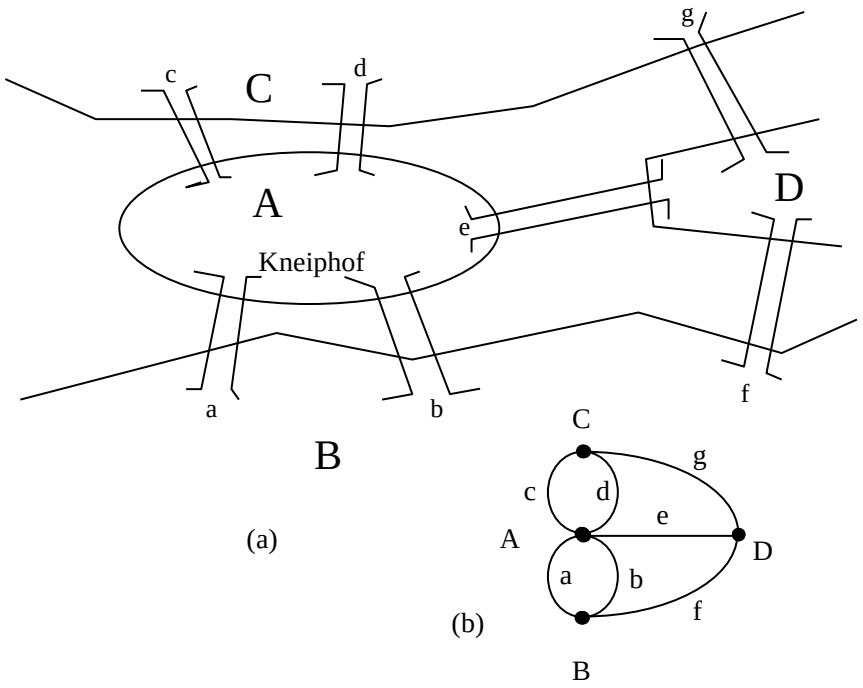
Introduction to Data Structures

Kyuseok Shim

ECE, SNU.

Graph Abstract Data Type

- Konigsberg bridge problem



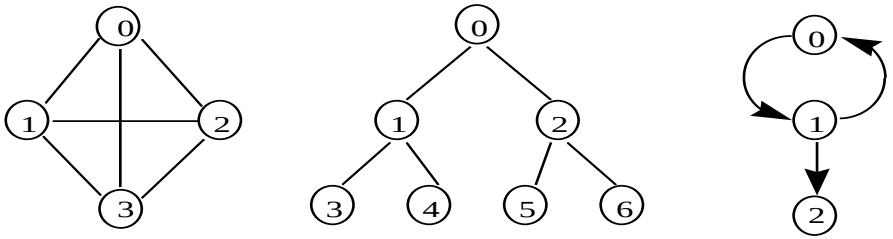
-Eulerian walk
Degree of each vertex is even

Figure 6.1 : (a) Section of the river Pregel in Konigsberg; (b) Euler's graph

Graph Abstract Data Type (Cont.)

- Graph G=(V, E)
  - V is a finite, nonempty set of vertices
  - E is a set of edges
  - An edge is a pair of vertices
  - V(G) is the set of vertices of G
  - E(G) is the set of edges of G

- Undirected (directed) graph
  - The pair of vertices representing an edge is unordered (ordered)

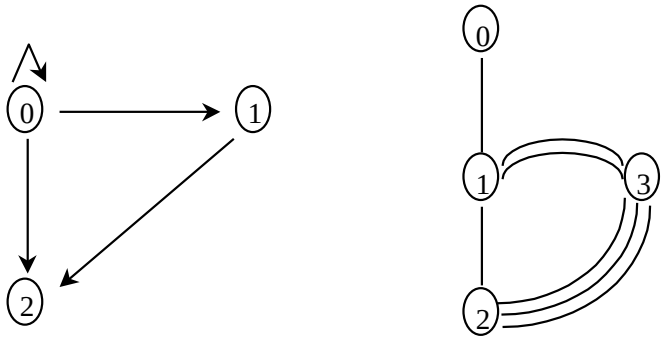


(a) G_1 (b) G_2 (c) G_3

Figure 6.2 : Three sample graphs

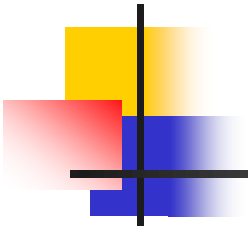
Graph Abstract Data Type (Cont.)

- Restriction
  - A graph may not have an edge from a vertex back to itself
  - A graph may not have multiple occurrences of the same edge



(a) Graph with a self edge (b) Multigraph

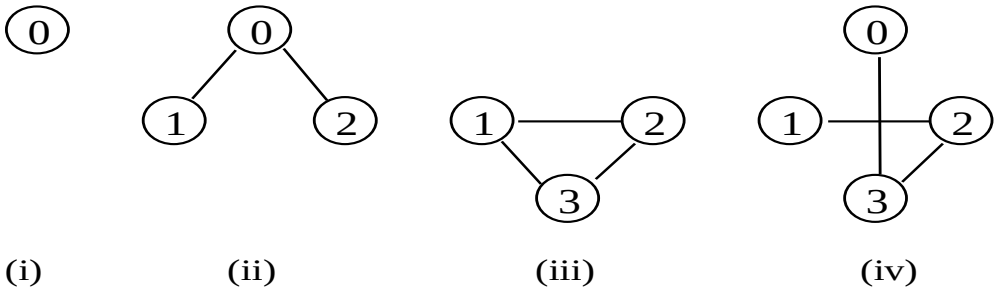
Figure 6.3 : Examles of graphlike structures



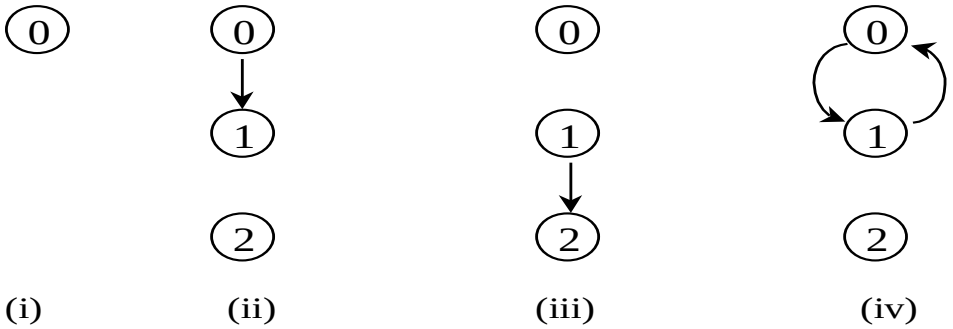
Graph Abstract Data Type (Cont.)

- Complete graph
  - n-vertex, undirected graph with n(n-1)/2 edges
- (u,v) is an edge in E(G)
  - Vertices u and v are adjacent
  - (u,v) is incident on vertices u and v
  - if (u, v) is a directed edge
    - u is adjacent to v
    - v is adjacent from u
- Subgraph of G
  - Graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$

Graph Abstract Data Type (Cont.)



(a) some of the subgraphs of G_1



(b) some of the subgraphs of G_3

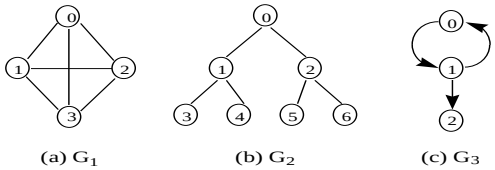
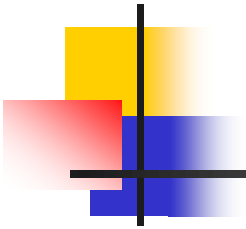


Figure 6.4 : Some subgraphs



Graph Abstract Data Type (Cont.)

- Path from u to v in G
  - A sequence of vertices u, i_1, i_2, ..., i_k, v such that (u,i_1) (i_1, i_2) ...(i_k, v) are edges in E(G)
  - Length of path is number of edges on it
  - Simple path is path in which all vertices except possibly the first and last are distinct
  - Cycle is a simple path in which the first and last vertices are the same

Graph Abstract Data Type (Cont.)

- Vertices u and v are connected in (undirected) graph G, there is a path in G from u to v
- Connected graph
  - For every pair of distinct vertices u and v in V(G) there is a path from u and v
- (connected) Component
  - A maximally connected subgraph
  - Maximal: no more vertices or edges can be added while preserving its connectivity

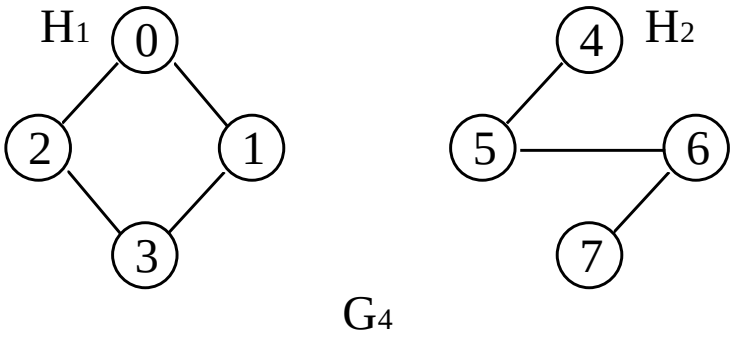


Figure 6.5 : A graph with two connected components

Graph Abstract Data Type (Cont.)

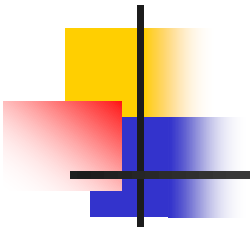
- Tree
  - Connected acyclic graph
- Degree of vertex
  - Number of edges incident to that vertex
- d_i is the degree of vertex i in G with n vertices and e edges

$$e = (\sum_{i=0}^{n-1} d_i)/2$$

Graph Abstract Data Type (Cont.)

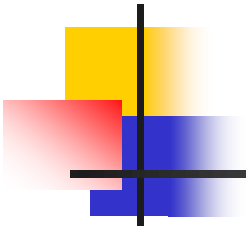
```
1.  class Graph
2.  {// objects: A nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices.
3.  public:
4.          virtual ~Graph() {}
5.                  // virtual destructor
6.          bool IsEmpty() const { return n == 0 };
7.                  // return true iff graph has no vertices
8.          int NumberOfVertices() const { return n };
9.                  // return number of vertices in the graph
10.         int NumberOfEdges() const { return e };
11.                 // return number of edges in the graph
12.         virtual int Degree(int u) const = 0;
13.                 // return number of edges incident to vertex u
14.         virtual bool ExistsEdge(int u, int v) const = 0;
15.                 // return true iff graph has the edge (u,v)
16.         virtual void InsertVertex(int v) = 0;
17.                 // insert vertex v into graph; v has no incident edges
18.         virtual void InsertEdge(int u, int v) = 0;
19.                 // insert edge (u,v) into graph
20.         virtual void DeleteVertex(int v) = 0;
21.                 // delete v and all edges incident to it
22.         virtual void DeleteEdge(int u, int v) = 0;
23.                 // deelte edge (u,v) from the graph
24. private:
25.                 int n;          //number of vertices
26.                 int e;          // number of edges
27. };
```

ADT 6.1 : Abstract data type Graph



Graph Representations

- Adjacency Matrix
- Adjacency Lists



Adjacency Matrix

- Definition
  - G=(V,B) is a graph with n vertices, n≥1
  - Adjacency matrix A of G
    - two dimensional n x n array
    - A[i][j]=1 iff edge(i, j) is in E(G)
- Properties
  - Space needed is n^2
  - A is symmetric for undirected G
    - Need only the upper or lower triangle of A

Adjacency Matrix (Cont.)

- Degree of vertex i for an undirected graph
  - $$i = \sum_{j=0}^{n-1} A[i][j]$$
- How many edges are in a directed graph?
  - Complexity of operations : $n^2 - n = O(n^2)$ since diagonal entries are zero

	0	1	2	3
0	0	1	1	1
1	1	0	1	1
2	1	1	0	1
3	1	1	1	0

(a) G_1

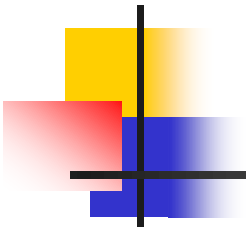
	0	1	2
0	0	1	0
1	1	0	1
2	0	0	0

(b) G_3

	0	1	2	3	4	5	6	7
0	0	1	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0
2	1	0	0	1	0	0	0	0
3	0	1	1	0	0	0	0	0
4	0	0	0	0	0	1	0	0
5	0	0	0	0	1	0	1	0
6	0	0	0	0	0	1	0	0
7	0	0	0	0	0	0	1	0

(c) G_4

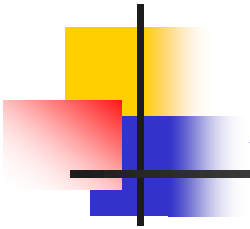
Figure 6.7 : Adjacency matrices



Adjacency Lists

- Representation
  - One list for each vertex in G
    - Nodes in list i represent vertices that are adjacent from vertex i
    - Each list has a head node
    - Vertices in a list are not ordered
  - Fields of node
    - data : index of vertex adjacent to vertex i
    - link
- Declaration in C++

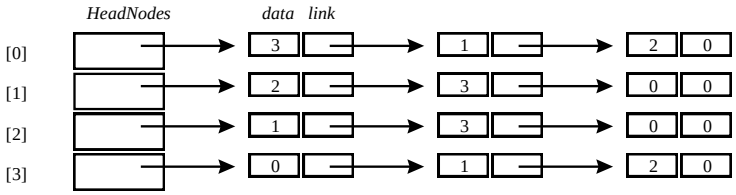
```
class Graph
{
private:
   List<int> *HeadNodes;
   int n;
public:
   Graph(const int vertices = 0) : n(vertices)
   { HeadNodes = new List<int>[n];};
};
```



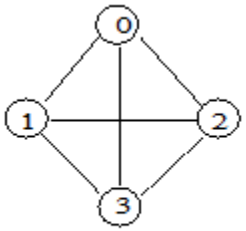
Adjacency Lists (Cont.)

- For n vertices and edges
  - Requires n head nodes and 2e list nodes
- Complexity of operations
  - Number of nodes in adjacency list = O(n+e)

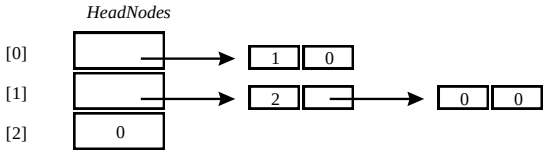
Adjacency Lists (Cont.)



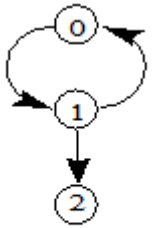
(a) G1



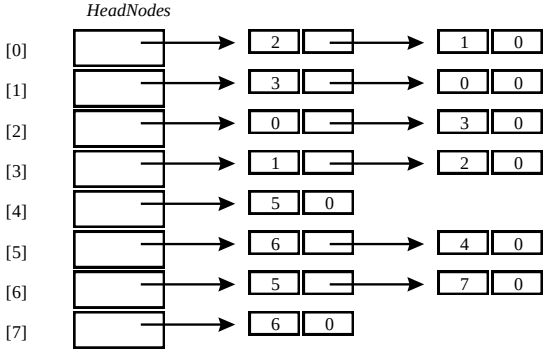
(a) G_1



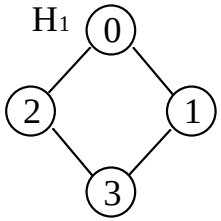
(b) G3

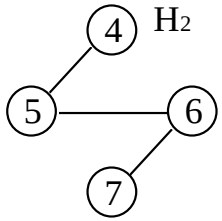


(c) G_3



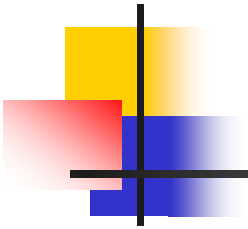
(c) G4





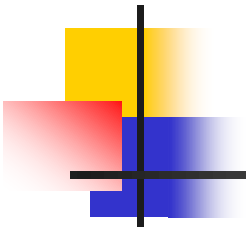
G_4

Figure 6.8 : Adjacency lists



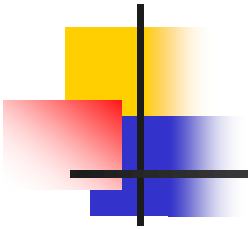
Weighted Edges

- Network
  - Graph with weighted edges
- Representation
  - Adjacency matrix
    - A[i][j] keeps weight
  - Adjacency list
    - Additional field in list node keeps weight



Elementary Graph Operations

- Graph traversal
  - Given G=(V, E) and a vertex v in V(G)
  - Visit all vertices reachable from v



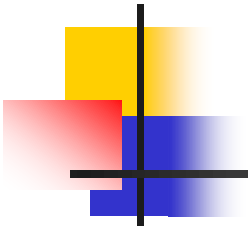
Depth-First-Search

- Procedure
  1. Visit start vertex v
  2. An unvisited vertex w adjacent to v is selected, and initiate DFS from w
  3. When u is reached such that all its adjacent vertices have been visited
     - Back up to the last vertex visited that has an unvisited vertex w adjacent to it
     - Initiate DFS from w
  4. Search terminates when no unvisited vertex can be reached from visited vertices

Depth-First-Search (Cont.)

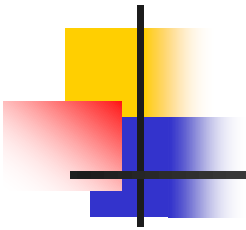
```
1.  virtual void Graph::DFS() // Driver
2.  {
3.          visited = new bool[n];
4.                  // visited is declared as a bool* data member of graph
5.          fill(visited, visited + n, false);
6.          for(int v=0; v<n; v++)
7.               if(visited[v] == false)
8.                    DFS(v); // start search at vertex 0
9.          delete [] visited;
10. }

11. virtual void Graph::DFS(const int v) // Workhorse
12. { // Visit all previously unvisited vertices that are reachable from vertex v.
13.         visited[v] = true;
14.         for(each vertex w adjacent to v) // actual code uses an iterator
15.              if(!visited[w]) DFS(w);
16. }
```

Program 6.1 : Depth-first search



Depth-First-Search (Cont.)

- Analysis
  - If adjacency list is used
    - Examines each node in the adjacency lists at most once
    - There are 2e list nodes
    - O(e)
  - If adjacency matrix is used
    - time to determine all adjacent vertices to v : O(n)
    - total time : $O(n^2)$



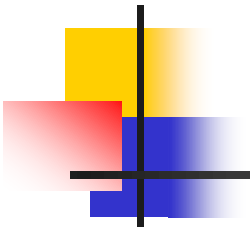
Breadth-First-Search

- Procedure
  1. Visit start vertex v
  2. Visit all unvisited vertices adjacent to v
  3. Visit unvisited vertices adjacent to the newly Visited vertices

Breadth-First-Search (Cont.)

```
1.  virtual void Graph::BFS() // Driver
2.  {
3.          visited = new bool[n];
4.                  // visited is declared as a bool* data member of graph
5.          fill(visited, visited + n, false);
6.          for(int v=0; v<n; v++)
7.              if(visited[v] == false)
8.                  BFS(v); // start search at vertex 0
9.          delete [] visited;
10. }

11. virtual void Graph::BFS(int v)
12. { // A breadth first search of the graph is carried out beginning at vertex v.
13.    // visited[i] is set to true when v is visited. The function uses a queue.
14.         visited[v] = true;
15.         Queue<int> q;
16.         q.Push(v);
17.         while(!q.IsEmpty()) {
18.                 v = q.Front();
19.                 q.Pop();
20.                 for(all vertices w adjacent to v) // actual code uses an iterator
21.                         if(!visited[w]) {
22.                                 q.Push(w);
23.                                 visited[w] = true;
24.                         }
25.         } // end of while loop
26. }
```

Program 6.2 : Breadth-first search



Breadth-First-Search (Cont.)

- Analysis
  - Adjacency matrix : $O(n^2)$
  - Adjacency list : $O(e)$

Example (DFS and BFS)

- DFS
  - 0 -> 1 -> 3 -> 7 -> 4 -> 5 -> 2 -> 6
- BFS
  - 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7

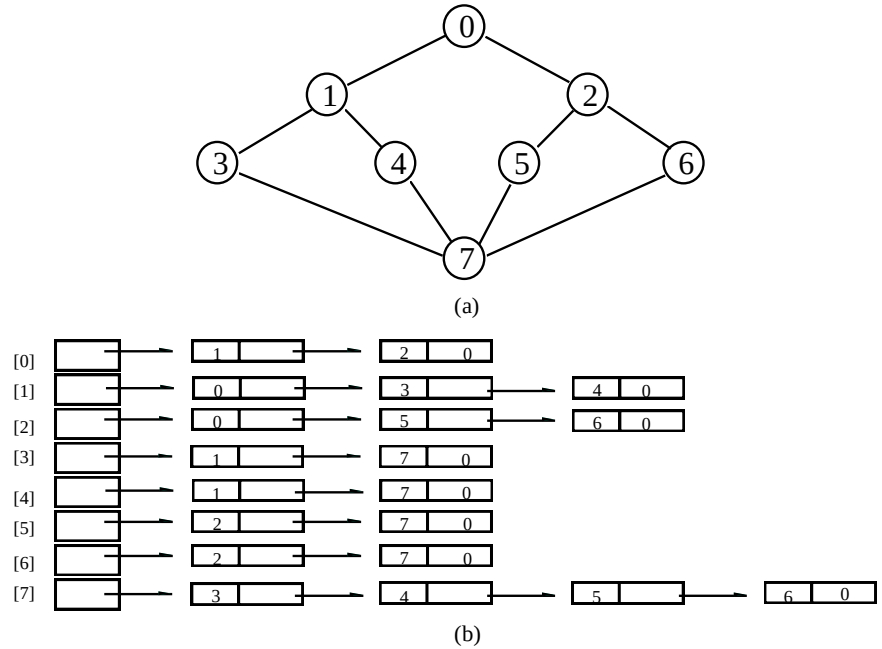
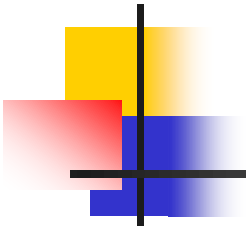


Figure 6.17 : Graph G and its adjacency lists



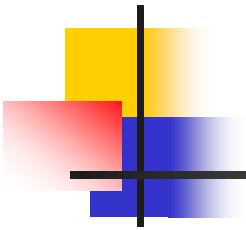
Connected Components

- If G is an undirected Graph, one can know its connectivity by simply making a call to either DFS or BFS
  - Making a call to either DFS or BFS and then determining if there is any unvisited vertex
- Determining connected components
  - Obtained by making repeated calls to either DFS or BFS
  - Start with a vertex that has not yet been visited

Connected Components (Cont.)

```
1.  virtual void Graph::Components()
2.  { // Determine the connected components of the graph.
3.          // visited is assumed to be declared as a bool* data member of Graph
4.          visited = new bool[n];
5.          fill(visited, visited + n, false);
6.          for(i =0; i < n; i++)
7.                  if(!visited[i]) {
8.                          DFS(i); // find a componenet
9.                          OutputNewComponents();
10.                 }
11.         delete [] visited;
12. }
```

Program 6.3 : Determining connected components



Connected Components (Cont.)

- Analysis
  - Adjacency matrix : $O(n^2)$
  - Adjacency list : $O(n+e)$

Spanning Trees

- Spanning tree
  - Tree consisting of edges in G and including all vertices
  - Depth-First-Spanning tree
  - Breadth-First-Spanning tree

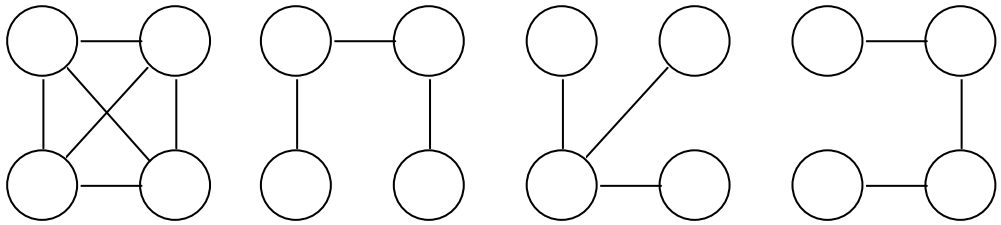
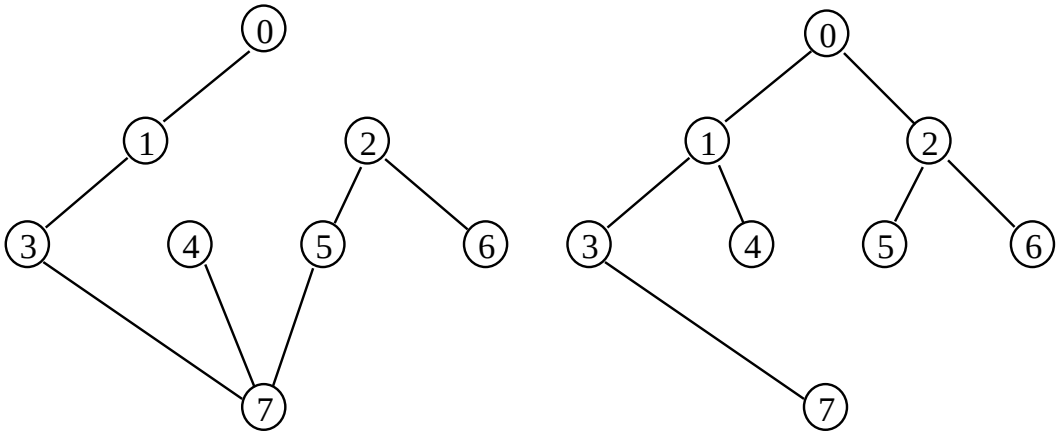


Figure 6.18 : A complete graph and three of its spanning trees

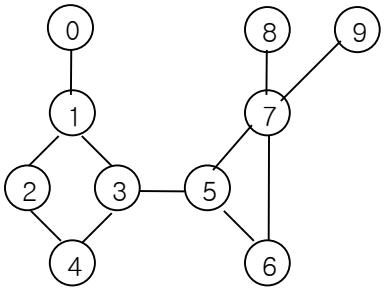
Spanning Trees (Cont.)



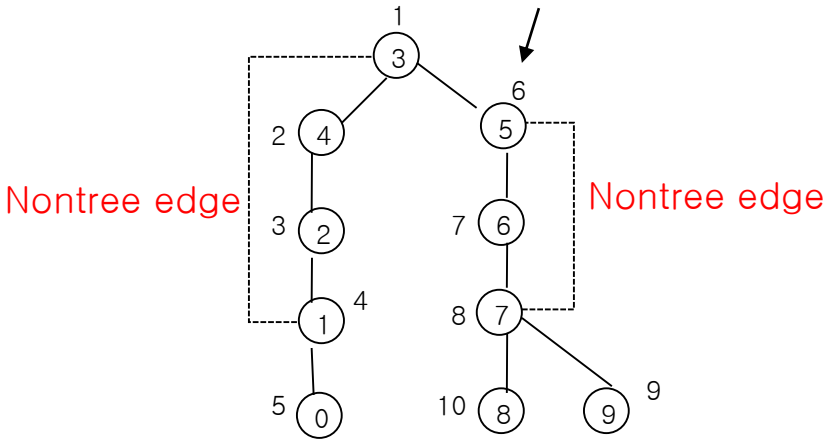
(a) *DFS*(0) spanning tree (b) *BFS*(0) spanning tree

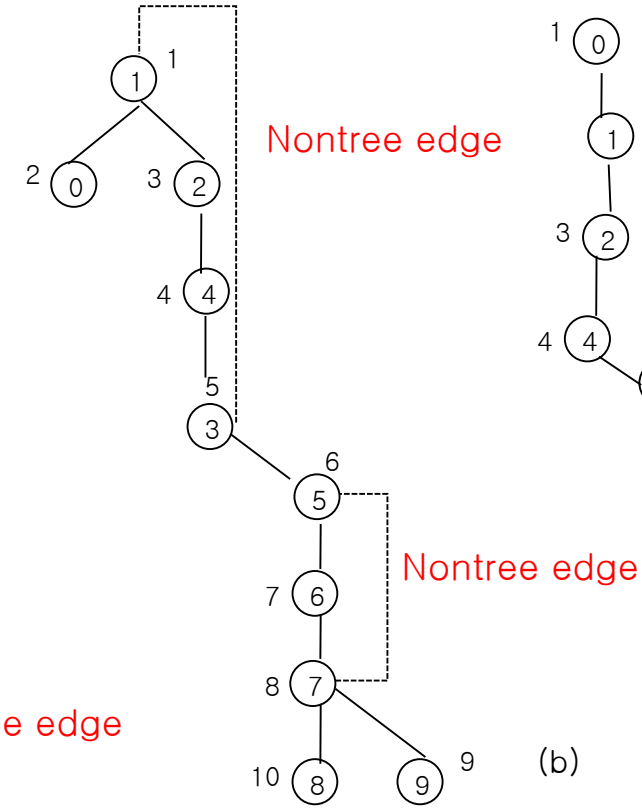
Figure 6.19 : Depth-first and breadth-first spanning trees for graph of Figure 6.17

Depth-first spanning trees

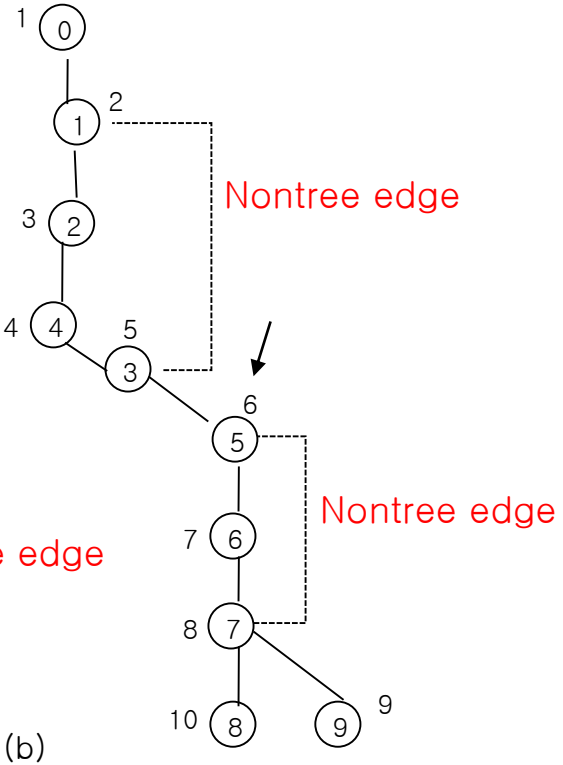


Original graph



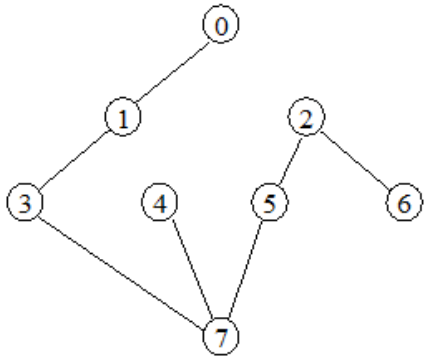


(b)

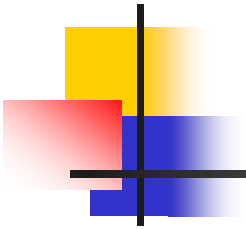


Spanning Trees (Cont.)

- Properties
  - If a non tree edge is introduced into any spanning tree, then a cycle is formed
    - Ex) If (7,6) edge is added to Fig 6.19(a), then the resulting cycle is 7,6,2,5,7
    - Used to obtain an independent set of circuit equations for an electrical network
  - A Spanning tree is a minimal subgraph G' of G such that
    - V(G') = V(G)
    - G’ is connected
  - Spanning tree has n-1 edges



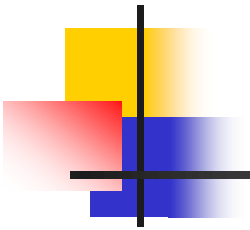
(a) *DFS*(0) spanning tree



Biconnected Components

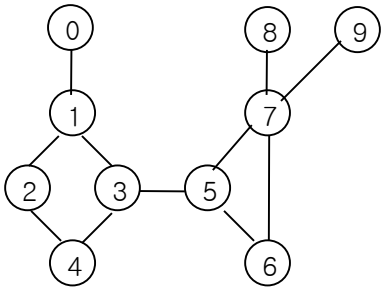
- Articulation point
  - A vertex v whose deletion operation leaves behind a graph that has at least two connected components
- Biconnected graph
  - A connected graph that has no articulation points
- Biconnected component
  - Maximal biconnected subgraph
  - The original graph contains no other biconnected subgraph

Biconnected Components (Cont.)

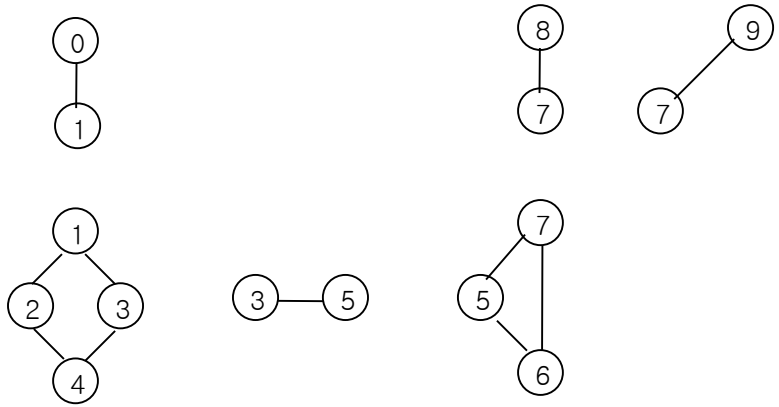


- A biconnected graph has just one biconnected component.
- Two biconnected components of the same graph can have at most one vertex in common.
- No edge can be in two or more biconnected components.
- Biconnected components of a graph G partition the edges of G.
- The biconnected components of a connected undirected graph G can be found by using any depth-first spanning tree of G.

Biconnected Components (Cont.)



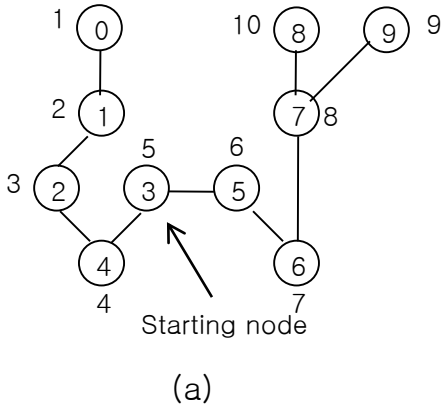
(a) A connected graph



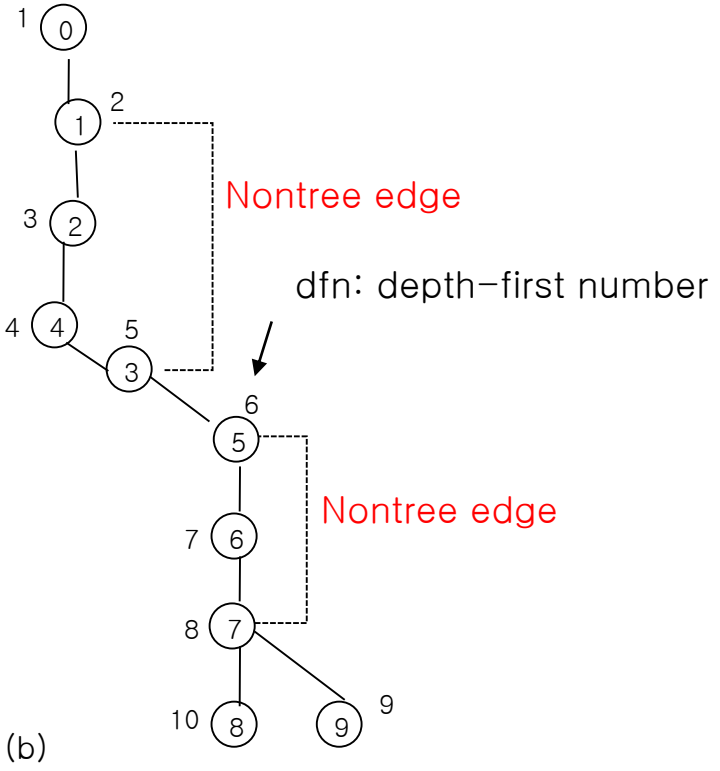
(b) Its biconnected components

Figure 6.20: A connected graph and its biconnected components

A Depth-first Spanning Tree with Root 0



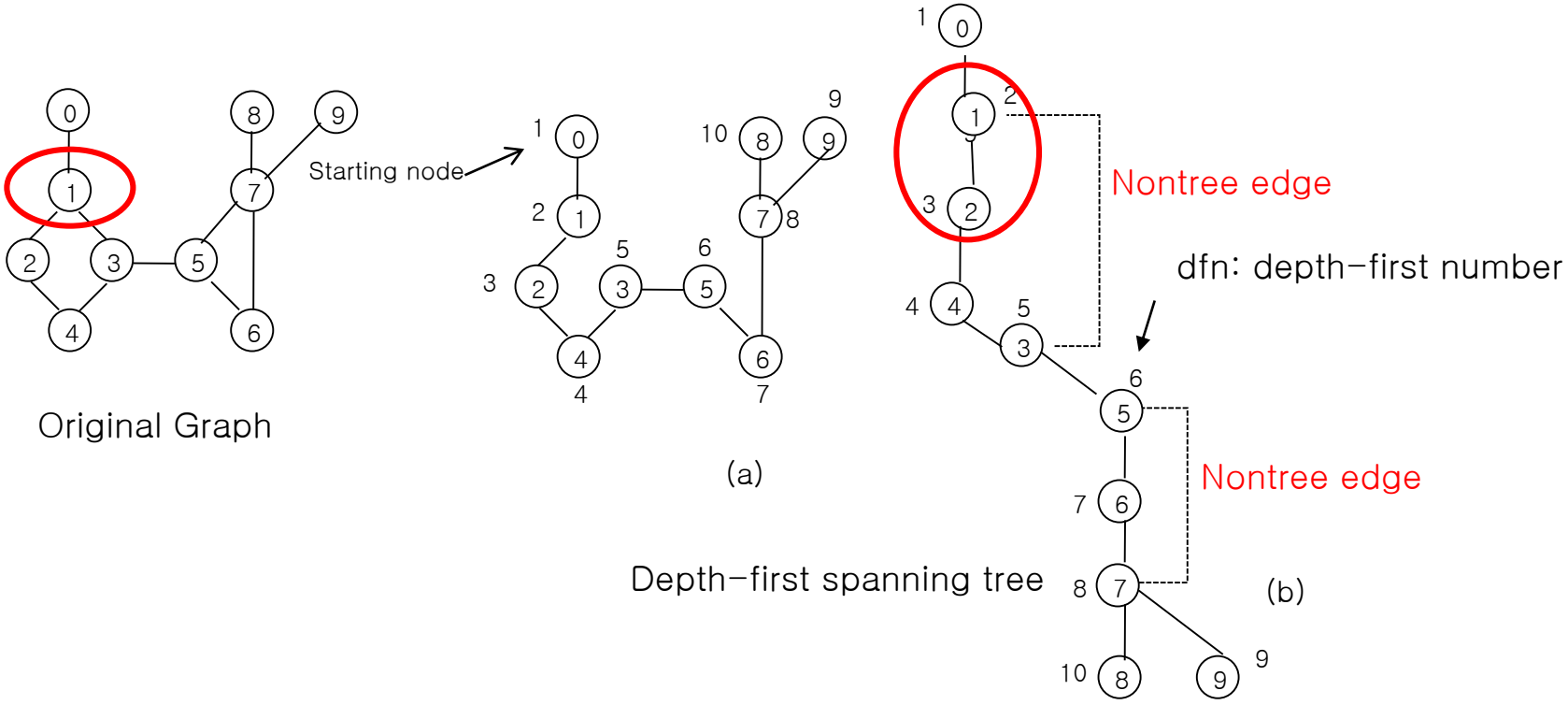
(a)



(b)

Figure 6.21: Depth-first spanning tree of Figure 6.20(a)

Determining Biconnected Components (Cont.)

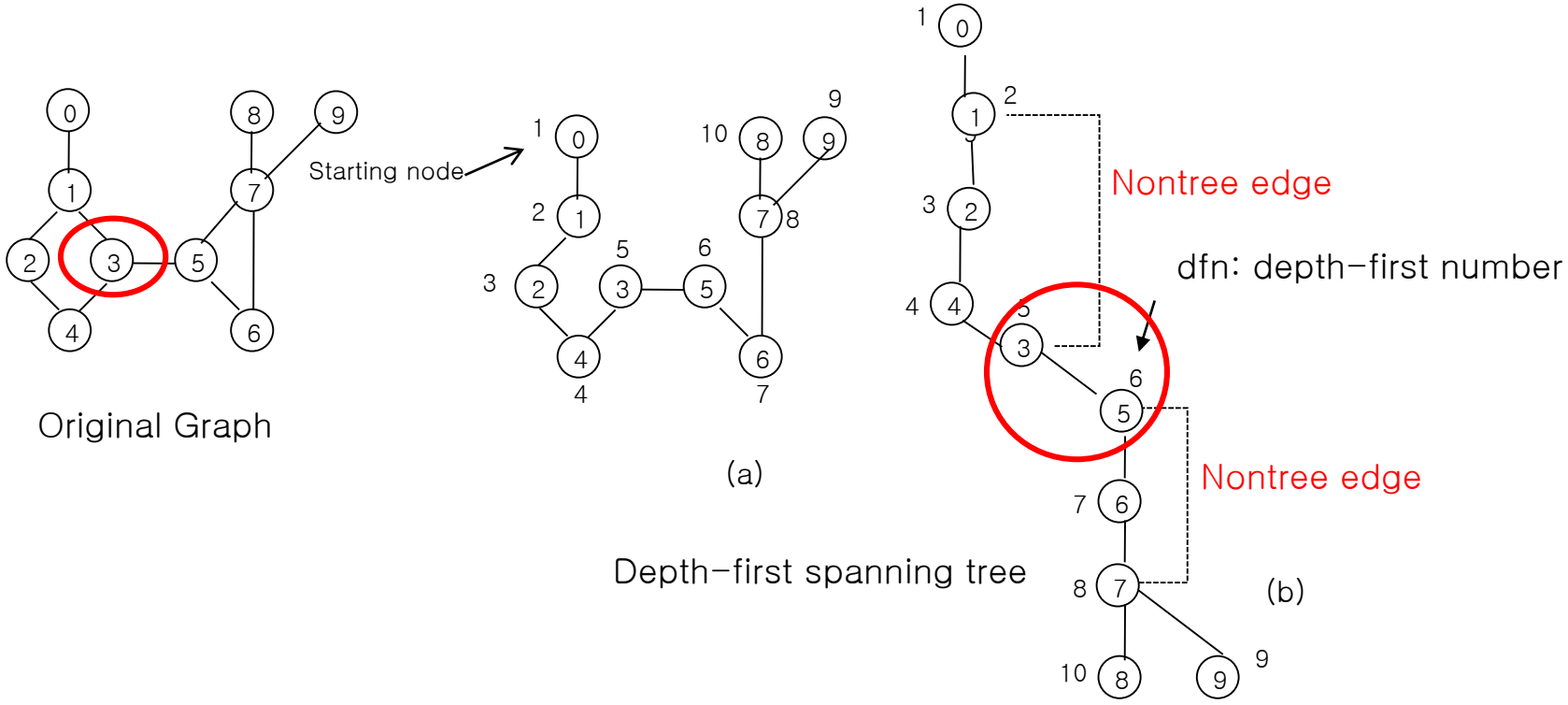


low(w) ≥ dfn(u)

Vertex	0	1	2	3	4	5	6	7	8	9
dfn	1	2	3	5	4	6	7	8	10	9
low	1	2	2	2	2	6	6	6	10	9

dfn and low values for the spanning tree

Determining Biconnected Components (Cont.)

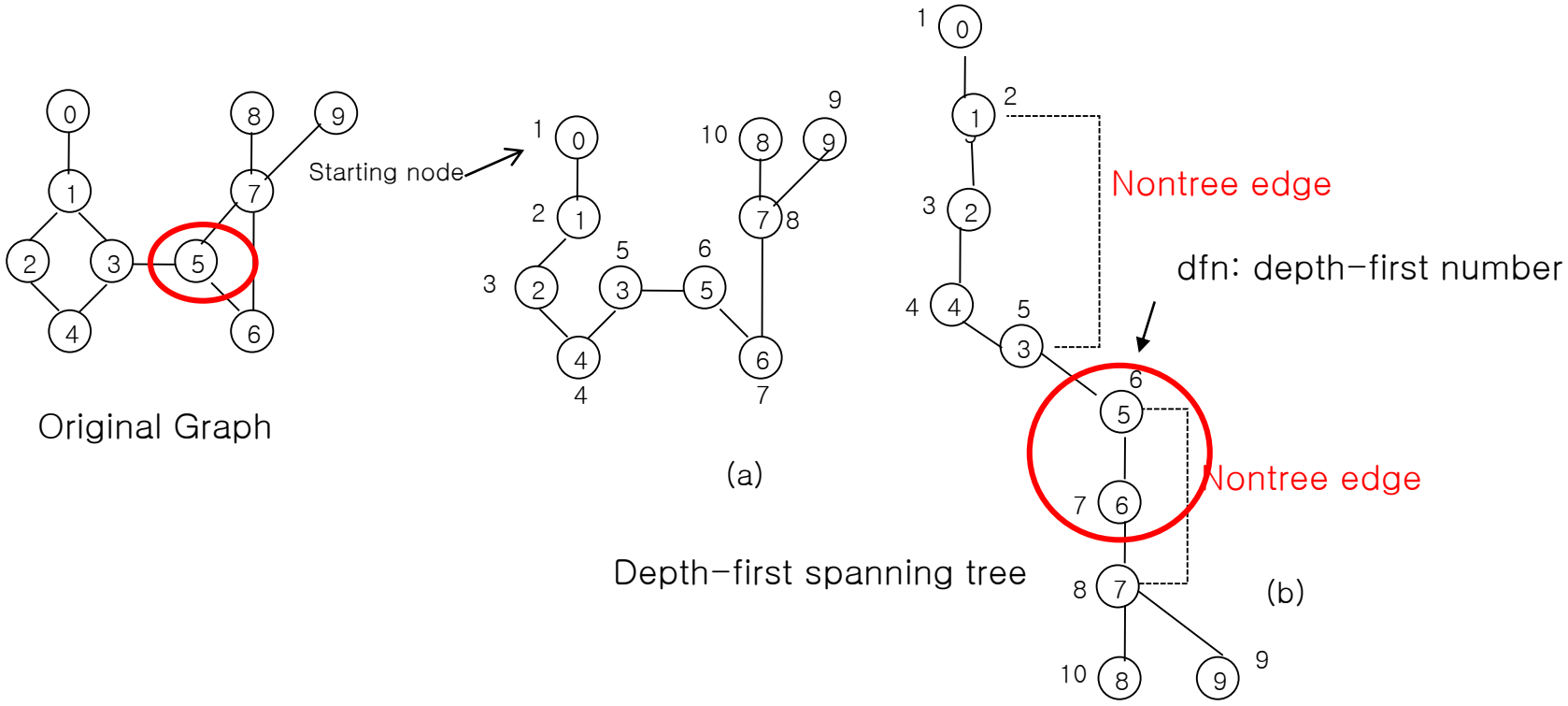


low(w) ≥ dfn(u)

Vertex	0	1	2	3	4	5	6	7	8	9
dfn	1	2	3	5	4	6	7	8	10	9
low	1	2	2	2	2	6	6	6	10	9

dfn and low values for the spanning tree

Determining Biconnected Components (Cont.)

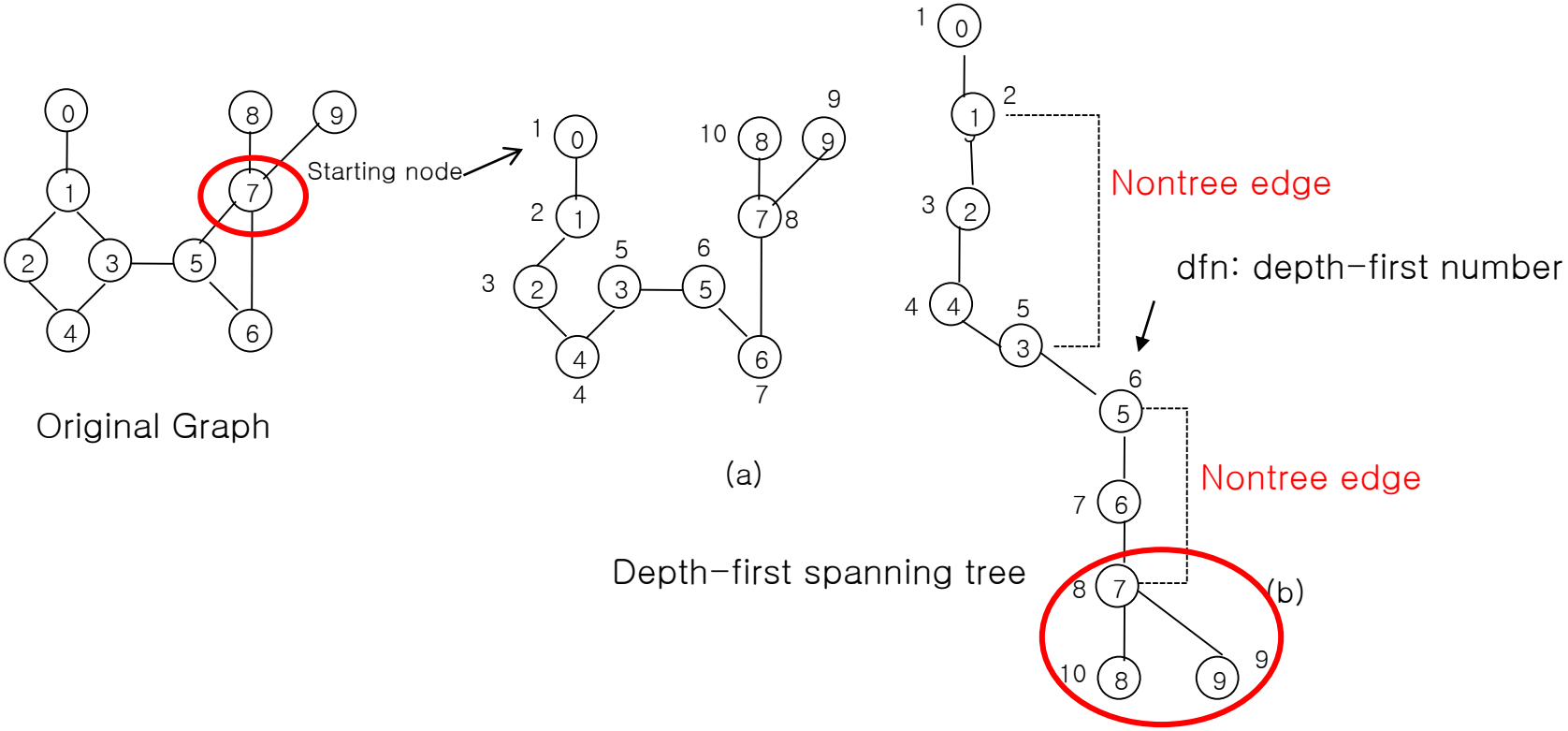


$low(w) \geq dfn(u)$

Vertex	0	1	2	3	4	5	6	7	8	9
dfn	1	2	3	5	4	6	7	8	10	9
low	1	2	2	2	2	6	6	6	10	9

dfn and low values for the spanning tree

Determining Biconnected Components (Cont.)

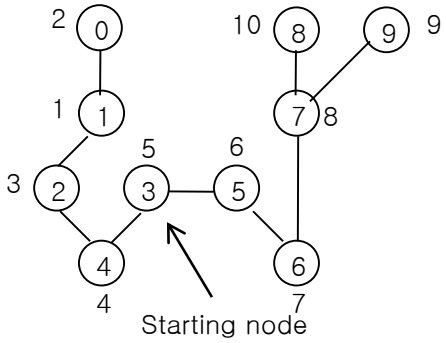


low(w) ≥ dfn(u)

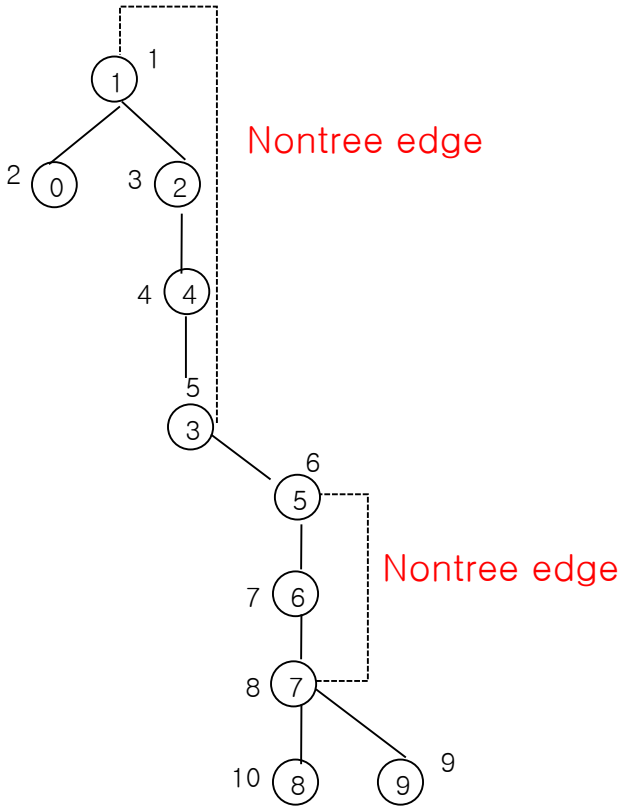
Vertex	0	1	2	3	4	5	6	7	8	9
dfn	1	2	3	5	4	6	7	8	10	9
low	1	2	2	2	2	6	6	6	10	9

dfn and low values for the spanning tree

A Depth-first Spanning Tree with Root 1



(a)



(b)

A Depth-first Spanning Tree with Root 3

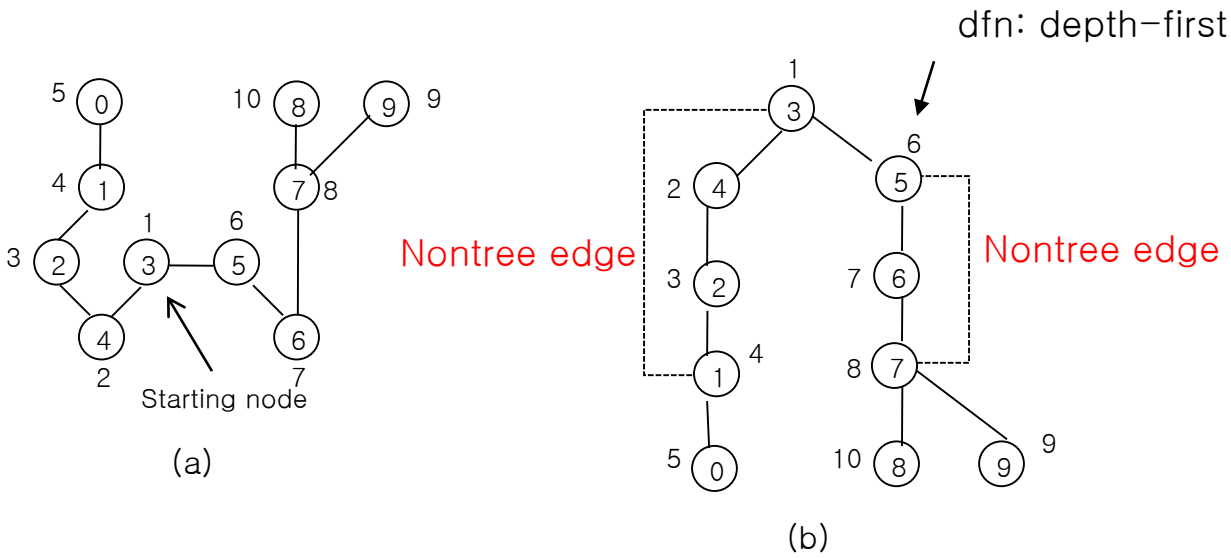
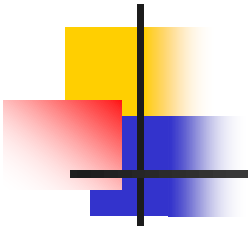
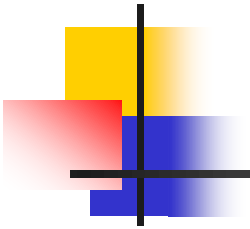


Figure 6.21: Depth-first spanning tree of Figure 6.20(a)



A Depth-first Spanning Tree

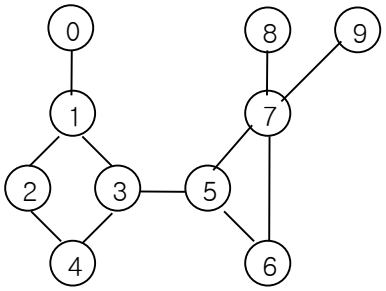
- A non-tree edge (u,v) is a back edge with respect to a spanning tree T iff either u is an ancestor of v or v is an ancestor of u.
- A nontree edge that is not a back edge is called a cross edge.
- The root node of the depth-first spanning tree is an articulation point iff it has at least two children.
- Any other vertex u is an articulation point iff it has at least one child w such that it is not possible to reach an ancestor of u using a path composed of w, descendants of w and a single back edge.



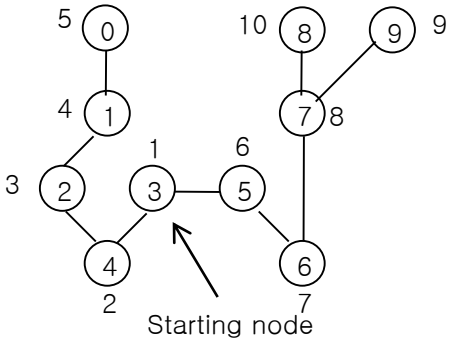
Determining Biconnected Components

- Depth-first Number
  - The sequence in which the vertices are visited during the DFS
- Back edge (u, v)
  - Nontree edge
  - Either u is an ancestor of v or v is an ancestor of u
- low(w) – the lowest depth-first number that can be reached from w using a path of descendants followed by at most one back edge
  - **min{** dfn(w), min{low(x)| x is a child of w},
    min{dfn(x)|(w, x) is a back edge} **}**
- Articulation point (2 cases)
  - vertex u is an articulation point iff
    1. If u is the root of the spanning tree and has two or more children
    2. If u has a child w such that low(w) ≥ dfn(u)

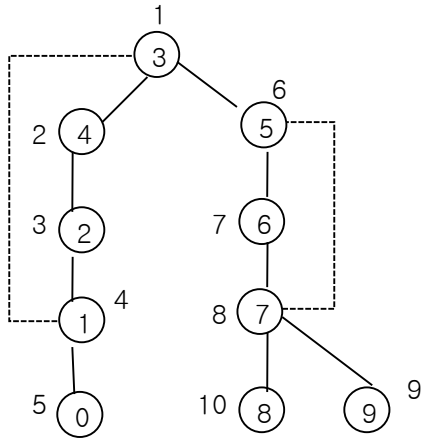
Determining Biconnected Components (Cont.)



Original Graph



(a)



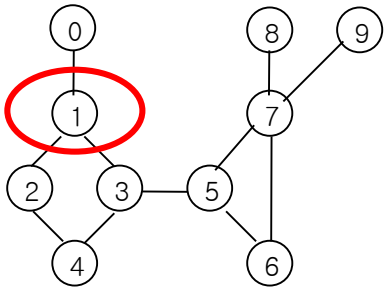
(b)

Depth-first spanning tree

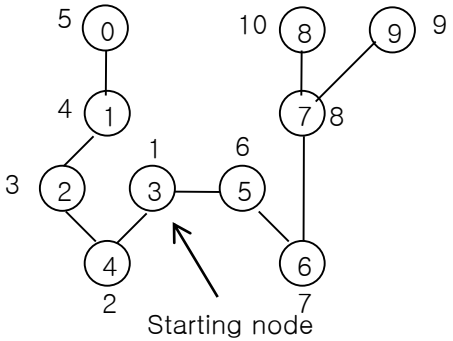
Vertex	0	1	2	3	4	5	6	7	8	9
dfn	5	4	3	1	2	6	7	8	10	9
low	5	1	1	1	1	6	6	6	10	9

Figure 6.22: dfn and low values for the spanning tree of Figure 6.21(b)

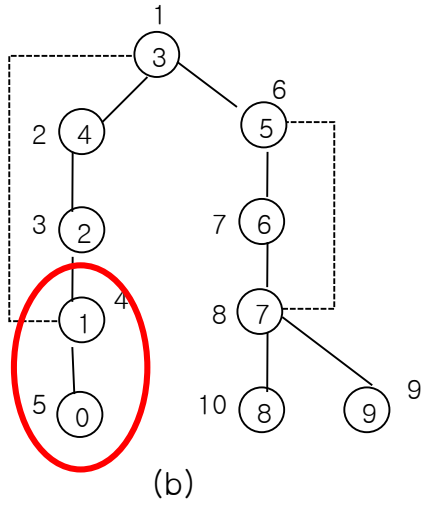
Determining Biconnected Components (Cont.)



Original Graph



(a)



(b)

Depth-first spanning tree

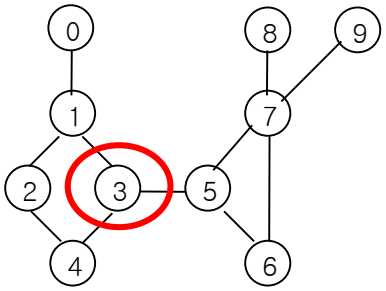
low(w) ≥ dfn(u)

Vertex	0	1	2	3	4	5	6	7	8	9
dfn	5	4	3	1	2	6	7	8	10	9
low	5	1	1	1	1	6	6	6	10	9

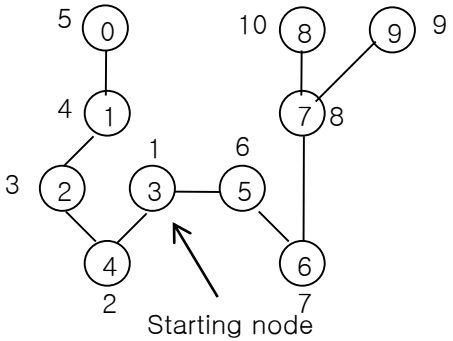
Figure 6.22: dfn and low values for the spanning tree of Figure 6.21(b)

Determining Biconnected Components (Cont.)

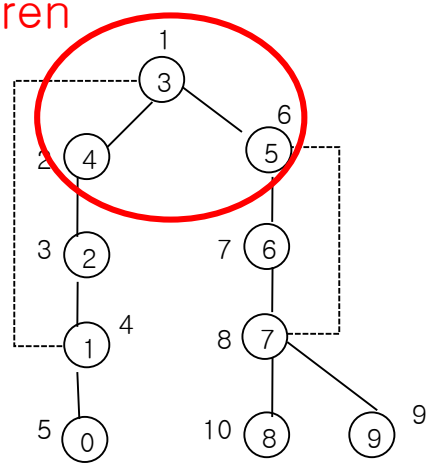
root of the spanning tree and has two or more children



Original Graph



(a)



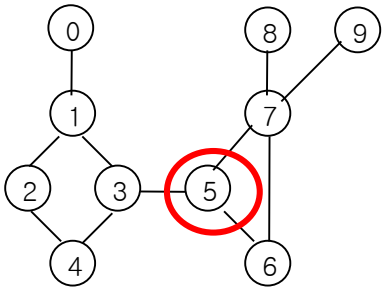
(b)

Depth-first spanning tree

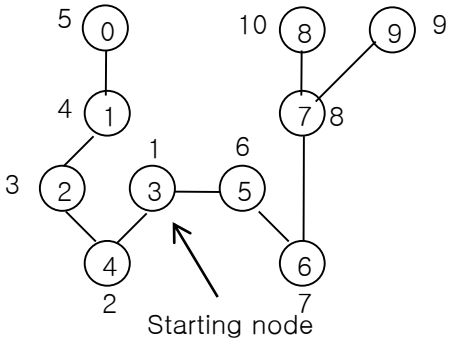
Vertex	0	1	2	3	4	5	6	7	8	9
dfn	5	4	3	1	2	6	7	8	10	9
low	5	1	1	1	1	6	6	6	10	9

Figure 6.22: dfn and low values for the spanning tree of Figure 6.21(b)

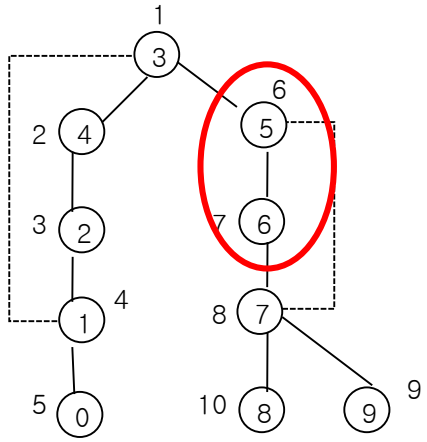
Determining Biconnected Components (Cont.)



Original Graph



(a)



(b)

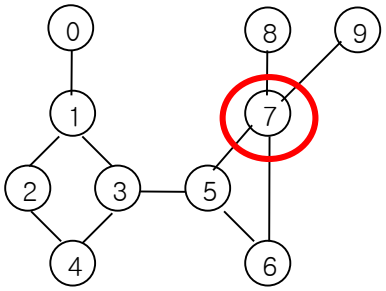
Depth-first spanning tree

low(w) ≥ dfn(u)

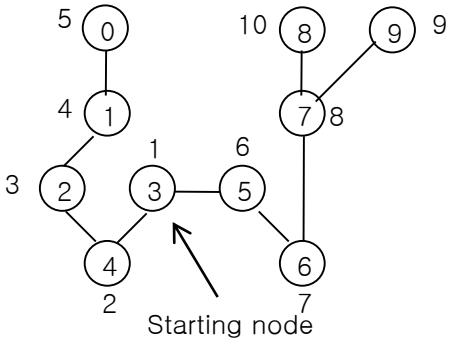
Vertex	0	1	2	3	4	5	6	7	8	9
dfn	5	4	3	1	2	6	7	8	10	9
low	5	1	1	1	1	6	6	6	10	9

Figure 6.22: dfn and low values for the spanning tree of Figure 6.21(b)

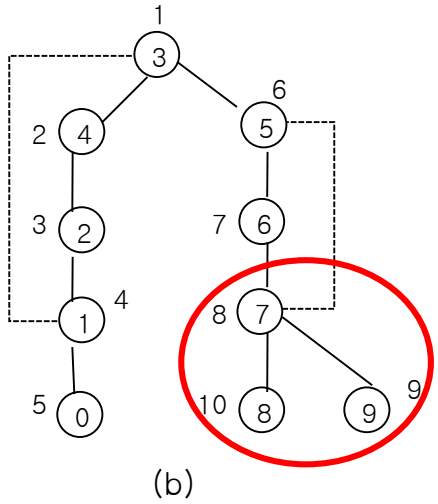
Determining Biconnected Components (Cont.)



Original Graph



(a)



(b)

Depth-first spanning tree

low(w) ≥ dfn(u)

Vertex	0	1	2	3	4	5	6	7	8	9
dfn	5	4	3	1	2	6	7	8	10	9
low	5	1	1	1	1	6	6	6	10	9

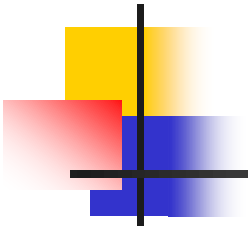
Figure 6.22: dfn and low values for the spanning tree of Figure 6.21(b)

Determining Biconnected Components (Cont.)

```
1.  virtual void Graph::DfnLow(const int x) // begin DFS at vertex x
2.  {
3.        num = 1;              // num is an int data member of Graph
4.        dfs = new int[n];     // dfn is declared as int* in Graph
5.        low = new int[n];     // low is declared as int* in Graph
6.        fill(dfn, dfn + n, 0);
7.        fill(low, low + n, 0);
8.        DfnLow(x, -1); // start at vertex x
9.        delete [] dfn;
10.       delete [] low;
11. }

12. void Graph::DfnLow(const int u, const int v)
13. { // Compute dfn and low while performing a depth first search beginning at vertex u.
14.   // v is the parent (if any) of u in the resulting spanning tree.
15.       dfn[u] = low[u] = num++;
16.       for(each vertex w adjacent from u) // actual code uses an iterator
17.             if(dfn[w] == 0) { // w is an unvisited vertex
18.                   DfnLow(w, u);
19.                   low[u] = min(low[u], low[w]);
20.             }
21.             else if(w != v) low[u] = min(low[u], dfn[w]); // back edge
22. }
```

Program 6.4: Computing dfn and low

Printing Biconnected Components

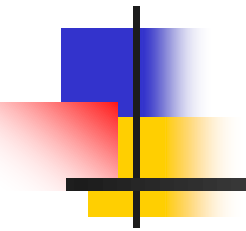


- Biconnected components in a graph can be determined by using the previous algorithm with a slight modification.
- That modification is to maintain a stack of edges.
- Keep adding edges to the stack in the order they are visited
- When an articulation point is detected
  - i.e., say a vertex u has a child v such that no vertex in the subtree rooted at v has a back edge (low[v]≥dfn[u])
  - Pop and print all the edges in the stack till the (u,v) is found, as all those edges including the edge (u,v) will form one biconnected component.

Printing Biconnected Components (Cont.)

```
1.  virtual void Graph::Biconnected()
2.  {
3.        num = 1;                              // num is an int datamember of Graph
4.        dfn = new int[n];     // dfn is declared as int* in Graph
5.        low = new int[n];     // low is declared as int* in Graph
6.        fill(dfn, dfn + n, 0);
7.        fill(low, low + n, 0);
8.        Biconnected(0,-1); // start at vertex 0
9.        delete [] dfn;
10.       delete [] low;
11. }
12. virtual void Graph::Biconnected(const int u, const int v)
13. { // Compute dfn and low, and output the edges of G by their biconnected components.
14.   // v is the parent (if any) of u in the resulting spanning tree.
15.   // s is an initially empty stack declared as a data member of Graph.
16.       dfn[u] = low[u] = num++;
17.       for(each vertex w adjacent from u) { // actual code uses an iterator
18.             if((v != w)&&(dfn[w]<dfn[u])) add (u,w) to stack s;
19.             if(dfn[w] == 0) { // w is an unvisited vertex
20.                   Biconnected(w, u);
21.                   low[u] = min(low[u], low[w]);
22.                   if(low[w] >= dfn[u]) {
23.                         cout << "New Biconnected Component: " << end;
24.                         do {
25.                               delete an edge from the stack s;
26.                               let this edge be (x, y);
27.                               cout <<  x << "," << y << endl;
28.                         } while( (x,y) and (u,w) are not the same edge)
29.                   }
30.             }
31.             else if (w != v) low[u] = min(low[u], dfn[w]); // back edge
32. }
```

Program 6.5: Outputting biconected components when n > 1



Minimum Spanning Trees

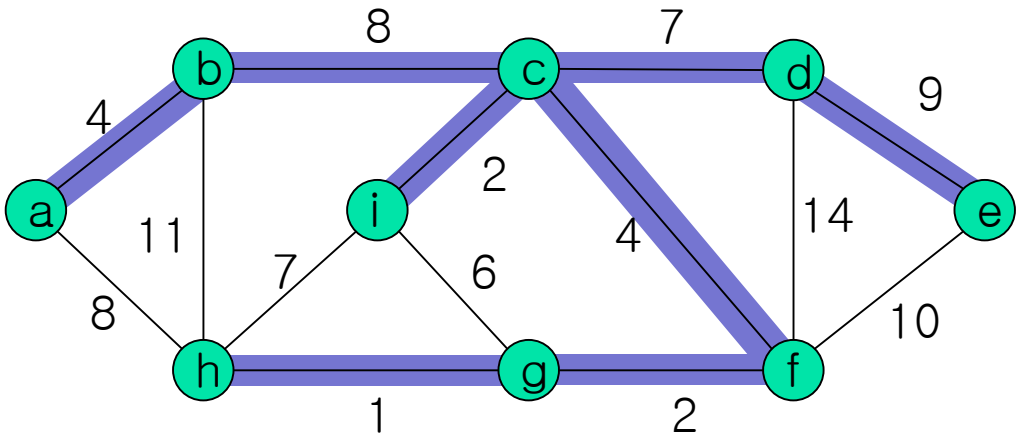
Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Minimum Spanning Tree

- In the design of electronic circuitry, it is often necessary to make the pins of several components electrically equivalent by wiring together.
- To interconnect a set of n pins, we can use an arrangement of n-1 pins.
- Of all such arrangements, the one using the least amount of wire is the most desirable.

Minimum Spanning Tree

- Given a connected, undirected, weighted graph
- Find a spanning tree using edges that minimizes the total weight



Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Minimum Spanning Tree

- We can model this wiring problem with a connected, undirected graph G=(V,E), where
  - V is the set of pins
  - E is the set of possible interconnections between pair of pins
  - A weight w(u,v) for each edge $(u,v) \in E$ that specifying the cost to connect u and v
- Find an acyclic subset $T \subseteq E$ that connects all of the vertices and whose total weight
  $w(T) = \sum_{(u,v)\in T} w(u,v)$ is minimized.
- Since T is acyclic and connects all the vertices, we call a minimum spanning tree.

Minimum Spanning Tree

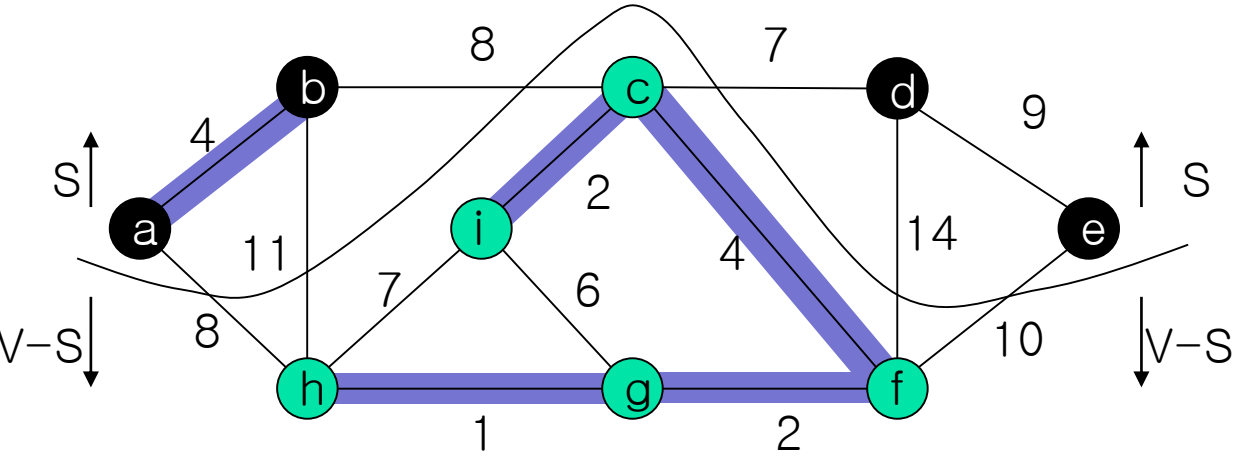
GENERIC-MST(G, w)

1. A = Ø
2. **while** A does not form a spanning tree
3. find an edge (u,v) that is safe for A
4. A = A U {(u,v)}
5. **return** A

Minimum Spanning Tree

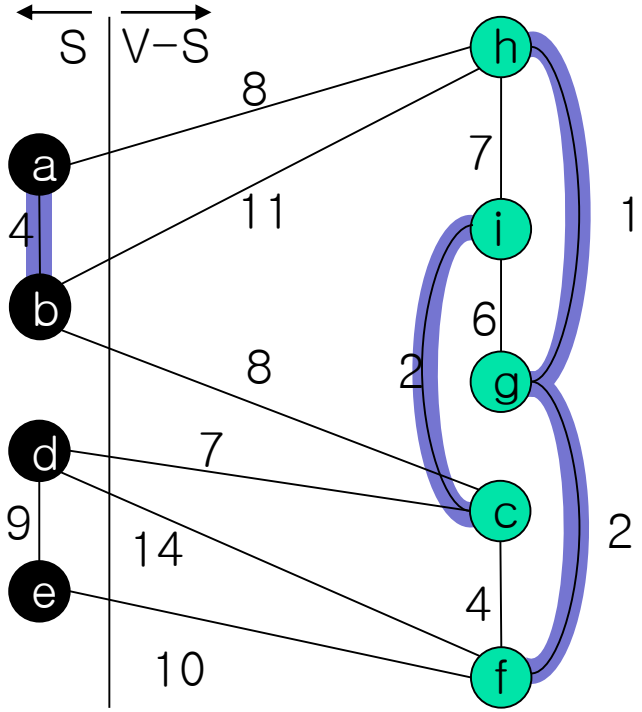
- If A U {(u,v)} is also a subset of a minimum spanning tree, we call the edge (u,v) a safe edge.
- A cut (S, V-S) of an undirected graph G=(V,E) is a partition of V.
- An edge (u,v)$\in$E crosses the cut (S, V-S) if one of its endpoints is in S and the other is in V-S.
- A cut respects a set A of edges if no edge in A crosses the cut.
- An edge is a light edge crossing a cut if its weight is the minimum of any edge crossing the cut.

One Way of Viewing a Cut (S, V-S)

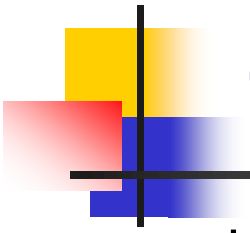


- Black vertices are in the set S, and green vertices are in V-S.
- The edge (d,c) is the unique light edge crossing the cut.

Another Way of Viewing a Cut (S, V-S)



Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009



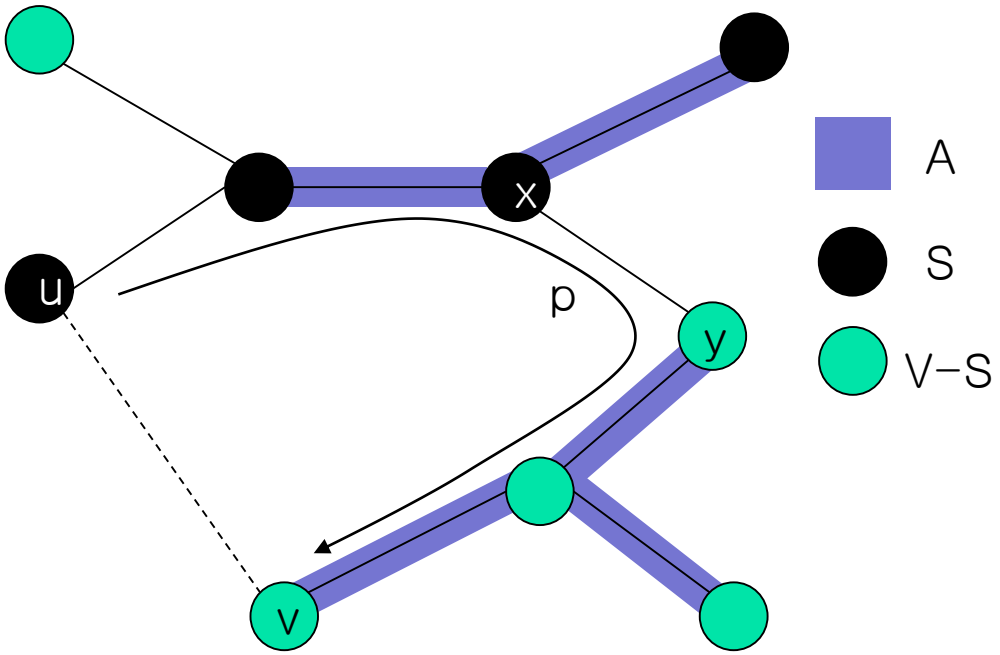
Theorem 23.1

- Let G=(V,E) be a connected undirected graph with a real-valued weight function w defined on E.
- Let A be a subset of E that is included in some minimum spanning tree for G.
- Let (S,V-S) be any cut of G that respects A and let (u,v) be a light edge crossing (S,V-S).
- Then, the edge (u,v) is safe for A

Theorem 23.1 (Proof)

- Let T be a minimum spanning tree that includes A
- Assume that T does not contain the light edge (u,v)
- Construct another minimum spanning tree T' that include A U {(u,v)} by using a cut-and-paste technique, thereby showing that (u,v) is a safe edge for A

Theorem 23.1 (Proof)



- The edge (x, y) is an edge on the unique simple path p from u to v in T and the edges in A are shaded.

Theorem 23.1 (Proof)

- The light edge (u,v) forms a cycle with the edge on the simple path p from u to v in T.
- Since u and v are on opposite sides of the cut (S, V-S), there is at least one edge in T on the simple path p that also crosses the cut. Let (x,y) be any such edge.
- The edge (x, y) is not in A, since the cut respects A.
- Since (x,y) is on the unique path from u to v in T, removing (x,y) breaks T into two components.
- Adding (u,v) re-connects them to form a new spanning tree $T' = T - \{(x,y)\} \cup \{(u,v)\}$.

Theorem 23.1 (Proof)

- We next show that $T' = T - \{(x,y)\} \cup \{(u,v)\}$ is a minimum spanning tree.
- Since (u, u) is a light edge crossing (S, V-S) and (x, y) also crosses this cut, we have $w(u,v) \leq w(x, y)$ resulting that $w(T') = w(T) - w(x,y) + w(u,v) \leq w(T)$.
- But $w(T) \leq w(T')$, since T is a minimum spanning tree.
- Thus, T' must be a minimum spanning tree.
- Because $A \subseteq T'$ and $A \cup \{(u,v)\} \subseteq T'$ where T' is a minimum spanning tree, (u,v) is safe for A.

Understanding of GENERIC-MST

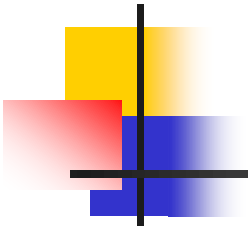
- As the method proceeds, the set A is always acyclic; otherwise, a minimum spanning tree including A would contain a cycle, which is a contradiction.
- At any point in the execution, the graph $G_A=(V, A)$ is a forest, and each of the connected components of G_A is a tree.
- Some of the trees may contain just one vertex, as is the case, for example, when the method begins: A is empty and the forest contains |V| trees, one for each vertex.
- Moreover, any safe edge (u,v) for A connects distinct components of G_A, since A $\cup$ {(u,v)} must be acyclic.

Understanding of GENERIC-MST

- The **while** loop in lines 2 - 4 of GENERIC-MST executes |V| - 1 times because it finds one of the |V|-1 edges of a minimum spanning tree in each iteration.
- Initially, when $A = \emptyset$, there are |V| trees in G_A, and each iteration reduces that number by 1.
- When the forest contains only a single tree, the method terminates.

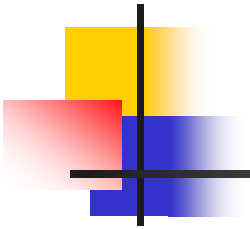
Corollary 23.2

- Let G=(V,E) be a connected, undirected graph with a real-valued weight function w defined on E.
- Let A be a subset of E that is included in some minimum spanning tree for G.
- Let $C=(V_C,E_C)$ be a connected component (tree) in the forest $G_A=(V,A)$.
- If (u,v) is a light edge connecting C to some other component in G_A, then (u,v) is safe for A



Corollary 23.2 (Proof)

- The cut (V_C, V-V_C) respects A and (u,v) is a light edge for this cut.
- Thus, (u,v) is safe for A.



Kruskal's and Prim's Algorithms

- They each use a specific rule to determine a safe edge in line 3 of GENERIC-MST.
- In Kruskal's algorithm,
  - The set A is a forest whose vertices are all those of the given graph.
  - The safe edge added to A is always a least-weight edge in the graph that connects two distinct components.
- In Prim's algorithm,
  - The set A forms a single tree.
  - The safe edge added to A is always a least-weight edge connecting the tree to a vertex not in the tree.

Kruskal's Algorithm

- A greedy algorithm since **at each step it adds to the forest an edge of least possible weight.**
- Find a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge (u,v) of least weight.
- Let two trees C_1 and C_2 are connected by (u,v).
- Since (u,v) must be a light edge connecting C_1 to some other tree, Corollary 23.2 implies that (u,v) is a safe edge for C_1.

Implementation of Kruskal's Algorithm

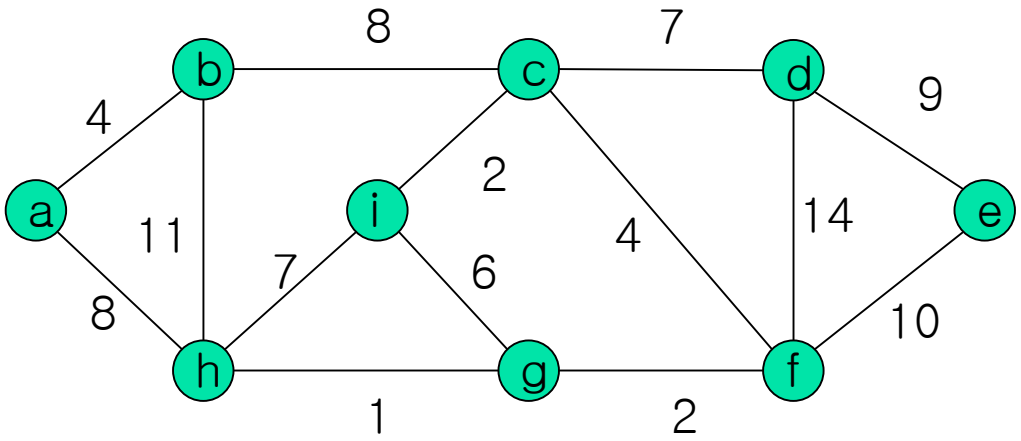
- It uses a disjoint-set data structure to maintain several disjoint sets of elements.
- Each set contains the vertices in one tree of the current forest.
- The operation FIND-SET(u) returns a representative element from the set that contains u.
- Thus, we can determine whether two vertices u and v belong to the same tree by testing whether FIND-SET(u) equals FIND-SET(v).
- To combine trees, Kruskal's algorithm calls the UNION procedure.

Kruskal's Algorithm

MST-KRUSKAL(G,w)

1. A = $\varnothing$
2. **for** each v $\in$ G.V
3. Make-Set(v)
4. sort the edges of G.E into nondecreasing order by weight w
5. **for** each edge (u,v) $\in$ G.E in sorted order
6. **if** Find-Set(u) $\neq$ Find-Set(v)
7. A = A U {{u,v}}
8. Union(u,v)
9. **return** A

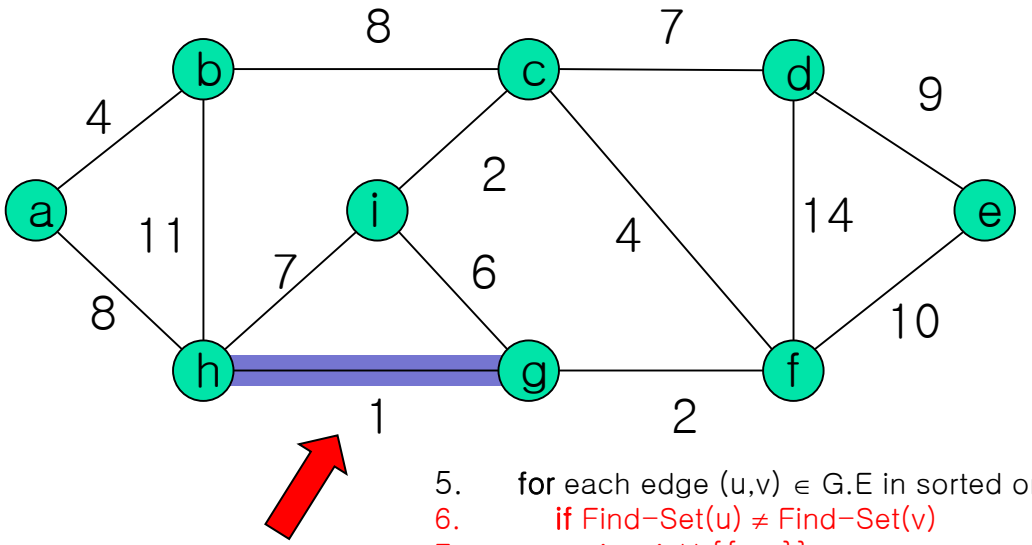
Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

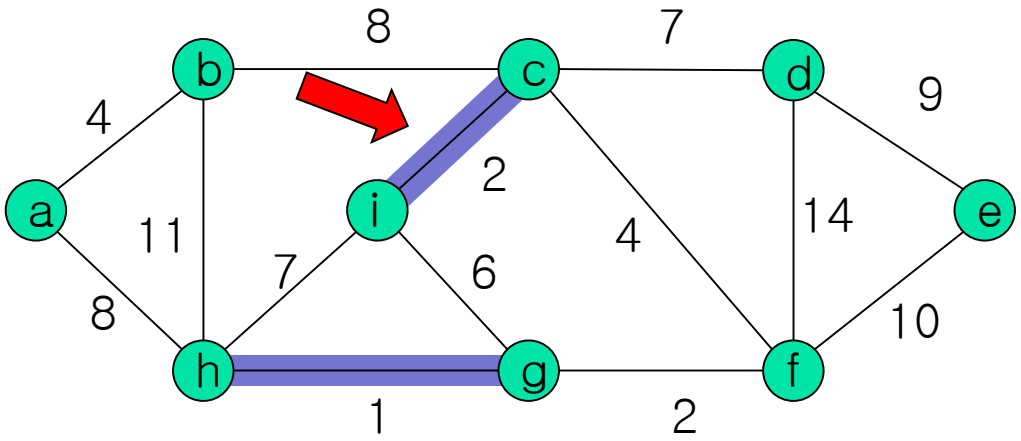
Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

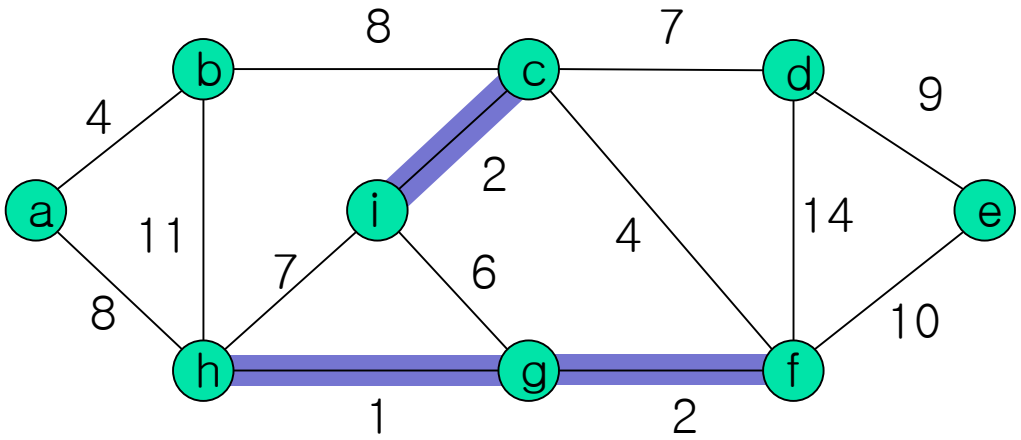
Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

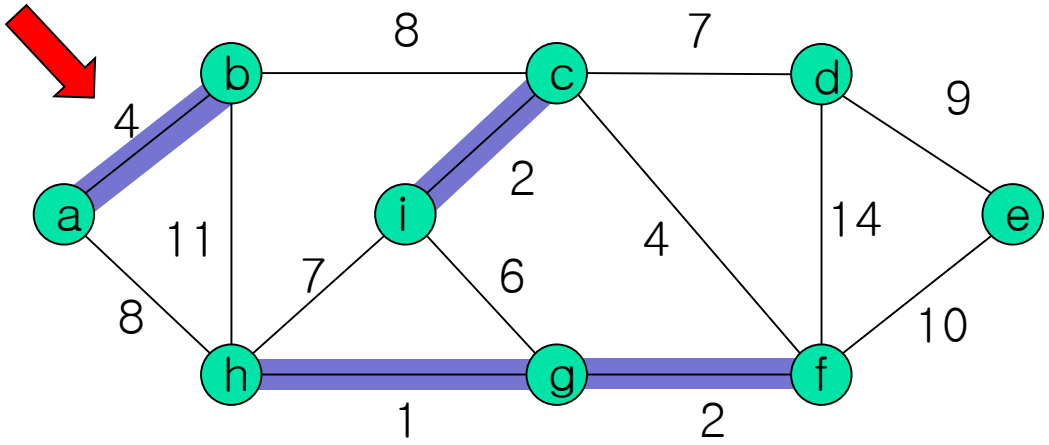
Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

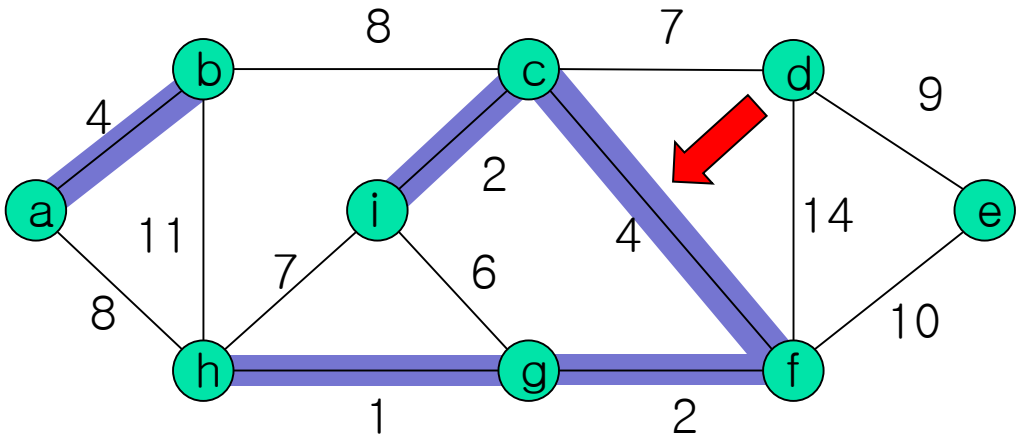
Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

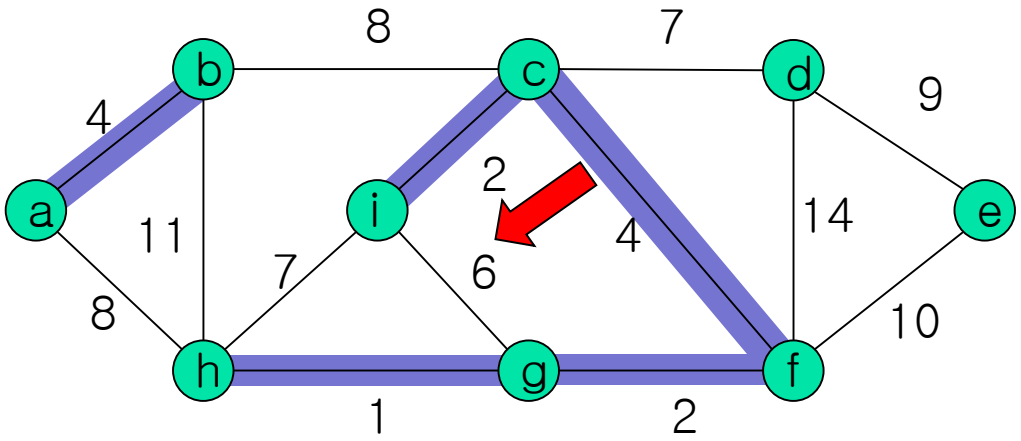
Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

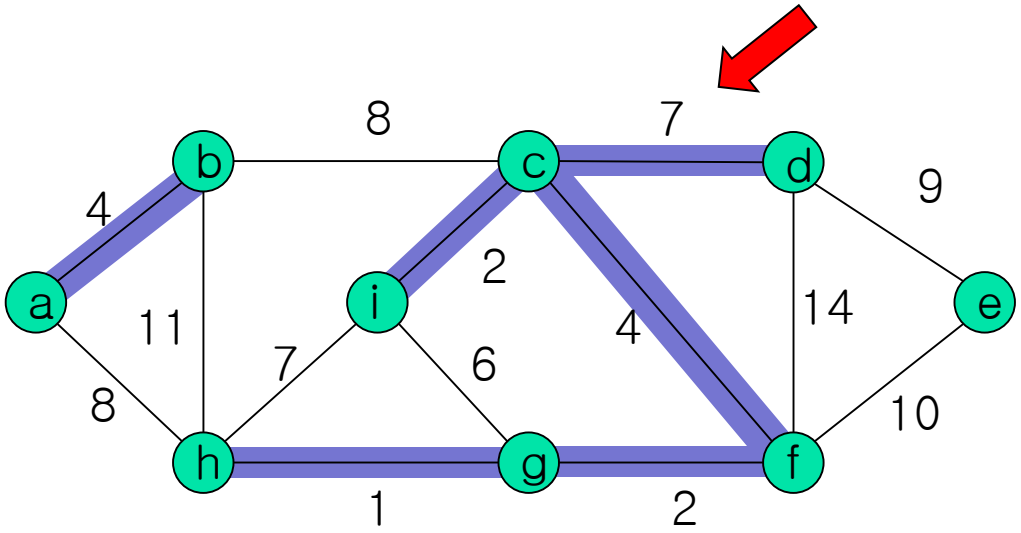
Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

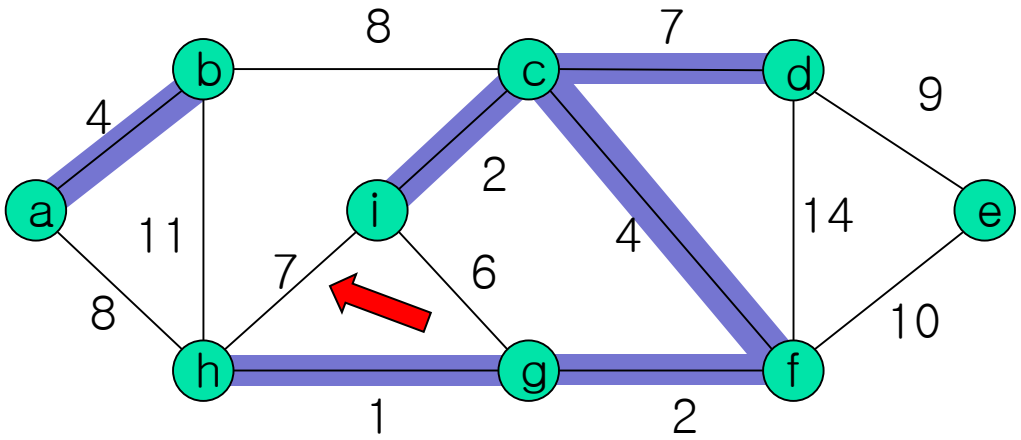
Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

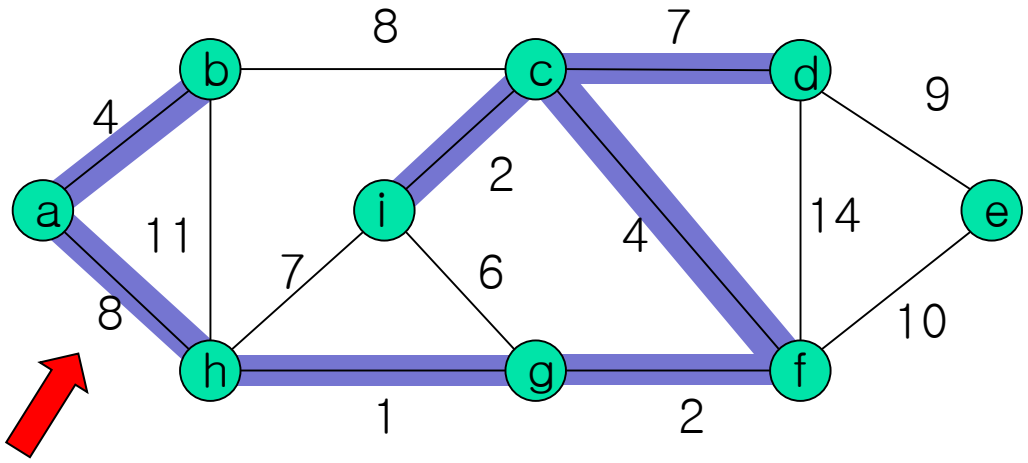
Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

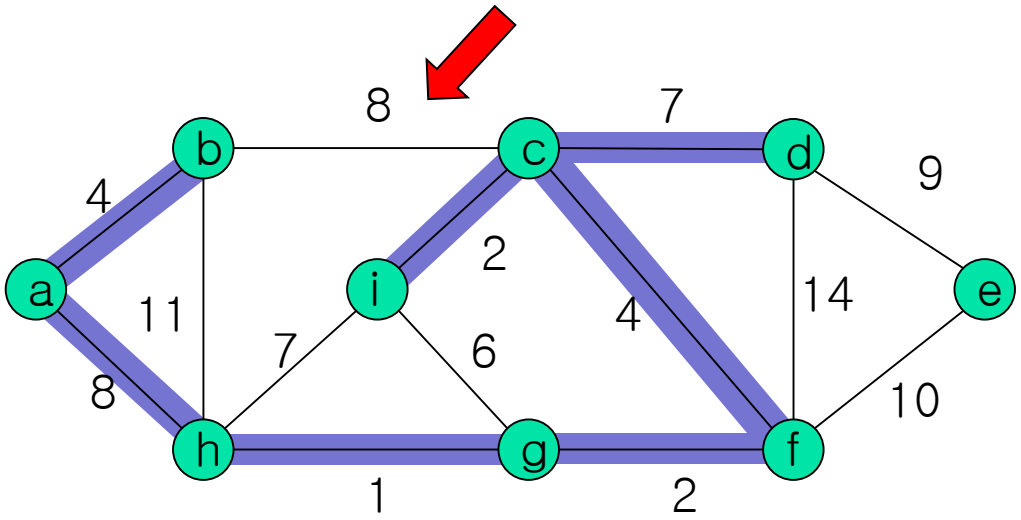
Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

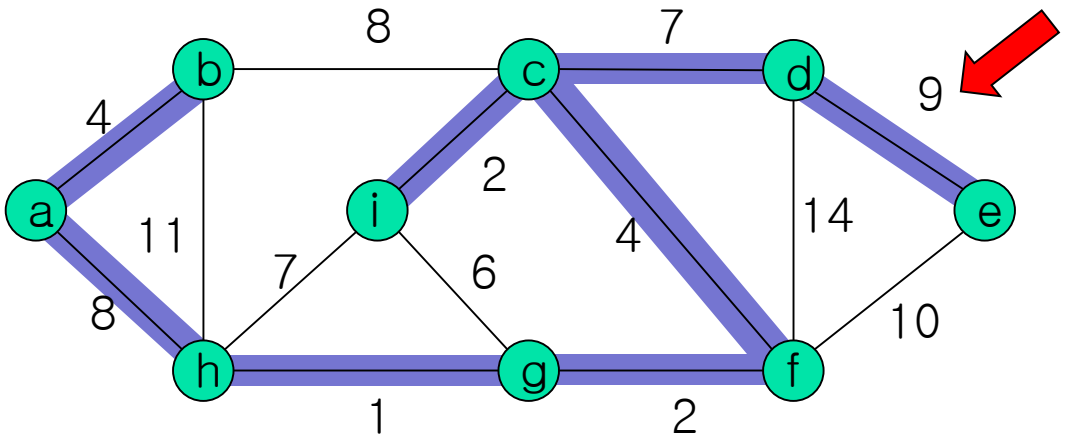
Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

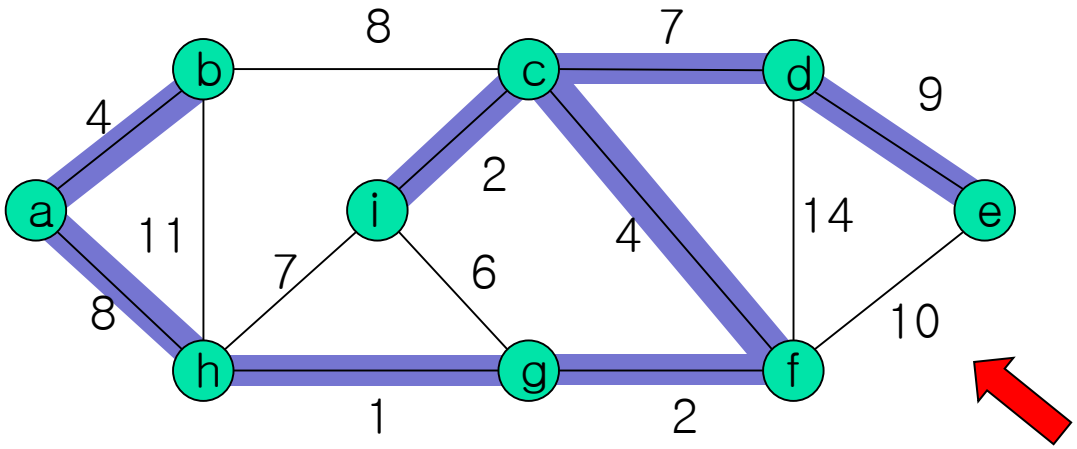
Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

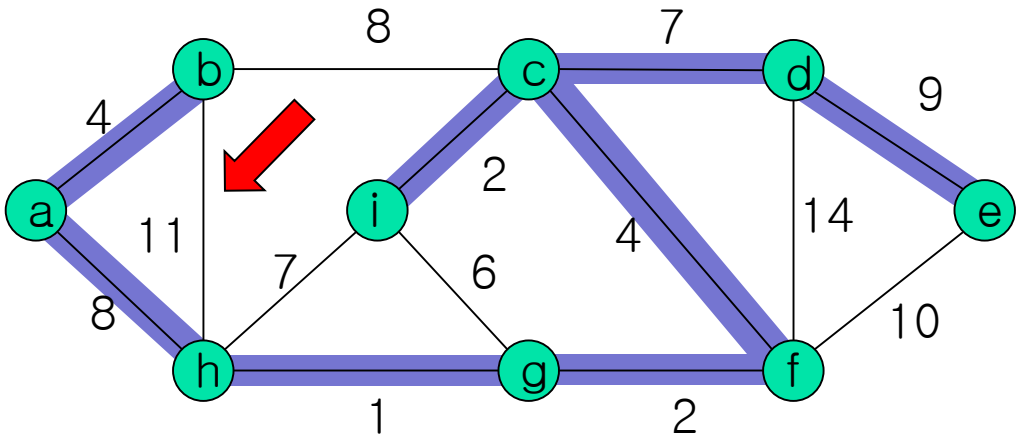
Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

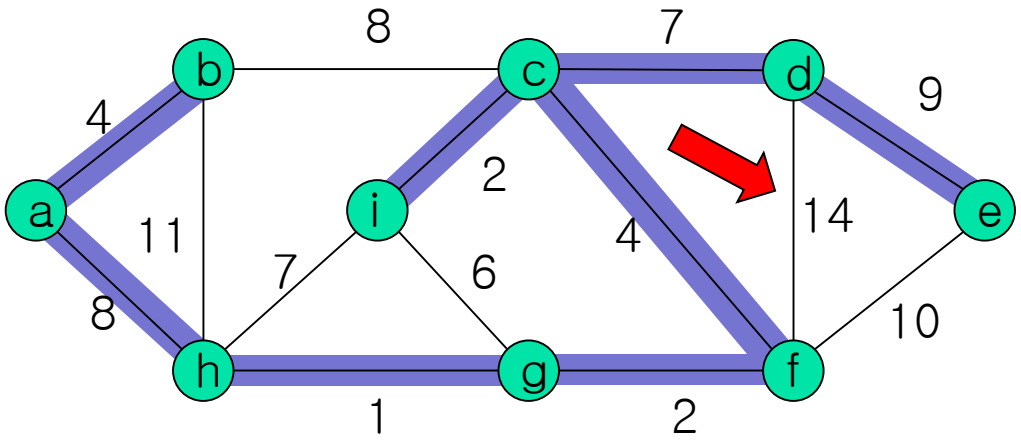
Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

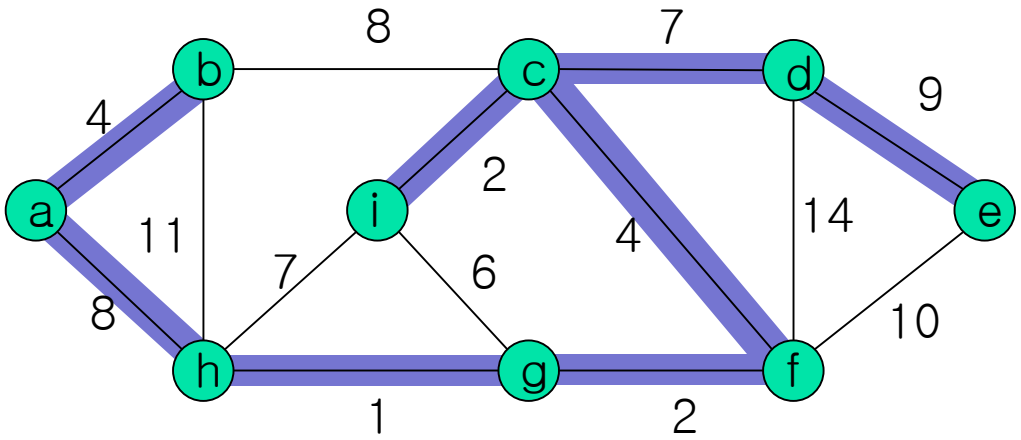
Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Kruskal's Algorithm




```
5.    for each edge (u,v) ∈ G.E in sorted order
6.        if Find-Set(u) ≠ Find-Set(v)
7.            A = A U {{u,v}}
8.            Union(u,v)
9.    return A
```

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Kruskal's Algorithm

MST-KRUSKAL(G,w)

1. $A = \varnothing$
2. **for** each $v \in G.V$
3. Make-Set(v)
4. sort the edges of G.E into nondecreasing order by weight w
5. **for** each edge $(u,v) \in G.E$ in sorted order
6. **if** Find-Set(u) $\neq$ Find-Set(v)
7. $A = A \cup \{\{u,v\}\}$
8. Union(u,v)
9. **return** A

O(V) Make-Set() calls

O(E lg E)

O(E) Find-Set() calls

O(V) Union() calls

Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2009

Running Time of Kruskal's Algorithm

- Use the disjoint-set-forest implementation with the union-by-rank and path-compression heuristics (Section 21.3).
- Sorting the edges in line 4 is $O(|E| \lg |E|)$.
- The disjoint-set operations takes $O((|V|+|E|)\,\alpha(|V|))$ time, where α is the very slowly growing function (Section 21.4).
  - The **for** loop (lines 2–3) performs $|V|$ MAKE-SET operations.
  - The **for** loop (lines 5–8) performs $O(|E|)$ FIND-SET and UNION operations.
- Since G is connected, we have $|E| \geq |V|-1$, the disjoint-set operations take $O(|E|\alpha(|V|))$ time.
- Moreover, since $\alpha(|V|) = O(\lg |V|) = O(\lg |E|)$, Kruskal's algorithm takes $O(|E| \lg |E|)$ time.
- Observing that $|E| < |V|^2 \Rightarrow \lg |E| = O(\lg V)$, the total running time of Kruskal's algorithm becomes $O(E \lg V)$.

Running Time of Kruskal's Algorithm

O(|V|) Make-Set() calls
O(|E|) Find-Set() calls
O(|V|) Union() calls

- In a nut shell,
  - Sort edges: O(|E| lg |E|)
  - Disjoint-set operations
    - O(|V|+|E|) operations $\Rightarrow$ O((|V|+|E|) α(|V|)) time
    - |E|$\geq$|V|-1 $\Rightarrow$ O(E| α(|V|)) time
    - Since α(n) can be upper bounded by the height of the tree,
      α(|V|)=O(lg |V|)=O(lg |E|).
- Thus, the total running time of Kruskal's algorithm is is O(|E| lg |E|)
- By observing that |E| < $|V|^2$ $\Rightarrow$ lg |E| = O(lg |V|), it becomes O(|E| lg |V|).