Chap 6. Graph (2)

Contents

6.2 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***
  - similar to a preorder tree traversal
  - uses *stack* or *recursion*

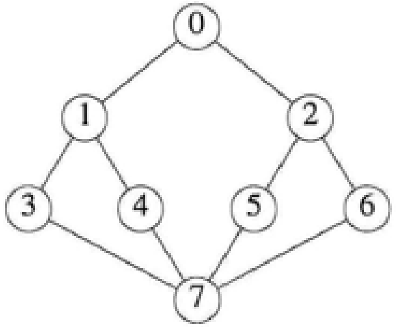
- ***Breadth First Search***
  - similar to a level order tree traversal
  - uses *queue*

- **We shall assumes that**
  - the linked adjacency list for graph is used

6.2.1 Depth First Search

Procedure

```
dfs(v){
  Label vertex v as reached.
  for (each unreached vertex u adjacent from v)
    dfs(u);
}
```

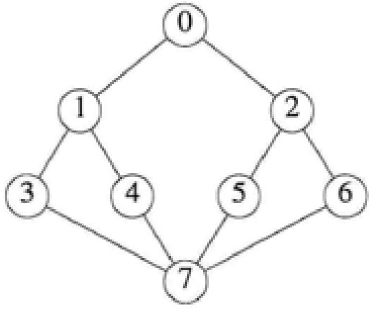


6.2.1 Depth First Search

```
#define FALSE 0
#define TRUE 1
short int visited[MAX_VERTICES];
```

```
void dfs(int v)
{/* depth first search of a graph beginning at v */
   nodePointer w;
   visited[v] = TRUE;
   printf("%5d",v);
   for (w = graph[v]; w; w = w→link)
      if (!visited[w→vertex])
         dfs(w→vertex);
}
```

Program 6.1: Depth first search



6.2.1 Depth First Search

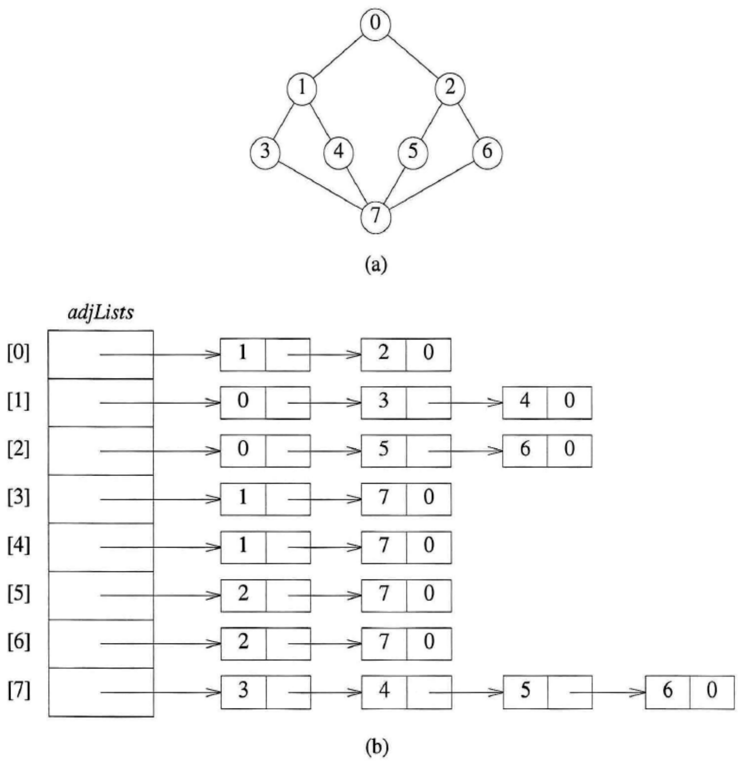
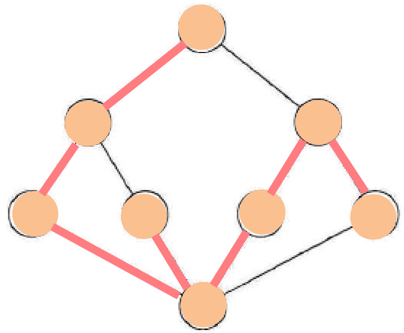


Figure 6.16: Graph *G* and its adjacency lists

dfs(0)



(a)

```
void dfs(int v)
{/* depth first search of a graph beginning at v */
   nodePointer w;
   visited[v] = TRUE;
   printf("%5d",v);
   for (w = graph[v]; w; w = w→link)
      if (!visited[w→vertex])
         dfs(w→vertex);
}
```

output 0 1 3 7 4 5 2 6

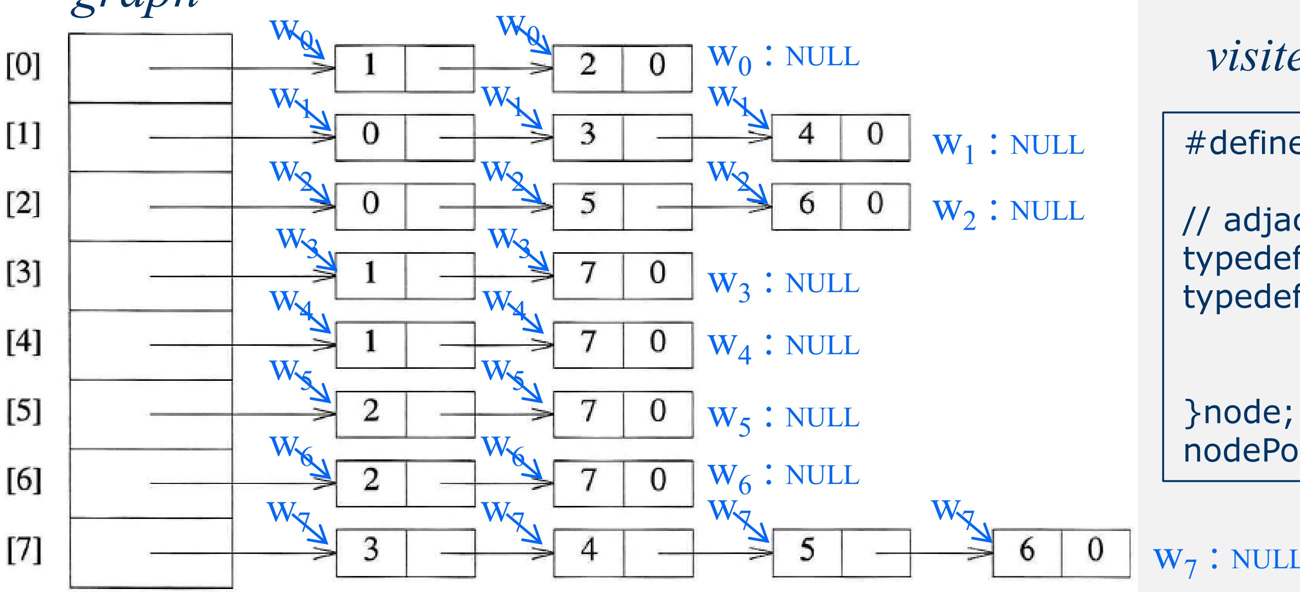
visited

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
T	T	T	T	T	T	T	T

```
#define MAX 100

// adjacency list of an undirected graph
typedef struct node* nodePointer;
typedef struct node{
            int vertex;
            nodePointer link;
}node;
nodePointer graph[MAX] = { NULL };
```

graph



(b)

Figure 6.16: Graph *G* and its adjacency lists

<*system stack*>

※ push/pop of the activation record of dfs()

6.2.1 Depth First Search

❖ **Analysis of *dfs***

- if adjacency list is used
  - search for adjacent vertices : $O(e)$
- if adjacency matrix is used
  - time to determine all adjacent vertices to v : $O(n)$
  - total time : $O(n^2)$

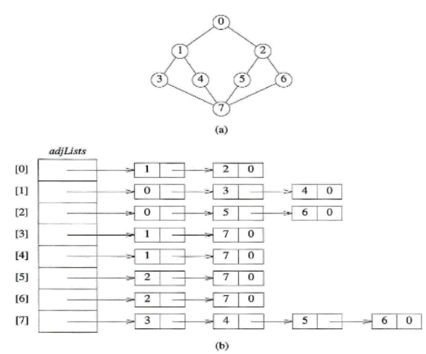


Figure 6.16: Graph *G* and its adjacency lists

0	1	1	0	0	0	0	0
1	0	0	1	1	0	0	0
1	0	0	0	0	1	1	0
0	1	0	0	0	0	0	1
0	1	0	0	0	0	0	1
0	0	1	0	0	0	0	1
0	0	1	0	0	0	0	1
0	0	0	1	1	1	1	0

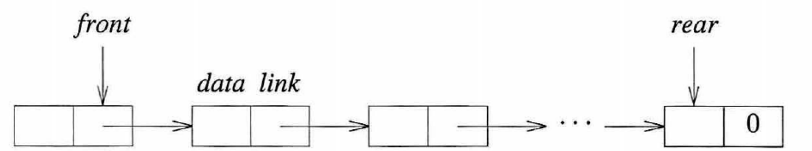
6.2.2 Breadth First Search

- **Procedure**
  - visit start vertex and put into a FIFO *queue*.
  - repeatedly remove a vertex from the queue,
    visit its unvisited adjacent vertices,
    put newly visited vertices into the queue.

6.2.2 Breadth First Search

The queue definition and the function prototypes used by *bfs* are:

```
typedef struct queue *queuePointer;
typedef struct queue {
        int vertex;
        queuePointer link;
        };
queuePointer front, rear;
void addq(int);
int deleteq();
```



(b) Linked queue

6.2.2 Breadth First Search

```
void bfs(int v)
{/* breadth first traversal of a graph, starting at v
    the global array visited is initialized to 0, the queue
    operations are similar to those described in
    Chapter 4, front and rear are global */
  nodePointer w;
  front = rear = NULL; /* initialize queue */
  printf("%5d",v);
  visited[v] = TRUE;  // non-empty queue
  addq(v);
  while (front) {
    v = deleteq();
    for (w = graph[v]; w; w = w→link)
      if (!visited[w→vertex]) {
        printf("%5d", w→vertex);
        addq(w→vertex);
        visited[w→vertex] = TRUE;
      }
    }
}
```

Program 6.2: Breadth first search of a graph

bfs(0)

```
void bfs(int v)
{  nodePointer w;
   front = rear = NULL;
   printf("%5d",v);
   visited[v] = TRUE;
   addq(v);
   while (front) {
      v = deleteq();
      for (w = graph[v]; w; w = w→link)
         if (!visited[w→vertex]) {
            printf("%5d", w→vertex);
            addq(w→vertex);
            visited[w→vertex] = TRUE;
         }
      }
}
```

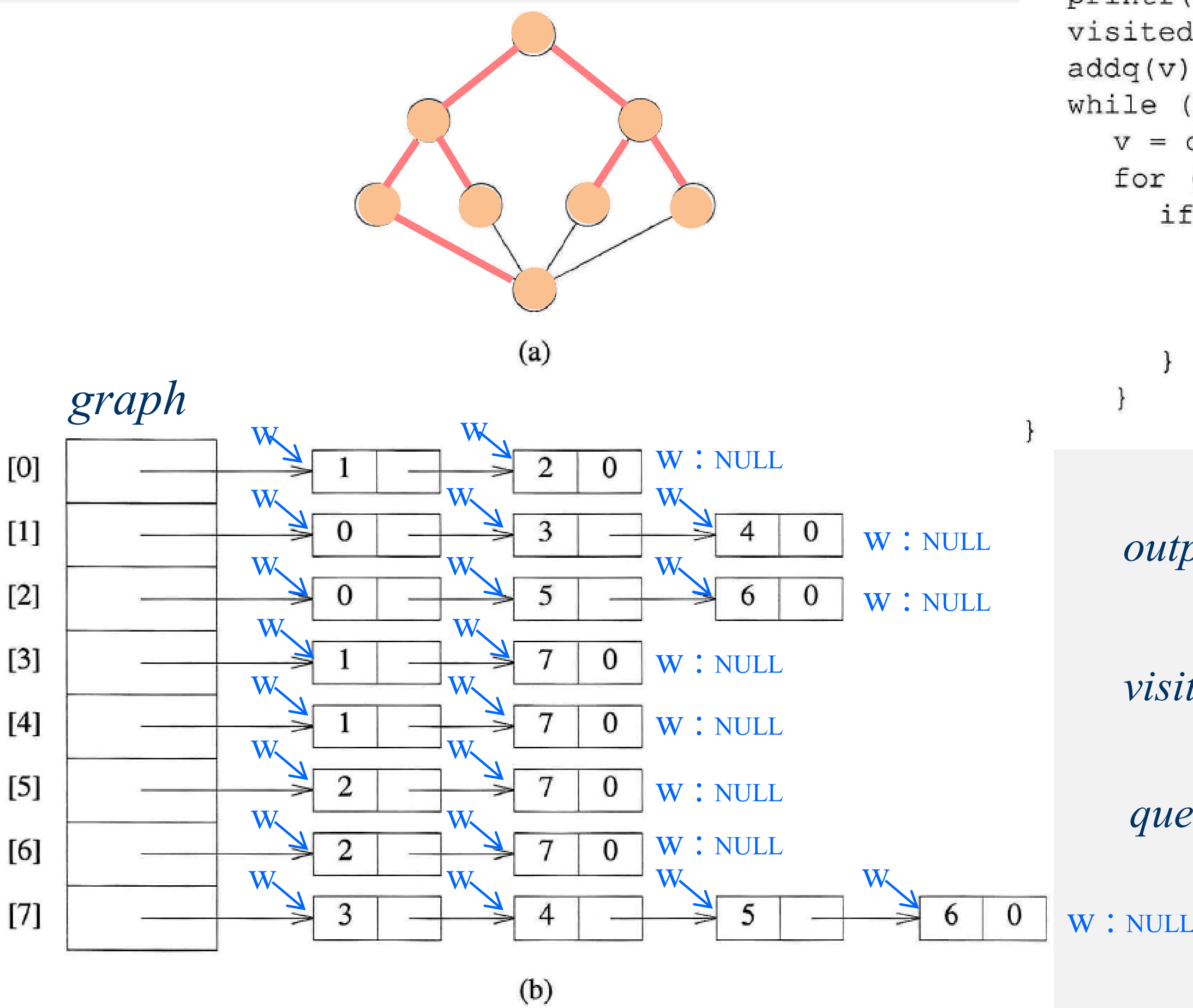


Figure 6.16: Graph *G* and its adjacency lists

output *0 1 2 3 4 5 6 7*

visited

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
T	*T*	*T*	*T*	*T*	*T*	*T*	*T*

queue

6.2.2 Breadth First Search

❖ Analysis of *bfs*

- adjacency list : O(e)
- adjacency matrix : O(n^2)

0	1	1	0	0	0	0	0
1	0	0	1	1	0	0	0
1	0	0	0	0	1	1	0
0	1	0	0	0	0	0	1
0	1	0	0	0	0	0	1
0	0	1	0	0	0	0	1
0	0	1	0	0	0	0	1
0	0	0	1	1	1	1	0

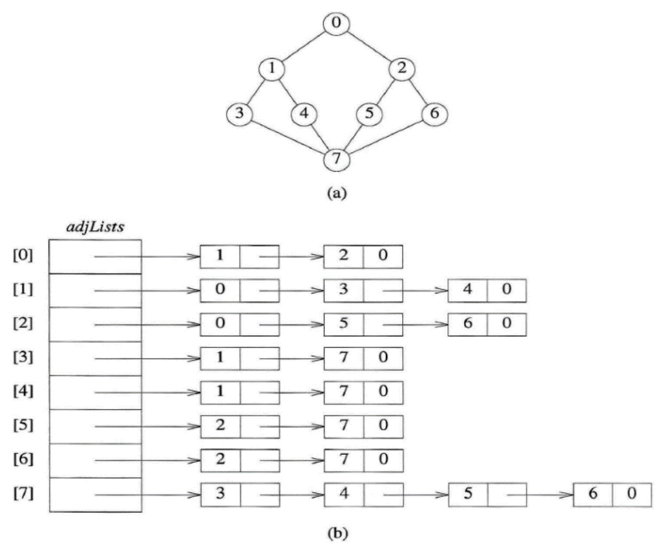


Figure 6.16: Graph G and its adjacency lists

6.2.3 Connected Components

- **determining if an undirected graph is connected**
  - calling *dfs*(0) or *bfs*(0) and then determining if there are any unvisited vertices
    - O(e) for total time taken by dfs
    - O(n+e) for generating all the connected component

- **listing the connected components of a graph**
  - making repeated calls to either *dfs*(v) or *bfs*(v) where v is an unvisited vertex. (Program 6.3)
    - O(n+e) for adjacency list
    - O(n^2) for adjacency matrix

6.2.3 Connected Components

```
void connected(void)
{/* determine the connected components of a graph */
int i;
for (i = 0; i < n; i++)
   if(!visited[i]) {
      dfs(i);
      printf("\n");
   }
}
```

Program 6.3: Connected components

6.2.4 Spanning Trees

❖ ***Spanning tree***

- any tree that consists solely of edges in G and that including all vertices

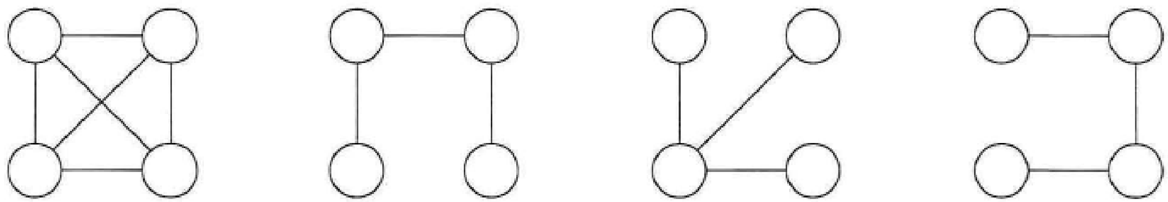
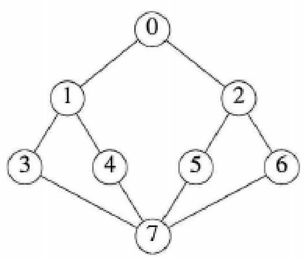
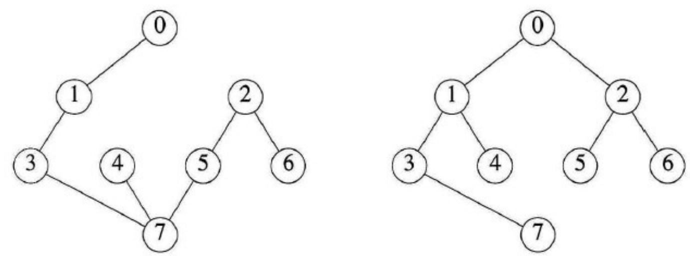


Figure 6.17: A complete graph and three of its spanning trees

6.2.4 Spanning Trees

- We may use *dfs* or *bfs* to create a spanning tree.

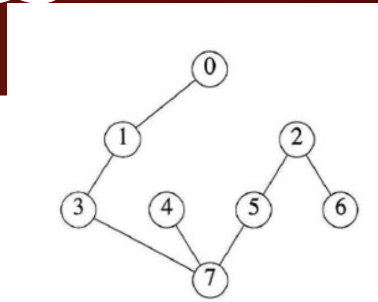




(a) *DFS* (0) spanning tree (b) *BFS* (0) spanning tree

Figure 6.18: Depth-first and breadth-first spanning trees for graph of Figure 6.16

6.2.4 Spanning Trees



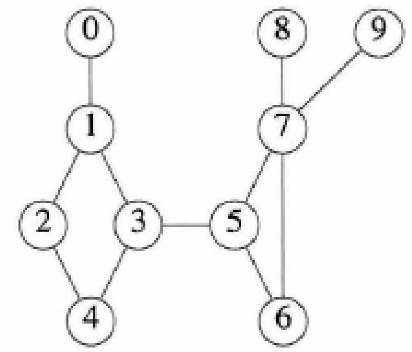
Properties

- If we add a nontree edge, (*v*,*w*), into any spanning tree, T, the result is a cycle that consists of the edge (*v*,*w*) and all the edges on the path from *w* to *v* in T.
- A spanning tree is **a *minimal subgraph* G' of G** such that V(G') = V(G) and G' is connected.
  - A *minimal subgraph* is defined as one with the fewest number of edges
- A spanning tree with *n* vertices has *n*-1 edges.

6.2.5 Biconnected Components

❖ ***articulation point***

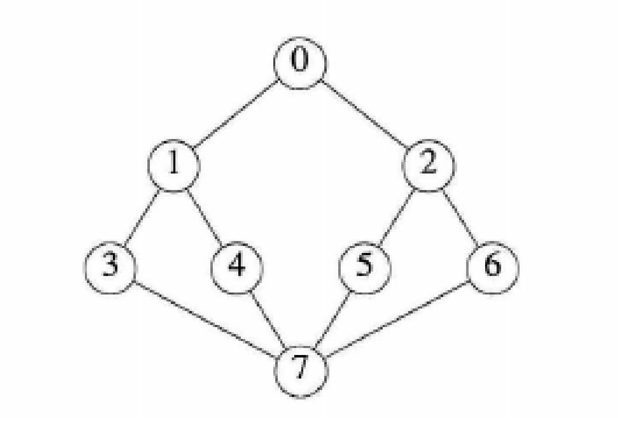
- a vertex *v* of G such that the deletion of *v,* together with all edges incident on *v,* produces a graph, G', that has at least two connected components (*maximal* connected subgraph)
- Figure (a) has four articulation points,
  - vertices **1,** 3, 5, and 7.



(a) Connected graph

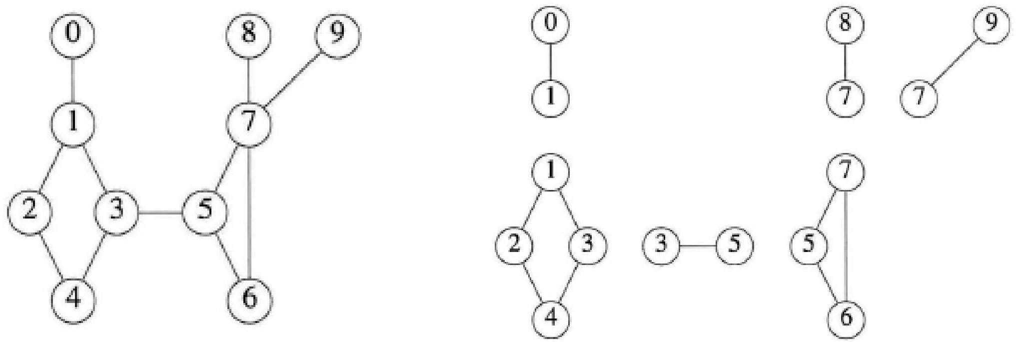
6.2.5 Biconnected Components

- ***biconnected graph***
  - a connected graph that has no articulation points.
- **following figure is a biconnected graph ?**



6.2.5 Biconnected Components

- ***biconnected component***
  - a connected undirected graph is a *maximal biconnected subgraph*
- **the graph of Figure (a) contains the six biconnected components shown in Figure (b).**



(a) Connected graph

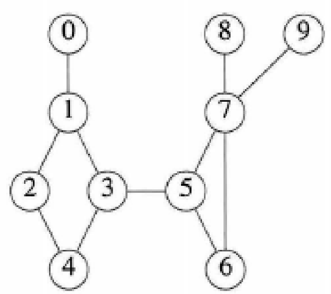
(b) Biconnected components

6.2.5 Biconnected Components

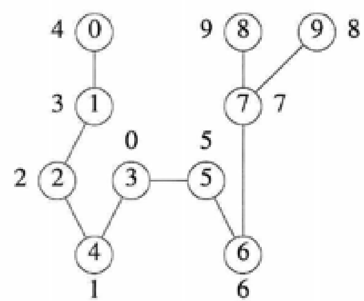
- **find the biconnected components of a conected undirected graph, G,**
  - by using any depth first spanning tree of G
  - *dfs* (3) applied to the graph of Figure 6.19(a)
    - redrawn the tree in Figure 6.20(b) to better reveal its tree structure.
    - The numbers outside the vertices in either figure give the sequence in which the vertices are visited during the depth first search.
    - call this number the *depth first number,* or *dfn,* of the vertex.

6.2.5 Biconnected Components

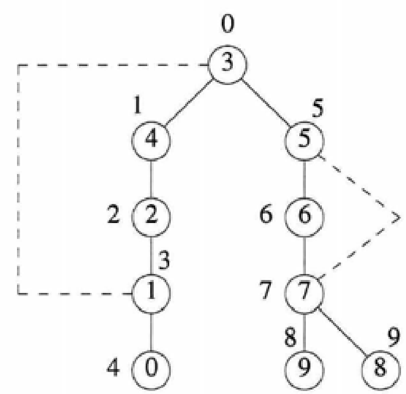
- find the biconnected components of a conected undirected graph, G,



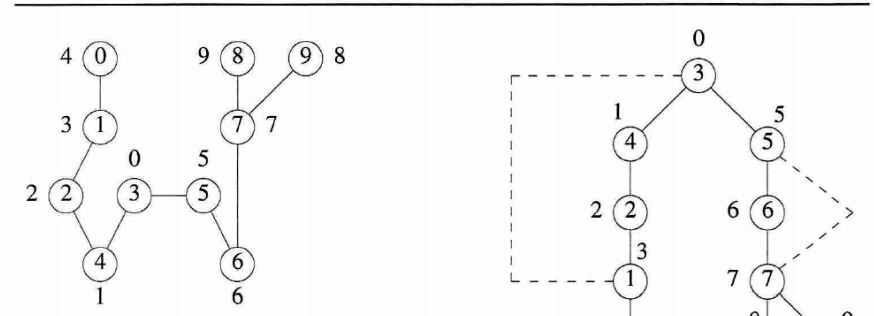
(a) Connected graph



(a) depth first spanning tree



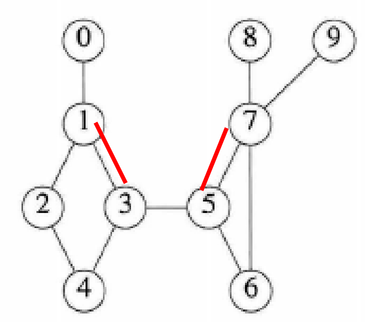
6.2.5 Biconnected Components



(a) depth first spanning tree

(b)

Figure 6.20: Depth first spanning tree of Figure 6.19(a)



(a) Connected graph

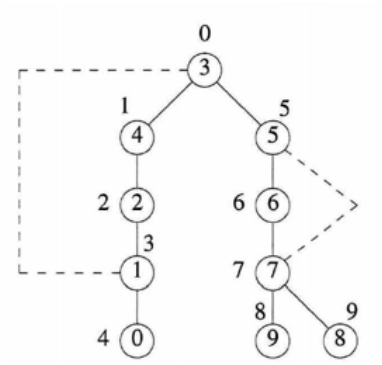
dfn (3) = 0, *dfn* (0) = 4, and *dfn* (9) = 8

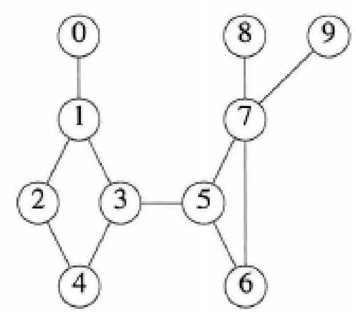
dfn (ancestor) < *dfn* (decedant)

broken lines in Figure 6.20(b) represent nontree edges(*back edge)*

6.2.5 Biconnected Components

- **find articulation point**
  - root of a 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 we cannot reach an ancestor of *u* using a path that consists of only *w*, descendants of w, and a single back edge.



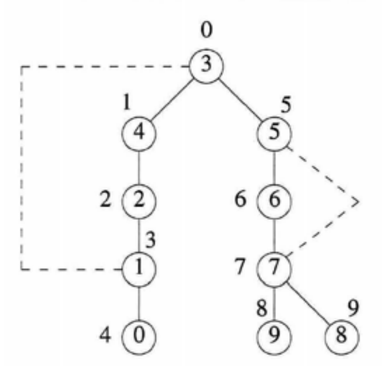


(a) Connected graph

6.2.5 Biconnected Components

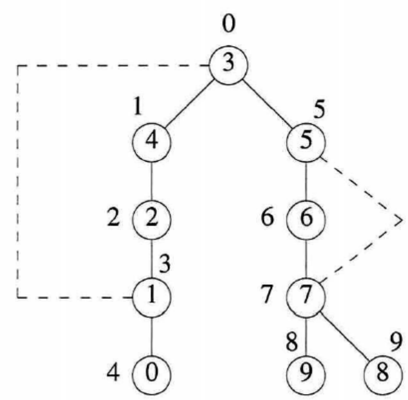
- ***low*** **(u) is the lowest depth first number that we can reach from *u* using a path of descendants followed by at most one back edge**

  low **(u) = *min{dfn*** **(u), *min{low*** **(w) | *w* is a child of *u},* min *{dfn*** **(w) | (u, *w)* is a back edge } }**
  - (u, *w) : an edge to upper direction, a back edge from u to w*
- ***u* is an articulation point**
  - *low* (w) $\geq$ *dfn* (u) ; *w* is a child of *u*
  - https://www.crocus.co.kr/1164



6.2.5 Biconnected Components

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



(b)

an articulation point

Vertex 1의 자식

vertex 3 : root node

vertex 1 : *low* (0) = 4 ≥ *dfn* (1) = 3.

vertex 5 : *low* (6) = 5 ≥ *dfn* (5) = 5

vertex 7 : *low* (8) = 9 ≥ *dfn* (7) = 7

low (9) = 8 ≥ *dfn* (7) = 7

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

vertex 4 : *low* (2) = 0 < *dfn* (4) = 1