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

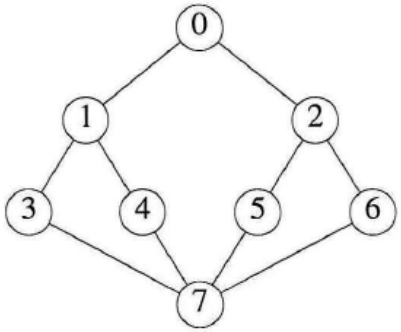
```
dfs(v){
   Label vertex v as reached.
   for (each unreached vertex u adjacent from v)
      dfs(u);
}
```

```
#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);
}
```

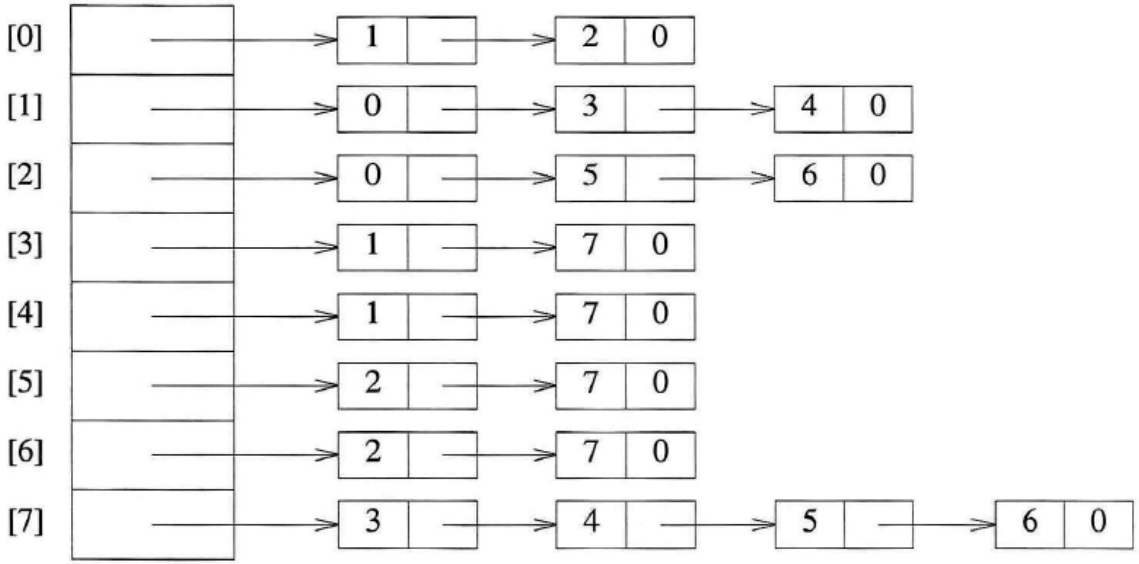
Program 6.1: Depth first search

Example 6.1



(a)

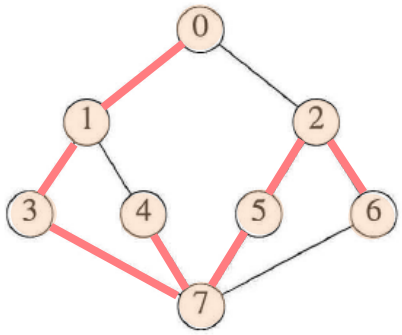
graph



(b)

Figure 6.16: Graph *G* and its adjacency lists

dfs(0)



(a)

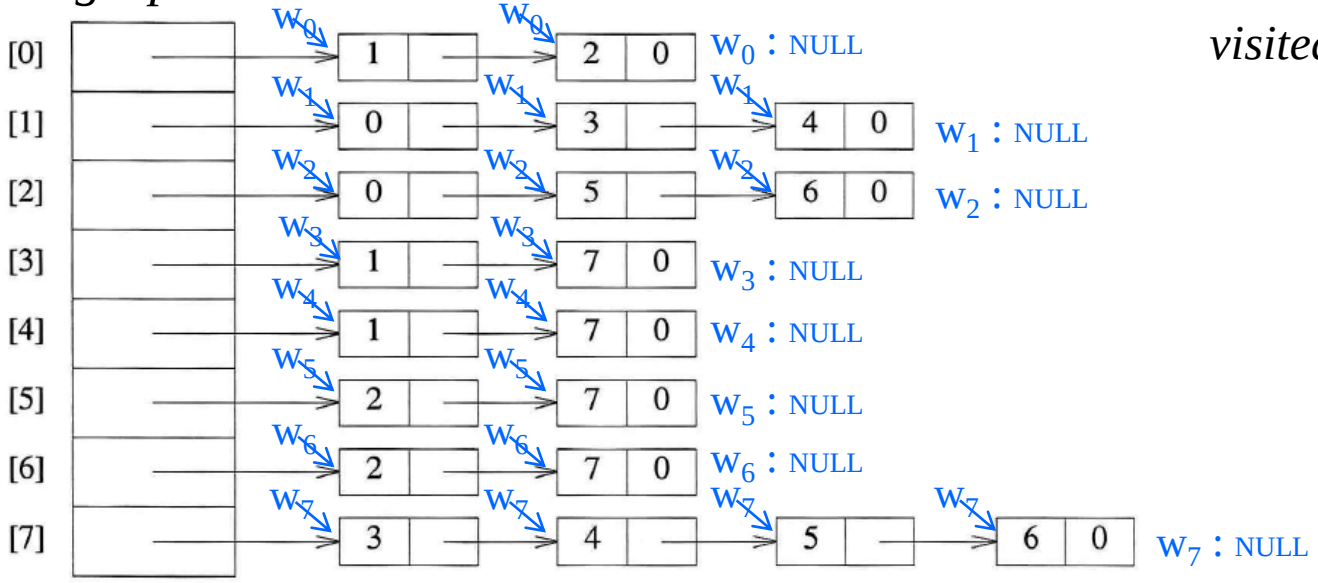
```
void dfs(int v)
{/* depth first search of a graph beginning at v */
   nodePointer w;
   visited[v] = TRUE;
   printf("%5d",v);
   for (w = graph[v]; w; w = w→link)
      if (!visited[w→vertex])
         dfs(w→vertex);
}
```

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

graph



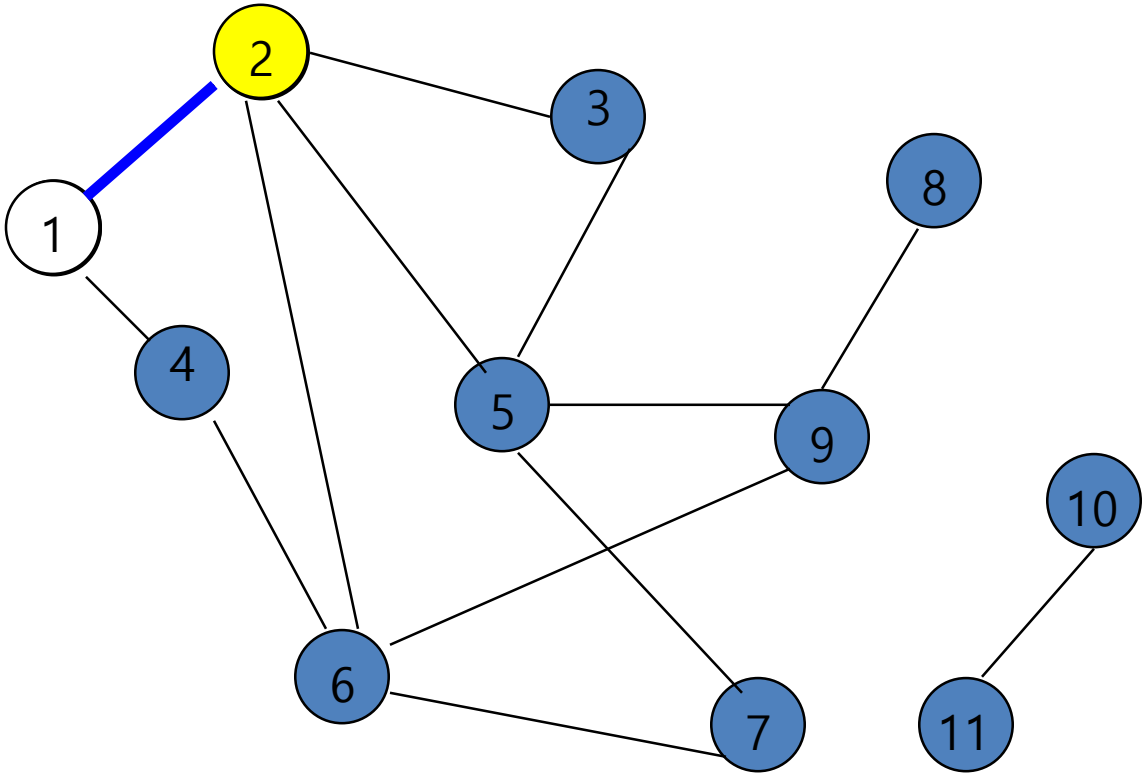
(b)

Figure 6.16: Graph *G* and its adjacency lists

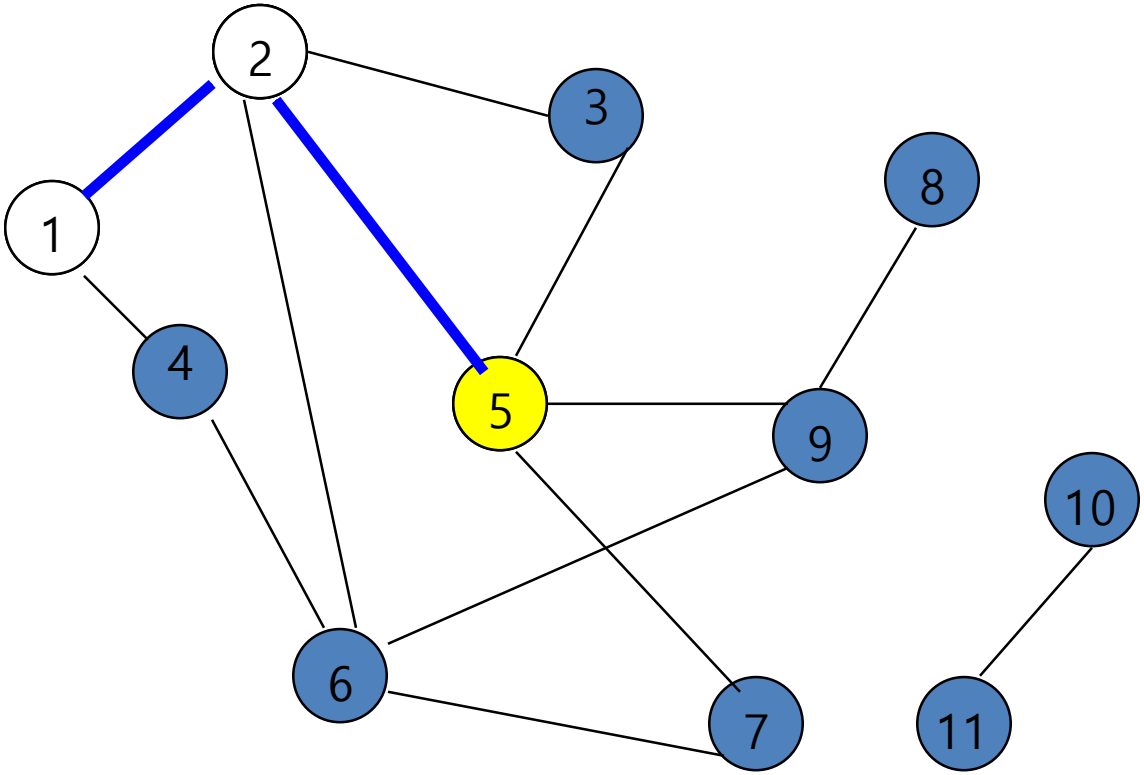
<system stack>

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

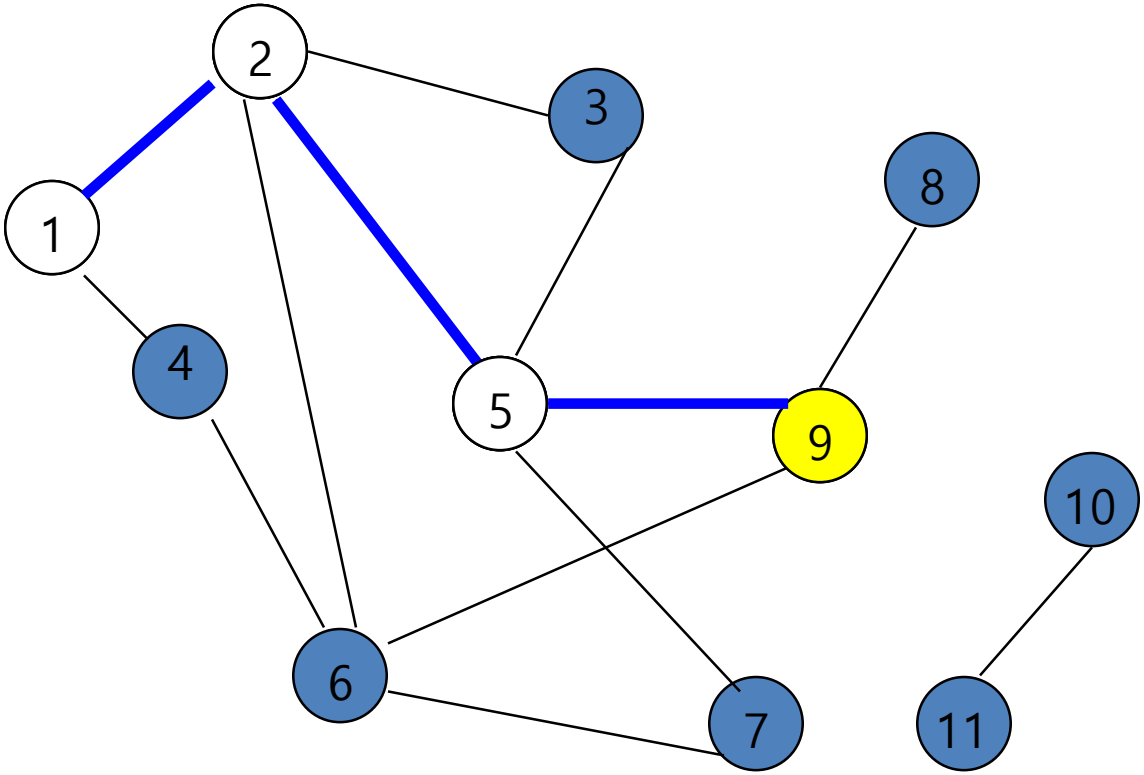
DFS : example



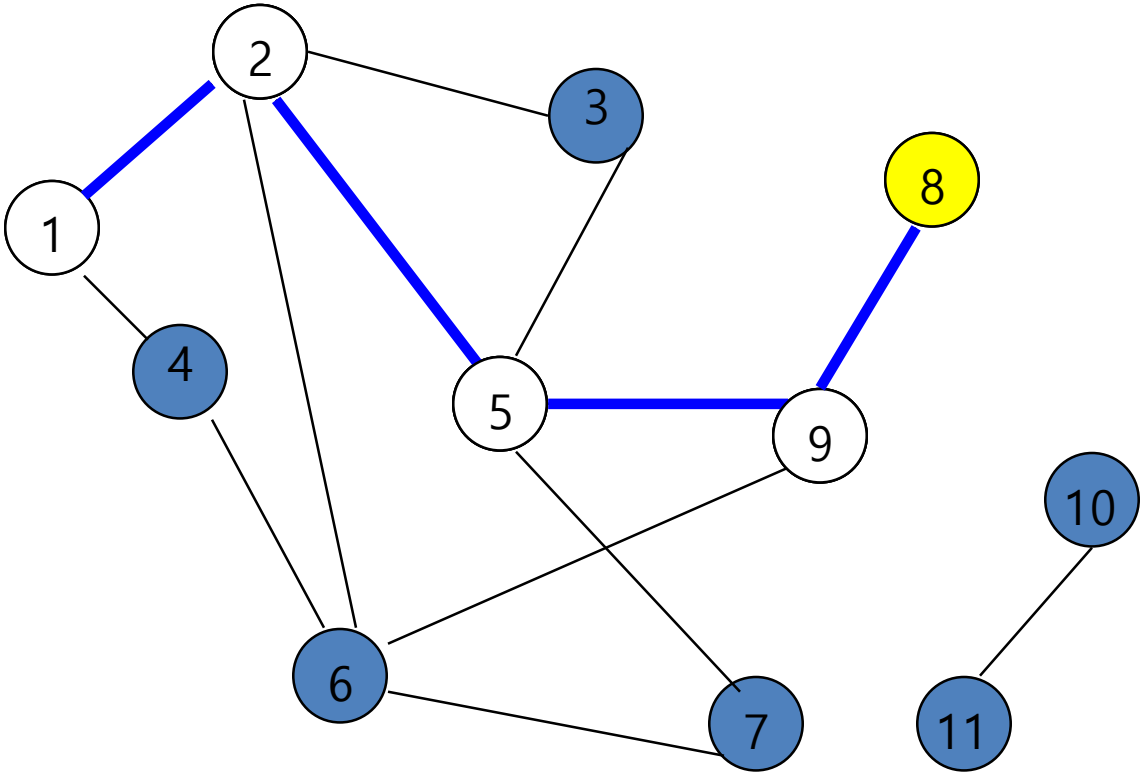
DFS : example



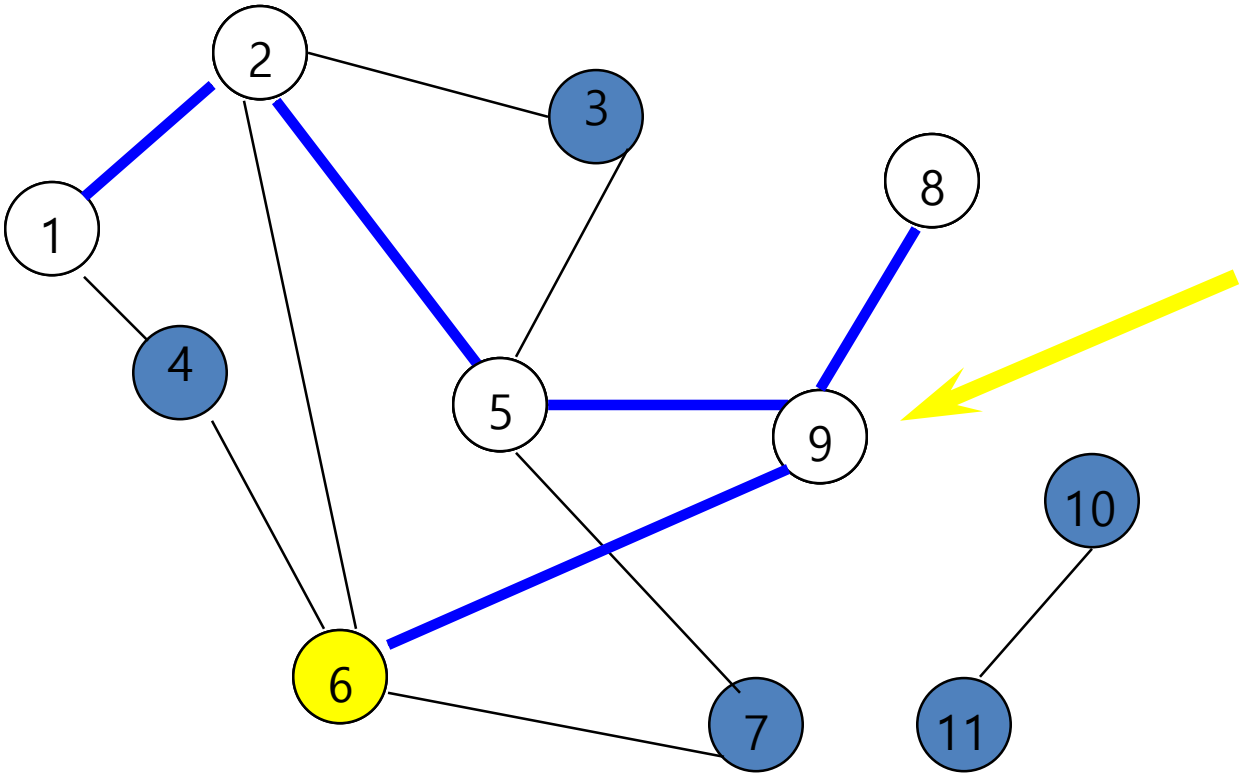
DFS : example



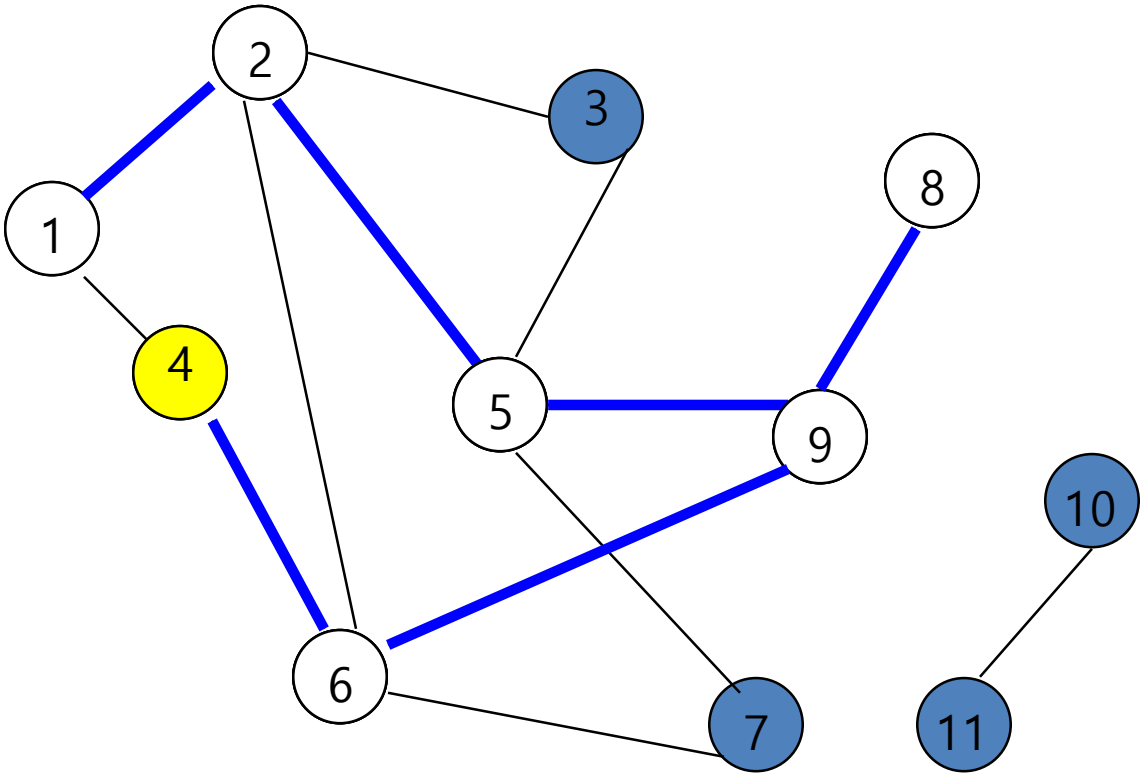
DFS : example



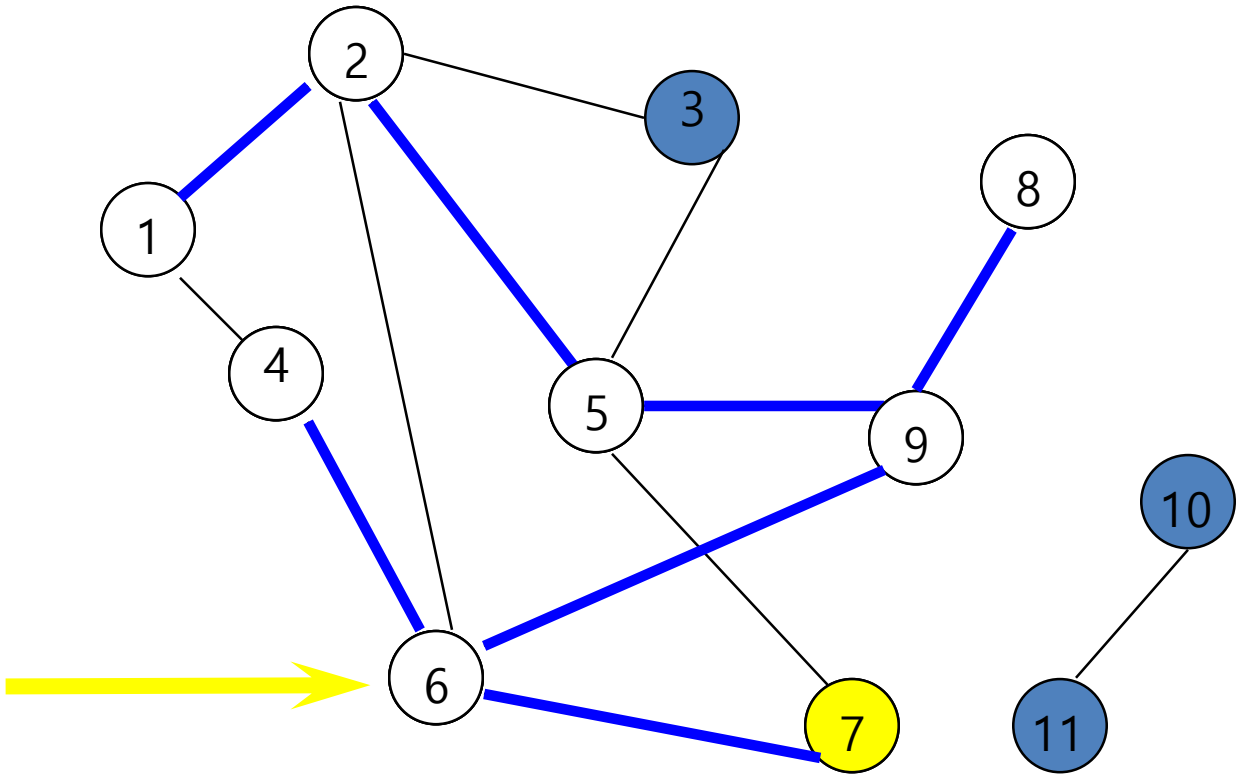
DFS : example



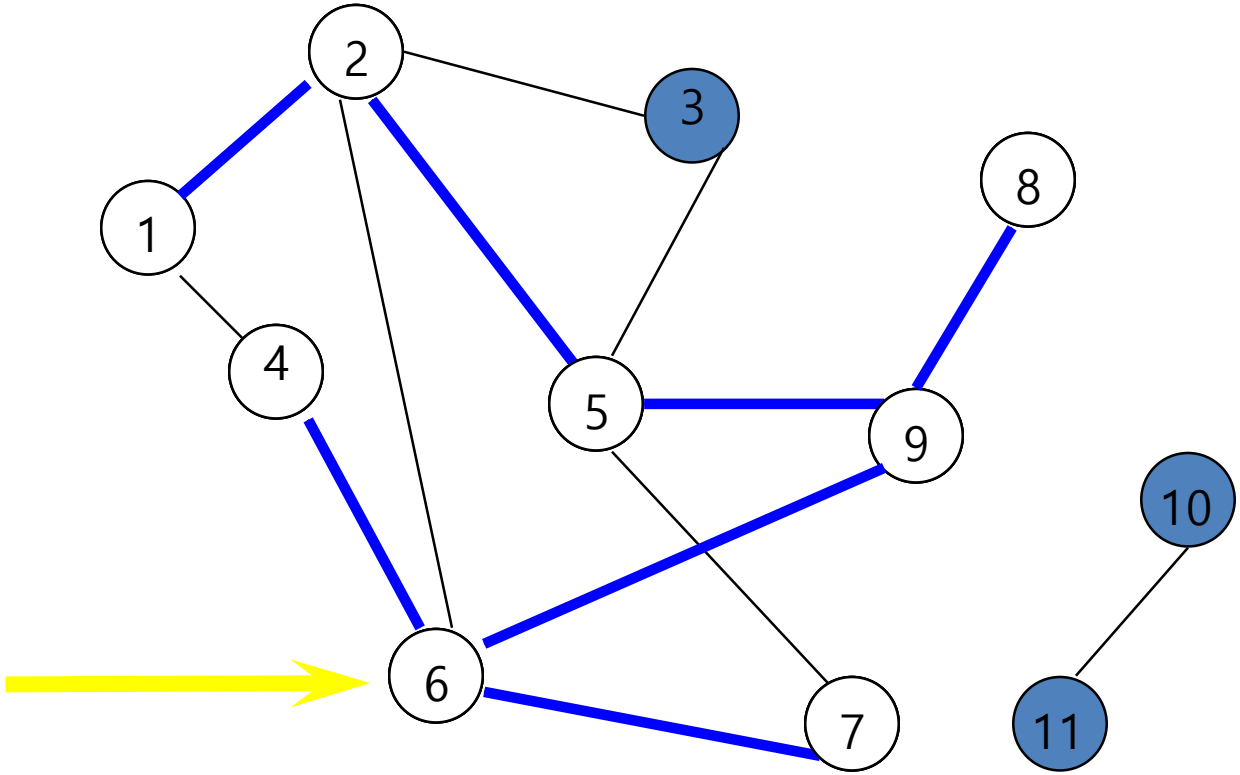
DFS : example



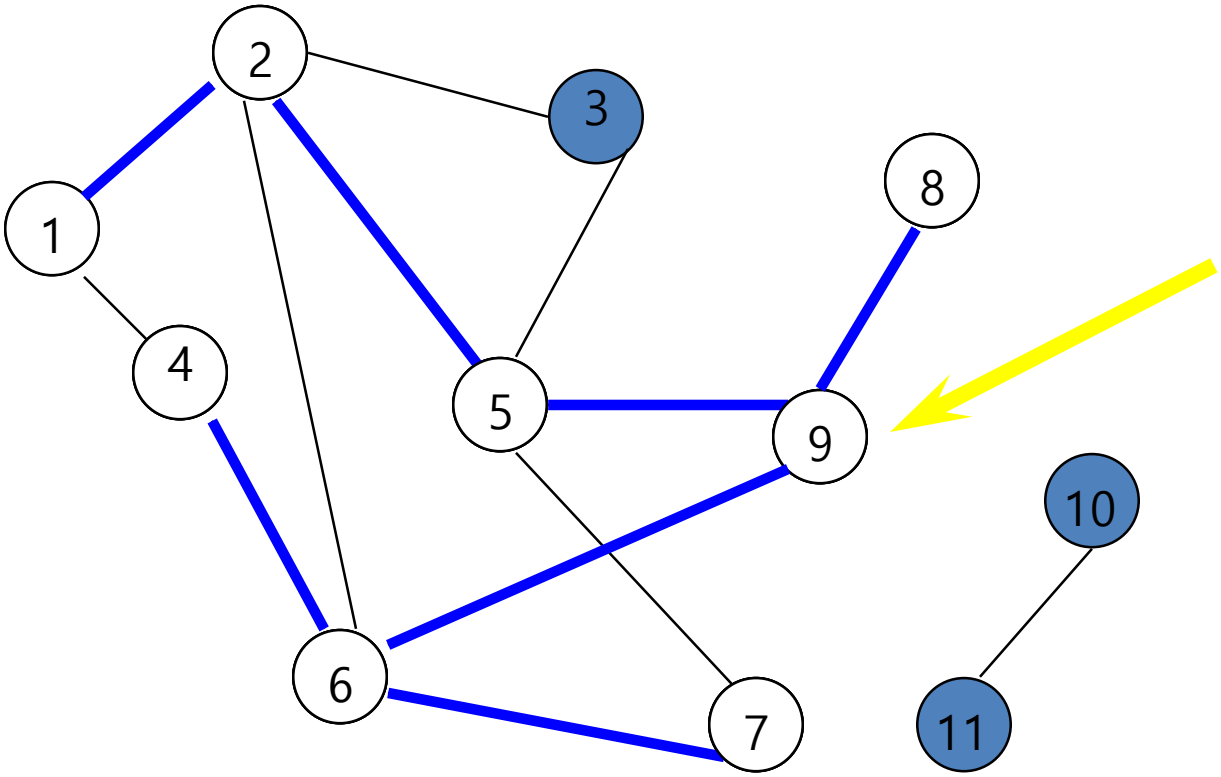
DFS : example



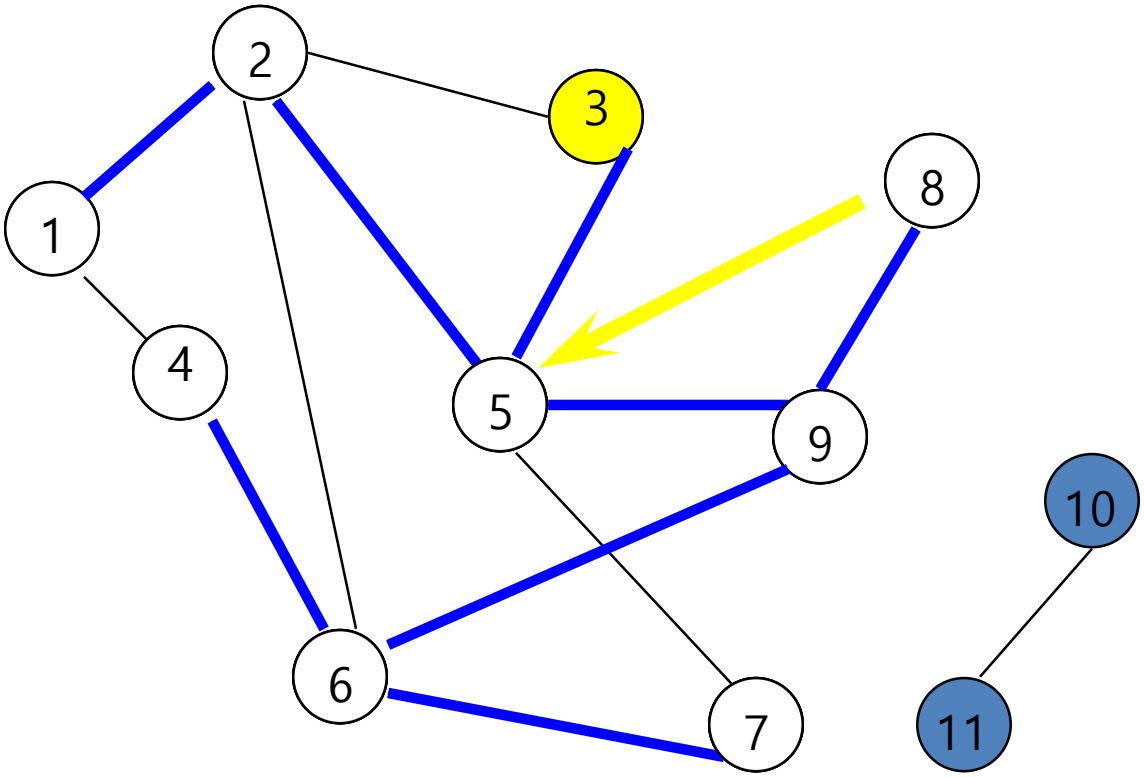
DFS : example



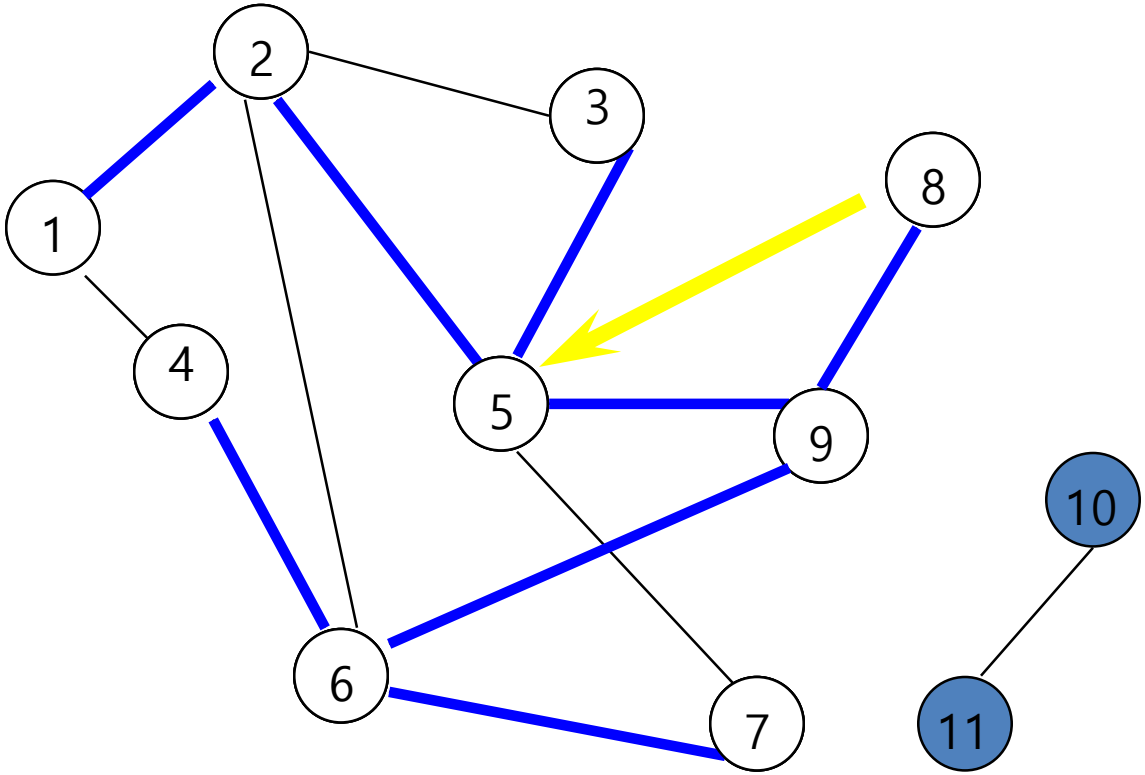
DFS : example



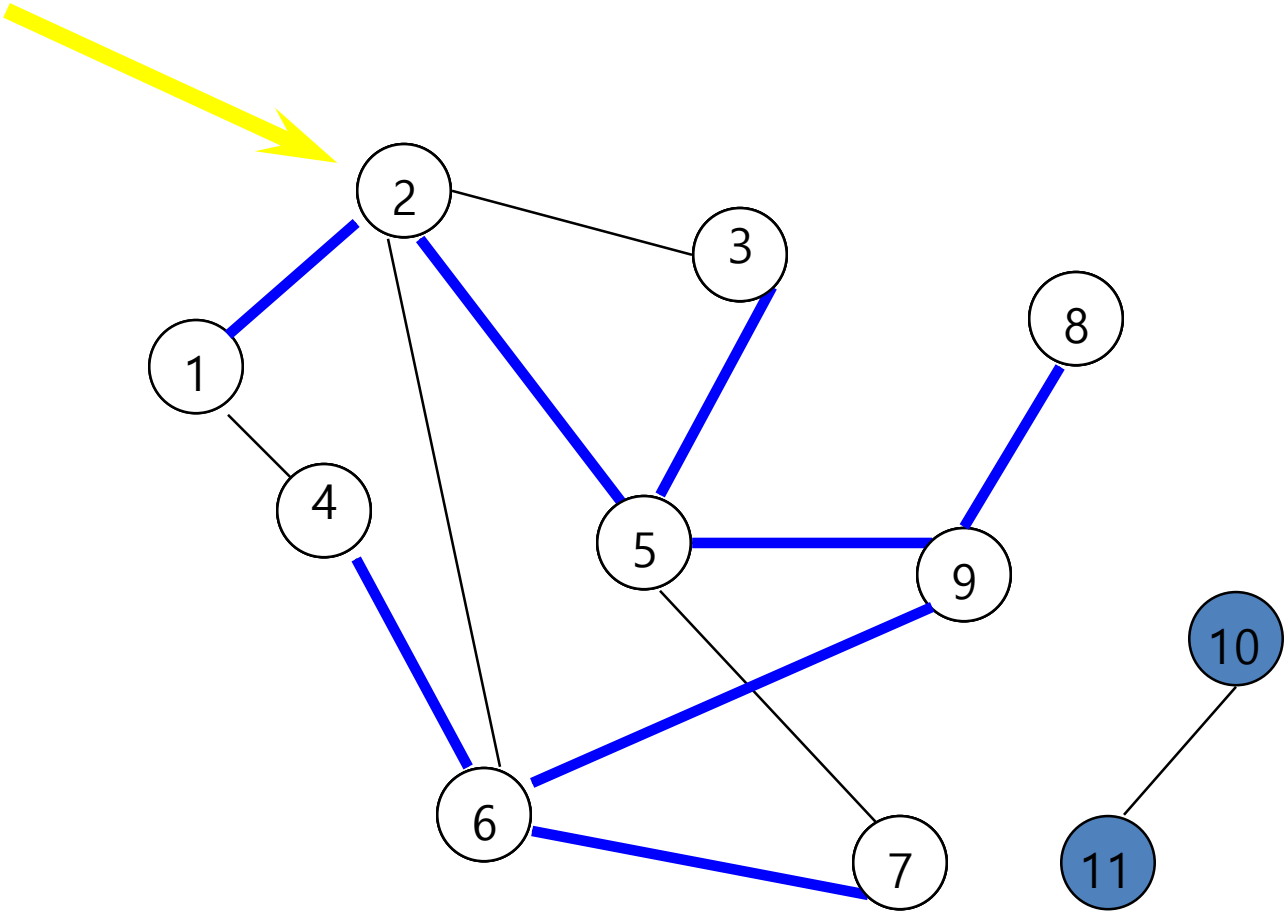
DFS : example



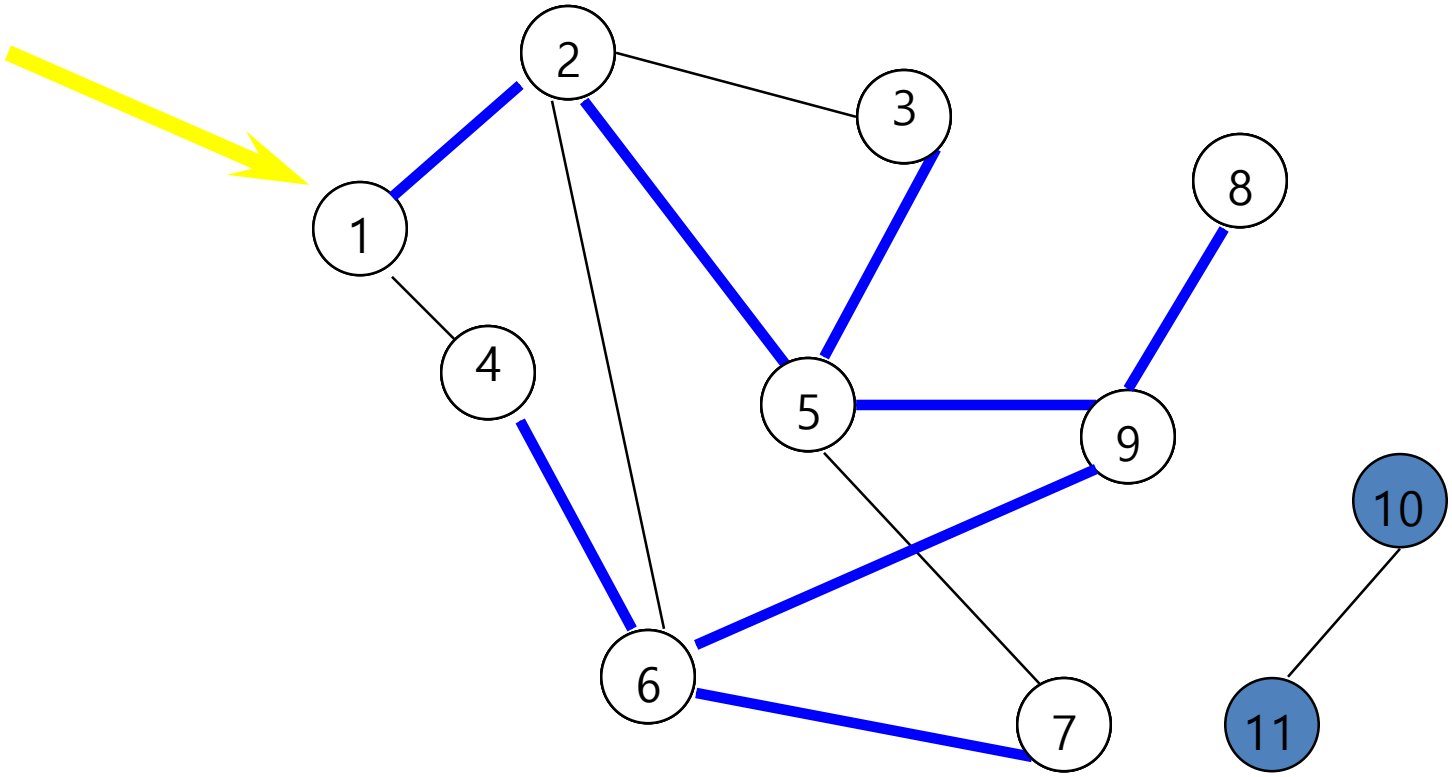
DFS : example



DFS : example



DFS : example



- 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 : $O(n^2)$

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.

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

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

– use a *dynamically linked queue* as in Chapter 4

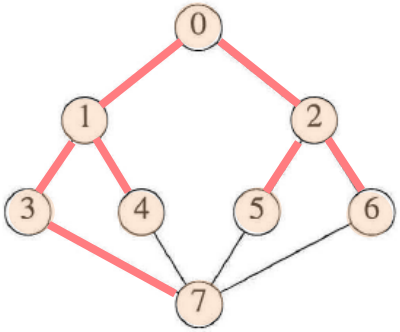
- Program 4.7, 4.8
  - replace all reference to *element* with int.

```
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;
  addq(v);
  while (front) { // non-empty queue
    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

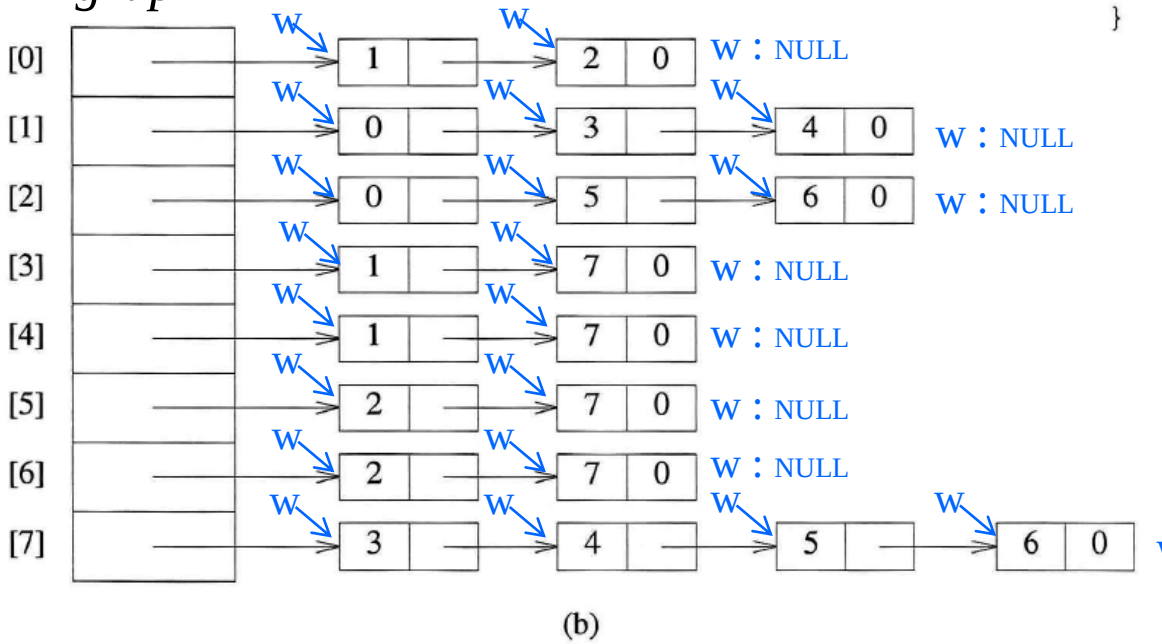
bfs(0)

```
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;
         }
      }
}
```



(a)

graph



(b)

output 0 1 2 3 4 5 6 7

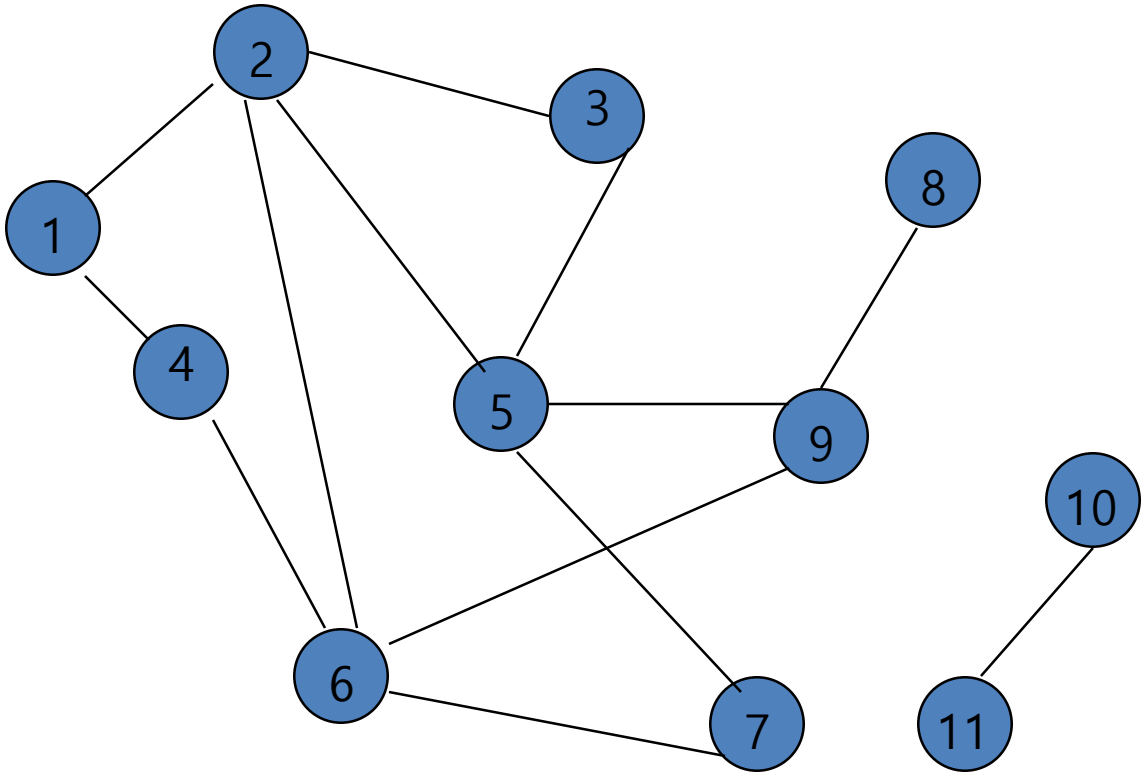
visited

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

queue

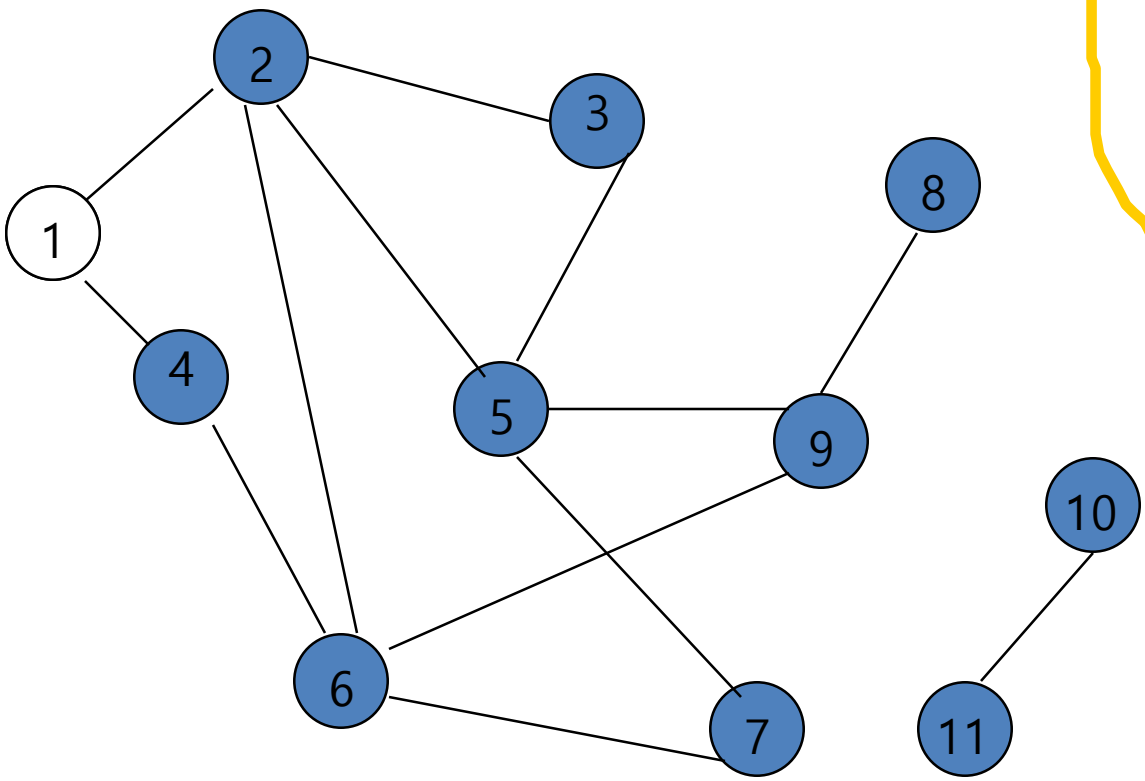
Figure 6.16: Graph *G* and its adjacency lists

BFS : Example



Start search at vertex 1.

BFS : Example

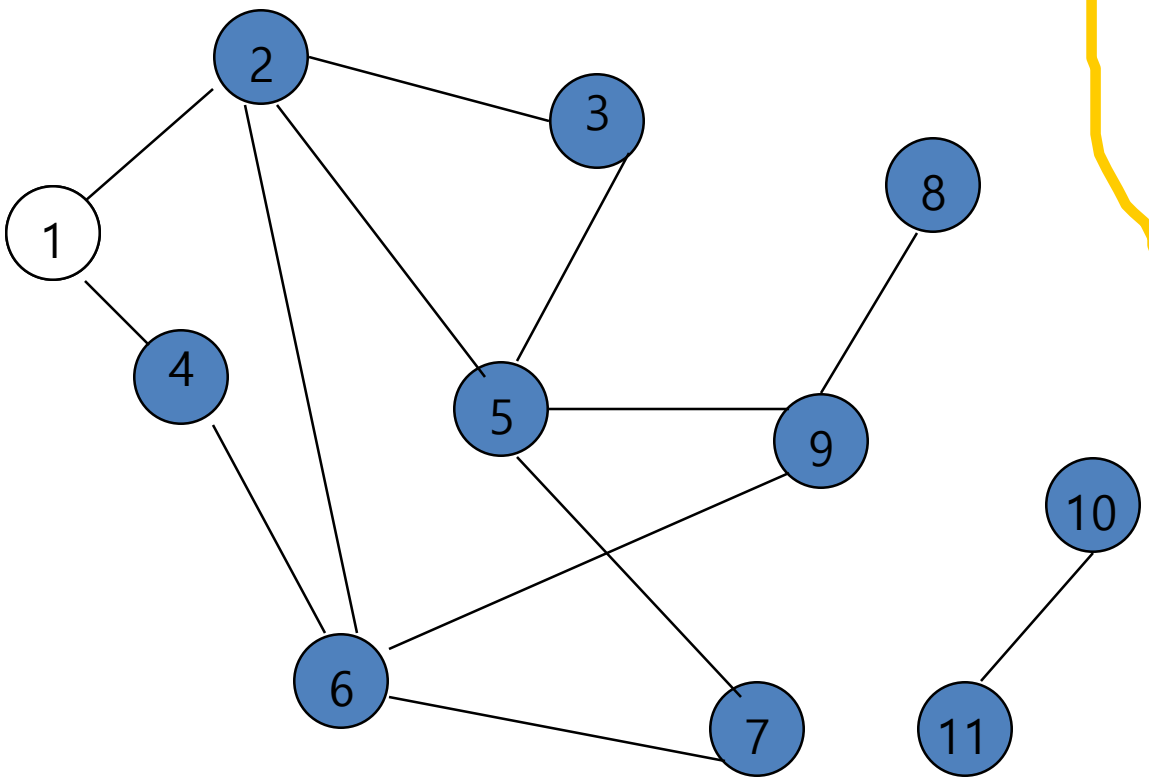


FIFO Queue

1

Visit/mark/label start vertex and put in a FIFO queue.

BFS : Example

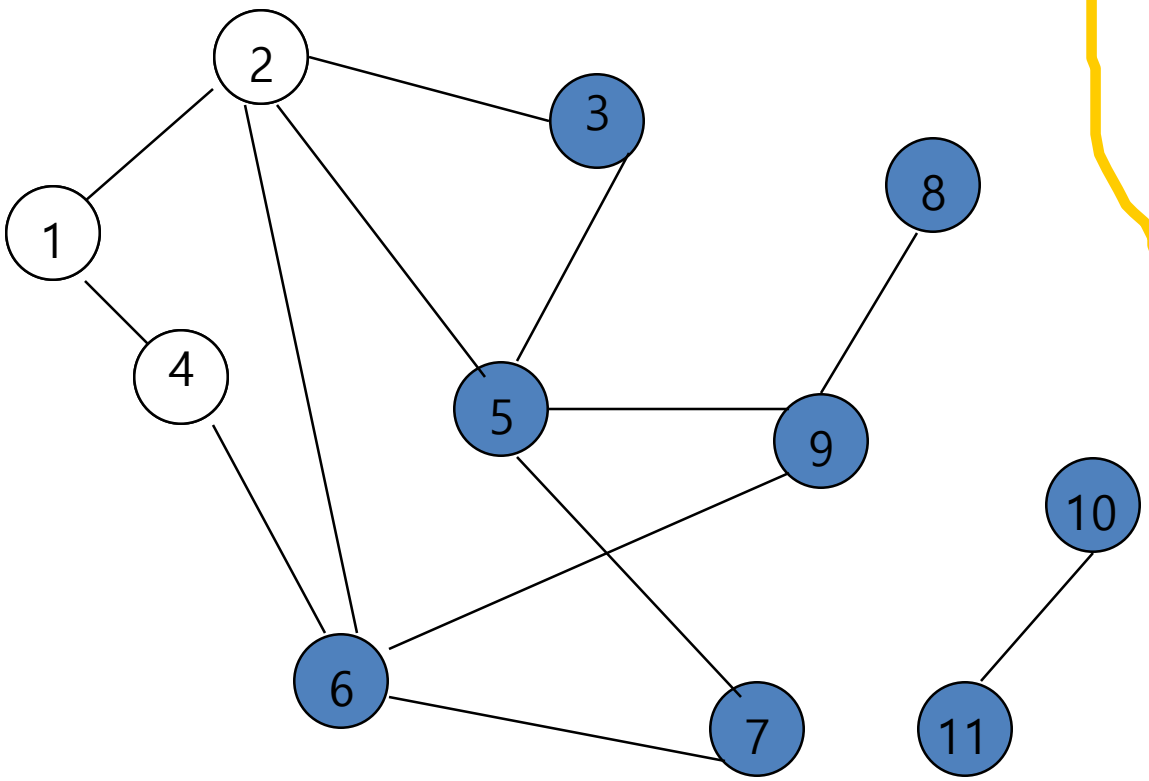


FIFO Queue

1

Remove 1 from Q; visit adjacent unvisited vertices; put in Q ;

BFS : Example

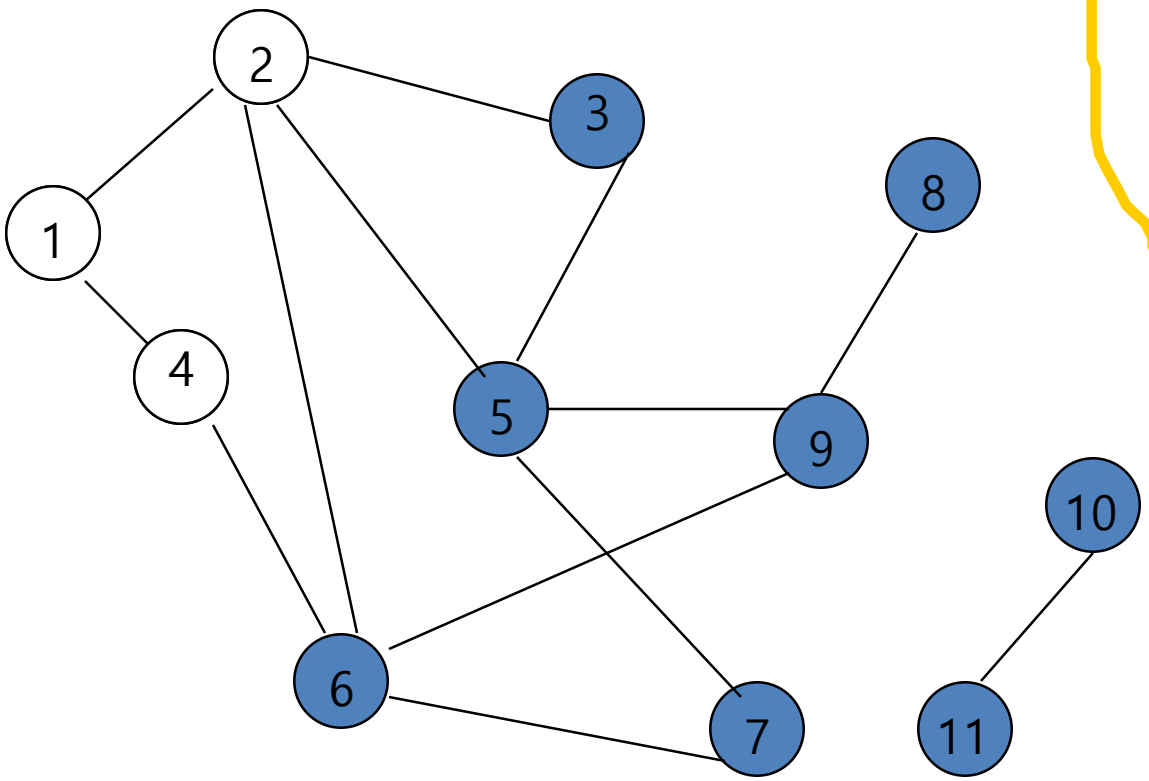


FIFO Queue

2 4

Remove 1 from Q; visit adjacent unvisited vertices; put in Q ;

BFS : Example

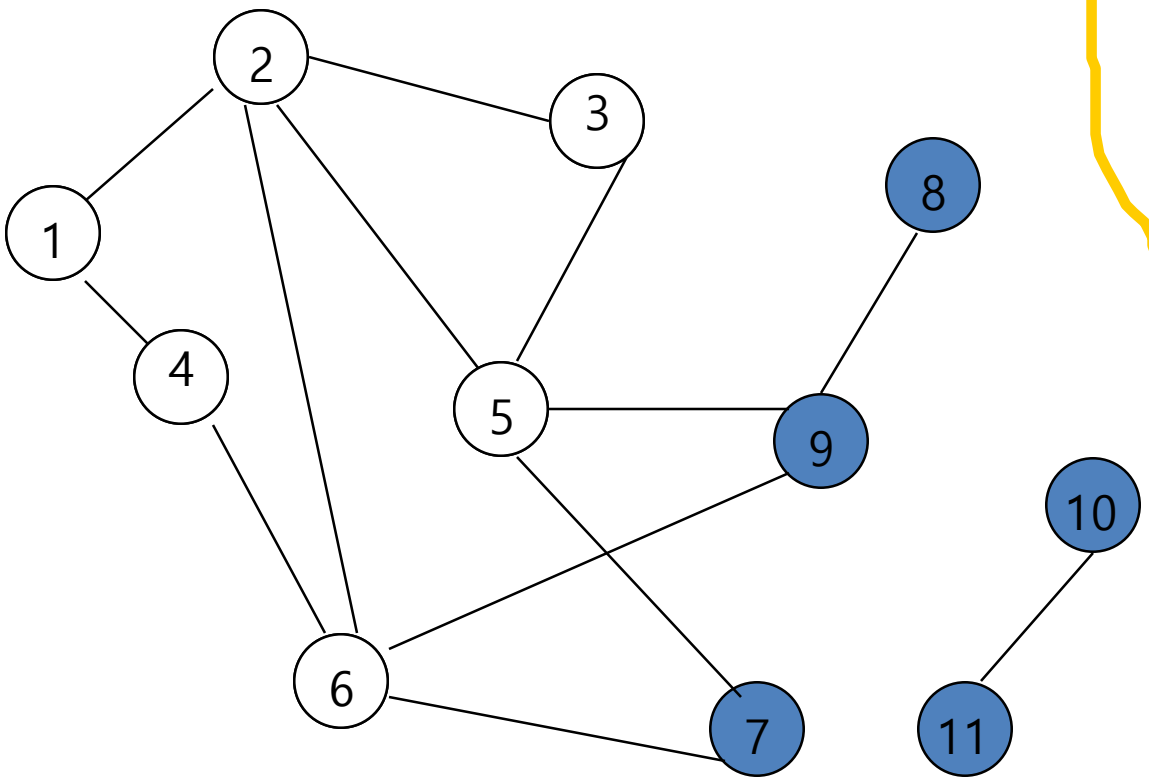


FIFO Queue

2 4

Remove 2 from Q; visit adjacent unvisited vertices; put in Q;

BFS : Example

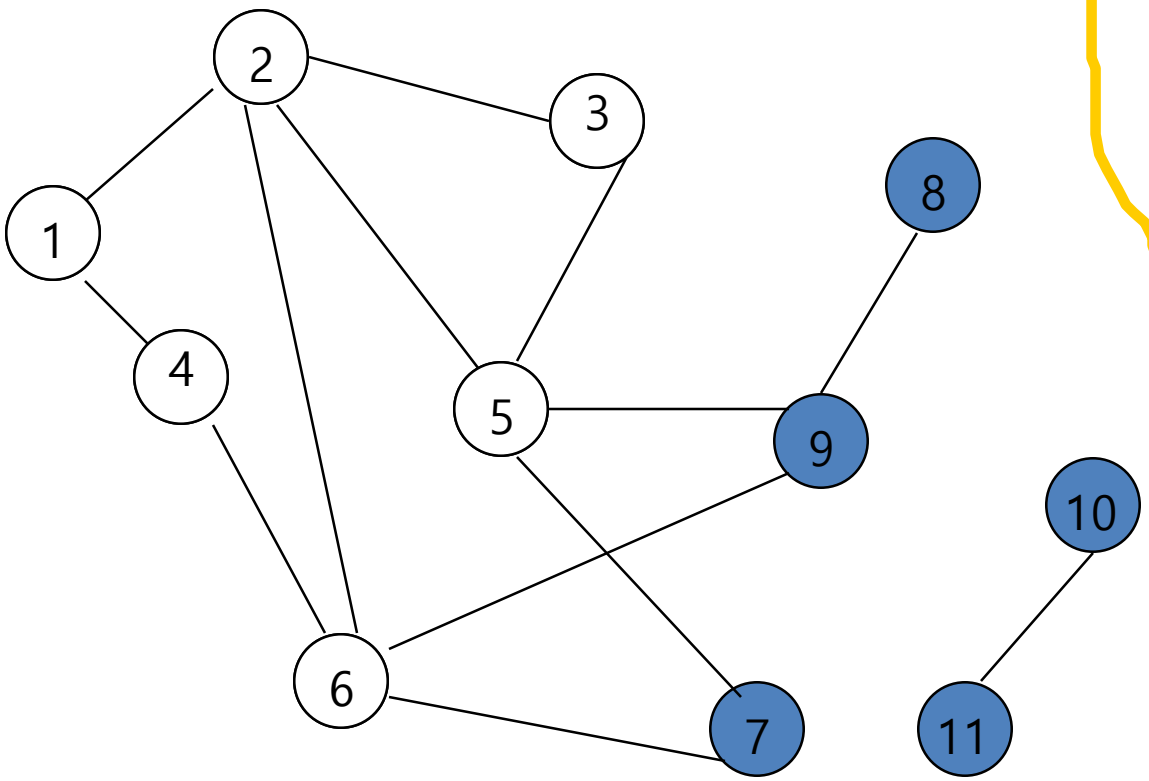


FIFO Queue

4 5 3 6

Remove 2 from Q; visit adjacent unvisited vertices; put in Q;

BFS : Example

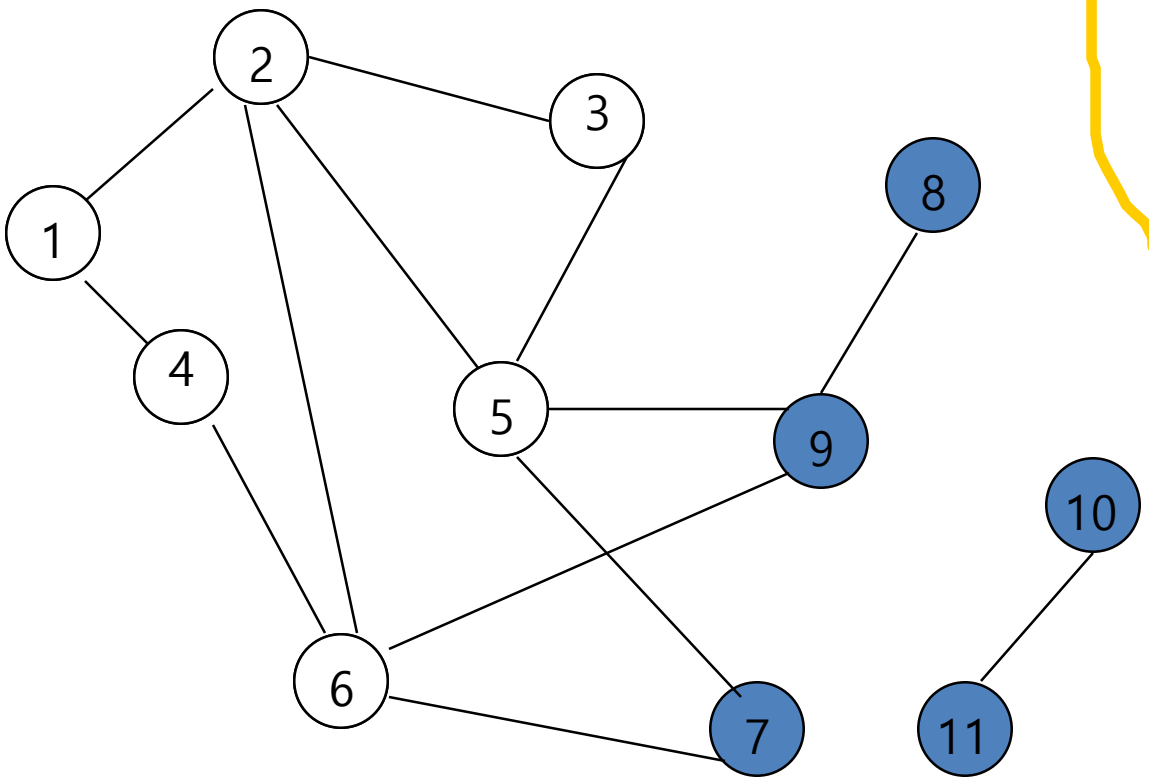


FIFO Queue

4 5 3 6

Remove 4 from Q; visit adjacent unvisited vertices; put in Q;

BFS : Example

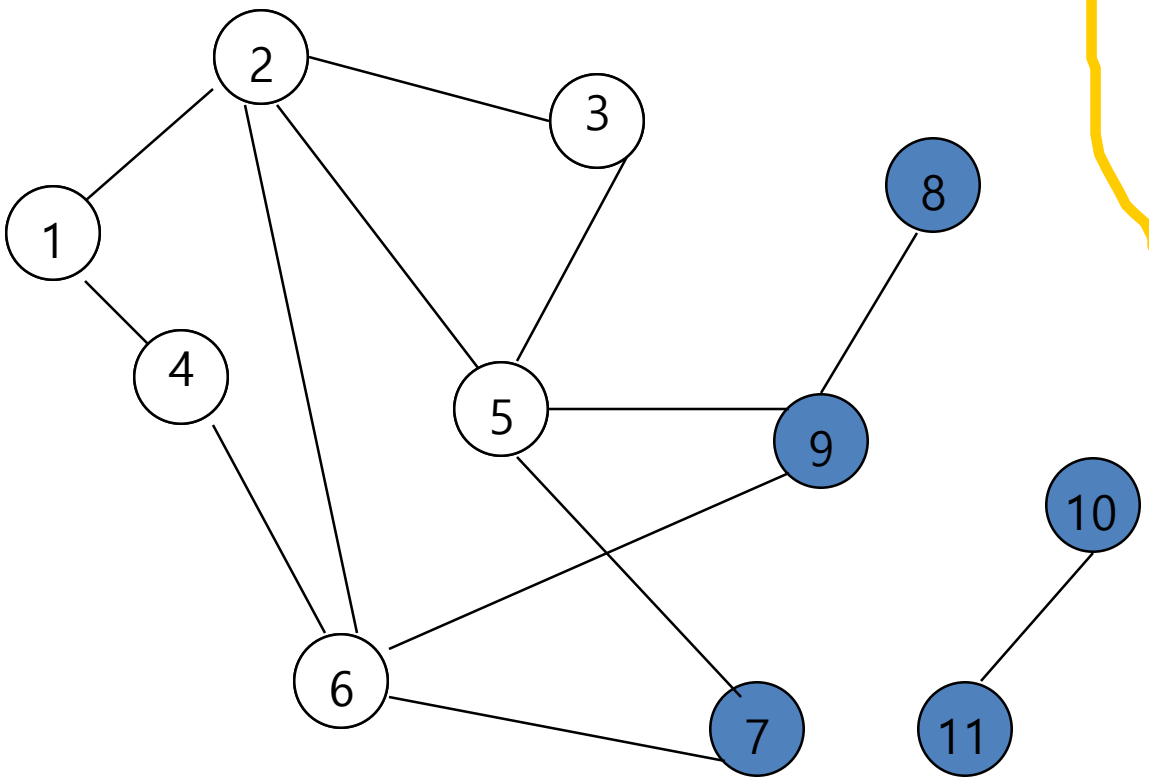


FIFO Queue

5 3 6

Remove 4 from Q; visit adjacent unvisited vertices; put in Q;

BFS : Example

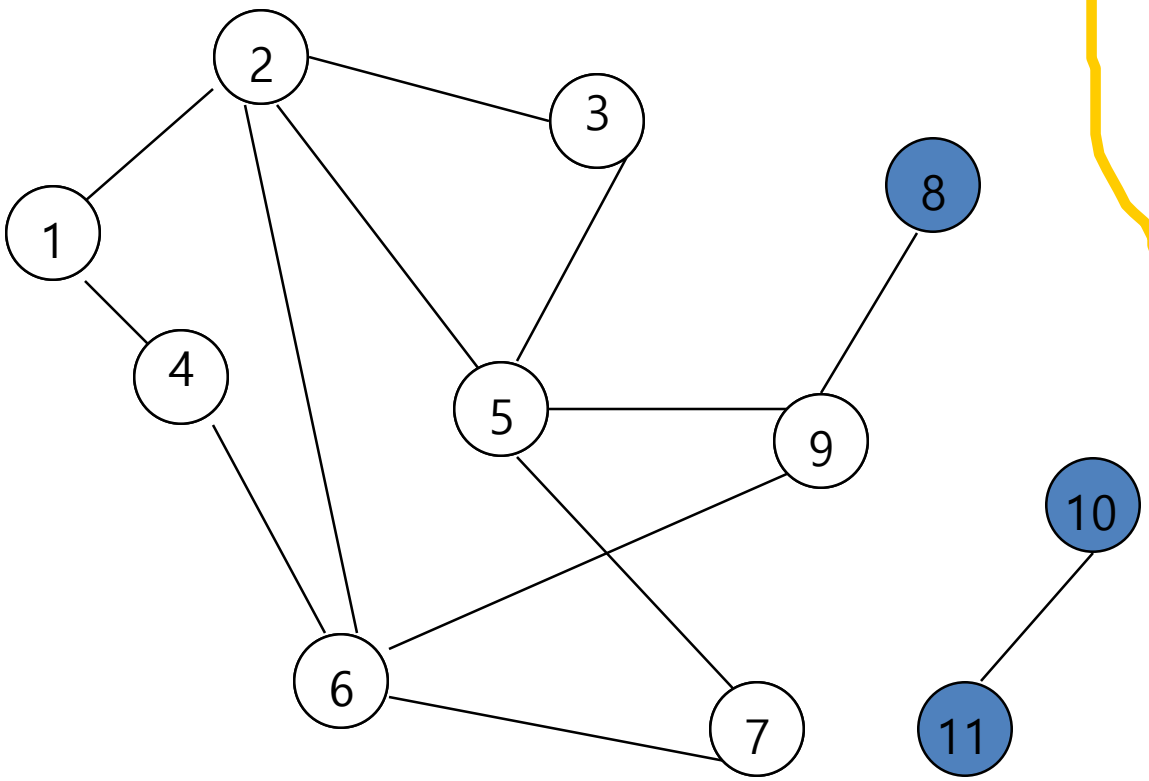


FIFO Queue

5 3 6

Remove 5 from Q; visit adjacent unvisited vertices; put in Q;

BFS : Example

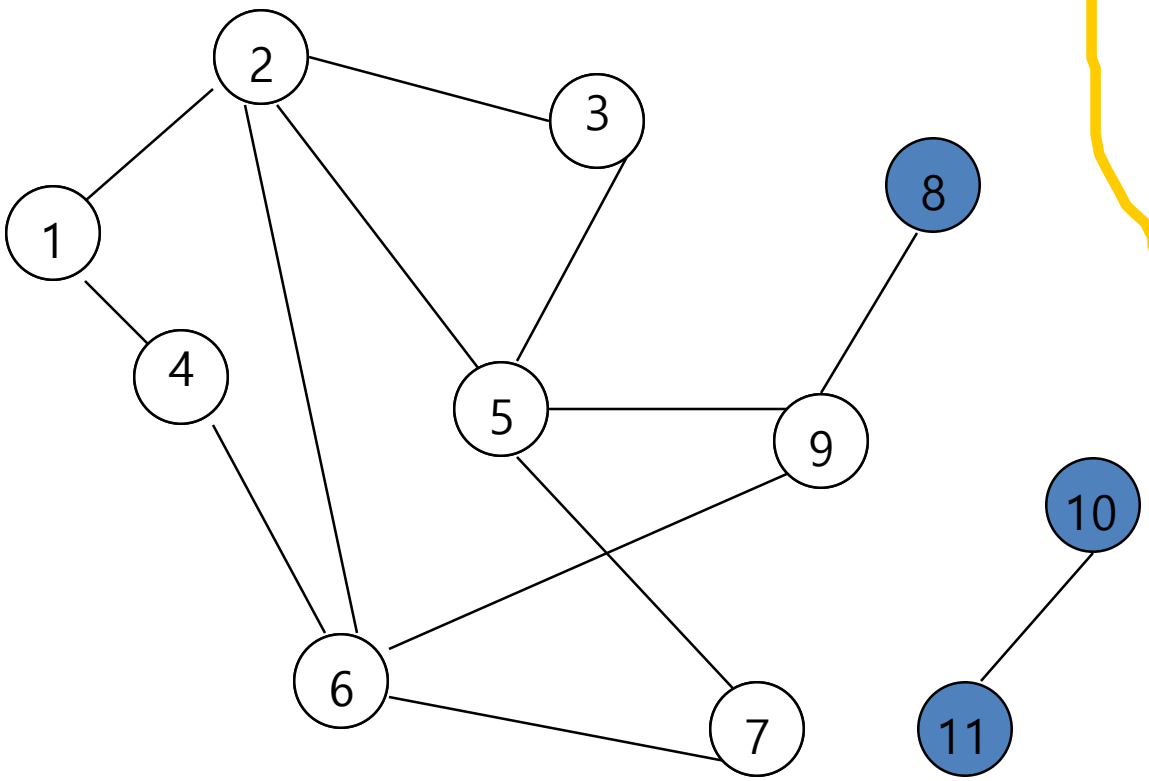


FIFO Queue

3 6 9 7

Remove 5 from Q; visit adjacent unvisited vertices; put in Q;

BFS : Example

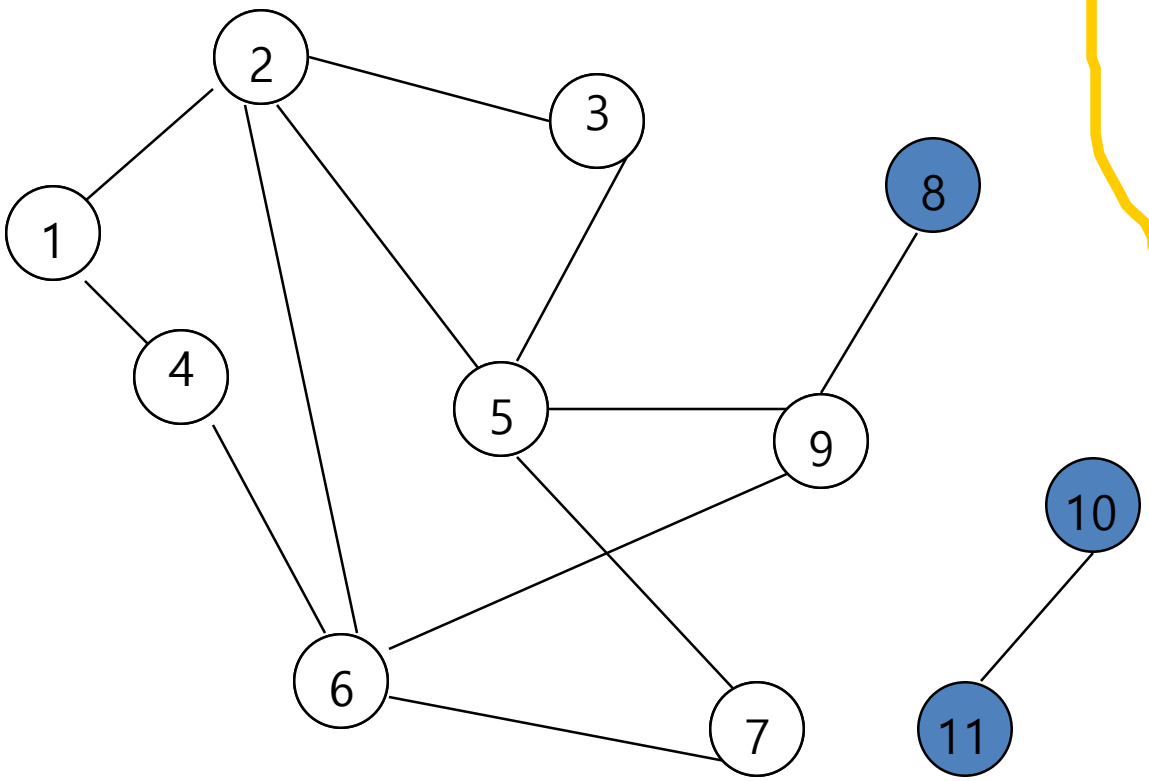


FIFO Queue

3 6 9 7

Remove 3 from Q; visit adjacent unvisited vertices; put in Q;

BFS : Example

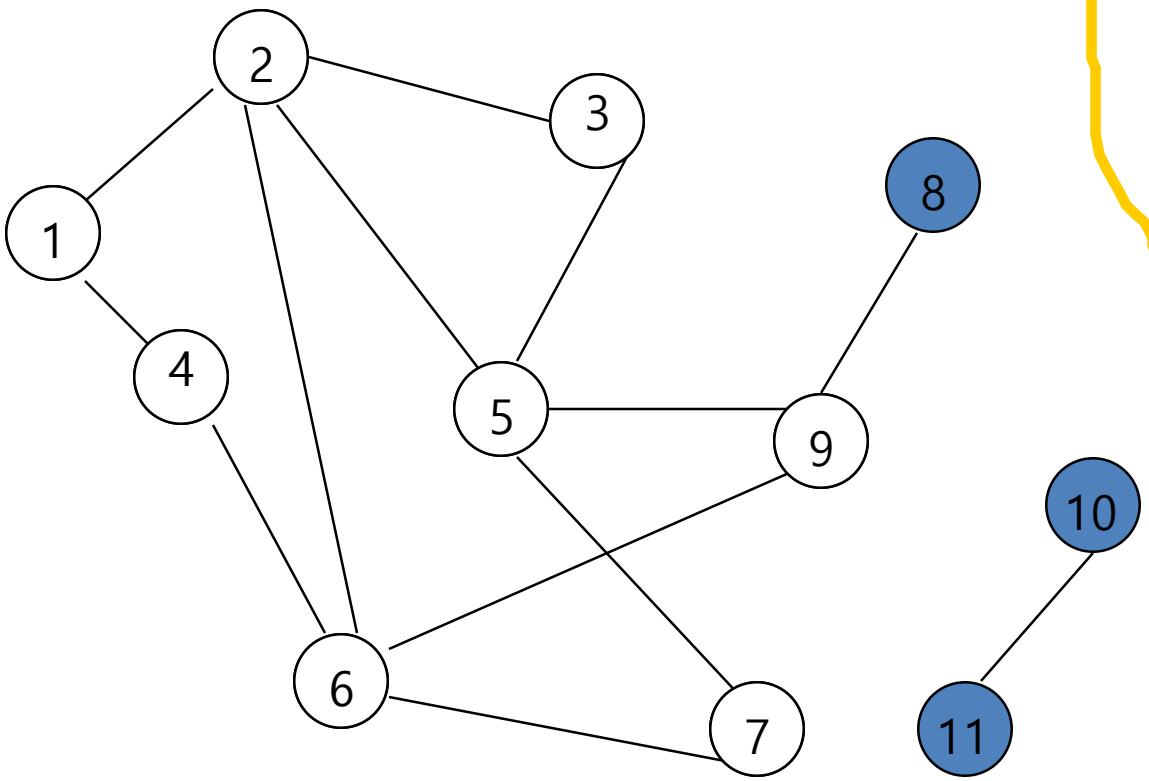


FIFO Queue

6 9 7

Remove 3 from Q; visit adjacent unvisited vertices; put in Q;

BFS : Example

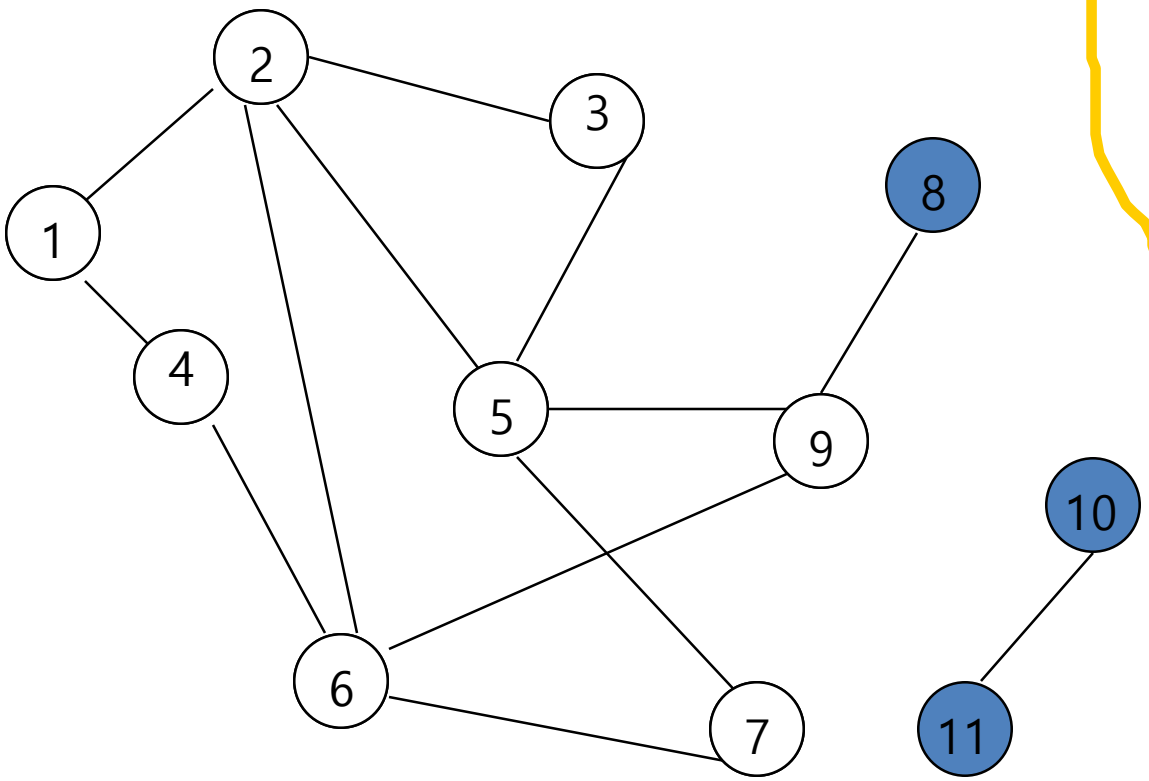


FIFO Queue

6 9 7

Remove 6 from Q; visit adjacent unvisited vertices; put in Q;

BFS : Example

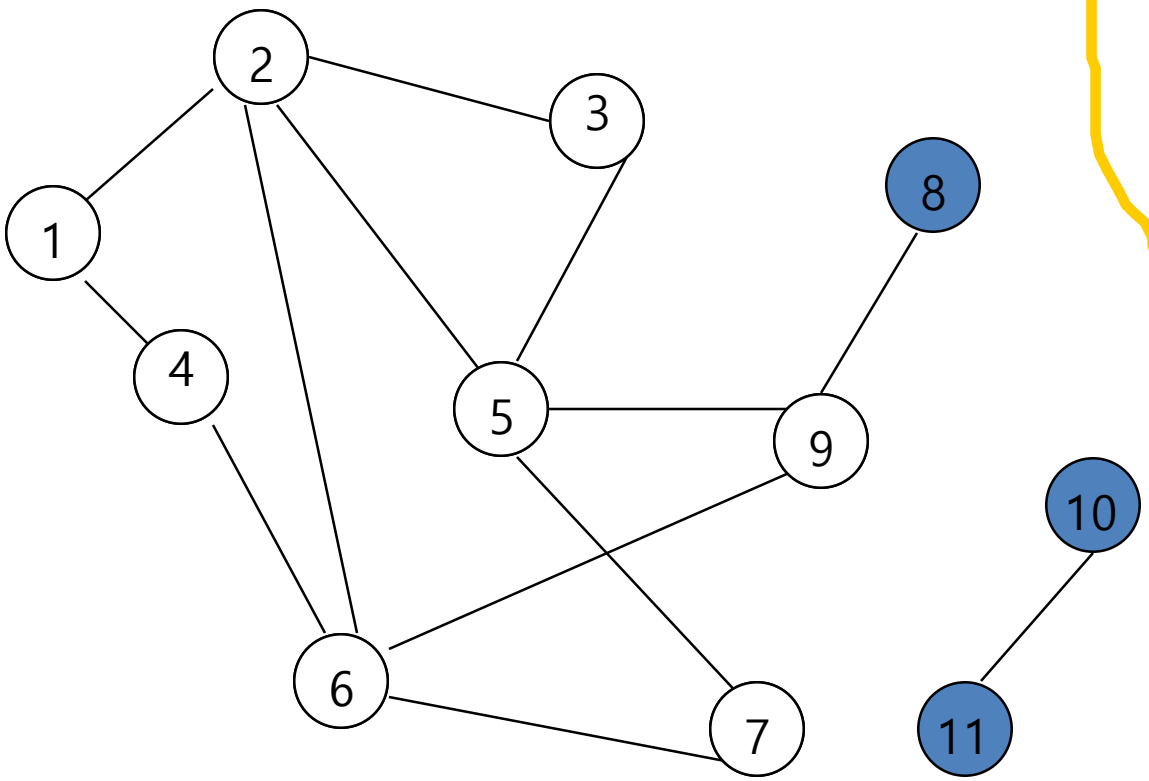


FIFO Queue

9 7

Remove 6 from Q; visit adjacent unvisited vertices; put in Q;

BFS : Example

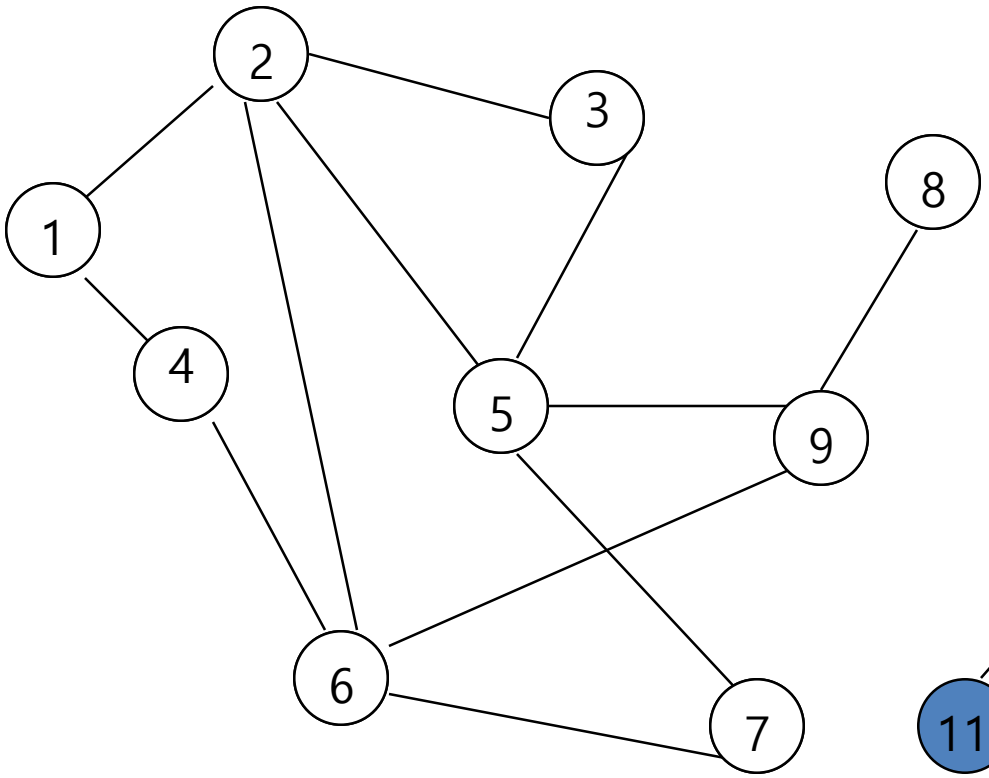


FIFO Queue

9 7

Remove 9 from Q; visit adjacent unvisited vertices; put in Q;

BFS : Example

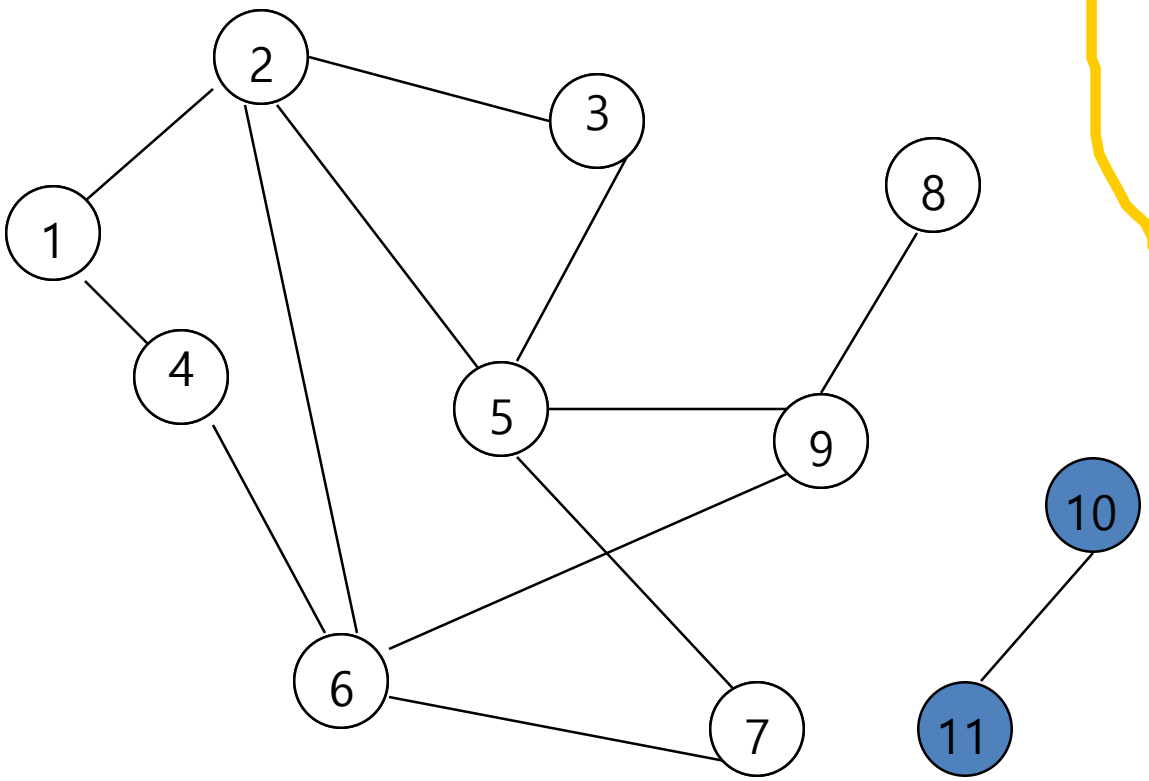


FIFO Queue

7 8

Remove 9 from Q; visit adjacent unvisited vertices; put in Q;

BFS : Example

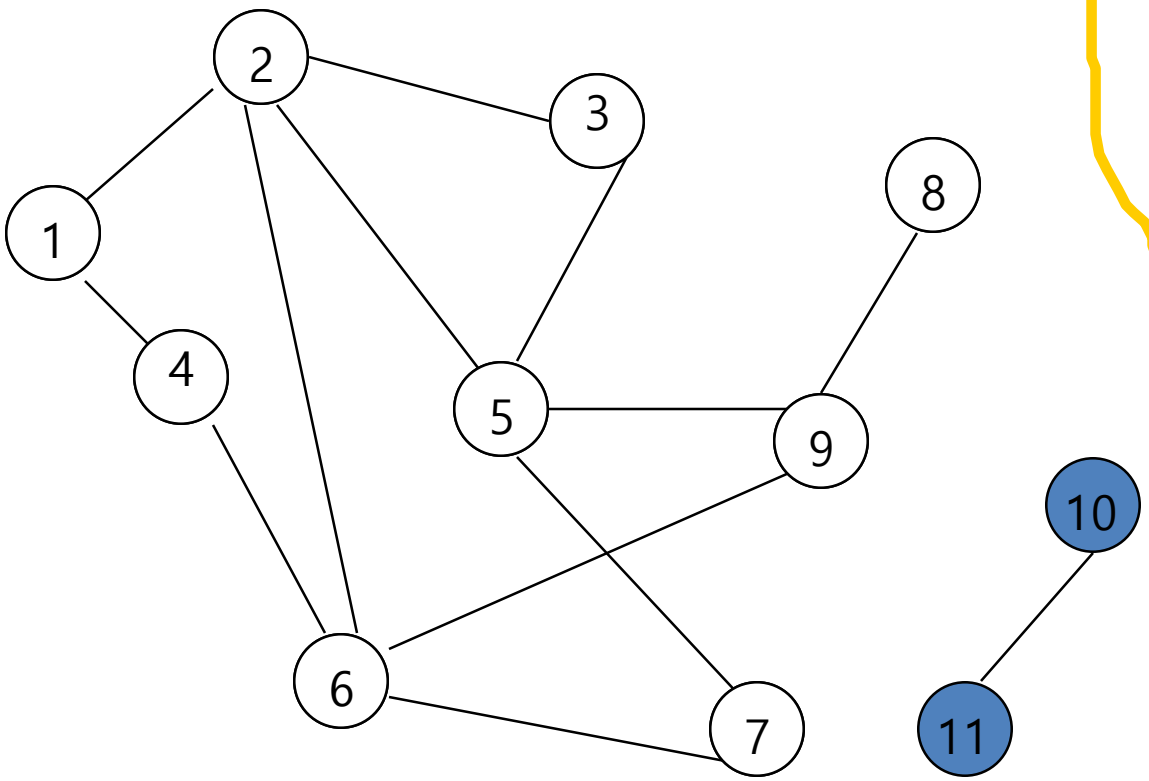


FIFO Queue

7 8

Remove 7 from Q; visit adjacent unvisited vertices; put in Q;

BFS : Example

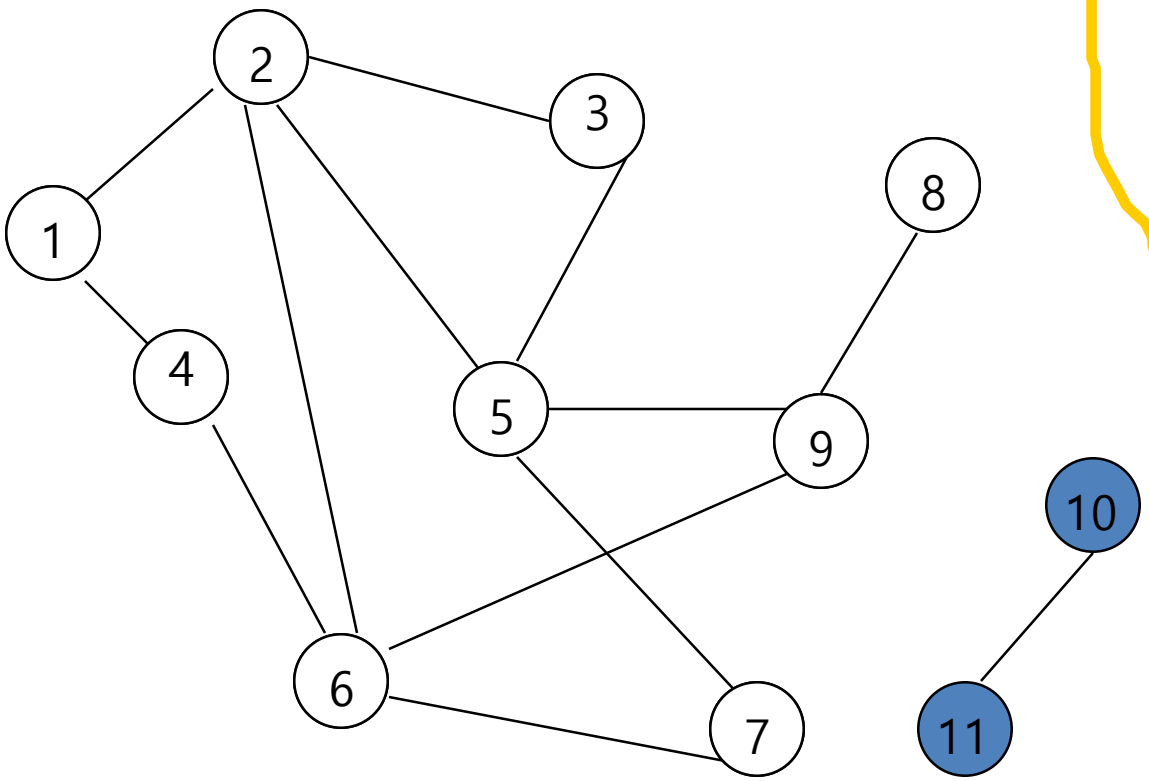


FIFO Queue

8

Remove 7 from Q; visit adjacent unvisited vertices; put in Q;

BFS : Example

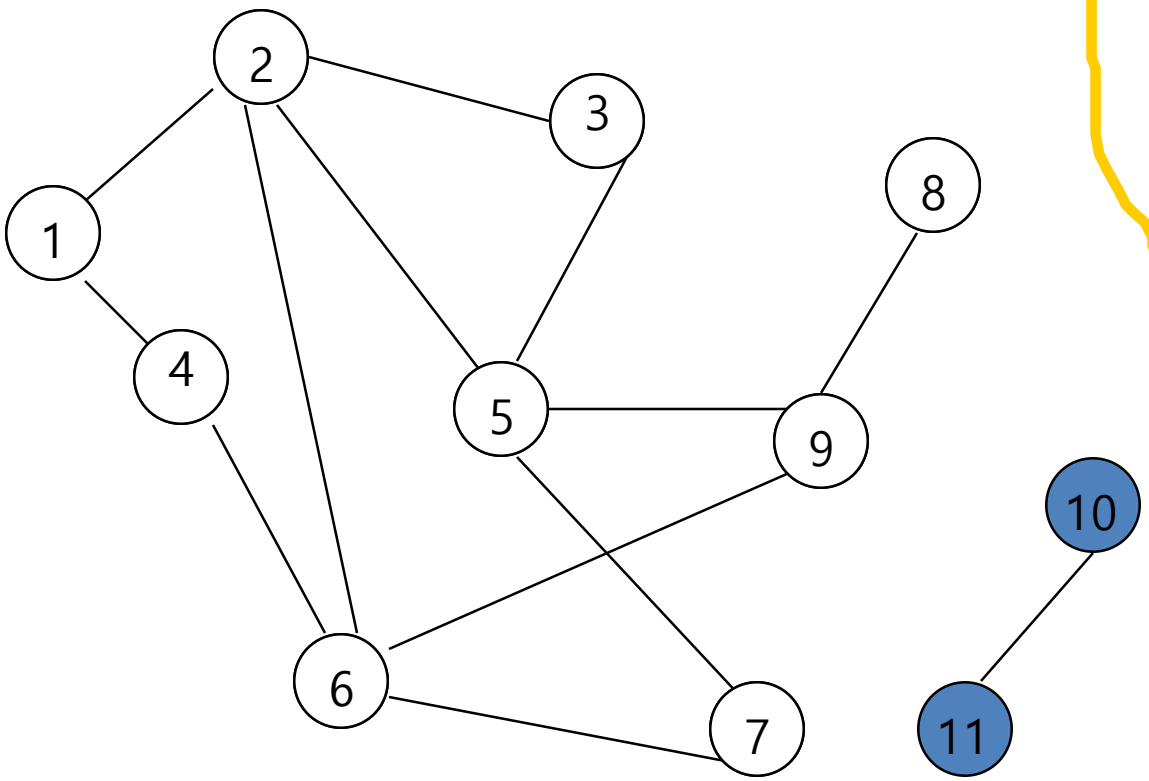


FIFO Queue

8

Remove 8 from Q; visit adjacent unvisited vertices; put in Q;

BFS : Example



FIFO Queue

Queue is empty. Search terminates.

- Analysis of *bfs*
  - adjacency list : $O(e)$
  - adjacency matrix : $O(n^2)$

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(n+e) for adjacency list

- 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

```
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

- 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

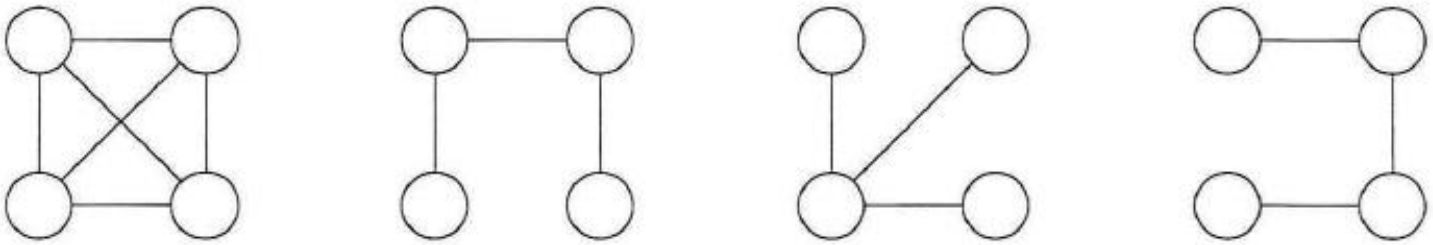
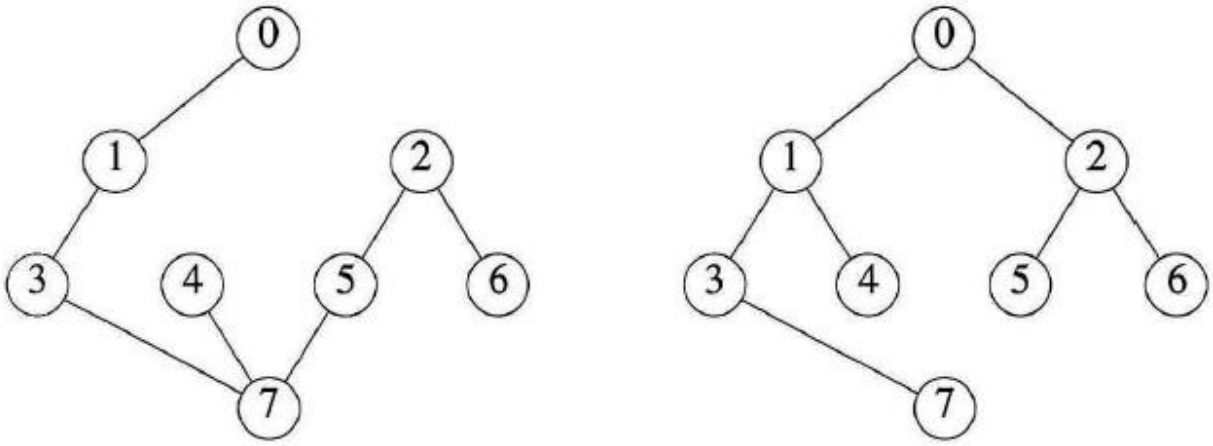


Figure 6.17: A complete graph and three of its spanning trees

- 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

- 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 with n vertices has n-1 edges.

6.3 Minimum Cost Spanning Trees

- *Cost* of a spanning tree
  - sum of the costs (weights) of the edges in the spanning tree

- *Minimum cost spanning tree*
  - a spanning tree of least cost

- Kruskal's, Prim's and Sollin's algorithms
  - three algorithms to build minimum cost spanning tree of a connected undirected graph
  - greedy method

- *Greedy method*
  - at each stage, make the best decision possible at the time
    - based on either *a least cost* or *a highest profit* criterion
  - make sure the decision will result in a feasible solution
    - satisfying the *constraints* of the problem

- To construct minimum cost spanning trees
  - best decision : least cost
  - constraints
    - use only edges within the graph
    - use exactly *n*-1 edges
    - may not use edges that produce a cycle

6.3.1 Kruskal's Algorithm

- Procedure
  - build a min-cost spanning tree T by adding edges to T one at a time
  - select edges for inclusion in T in nondecreasing order of their cost
  - edge is added to T if it does not form a cycle

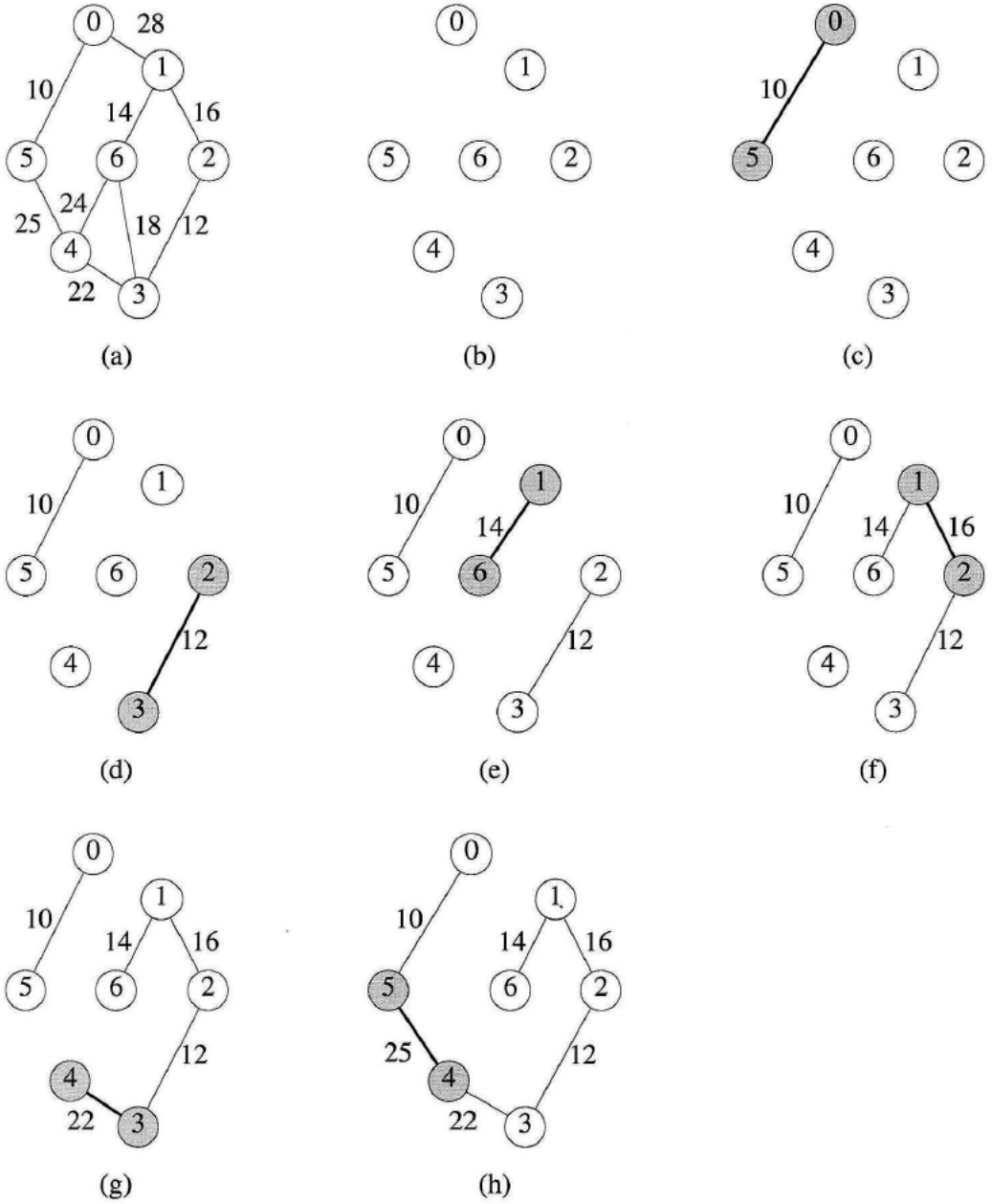


Figure 6.22: Stages in Kruskal's algorithm

Edge	Weight	Result	Figure
----	---	initial	Figure 6.22(b)
(0,5)	10	added to tree	Figure 6.22(c)
(2,3)	12	added	Figure 6.22(d)
(1,6)	14	added	Figure 6.22(e)
(1,2)	16	added	Figure 6.22(f)
(3,6)	18	discarded	
(3,4)	22	added	Figure 6.22(g)
(4,6)	24	discarded	
(4,5)	25	added	Figure 6.22(h)
(0,1)	28	not considered	

Summary of Kruskal's algorithm applied to Figure 6.22(a)

```
T = {};
while (T contains less than n-1 edges && E is not empty) {
   choose a least cost edge (v,w) from E;
   delete (v,w) from E;
   if ((v,w) does not create a cycle in T)
      add (v,w) to T;
   else
      discard (v,w);
}
if (T contains fewer than n-1 edges)
   printf("No spanning tree\n");
```

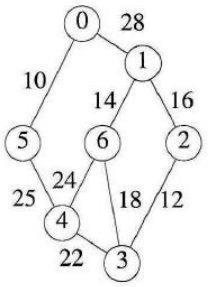
Program 6.7: Kruskal's algorithm

- **Theorem** :
  - Let G be any undirected, connected graph. Kruskal's algorithm generates a minimum-cost spanning tree.

6.3.2 Prim's Algorithm

```
T = {};
TV = {0}; /* start with vertex 0 and no edges */
while (T contains fewer than n-1 edges) {
   let (u, v) be a least cost edge such that u ∈ TV and
   v ∉ TV;
   if (there is no such edge)
      break;
   add v to TV;
   add (u, v) to T;
}
if (T contains fewer than n-1 edges)
   printf("No spanning tree\n");
```

Program 6.8: Prim's algorithm



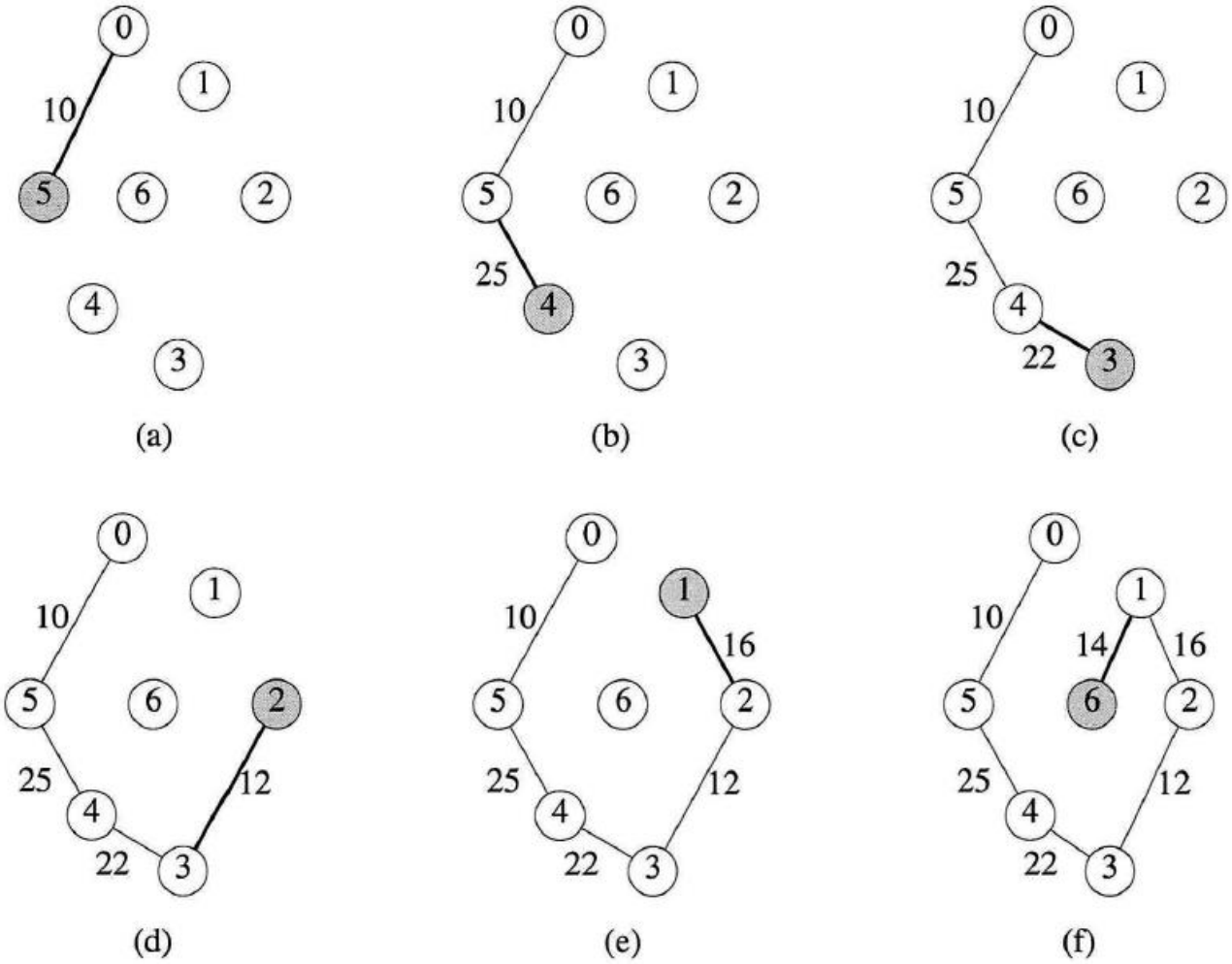


Figure 6.24: Stages in Prim's algorithm