CHAPTER 6

GRAPHS

GRAPHS

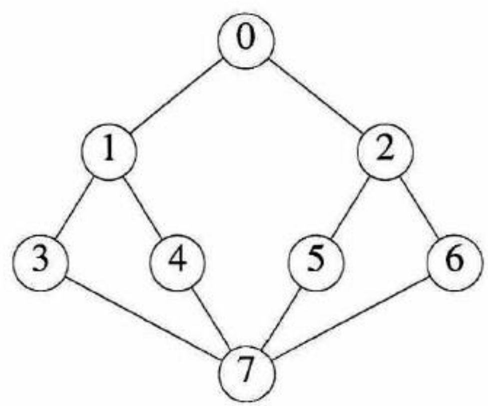
6.1 The Graph Abstract Data Type

6.2 Elementary Graph Operations

6.3 Minimum Cost Spanning Trees

6.2 Elementary Graph Operations

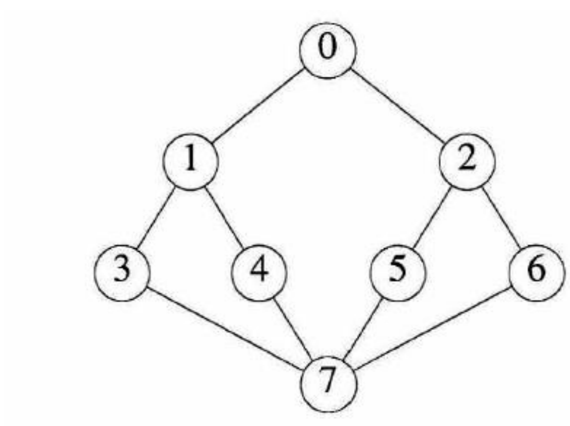
- Graph Traversal
  - Given an undirected graph G=(V, E) and a vertex *v* in V(G), visit all vertices reachable from *v*
  - Depth First Search (DFS) 깊이 우선
    - Similar to a preorder tree traversal
    - Uses stack or recursion
  - Breadth First Search (BFS) 넓이 우선
    - Similar to a level order tree traversal
    - Uses queue

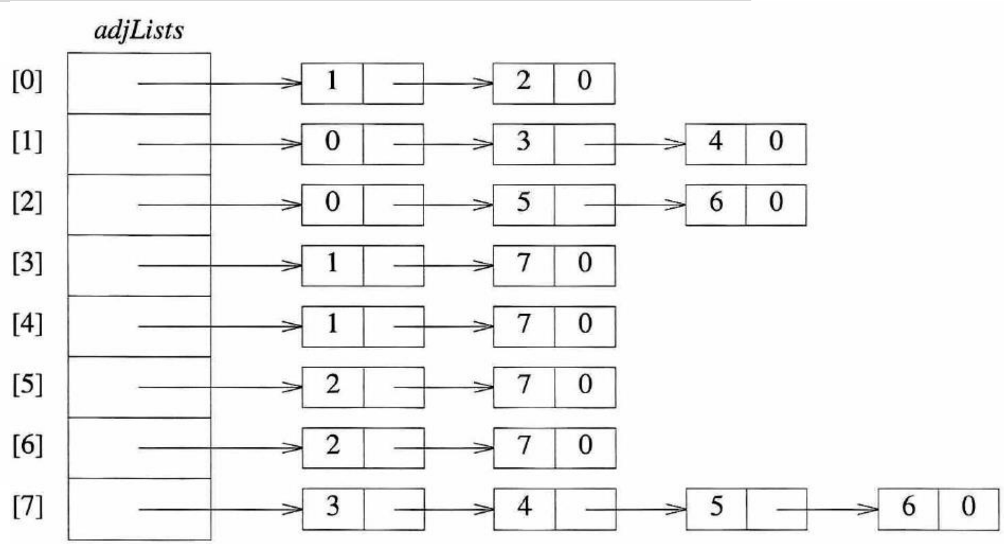


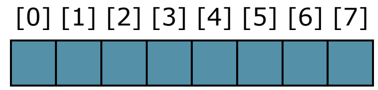
6.2.1 Depth First Search

- Procedure

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





visited: 


```
#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])    방문하지 않았으면
            dfs (w→vertex);
}
```

Program 6.1: Depth first search

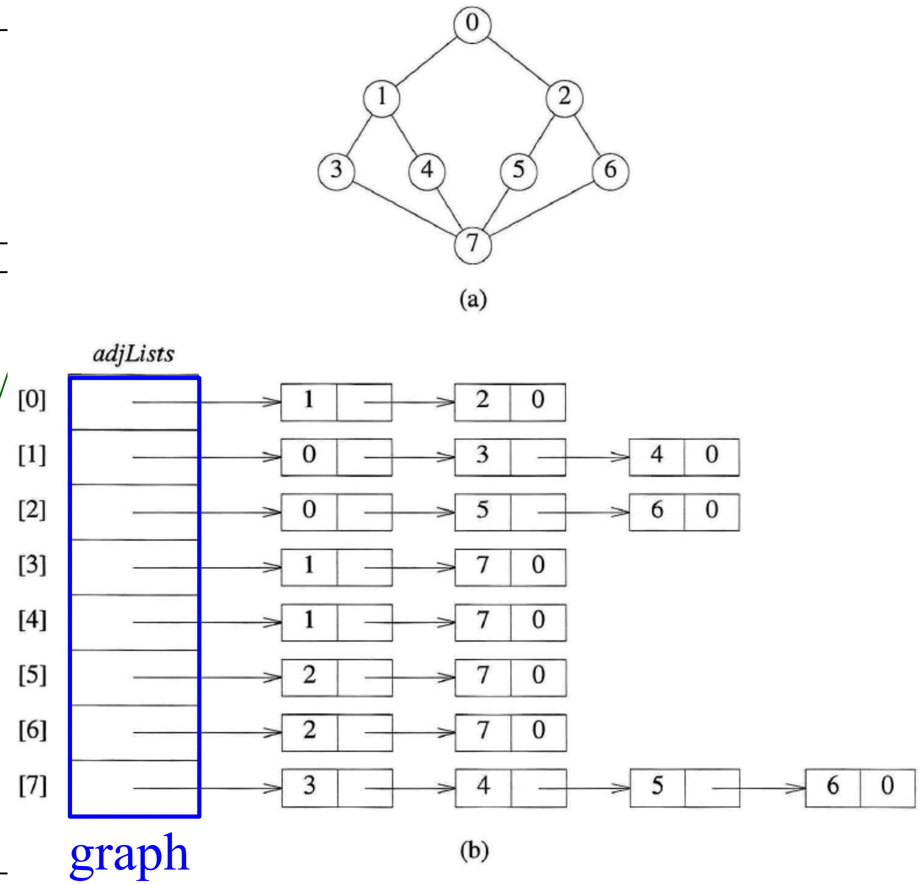
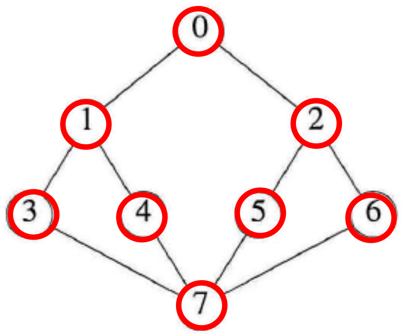


Figure 6.16: Graph *G* and its adjacency lists

output: 0 1 3 7 4 5 2 6

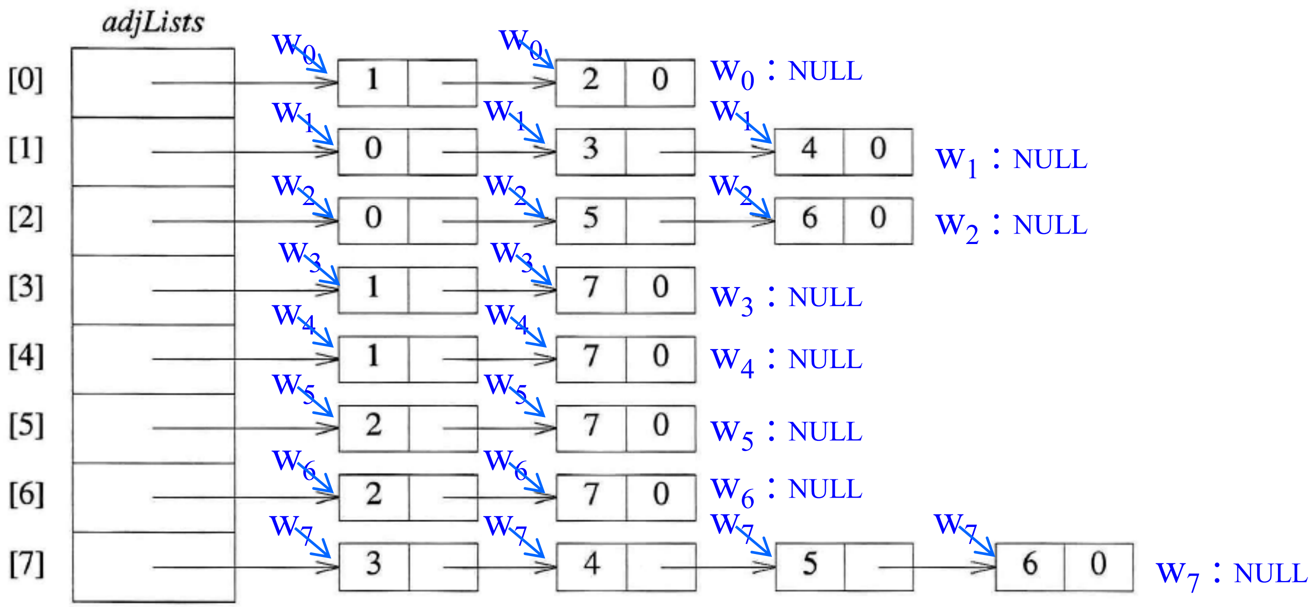
visited	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
	T	T	T	T	T	T	T	T



(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



graph

(b)

Figure 6.16: Graph *G* and its adjacency lists

- 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)

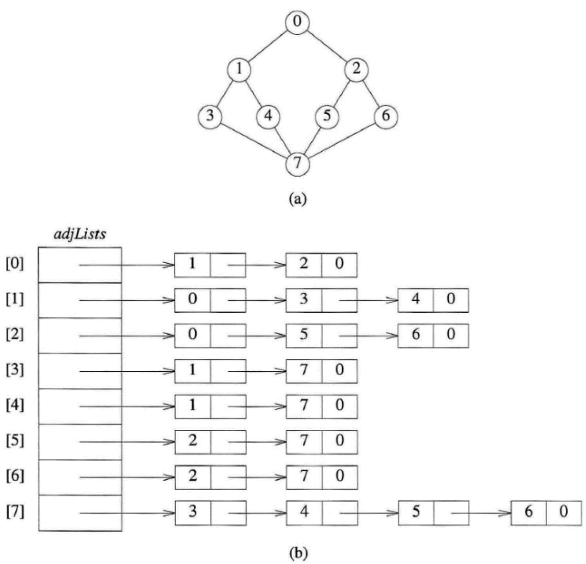


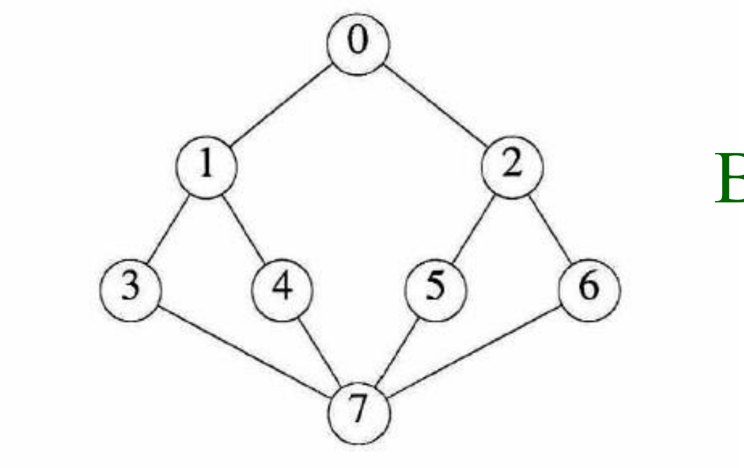
Figure 6.16: Graph G and its adjacency lists

```
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);
}
```

6.2.2 Breadth First Search

- Procedure

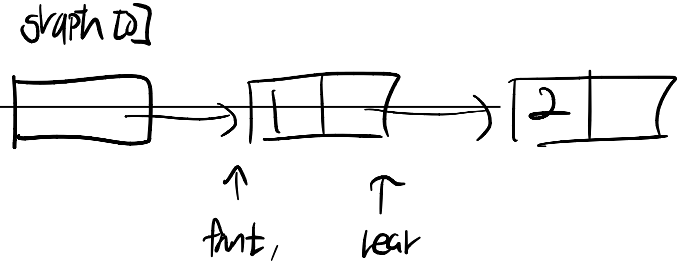
BFS(**v**) {
  mark ***v*** as visited and put ***v*** into **queue Q**
  While(Q is non-empty)
    remove a vertex ***w*** of Q
    mark and enqueue all (unvisited) neighbours of ***w***
}



BFS: 0 1 2 3 4 5 6 7

- To implement BFS
  - Dynamically linked queue (pg 4.7, 4.8)

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




```
void bfs(int v)
{
    nodePointer w;
    front = rear = NULL;  /* initialize queue */
    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;
            }
        }
    }
}
```

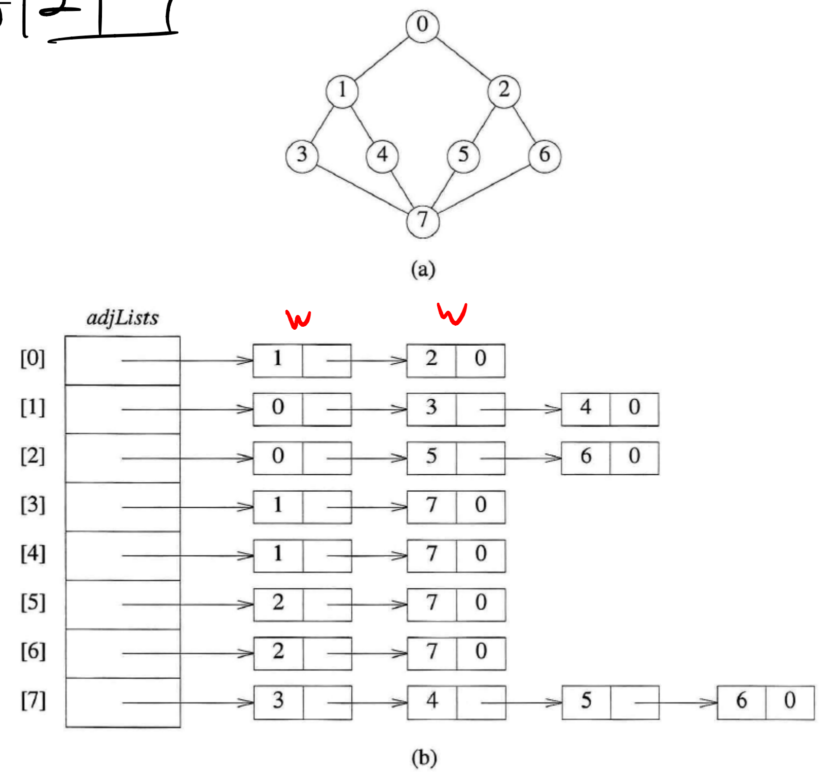


Figure 6.16: Graph *G* and its adjacency lists

Program 6.2: Breadth first search of a graph

bfs(0)

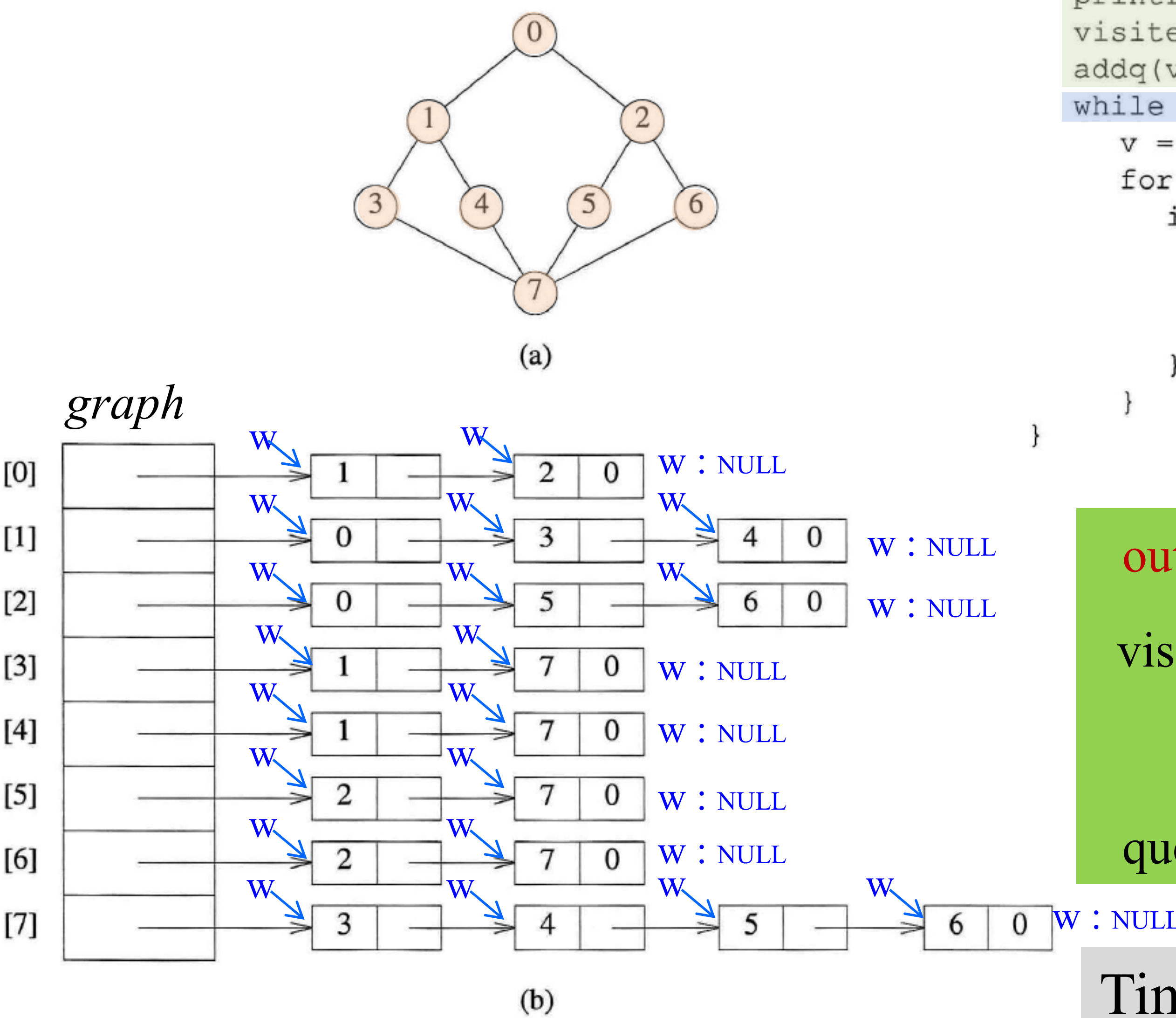


Figure 6.16: Graph G and its adjacency lists

```
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;
         }
      }
}
```

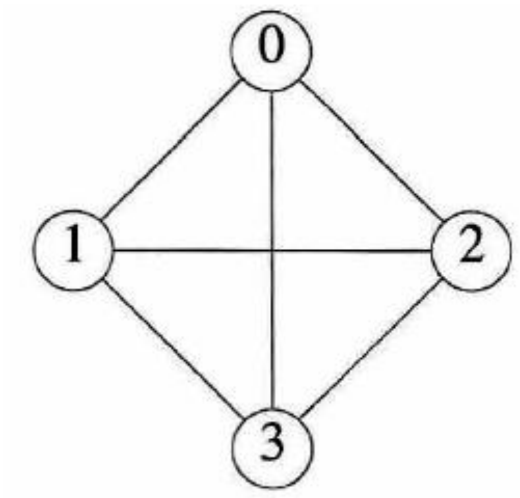
output: 0 1 2 3 4 5 6 7

visited

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

queue:

Time complexity : O(...)



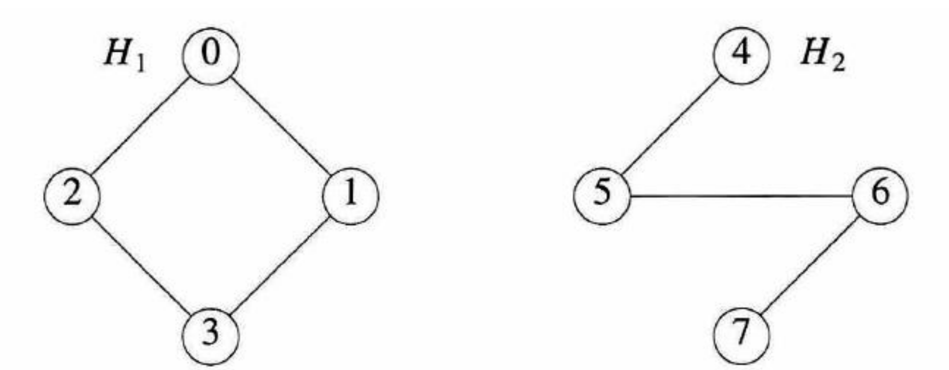
ADJACENCY LIST

0 -> 1 -> 2 -> 3 -> NULL
1 -> 2 -> 3 -> 0 -> NULL
2 -> 3 -> 1 -> 0 -> NULL
3 -> 2 -> 1 -> 0 -> NULL

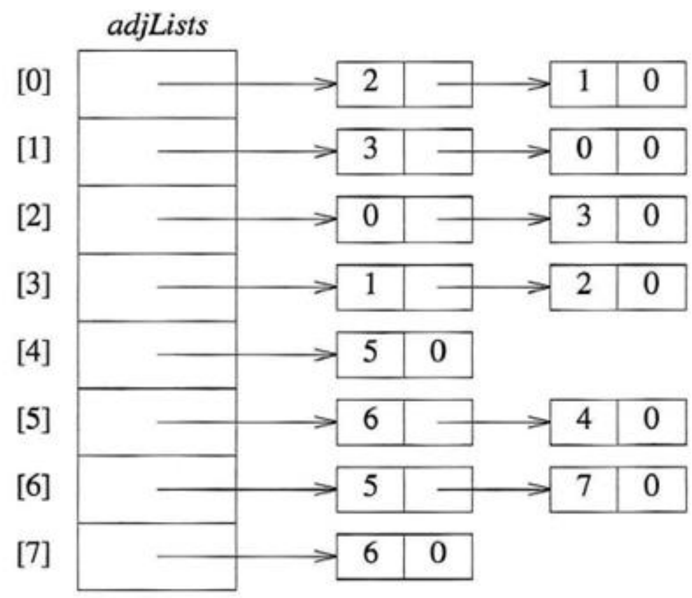
0 → 1 → 2 → 3

6.2.3 Connected Components

- Determining whether or not an undirected graph is connected :
  - Call *dfs*(0) or *bfs*(0) Conected면 어떤 vertex든 call 한번하면 다 돌 수있음
  - Determine if there are any unvisited vertices



G_4

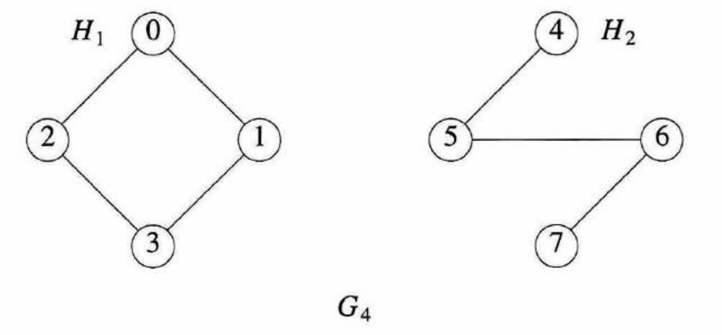


(c) G_4

- List the connected components of a graph

```
void connect(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



Adjacency list: O(n+e)
adjacency matirx: O(n^2)

6.2.4 Spanning Trees

- Any tree that consists solely of edges in *G* and that includes all the vertices in *G*
  - $E(G)$: T (tree edges) + N (nontree edges)

Vertex-1

vertex 4개
Edge 3개

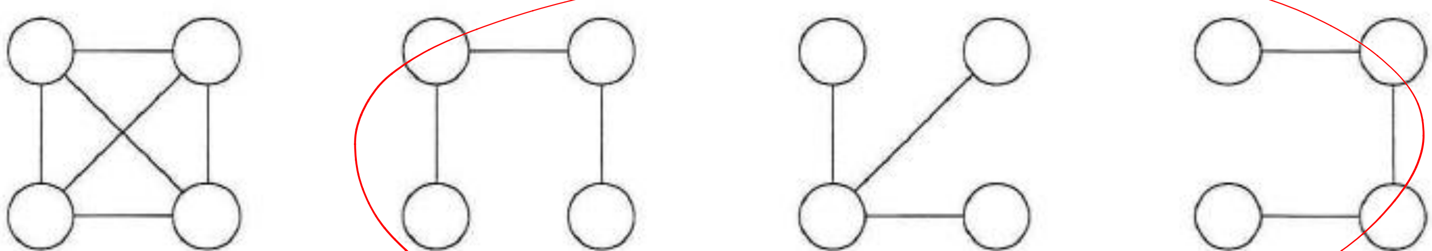
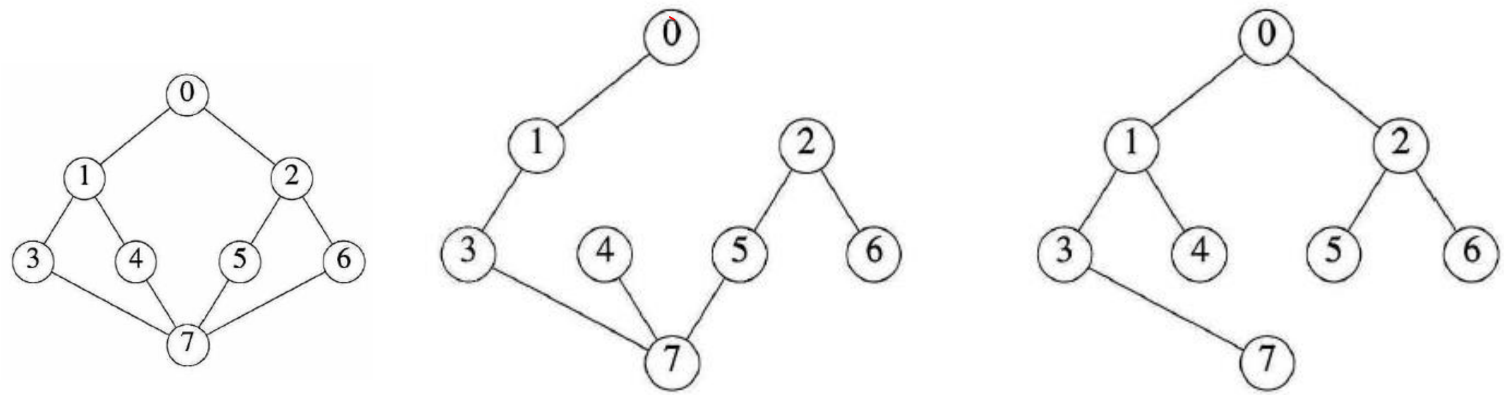


Figure 6.17: A complete graph and three of its spanning trees

- Use *dfs* or *bfs* to create a spanning tree
  - Depth first spanning tree
  - Breadth first spanning tree

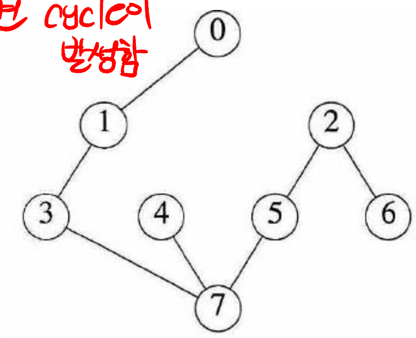


(a) *DFS* (0) spanning tree

(b) *BFS* (0) spanning tree

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

- Properties of spanning trees
  - If we add a nontree edge (v,w) into any spanning tree T, the result is a cycle 아무 Edge나 추가하면 cycle이 발생함
    - Ex) Add the nontree edge (7, 6)



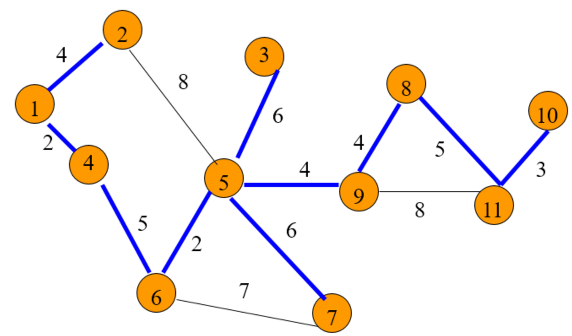
  - A spanning tree is a **minimal subgraph G'** of G such that V(G') = V(G) and G' is **connected**; Connected한 minimal subgraph
    - Subgraph with the fewest number of edges
  - A spanning tree with n vertices has $\boldsymbol{n}$**-1** edges
- Application
  - The design of communication networks

6.3 Minimum Cost Spanning Trees

- Cost of a spanning tree
  - Sum of the costs (weights) of the edges

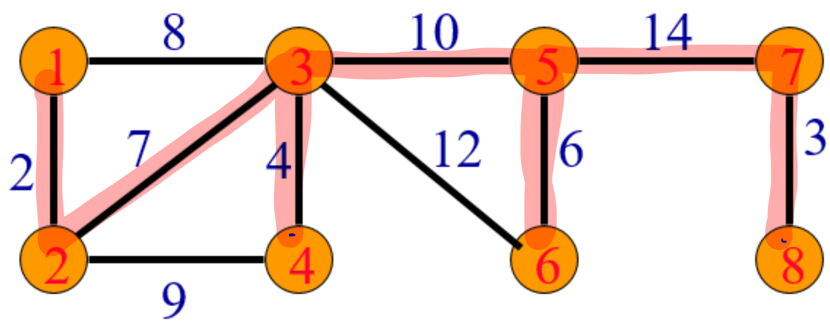
파란색 선

- Minimum cost spanning tree
  - A spanning tree of least cost



Spanning tree cost = 41.

Vertex 8개
Edge 7개



- To obtain a minimum cost spanning tree
  - Kruskal’s method
  - Prim’s method
  - Sollin’s method

- All three use an algorithm design strategy called the *greedy method*

- Greedy method
  – Construct an optimal solution in stages 최적의 solution을 찾음
  – At each stage, make the best decision possible
    - based on either **a least cost or a highest profit** criterion 최소비용 or 최대이익
  – Make sure the decision will result in a feasible solution
    - it cannot change this decision later
- Constraints
  – Must use only edges within the graph
  – Must use exactly n-1 edges
  – May not use edges that produce a cycle

6.3.1 Kruskal's Algorithm

- Algorithm
  - Build a min-cost spanning tree T by adding edges to T one at a time
  - Select the edges for inclusion in T in nondecreasing order of their of cost
  - An edge is added to T if it does not form a cycle with the edges that are already in T

- Since G is connected and has $n > 0$ vertices
  - exactly n-1 edges will be selected

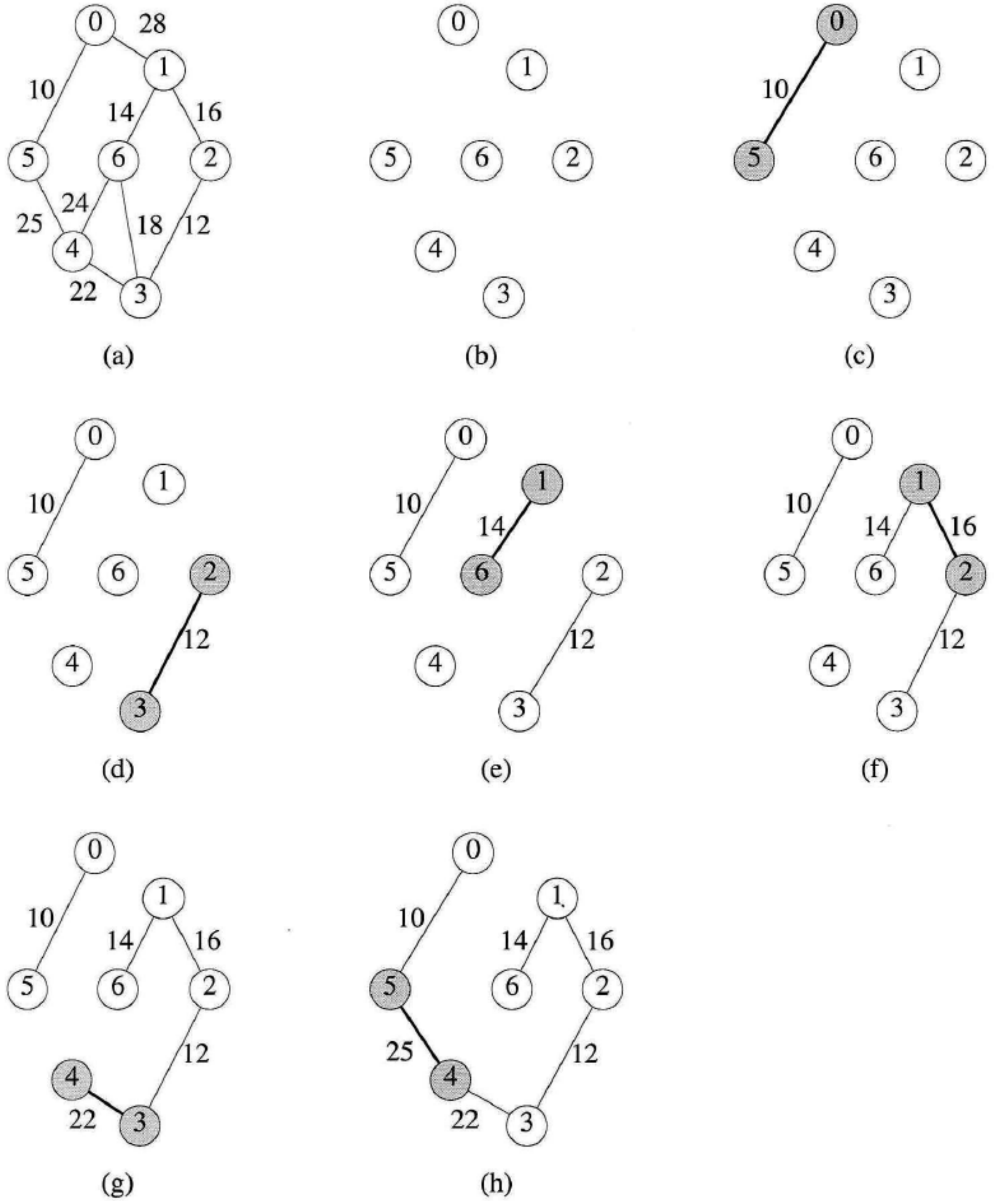


Figure 6.22: Stages in Kruskal's algorithm

```
T = {};   // T is the set of tree edges
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

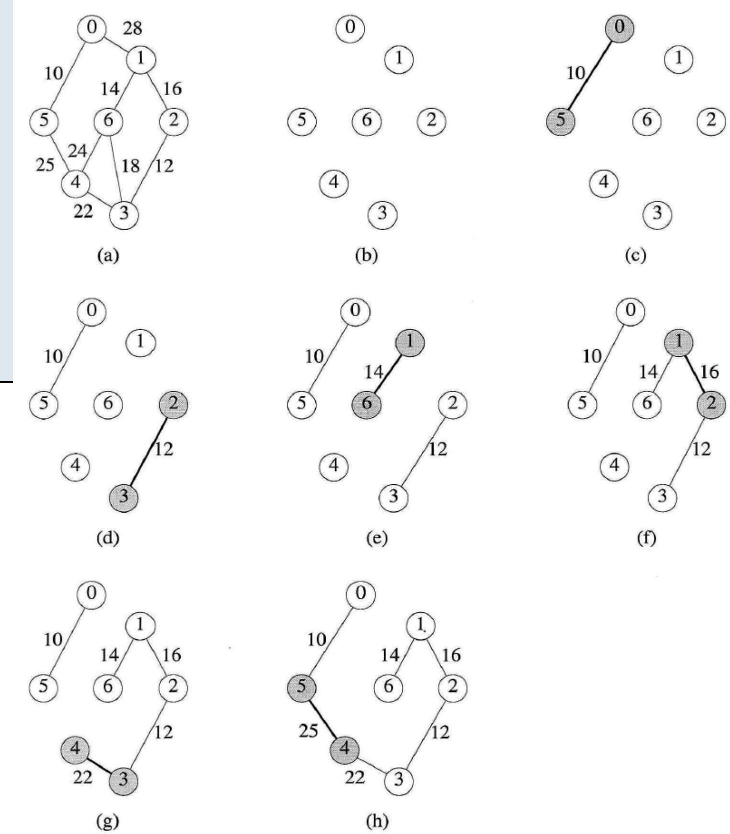


Figure 6.22: Stages in Kruskal's algorithm

6.3.2 Prim's Algorithm

- Algorithm
  - Build a minimum cost spanning tree *T* by adding edges to *T* one at a time.
  - At each stage, add a least cost edge to *T* such that the set of selected edges is also a tree.
  - Repeat the edge addition step until *T* contains n-1 edges.

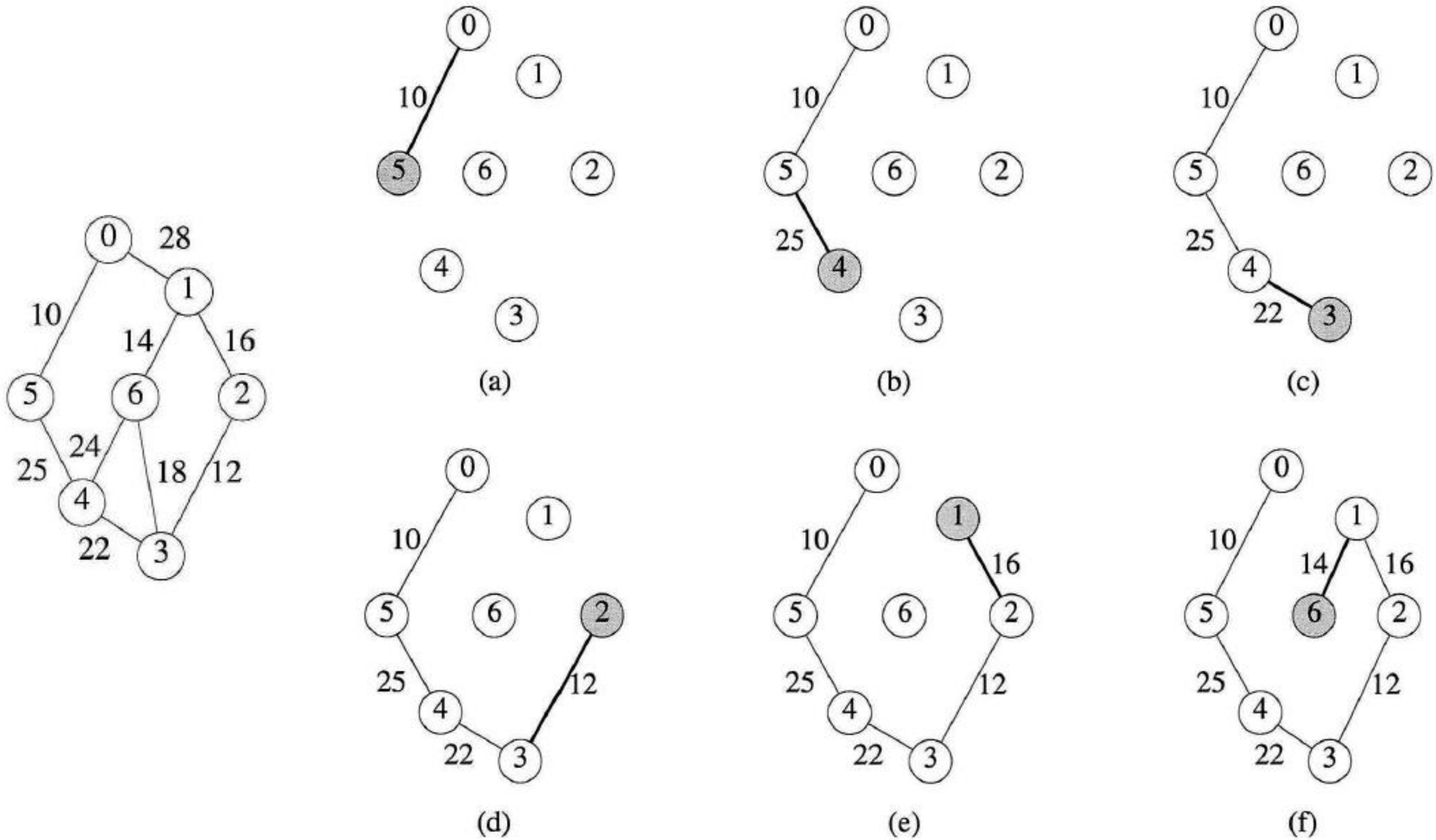
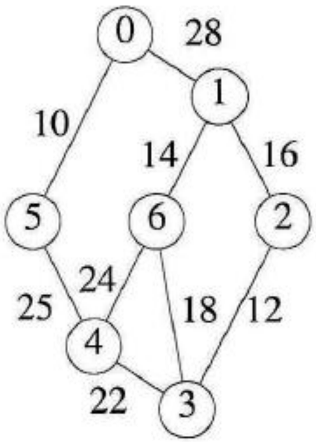


Figure 6.24: Stages in Prim's algorithm




```
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");
```

→ vertex 집합

Program 6.8: Prim's algorithm

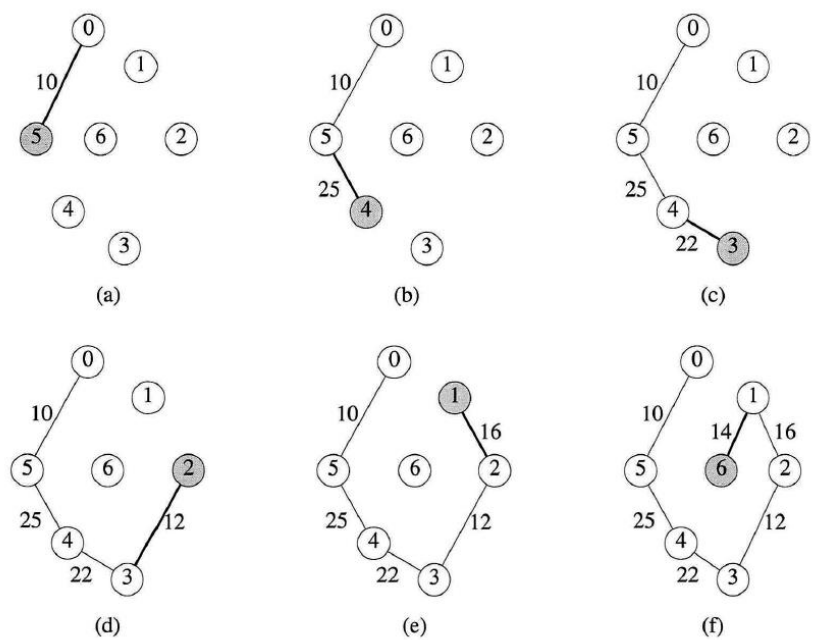


Figure 6.24: Stages in Prim's algorithm

GRAPHS

6.1 The Graph Abstract Data Type

6.2 Elementary Graph Operations

6.3 Minimum Cost Spanning Trees