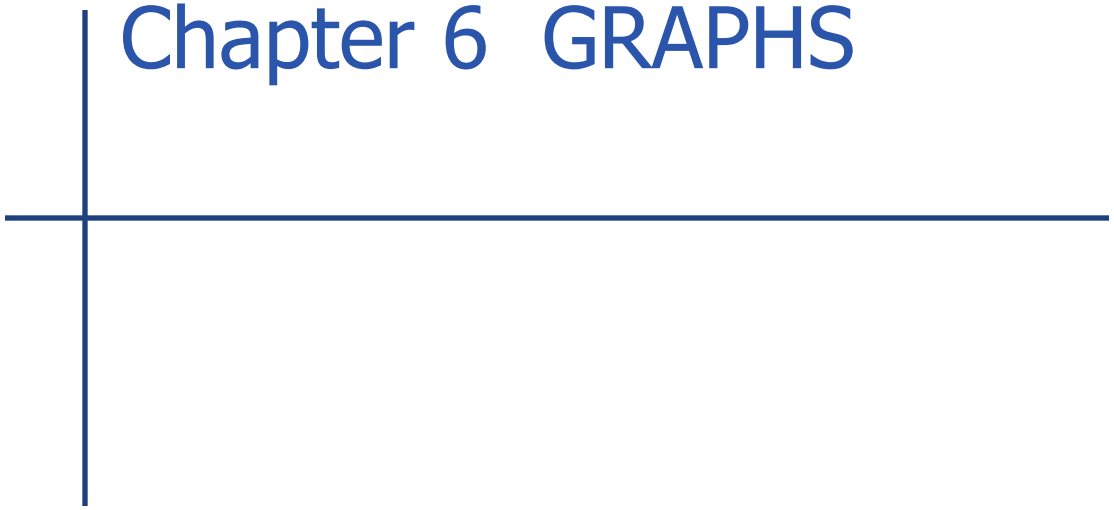
Chapter 6 GRAPHS



6.1 THE GRAPH ABSTRACT DATA TYPE

6.1.1 Introduction

The first recorded evidence of the use of graphs dates back to 1736 when *Leonhard Euler* used them to solve the classical *Königsberg* bridge problem.

The Königberg program is to determine whether, starting at one land area, it is possible to walk across all the bridges exactly once in returning to the starting land area. (Figure 6.1(a))

Euler solved the problem by representing the land areas as vertices and the bridges as edges in a graph. (Figure 6.1(b))

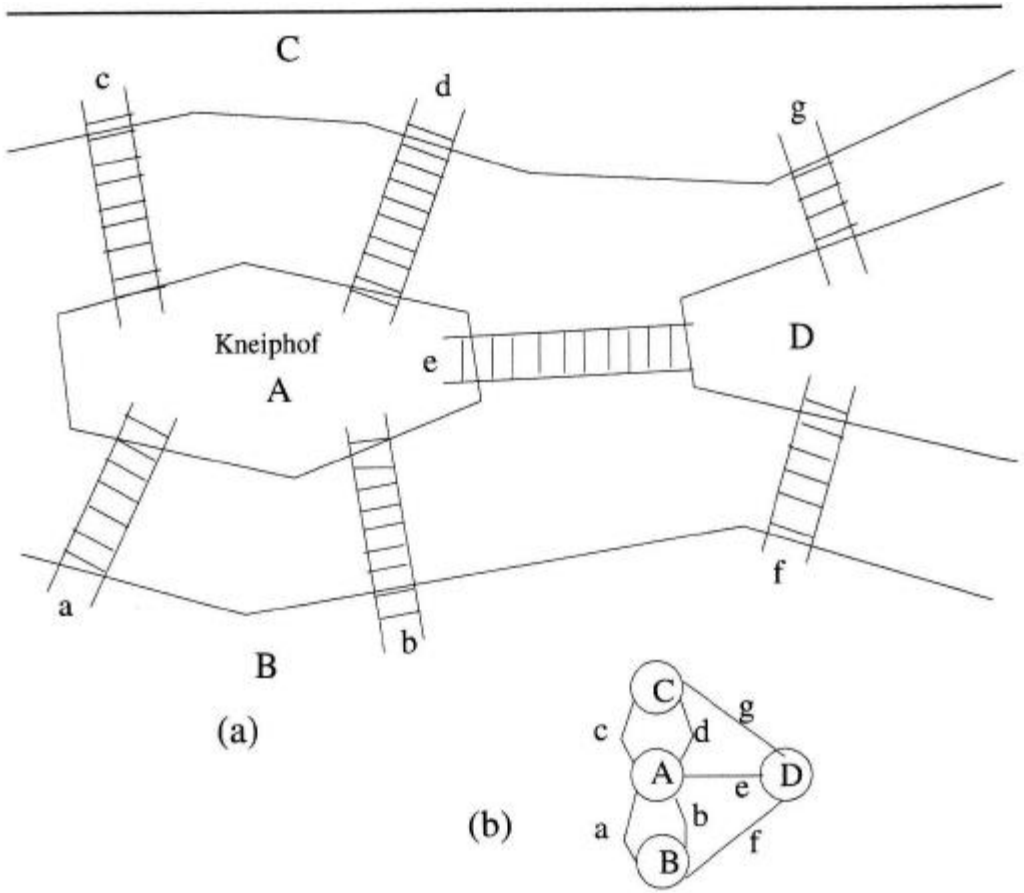


Figure 6.1: (a) Section of the river Pregel in Königsberg; (b) Euler's graph

Defining the *degree* of a vertex to be the number of edges incident to it,
Euler showed that there is a walk starting at any vertex, going through each edge exactly once and terminating at the start vertex iff the degree of each vertex is even.

A walk that does this is called *Eulerian.*

Graphs are the most widely used mathematical structure.

Applications

- Electrical circuit analysis
- Finding shortest routes
- Project planning
- Identification of chemical compounds
- Network flow design
- Gene/protein interactions, etc.

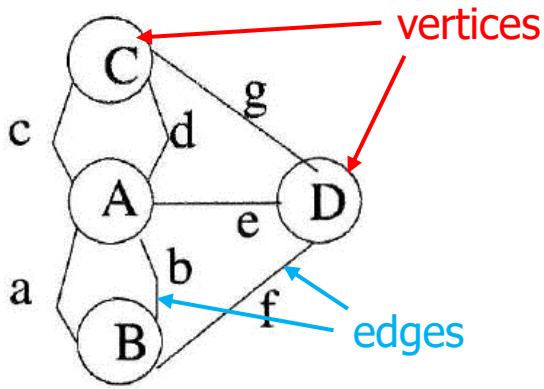
6.1.2 Definitions

A graph, G, consists of two sets :

V(G), a finite, nonempty set of vertices, and

E(G), a set of pairs of vertices called edges.

We may write **G=(V,E)** to represent a graph.



An undirected graph – the pair of vertices representing any edge is unordered.

e.g., the pairs (v_0, v_1) and (v_1, v_0) represent the same edge.

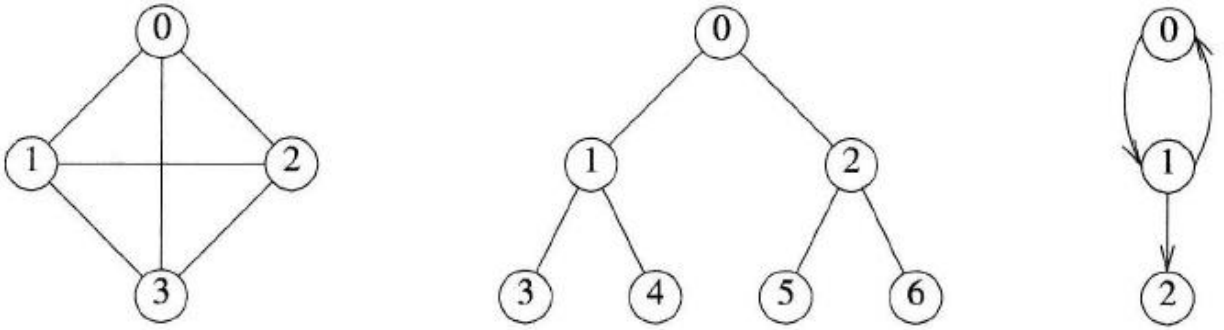
A directed graph – each edge is represented by a directed pair.

e.g., the pair $< v_0, v_1 >$ represents an edge in which

v_0 is the tail and v_1 is the head.

Therefore $< v_0, v_1 >$ and $< v_1, v_0 >$ represent two different edges.

[Figure 6.2] Three sample graphs



The set representation of each of these graphs is :

$V(G_1) = \{0, 1, 2, 3\}$; $E(G_1) = \{(0,1), (0,2), (0,3), (1,2), (1,3), (2,3)\}$

$V(G_2) = \{0, 1, 2, 3, 4, 5, 6\}$; $E(G_2) = \{(0,1), (0,2), (1,3), (1,4), (2,5), (2,6)\}$

$V(G_3) = \{0, 1, 2\}$; $E(G_3) = \{< 0,1 >, < 1,0 >, < 1,2 >\}$

Notice that G_2 is a tree.

We can define trees as a special case of graphs.

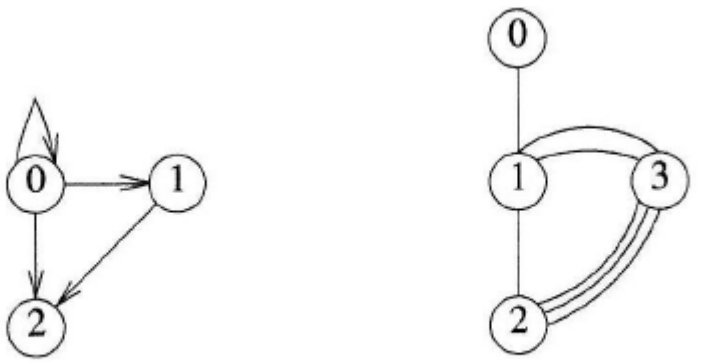
Restrictions imposed on graphs :

1. No *self* loops - A graph may not have an edge from a vertex, v, back to itself.

2. No multiple edges - A graph may not have multiple occurrences of the same edge.

   If we remove this restriction, we obtain a data object referred to as a *multigraph.*

[Figure 6.3] Examples of graphlike structures

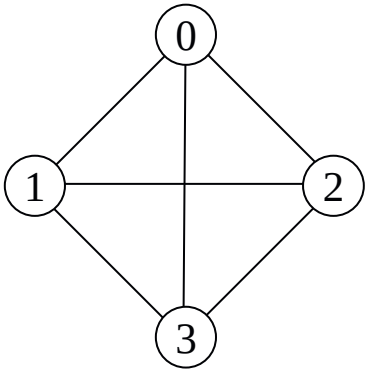


(a) Graph with a self edge (b) Multigraph

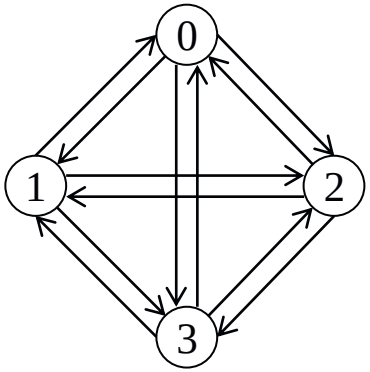
Complete graph

For an undirected graph,
a complete graph with n vertices has $n(n-1)/2$ different edges.

For a directed graph,
a complete graph with n vertices has $n(n-1)$ different edges.

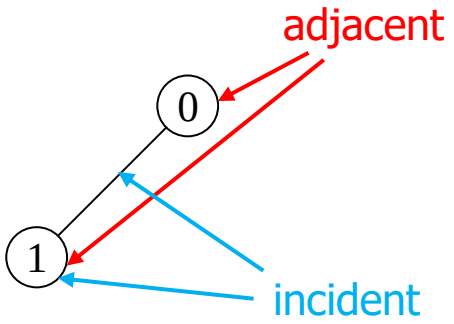


Complete undirected graph

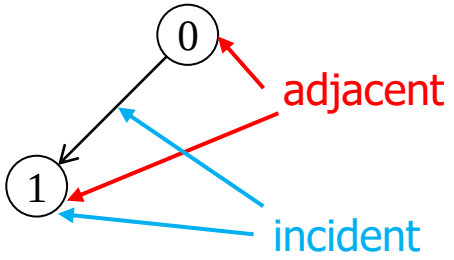


Complete directed graph

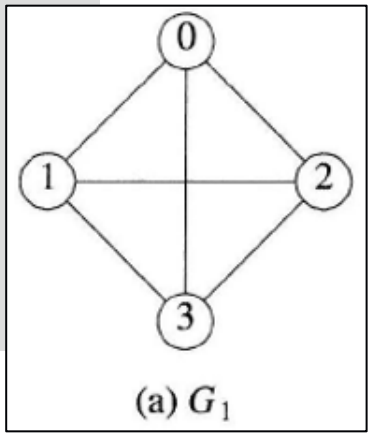
If (v_0, v_1) is an edge in an undirected graph,
then the vertices v_0 and v_1 are *adjacent* and
the edge (v_0, v_1) is *incident* on vertices v_0 and v_1.



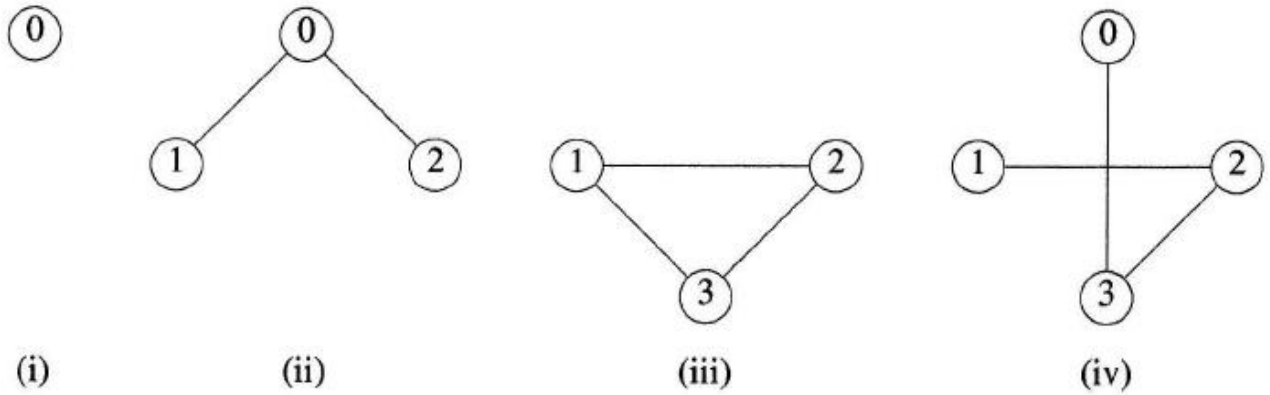
If $< v_0, v_1 >$ is a directed edge,
then vertex v_0 is *adjacent to* vertex v_1, while v_1 is *adjacent from* vertex v_0.
The edge $< v_0, v_1 >$ is incident to v_0 and v_1.



A *subgraph* of G is a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$.

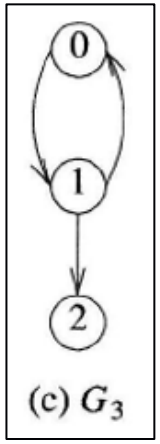


(a) G_1

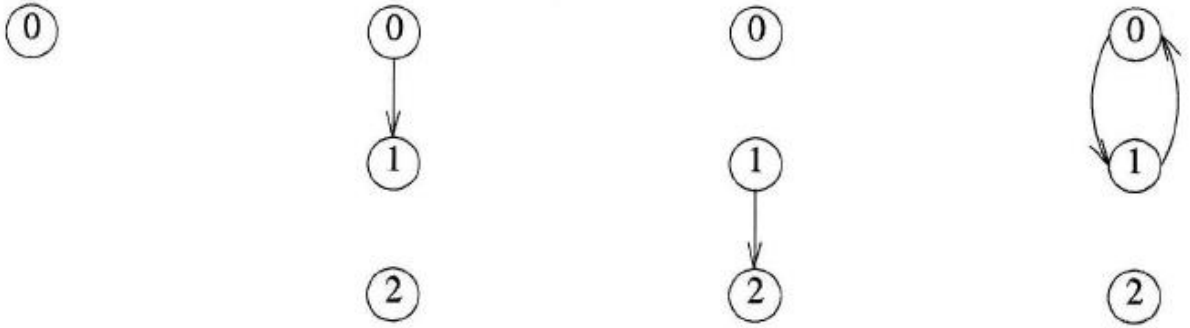


(i) (ii) (iii) (iv)

(a) Some of the subgraphs of G_1



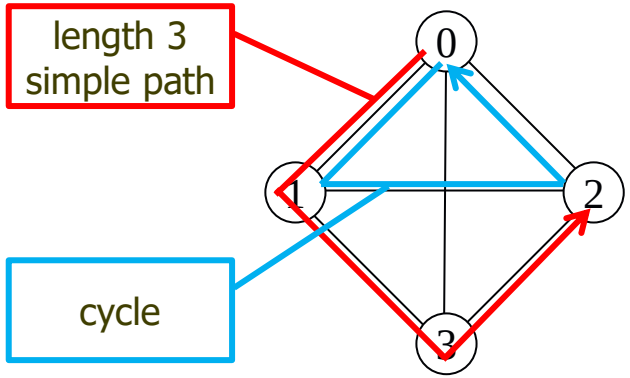
(c) G_3



(i) (ii) (iii) (iv)

(b) Some of the subgraphs of G_3

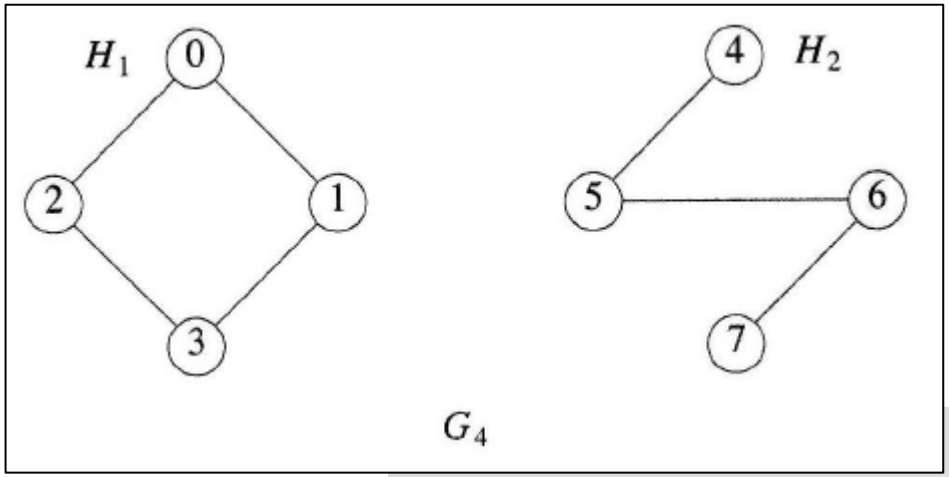
- A path from vertex v_p to a vertex v_q in a graph, G, is a sequence of vertices, $v_p, v_{i1}, v_{i2}, \ldots, v_{in}, v_q$, such that $(v_p, v_{i1}), (v_{i1}, v_{i2}), \ldots, (v_{in}, v_q)$ are edges in $E(G)$. If G' is a directed graph, then the path consists of $< v_p, v_{i1} >, < v_{i1}, v_{i2} >, \ldots, < v_{in}, v_q >$ edges in $E(G')$.
- The length of a path is the number of edges on it.
- A simple path is a path in which all vertices except possibly the first and last are distinct.
- A cycle is a simple path in which the first and last vertices are the same.



- A path (0, 1), (1, 3), (3, 2), also written as 0, 1, 3, 2, is of length 3.
- A path 0, 1, 3, 2 is a simple path.
- A path 0, 1, 2, 0 is a cycle.

- In an undirected graph G, two vertices, v_0 and v_1 are *connected* iff there is a path in G from v_0 to v_1.
- An undirected graph G is connected iff, for every pair of distinct vertices v_i, v_j, there is a path from v_i to v_j in G.
- A connected component, or simply a component, of an undirected graph is a *maximal* connected subgraph.
- A *tree* is a connected acyclic (i.e., has no cycles) graph.

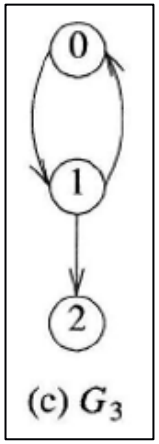
[Figure 6.5] A graph with two connected components



G_4 is not connected.

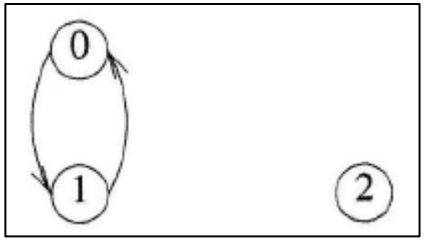
- A directed graph is *strongly connected* iff, for every pair of distinct vertices v_i, v_j in $V(G)$, there is a directed path from v_i to v_j and from v_j to v_i.
- A strongly connected component is a maximal subgraph which is strongly connected.
- The degree of a vertex is the number of edges incident to that vertex.
- For a directed graph, we define the in-degree of a vertex v as the number of edges that have v as the head, and the out-degree of a vertex v as the number of edges that have v as the tail.
- If d_i is the degree of a vertex i in a graph G with n vertices and e edges, then the number of edges is

$$e = \frac{1}{2}\sum_{i=0}^{n-1} d_i.$$



(c) G_3

[Figure 6.6] Strongly connected components of G_3



[ADT 6.1] Abstract data type *Graph*

ADT *Graph* is

objects : a nonempty set of vertices and a set of undirected edges,
where each edge is a pair of vertices.

functions : for all $graph \in Graph$, v, v_1, and $v_2 \in Vertices$

Graph Create() ::= **return** an empty graph.

Graph InsertVertex(graph, v) ::= **return** a graph with v inserted. v has no incident edges.

Graph InsertEdge(graph, v_1, v_2) ::= **return** a graph with a new edge between v_1 and v_2.

Graph DeleteVertex(graph, v) ::= **return** a graph in which v and all edges incident to it are removed.

Graph DeleteEdge(graph, v_1, v_2) ::= **return** a graph in which the edge (v_1, v_2) is removed. Leave the incident vertices in the graph.

Boolean IsEmpty(graph) ::= **if** (graph == empty graph) **return** TRUE **else return** FALSE.

List Adjacent(graph, v) ::= **return** a list of all vertices that are adjacent to v.

6.1.3 Graph Representations

6.1.3.1 Adjacency Matrix

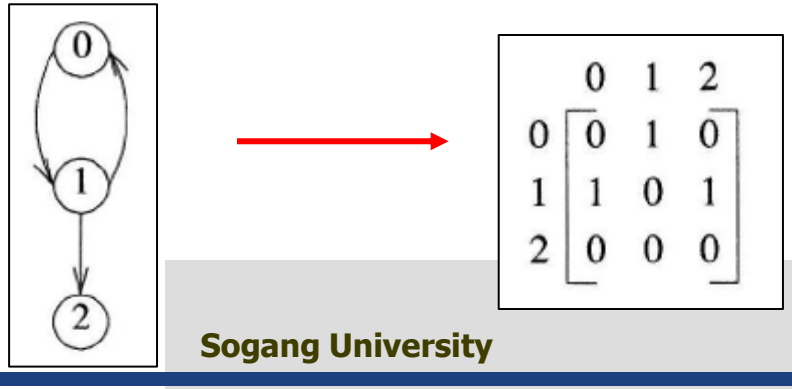
Let $G = (V, E)$ be a graph with n vertices, $n \geq 1$.
The adjacency matrix of G is a two-dimensional $n \times n$ array, say *adj_mat*, defined as:

adj_mat[i][j] = 1 if the edge (v_i, v_j) is in $E(G)$
0 if there is no such edge.

The same definition can be used for a directed graph except that the edge $< v_i, v_j >$ is directed.

The adjacency matrix for an undirected graph is symmetric, since the edge (v_i, v_j) is in $E(G)$ iff the edge (v_j, v_i) is also in $E(G)$.

For undirected graphs, we can save space by storing only the upper or lower triangle of the matrix.



From the adjacency matrix, we can determine:

- if there is an edge connecting any two vertices.
- the degree of a vertex.

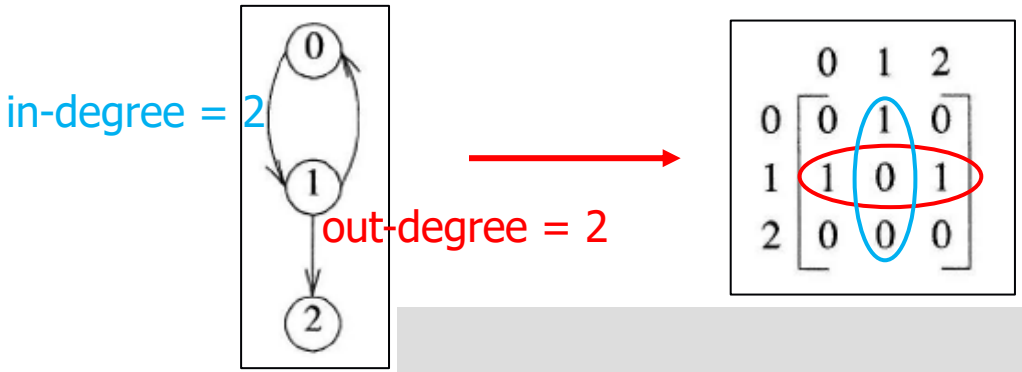
For an undirected graph,

the degree of any vertex i, is its row sum:

$$\sum_{j=0}^{n-1} adj\_mat[i][j]$$

For a directed graph,

the row sum is the out-degree, while the column sum is the in-degree.



Suppose we want to answer a nontrivial question about graph, such as,
How many edges are there in G, or, Is G connected?

Adjacency matrices will require at least O(n^2) time,
as $n^2 - n$ entries of the matrix (diagonal entries are zero)
have to be examined.

For a sparse graph, its adjacency matrix becomes sparse.
We might expect that the above questions would be answerable in
significantly less time, say O($e + n$) time,
where e is the number of edges in G, and $e \ll n^2/2$.

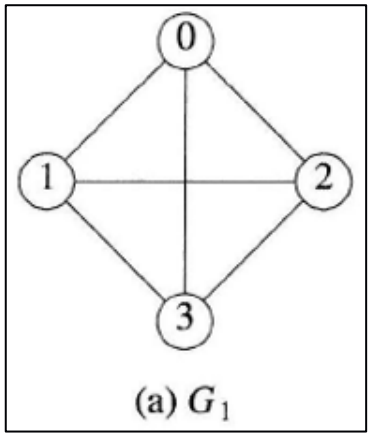
Adjacency Lists

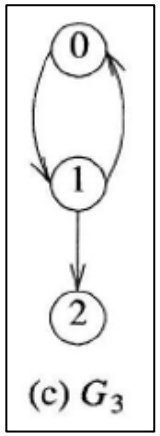
In this representation,
we replace the n rows of the adjacency matrix with n linked lists, one for each vertex in G.

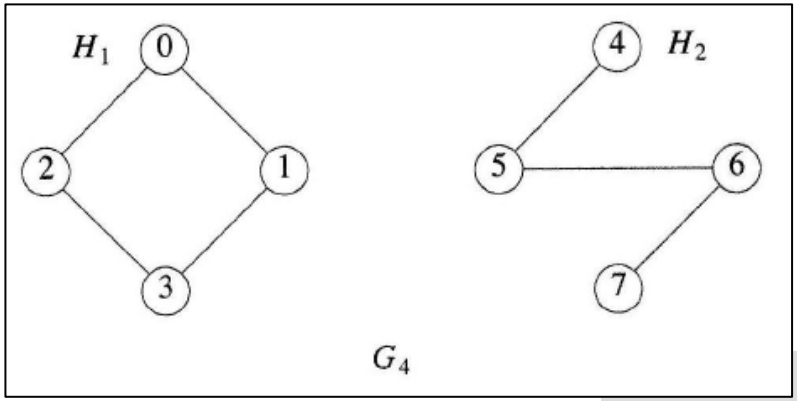
The nodes in chain i represent the vertices that are adjacent from vertex i.

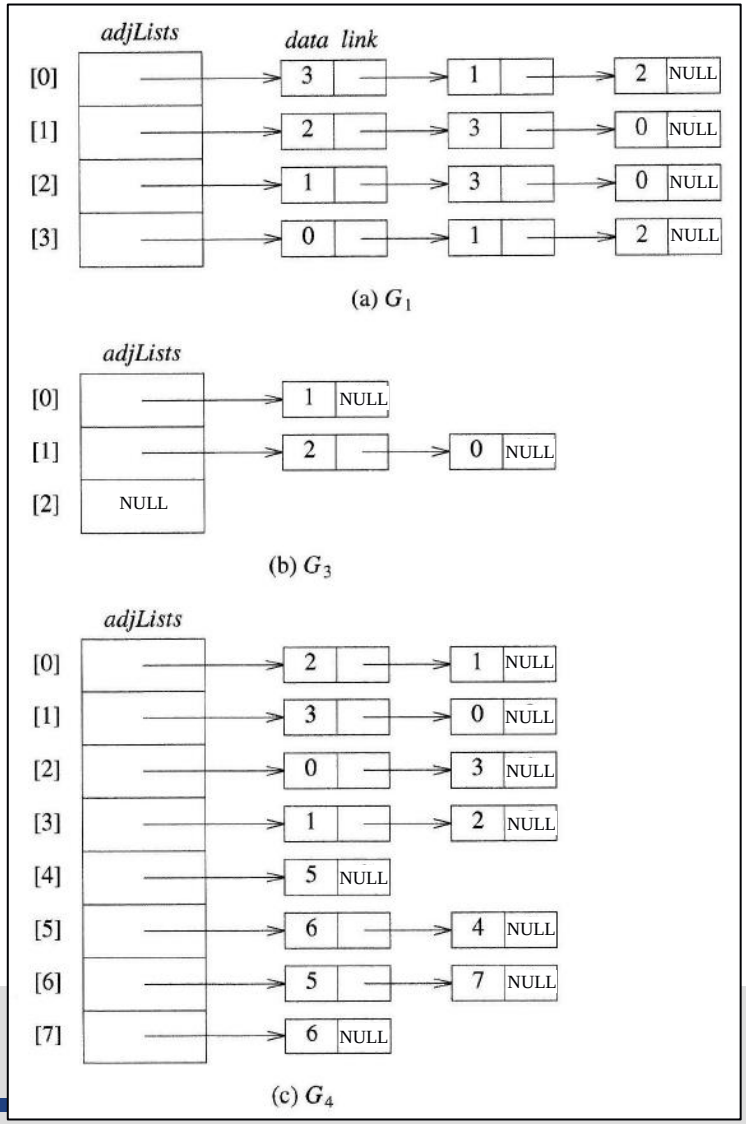
See Figure 6.8.

[Figure 6.8] Adjacency lists









The C declaration for the adjacency list representation :

```
#define MAX_VERTICES 50  /*maximum number of vertices*/
  typedef struct node *node_pointer;
  typedef struct node {
              int vertex;
              node_pointer link;
              };
  node_pointer graph[MAX_VERTICES];
  int n=0;  /* vertices currently in use */
```

In case of an undirected graph with n vertices and e edges,
this representation requires an array of size n and $2e$ list nodes.

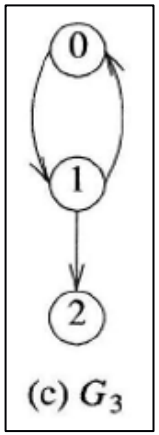
Each list node has two fields.

The degree of any vertex in an undirected graph may be determined by just counting the number of nodes in its adjacency list.

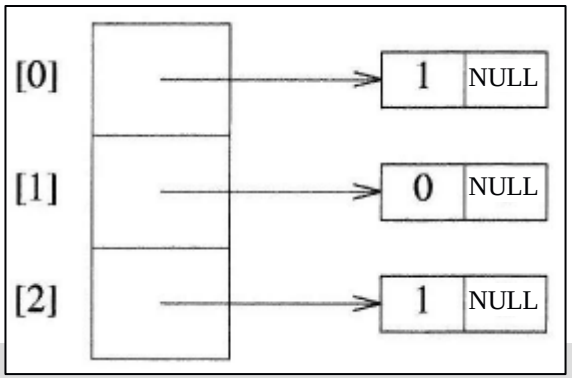
For a directed graph,

- the out-degree of any vertex may be determined by counting the number of nodes on its adjacency list.
- determining the in-degree of a vertex is a little more complex.

→ We can maintain another set of lists, called *inverse adjacency lists*, in addition to the adjacency lists.



[Figure 6.10] Inverse adjacency lists for G_3



Weighted Edges

Weights can be assigned to the edges of a graph.

These weights may represent the distance from one vertex to another or the cost of going from one vertex to an adjacent vertex.

In this case, we may modify our representation to keep the weight information.

A graph with weighted edges is called a *network*.

6.2 ELEMENTARY GRAPH OPERATIONS

Given an undirected graph, $G = (V, E)$, and a vertex, v, in $V(G)$, we wish to visit all vertices in G that are reachable from v, that is, all vertices that are connected to v.

We shall look at two ways of doing this:

- *Depth First Search*
- *Breadth First Search*

In our discussion of depth first search and breadth first search, we shall assume that the linked adjacency list representation for graphs is used.

6.2.1 Depth First Search

- We begin the search by visiting the start vertex, v.
- Next, we select an unvisited vertex, w, from v's adjacency list and carry out a depth first search on w.
- We preserve our current position in v's adjacency list by placing it on a **stack**.
- Eventually our search reaches a vertex, u, that has no unvisited vertices on its adjacency list.
- At this point, we remove a vertex from the stack and continue processing its adjacency list.
- Previously visited vertices are discarded;
  unvisited vertices are visited and placed on the stack.
- This search terminates when the stack is empty.

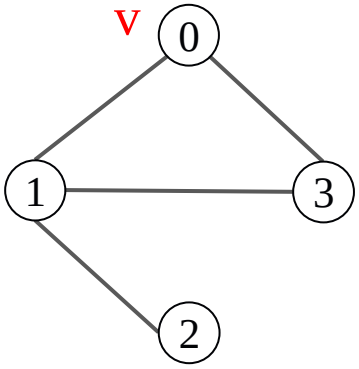
Declaration needed :

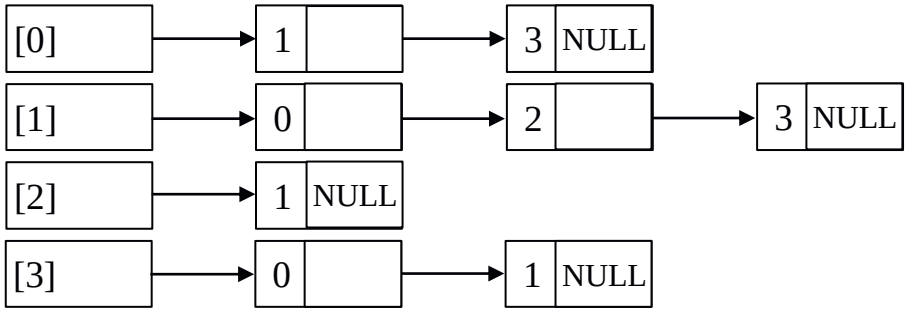
```
#define FALSE 0
#define TRUE 1
short int visited[MAX_VERTICES];
```

[Program 6.1] Depth first search

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

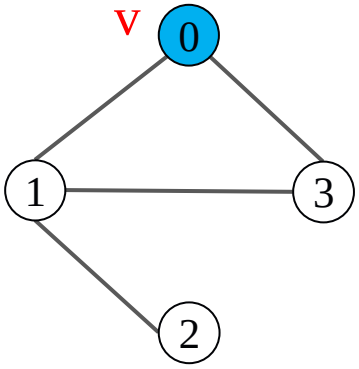
```
void dfs(int v)
{  /* depth first search of a graph beginning at v. */
        node_pointer w;
        visited[v] = TRUE;
        printf("%5d", v);
        for (w=graph[v]; w; w=w->link)
            if (!visited[w->vertex])
                dfs(w->vertex);
 }
```

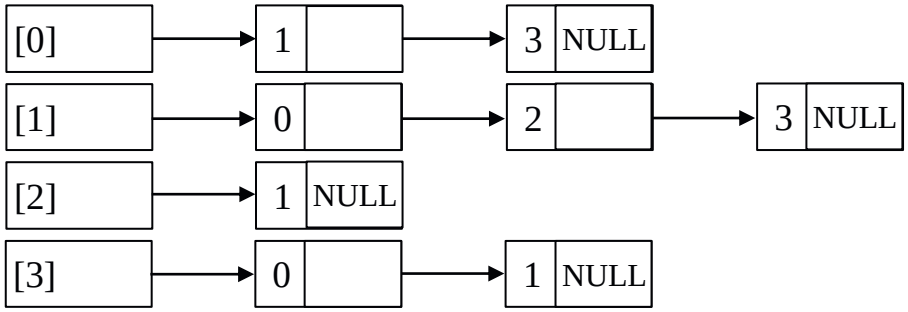




output:

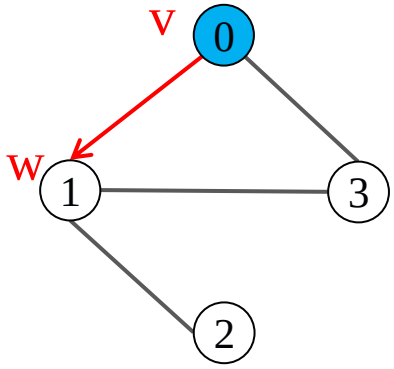
```
void dfs(int v)
{  /* depth first search of a graph beginning at v. */
        node_pointer w;
        visited[v] = TRUE;
        printf("%5d", v);
        for (w=graph[v]; w; w=w->link)
            if (!visited[w->vertex])
                dfs(w->vertex);
 }
```

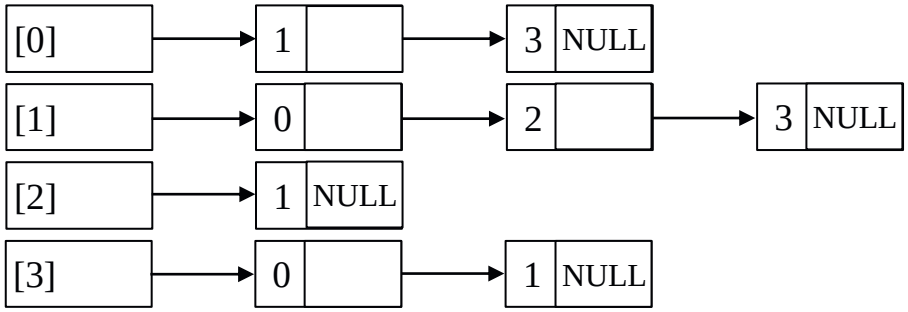




output: 0

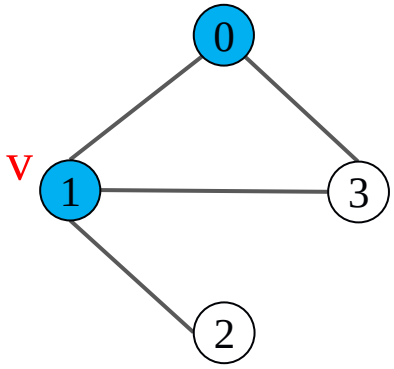
```
void dfs(int v)
{  /* depth first search of a graph beginning at v. */
        node_pointer w;
        visited[v] = TRUE;
        printf("%5d", v);
        for (w=graph[v]; w; w=w->link)
            if (!visited[w->vertex])
                dfs(w->vertex);
 }
```

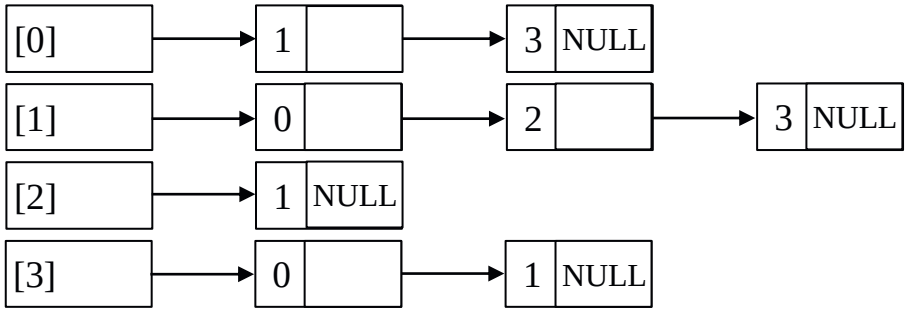




output: 0

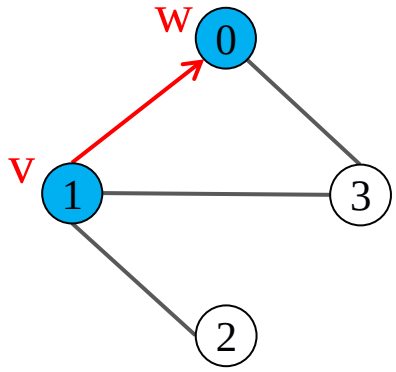
```
void dfs(int v)
{  /* depth first search of a graph beginning at v. */
        node_pointer w;
        visited[v] = TRUE;
        printf("%5d", v);
        for (w=graph[v]; w; w=w->link)
            if (!visited[w->vertex])
                dfs(w->vertex);
 }
```

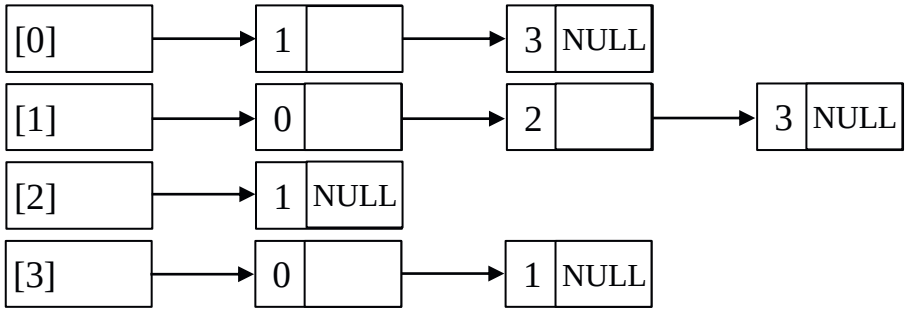




output: 0 1

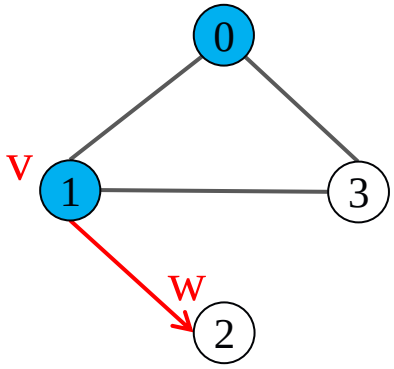
```
void dfs(int v)
{  /* depth first search of a graph beginning at v. */
        node_pointer w;
        visited[v] = TRUE;
        printf("%5d", v);
        for (w=graph[v]; w; w=w->link)
            if (!visited[w->vertex])
                dfs(w->vertex);
 }
```

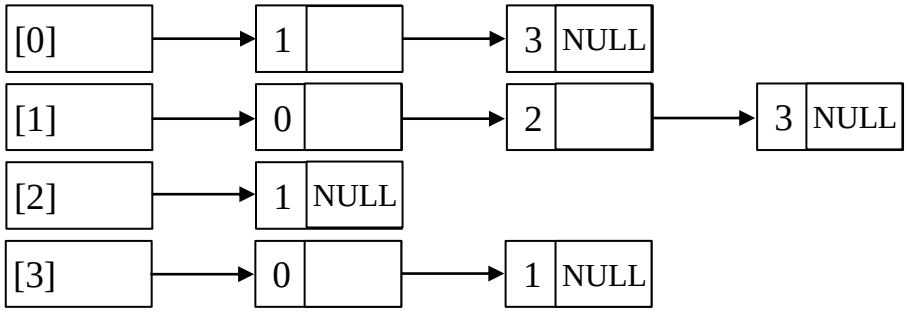




output: 0 1

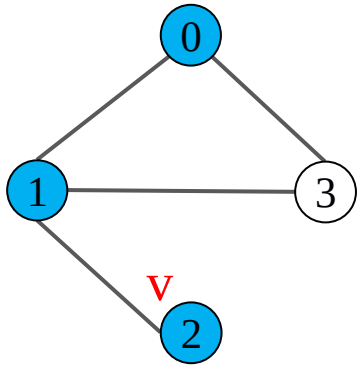
```
void dfs(int v)
{  /* depth first search of a graph beginning at v. */
        node_pointer w;
        visited[v] = TRUE;
        printf("%5d", v);
        for (w=graph[v]; w; w=w->link)
            if (!visited[w->vertex])
                dfs(w->vertex);
 }
```

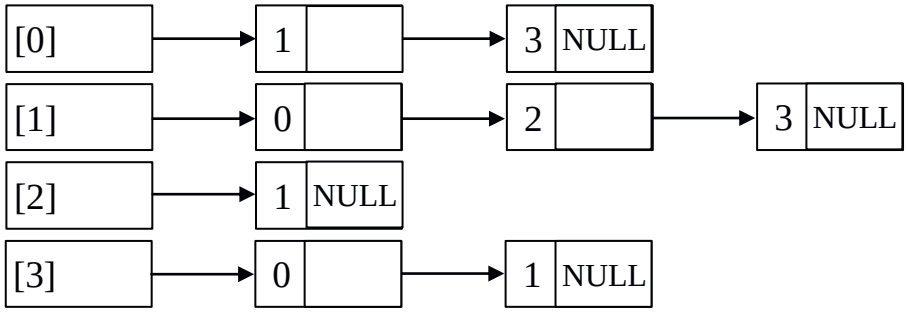




output: 0 1

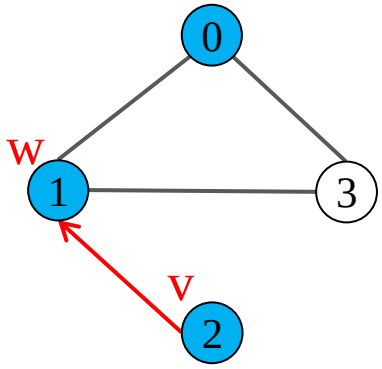
```
void dfs(int v)
{  /* depth first search of a graph beginning at v. */
        node_pointer w;
        visited[v] = TRUE;
        printf("%5d", v);
        for (w=graph[v]; w; w=w->link)
            if (!visited[w->vertex])
                dfs(w->vertex);
 }
```

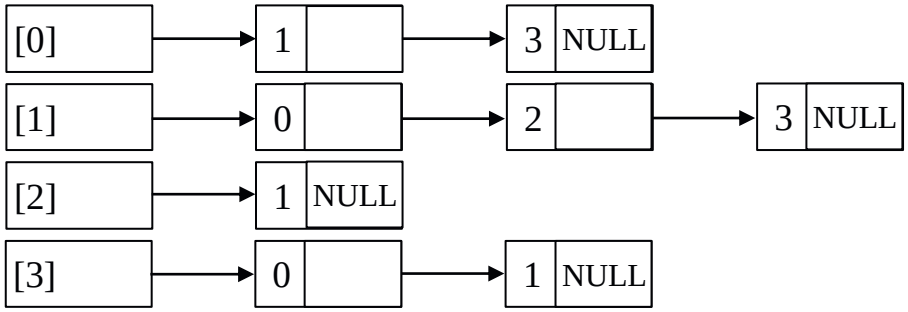




output: 0 1 2

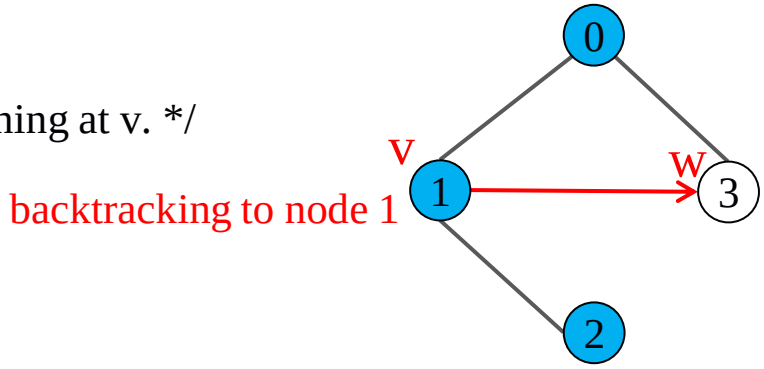
```
void dfs(int v)
{  /* depth first search of a graph beginning at v. */
        node_pointer w;
        visited[v] = TRUE;
        printf("%5d", v);
        for (w=graph[v]; w; w=w->link)
            if (!visited[w->vertex])
                dfs(w->vertex);
 }
```



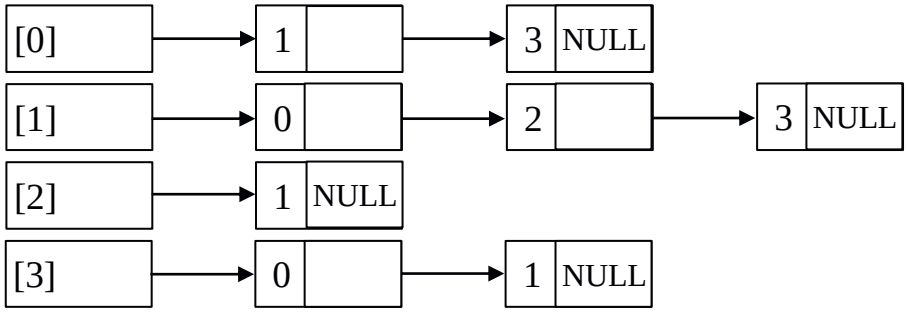


output: 0 1 2

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

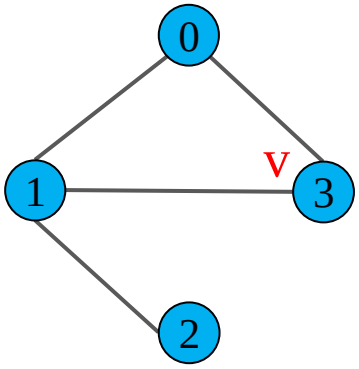


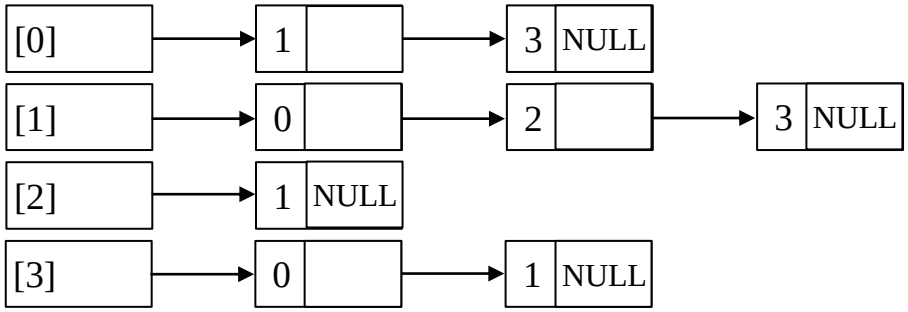
backtracking to node 1



output: 0 1 2

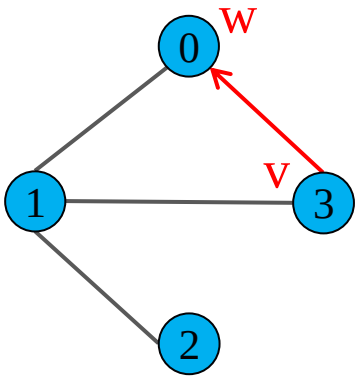
```
void dfs(int v)
{  /* depth first search of a graph beginning at v. */
        node_pointer w;
        visited[v] = TRUE;
        printf("%5d", v);
        for (w=graph[v]; w; w=w->link)
            if (!visited[w->vertex])
                dfs(w->vertex);
 }
```

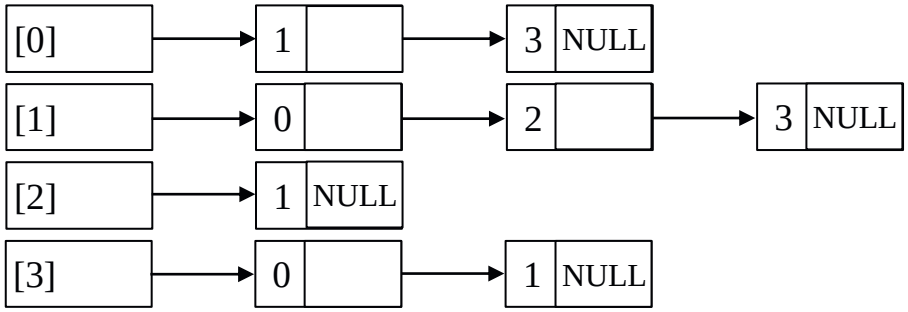




output: 0 1 2 3

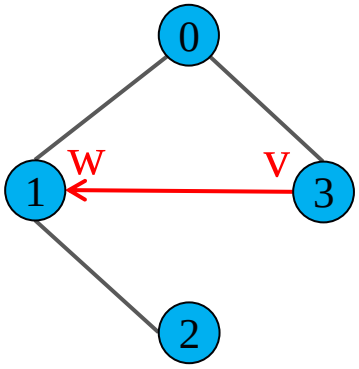
```
void dfs(int v)
{  /* depth first search of a graph beginning at v. */
        node_pointer w;
        visited[v] = TRUE;
        printf("%5d", v);
        for (w=graph[v]; w; w=w->link)
            if (!visited[w->vertex])
                dfs(w->vertex);
 }
```

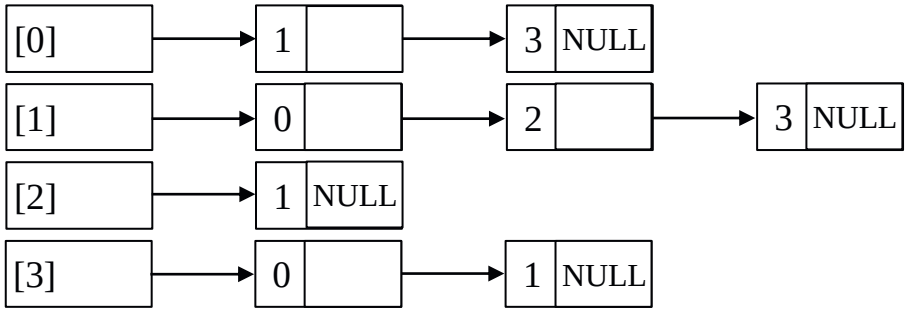




output: 0 1 2 3

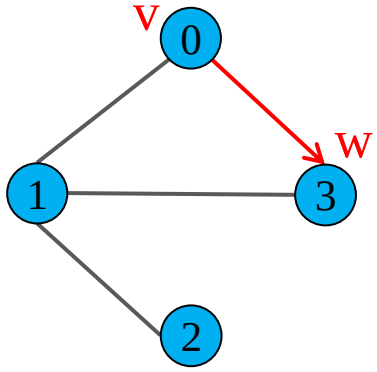
```
void dfs(int v)
{  /* depth first search of a graph beginning at v. */
        node_pointer w;
        visited[v] = TRUE;
        printf("%5d", v);
        for (w=graph[v]; w; w=w->link)
            if (!visited[w->vertex])
                dfs(w->vertex);
 }
```



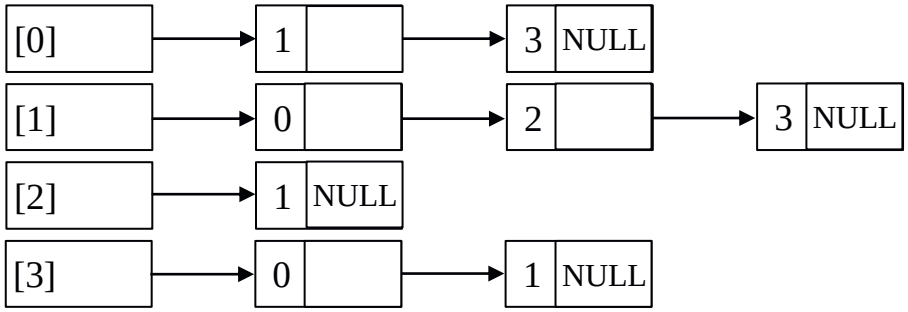


output: 0 1 2 3

backtracking to node 0




```
void dfs(int v)
{  /* depth first search of a graph beginning at v. */
        node_pointer 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 2 3

Analysis of *dfs* :

If we use adjacency lists,
since *dfs* examines each node in the adjacency lists at most once,
the time to complete the search is $O(e)$.

If we use adjacency matrix,
then determining all vertices adjacent to v requires $O(n)$ time.
Since we visit at most n vertices, the total time is $O(n^2)$.

6.2.2 Breadth First Search

- Breadth first search starts at vertex v and marks it as visited.
- It then visits each of the vertices on v's adjacency list.
- When we have visited all the vertices on v's adjacency list, we visit all the unvisited vertices that are adjacent to the first vertex on v's adjacency list.
- To implement this scheme, as we visit each vertex we place the vertex in a queue.
- When we have exhausted an adjacency list, we remove a vertex from the queue and proceed by examining each of the vertices on its adjacency list.
- Unvisited vertices are visited and then placed on the queue;
  visited vertices are ignored.
- When the queue is empty, the search is finished.

To implement breadth first search,
we use a *dynamically linked queue*.

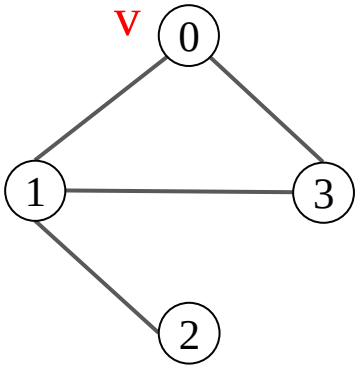
Necessary declarations:

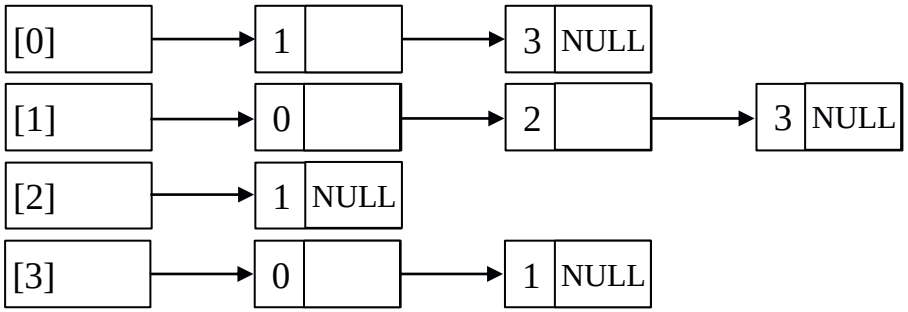
```
typedef struct queue *queue_pointer;
typedef struct queue {
        int vertex;
        queue_pointer link;
        };
queue_pointer front, rear;
void addq(int);
int deleteq();
```

[program 6.2] Breadth first search

```
void bfs(int v)
{ /* breadth first traversal of a graph, staring with node 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 */
    node_pointer 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;
            }
        }
    }
}
```

```
void bfs(int v)
{
    node_pointer 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;
            }
        }
    }
}
```

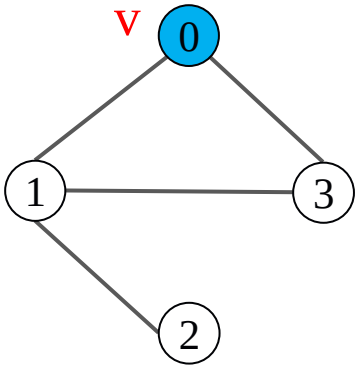


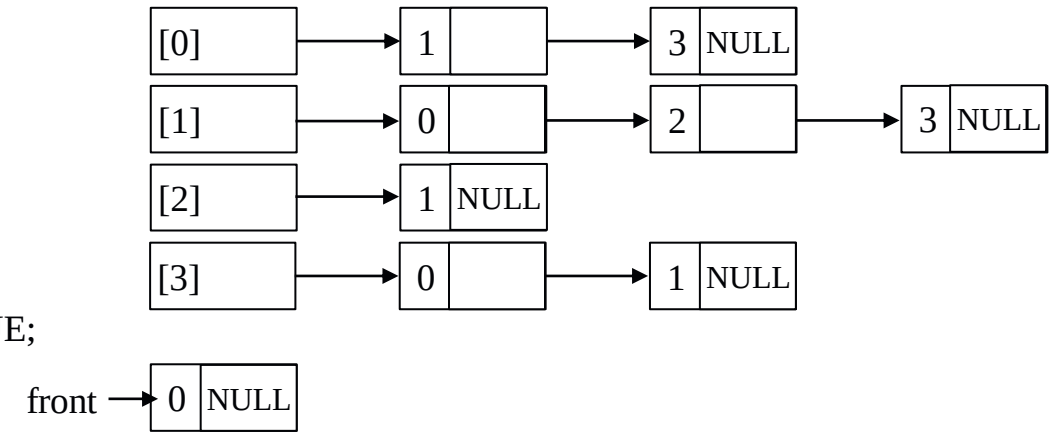


front = NULL

output:

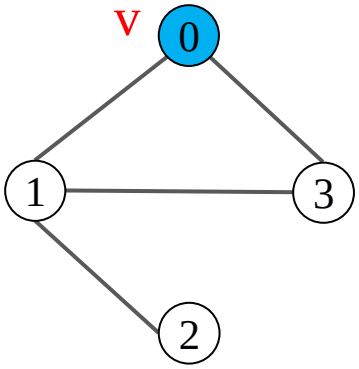
```
void bfs(int v)
{
    node_pointer w;
    front = rear = NULL; /* initialize queue */
    printf("%5d",v);
    visited[v] = TRUE;
    addq(v);
    while (front) {
        v = deleteq();
        for (w=graph[v]; w; w=w->link)
            if (!visited[w->vertex]) {
                printf("%5d",w->vertex);
                addq(w->vertex);
                visited[w->vertex] = TRUE;
            }
        }
    }
}
```

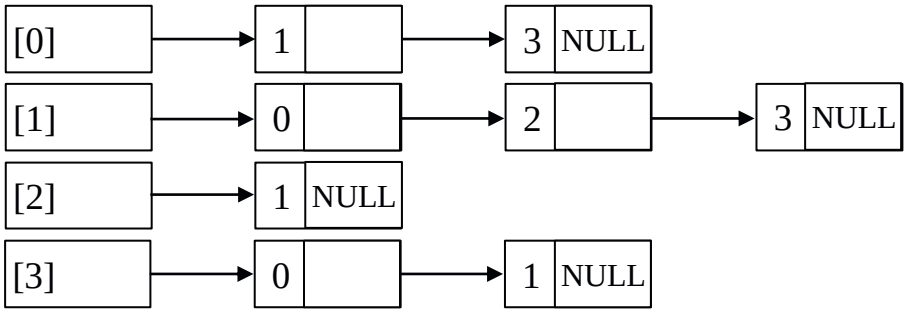




output: 0

```
void bfs(int v)
{
    node_pointer 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;
            }
        }
    }
}
```

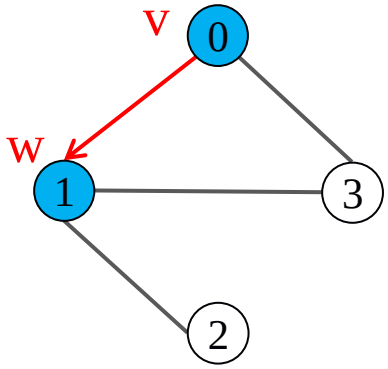


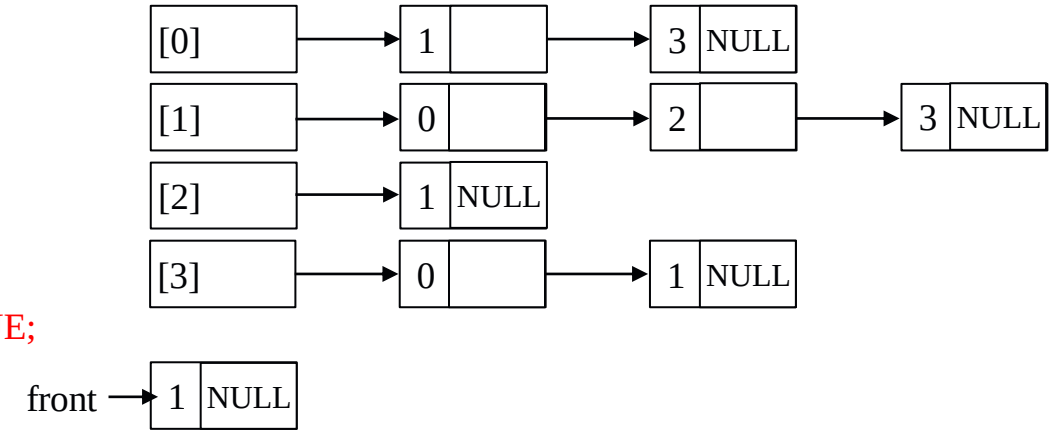


front = NULL

output: 0

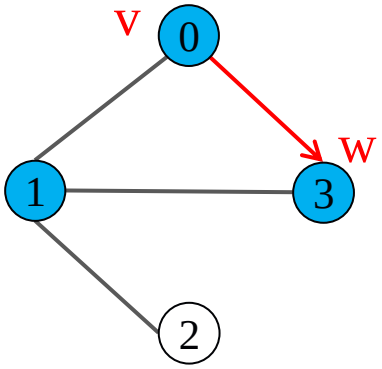
```
void bfs(int v)
{
     node_pointer w;
     front = rear = NULL; /* initialize queue */
     printf("%5d",v);
     visited[v] = TRUE;
     addq(v);
     while (front) {
          v = deleteq();
          for (w=graph[v]; w; w=w->link)
               if (!visited[w->vertex]) {
                    printf("%5d",w->vertex);
                    addq(w->vertex);
                    visited[w->vertex] = TRUE;
               }
          }
     }
}
```

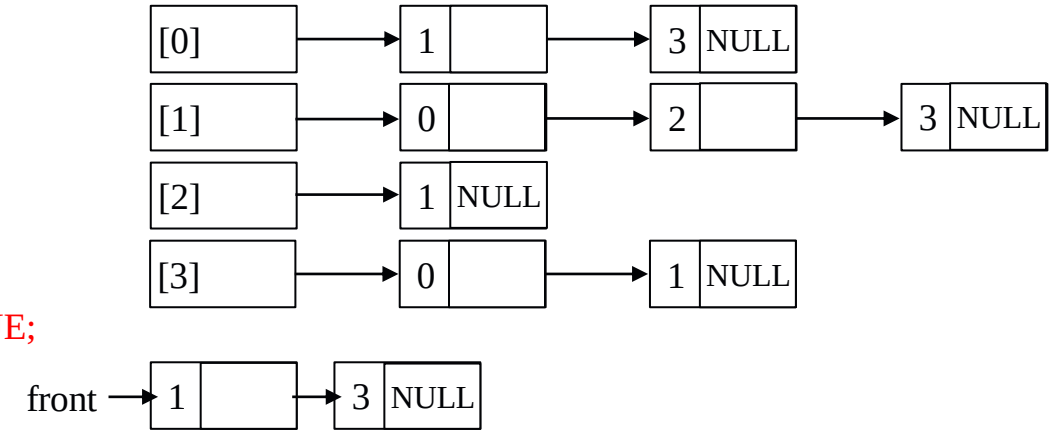




output: 0 1

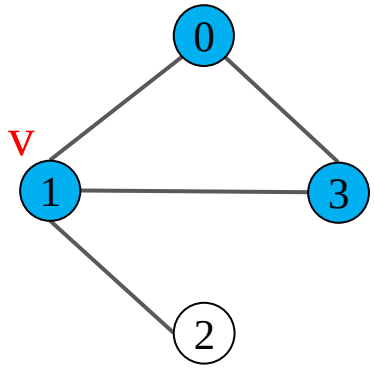
```
void bfs(int v)
{
     node_pointer w;
     front = rear = NULL; /* initialize queue */
     printf("%5d",v);
     visited[v] = TRUE;
     addq(v);
     while (front) {
          v = deleteq();
          for (w=graph[v]; w; w=w->link)
               if (!visited[w->vertex]) {
                    printf("%5d",w->vertex);
                    addq(w->vertex);
                    visited[w->vertex] = TRUE;
               }
          }
     }
}
```

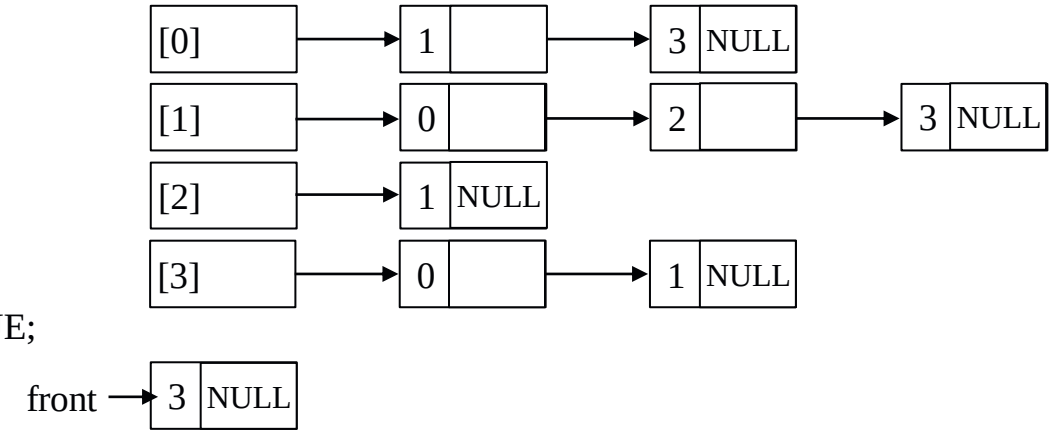




output: 0 1 3

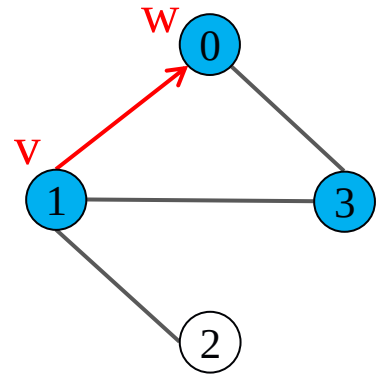
```
void bfs(int v)
{
    node_pointer w;
    front = rear = NULL; /* initialize queue */
    printf("%5d",v);
    visited[v] = TRUE;
    addq(v);
    while (front) {
        v = deleteq();
        for (w=graph[v]; w; w=w->link)
            if (!visited[w->vertex]) {
                printf("%5d",w->vertex);
                addq(w->vertex);
                visited[w->vertex] = TRUE;
            }
        }
    }
}
```

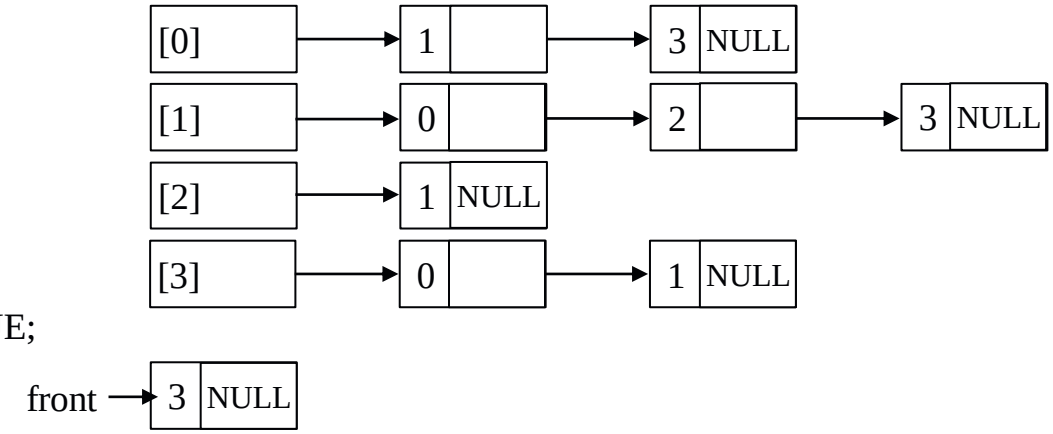




output: 0 1 3

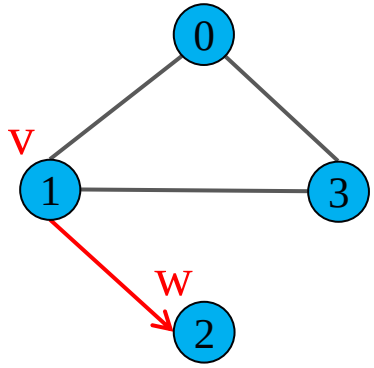
```
void bfs(int v)
{
    node_pointer w;
    front = rear = NULL; /* initialize queue */
    printf("%5d",v);
    visited[v] = TRUE;
    addq(v);
    while (front) {
        v = deleteq();
        for (w=graph[v]; w; w=w->link)
            if (!visited[w->vertex]) {
                printf("%5d",w->vertex);
                addq(w->vertex);
                visited[w->vertex] = TRUE;
            }
        }
    }
}
```

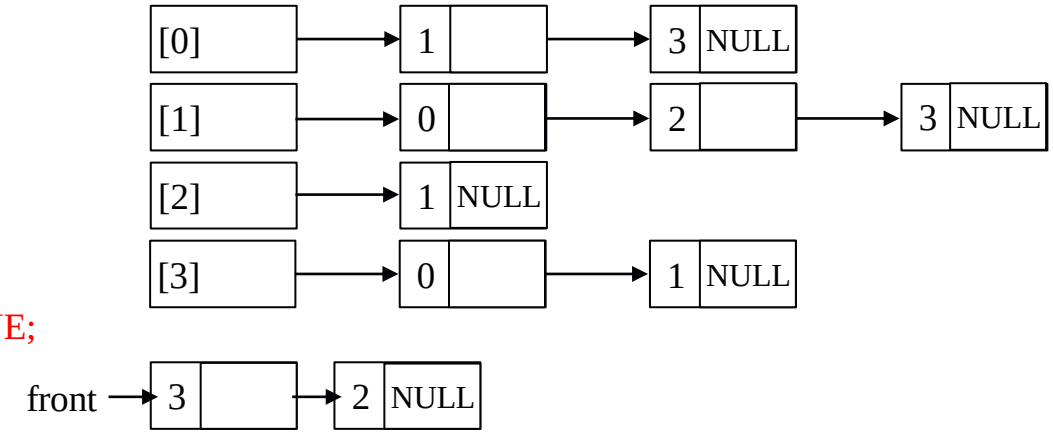




output: 0 1 3

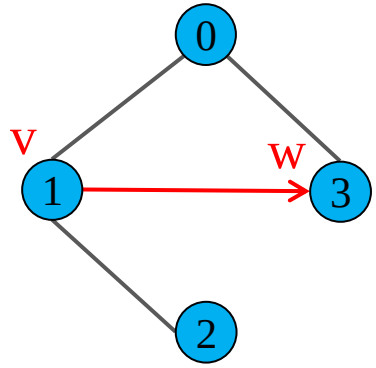
```
void bfs(int v)
{
    node_pointer w;
    front = rear = NULL; /* initialize queue */
    printf("%5d",v);
    visited[v] = TRUE;
    addq(v);
    while (front) {
        v = deleteq();
        for (w=graph[v]; w; w=w->link)
            if (!visited[w->vertex]) {
                printf("%5d",w->vertex);
                addq(w->vertex);
                visited[w->vertex] = TRUE;
            }
        }
    }
}
```

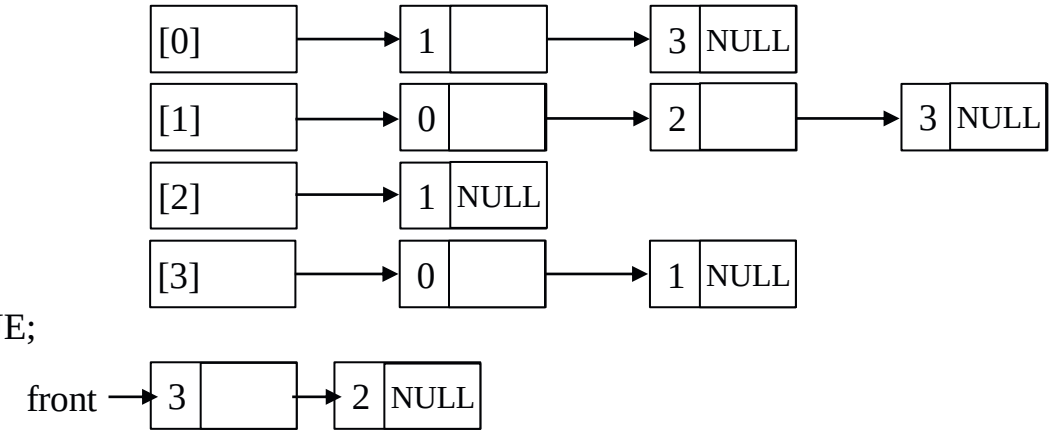




output: 0 1 3 2

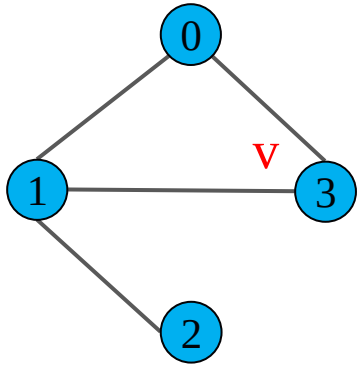
```
void bfs(int v)
{
    node_pointer w;
    front = rear = NULL; /* initialize queue */
    printf("%5d",v);
    visited[v] = TRUE;
    addq(v);
    while (front) {
        v = deleteq();
        for (w=graph[v]; w; w=w->link)
            if (!visited[w->vertex]) {
                printf("%5d",w->vertex);
                addq(w->vertex);
                visited[w->vertex] = TRUE;
            }
        }
    }
}
```

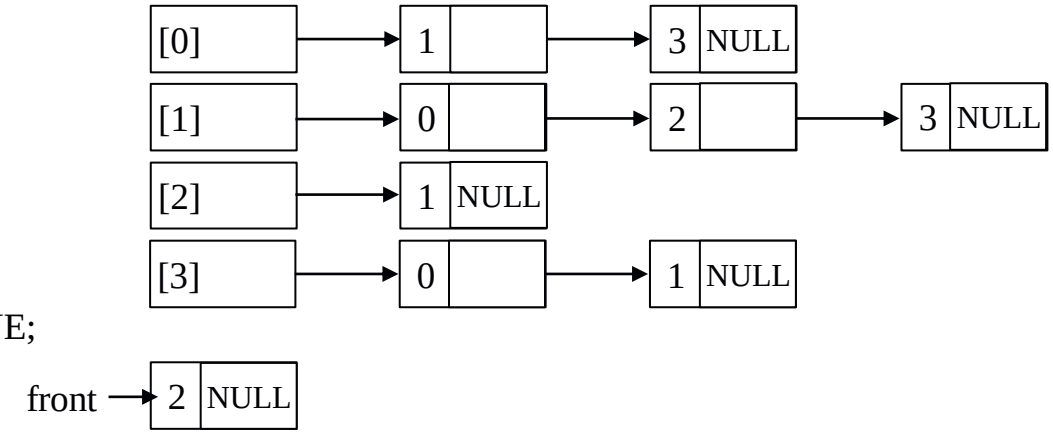




output: 0 1 3 2

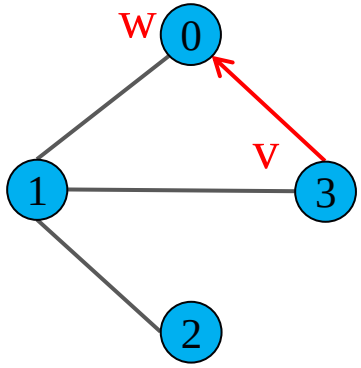
```
void bfs(int v)
{
    node_pointer w;
    front = rear = NULL; /* initialize queue */
    printf("%5d",v);
    visited[v] = TRUE;
    addq(v);
    while (front) {
        v = deleteq();
        for (w=graph[v]; w; w=w->link)
            if (!visited[w->vertex]) {
                printf("%5d",w->vertex);
                addq(w->vertex);
                visited[w->vertex] = TRUE;
            }
        }
    }
}
```

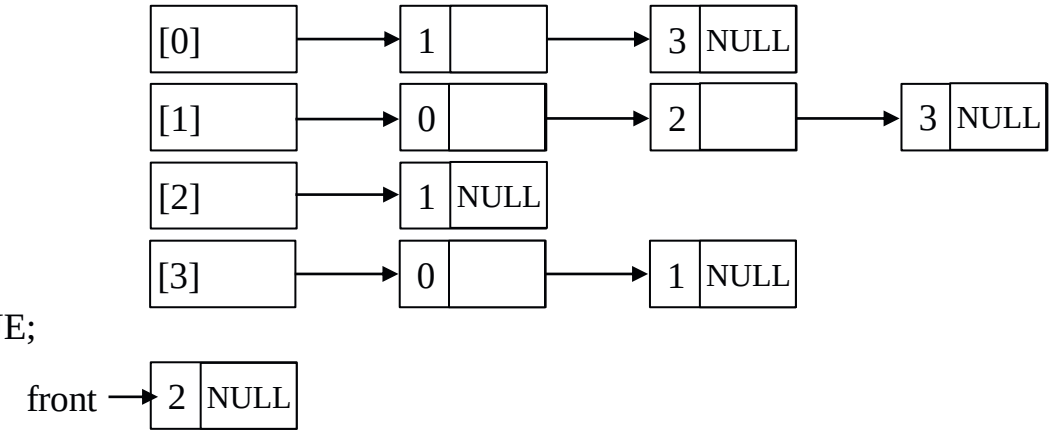




output: 0 1 3 2

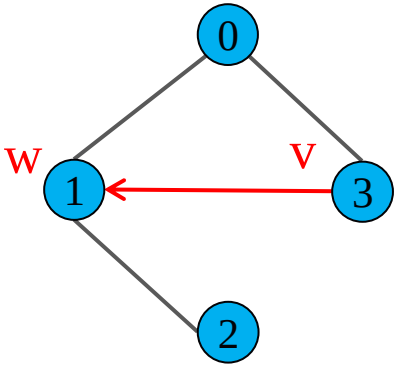
```
void bfs(int v)
{
     node_pointer w;
     front = rear = NULL; /* initialize queue */
     printf("%5d",v);
     visited[v] = TRUE;
     addq(v);
     while (front) {
          v = deleteq();
          for (w=graph[v]; w; w=w->link)
               if (!visited[w->vertex]) {
                    printf("%5d",w->vertex);
                    addq(w->vertex);
                    visited[w->vertex] = TRUE;
               }
          }
     }
}
```

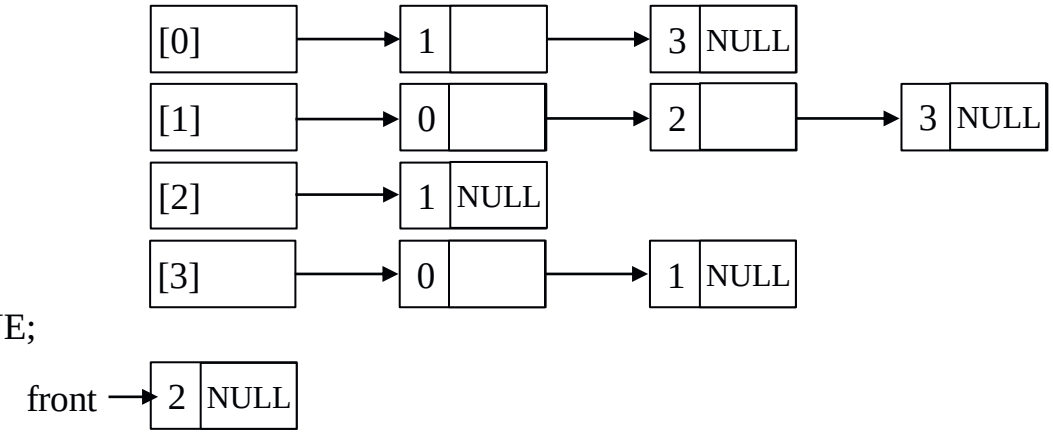




output: 0 1 3 2

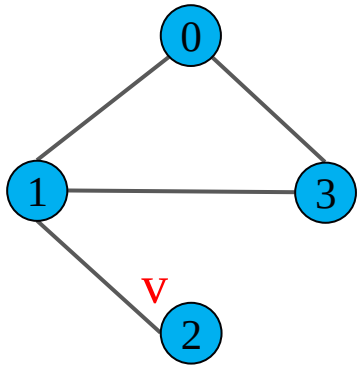
```
void bfs(int v)
{
    node_pointer w;
    front = rear = NULL; /* initialize queue */
    printf("%5d",v);
    visited[v] = TRUE;
    addq(v);
    while (front) {
        v = deleteq();
        for (w=graph[v]; w; w=w->link)
            if (!visited[w->vertex]) {
                printf("%5d",w->vertex);
                addq(w->vertex);
                visited[w->vertex] = TRUE;
            }
        }
    }
}
```

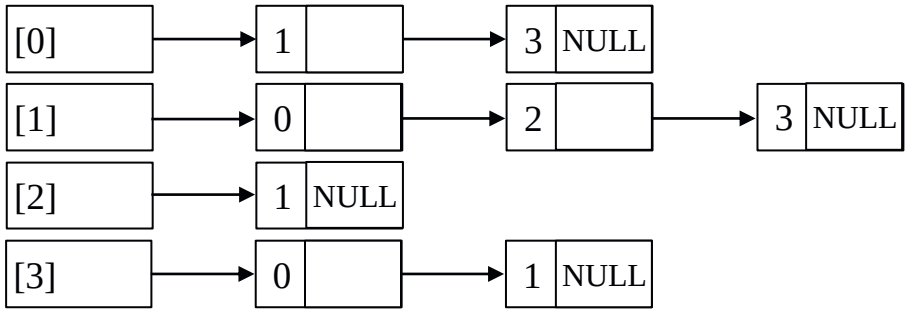




output: 0 1 3 2

```
void bfs(int v)
{
     node_pointer 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;
               }
          }
     }
}
```

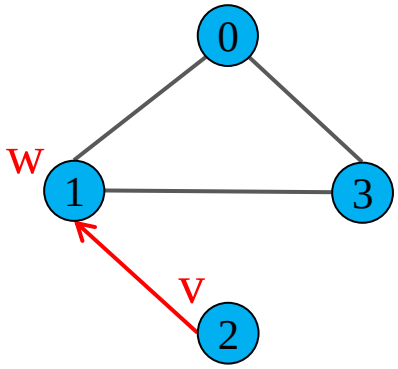


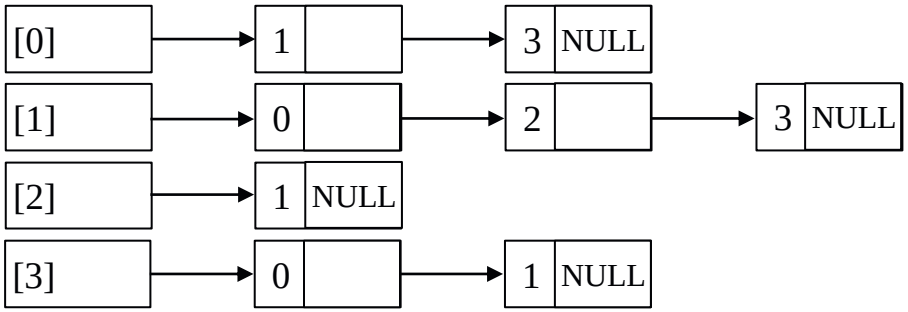


front = NULL

output: 0 1 3 2

```
void bfs(int v)
{
    node_pointer 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;
            }
        }
    }
}
```

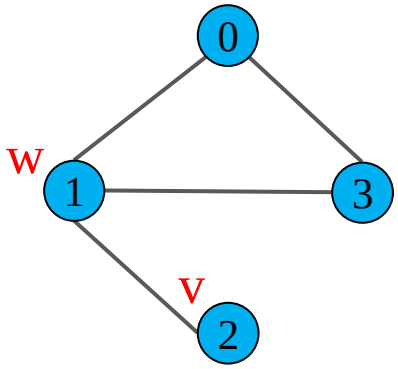


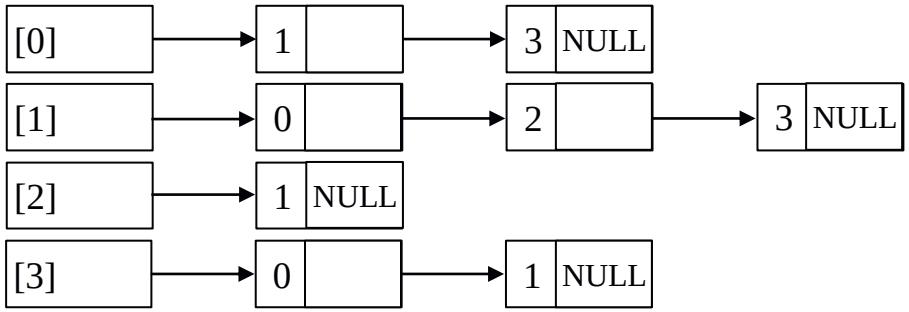


front = NULL

output: 0 1 3 2

```
void bfs(int v)
{
    node_pointer w;
    front = rear = NULL; /* initialize queue */
    printf("%5d",v);
    visited[v] = TRUE;
    addq(v);
    while (front) {        →break
        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;
            }
        }
    }
}
```





front = NULL

output: 0 1 3 2

Analysis of *bfs* :

Since each vertex is placed on the queue exactly once,
the **while** loop is iterated at most n times.

For the adjacency list representation, this loop has a total cost of $d_0 + \cdots + d_{n-1} = O(e)$, where $d_i = degree(v_i)$.

For the adjacency matrix representation,
the while loop takes $O(n)$ time for each vertex visited.
Therefore, the total time is $O(n^2)$.

6.2.3 Connected Components

Note that in both depth first search and breadth first search,
all vertices visited, together with all edges incident to them,
form a connected component of G.

This property allows us to determine whether or not an directed graph is connected.

We can implement this operation by simply calling either *dfs*(0) or *bfs*(0)
and then determining if there are any unvisited vertices.

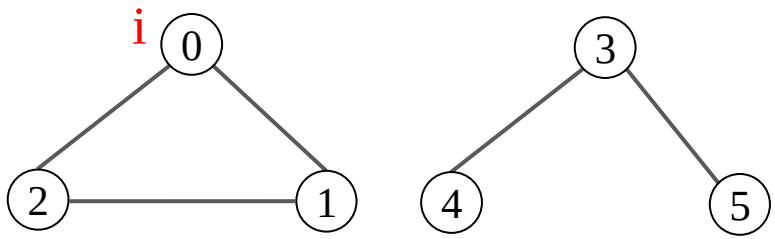
This takes $O(n + e)$ time if adjacency lists are used.

A closely related problem is that of listing the connected components of a graph.

[Program 6.3] Connected components

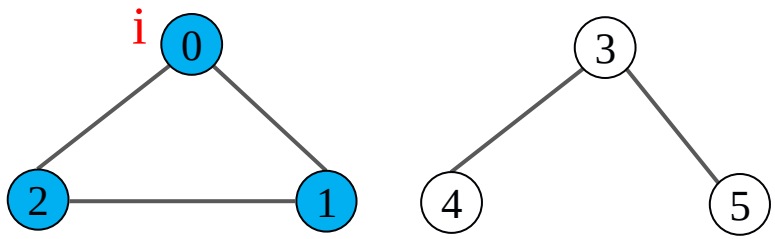
```
void connected(void)
{ /* determine the connected components of a graph */
     int i;
     for (i=0; i<n; i++)
          if (!visited[i]) {
               dfs(i);
               printf("\n");
          }
}
```

```
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");
          }
}
```



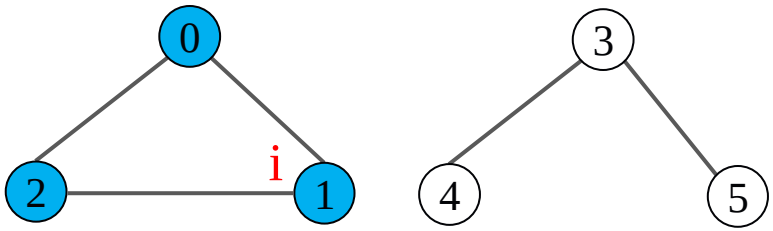
output:

```
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");
          }
}
```



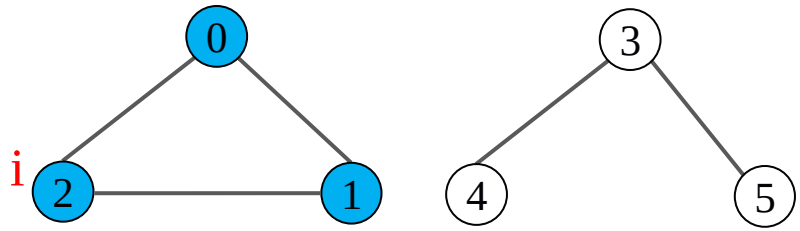
output:
0 1 2

```
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");
          }
}
```



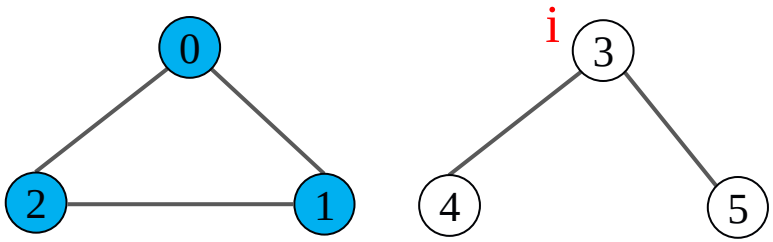
output:
0 1 2

```
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");
          }
}
```



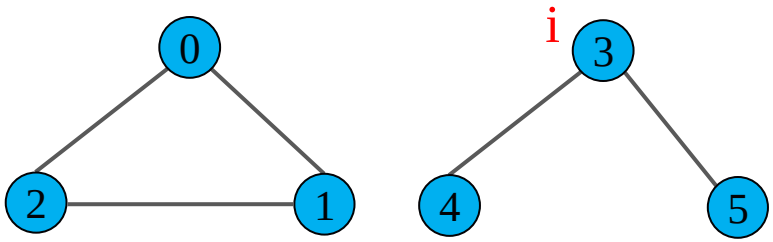
output:
0 1 2

```
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");
          }
}
```



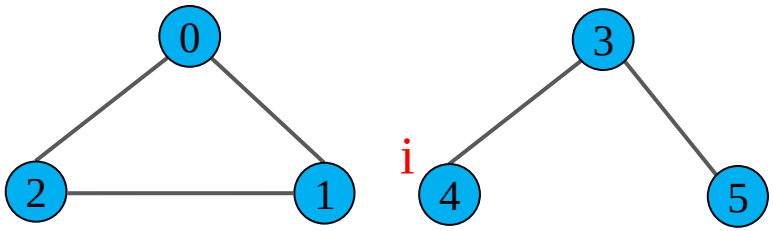
output:
0 1 2

```
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");
          }
}
```



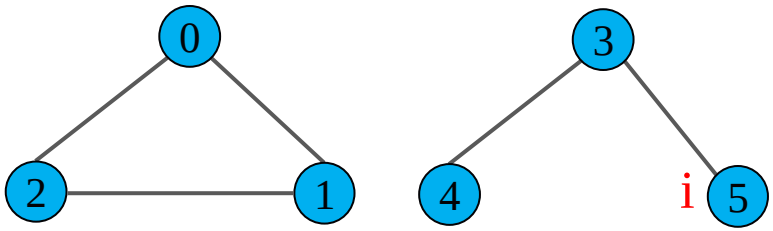
output:
0 1 2
3 4 5

```
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");
          }
}
```



output:
0 1 2
3 4 5

```
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");
          }
}
```



output:
0 1 2
3 4 5

Analysis of *connected* :

If G is represented by its adjacency lists,
then the total time taken by *dfs* is $O(e)$.

Since the **for** loop takes $O(n)$ time,
the total time needed to generate all the connected components is $O(n+e)$.

If G is represented by its adjacency matrix,
then the time needed to determine the connected components is $O(n^2)$.

6.2.4 Spanning Trees

When G is connected, a depth first search or breadth first search starting at any vertex visits all the vertices in G.

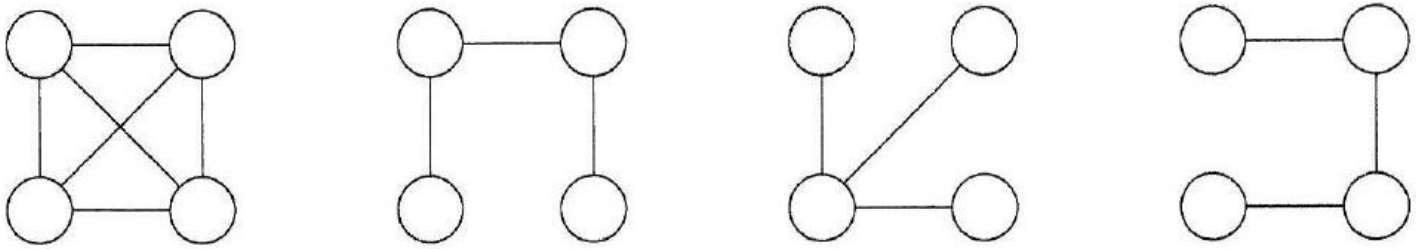
The search implicitly partitions the edges in G into two sets :

- T (for tree edges) is the set of edges used or traversed during the search.
- N (for nontree edges) is the set of remaining edges.

The edges in T form a tree that include all vertices of G.

A *spanning tree* is any tree that consists solely of edges in G and that includes all the vertices in G.

[Figure 6.17] A complete graph and three of its spanning trees

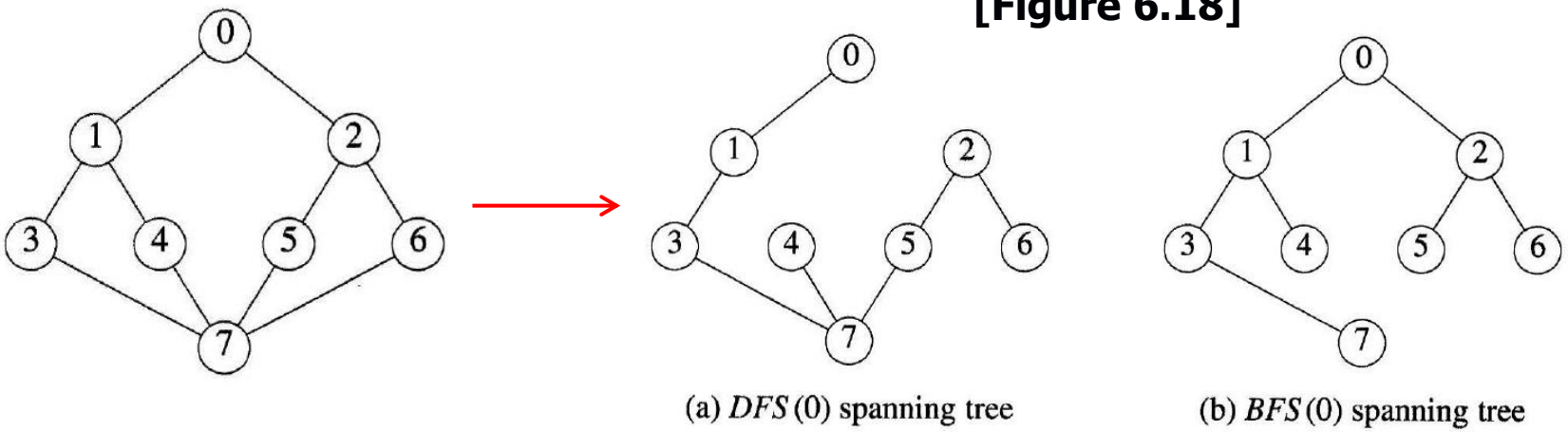


We may use either *dfs* or *bfs* to create a spanning tree;

depth first spanning tree

breadth first spanning tree

[Figure 6.18]



(a) *DFS* (0) spanning tree

(b) *BFS* (0) spanning tree

If we add a nontree edge (v, w) into any spanning tree T,
it creates a cycle that consists of the edge (v, w) and all edges on the path from w to v in T.

Another property of spanning trees:

A spanning tree is a *minimal subgraph,* G', of G such that $V(G') = V(G)$ and G' is connected.

Any connected graph with n vertices must have at least $n - 1$ edges, and all connected graph with $n - 1$ edges are trees.

Therefore, the spanning tree has $n - 1$ edges.

Constructing minimal subgraphs finds frequent application in the design of communication networks.

6.3 MINIMUM COST SPANNING TREES

The *cost* of a spanning tree of a weighted undirected graph is the sum of the costs (weights) of the edges in the spanning tree.

A *minimum cost spanning tree* is a spanning tree of least cost.

Three greedy algorithms to obtain a minimum cost spanning tree:

- Kruskal's algorithm,
- Prim's algorithm,
- Sollin's algorithm.

In the greedy method, we construct an optimal solution in stages.

At each stage, we make a decision that is the best decision (using some criterion) at this time.

Since we cannot change this decision later,
we make sure that the decision will result in a feasible solution.

Typically, the selection of an item at each stage is based on either a least cost or a highest profit criterion.
A feasible solution is one which works within the constraints specified by the problem.

For spanning trees, we use a least cost criterion.

Our solution must satisfy the following constraints:

(1) we must use only edges within the graph
(2) we must use exactly $n - 1$ edges
(3) we may not use edges that would produce a cycle

6.3.1 Kruskal's Algorithm

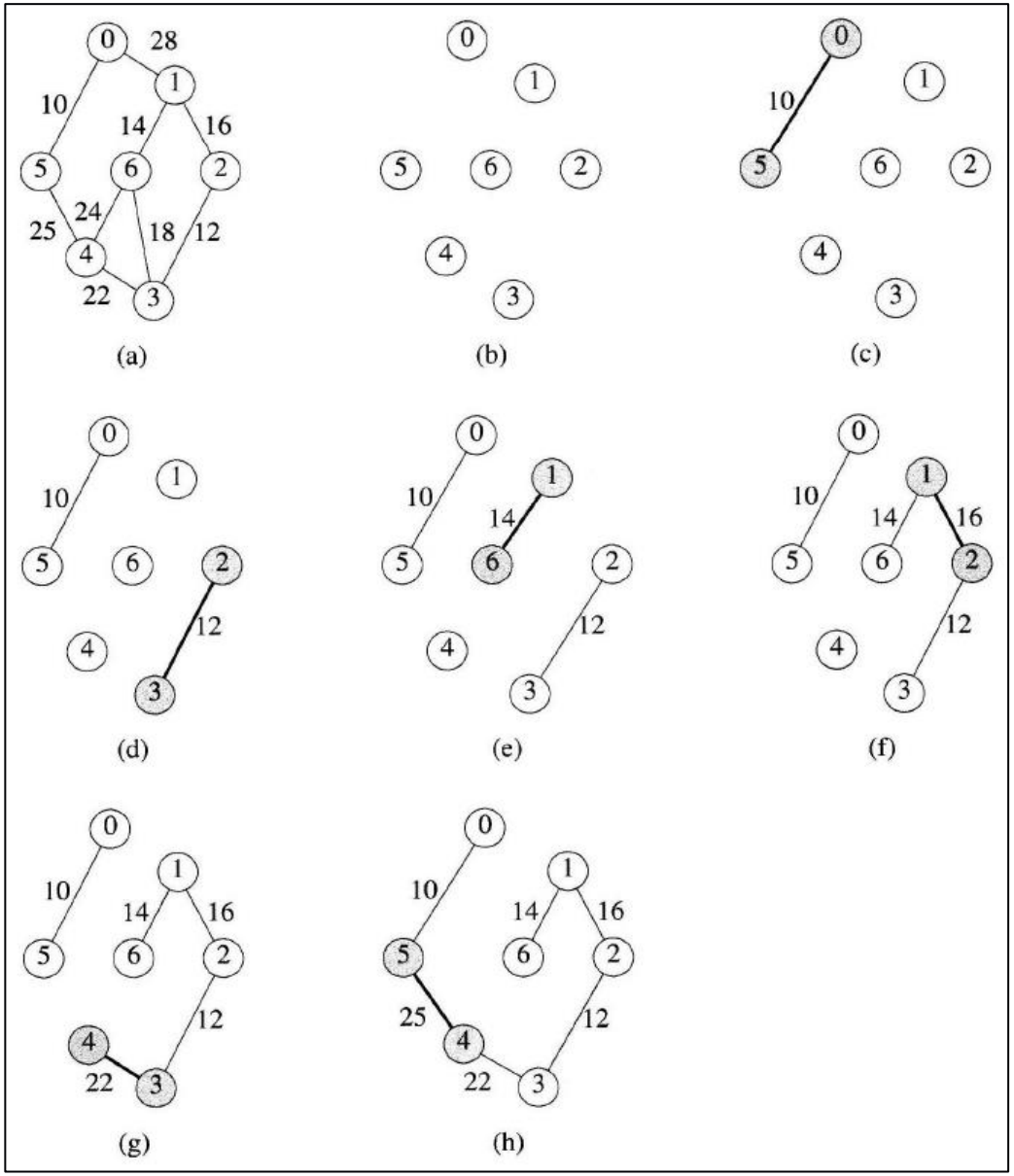
Kruskal's algorithm builds a minimum cost spanning tree T by adding edges to T one at a time.

The algorithm selects the edges for inclusion in T in nondecreasing order of their cost.

An edge is added to T if it does not form a cycle with edges that are already in T.

Since G is connected and has $n > 0$ vertices,
exactly $n - 1$ edges will be selected for inclusion in T.

[Figure 6.22] Stages in Kruskal's algorithm



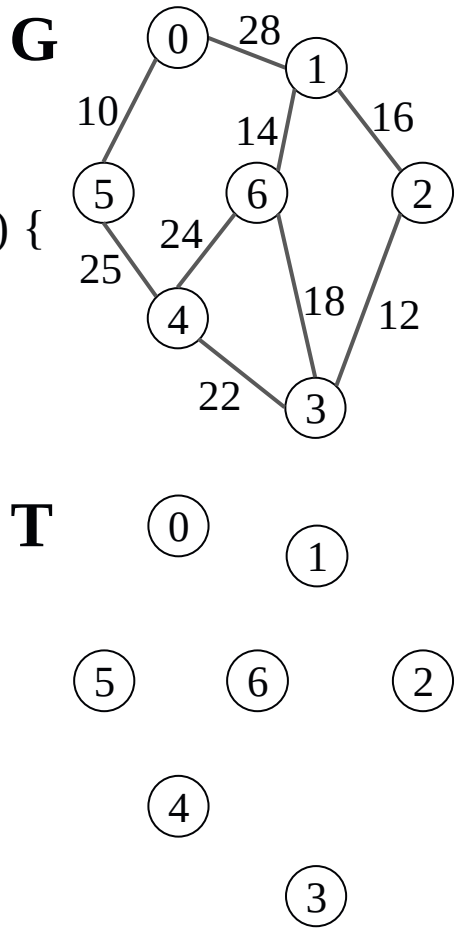
[Figure 6.23]

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	

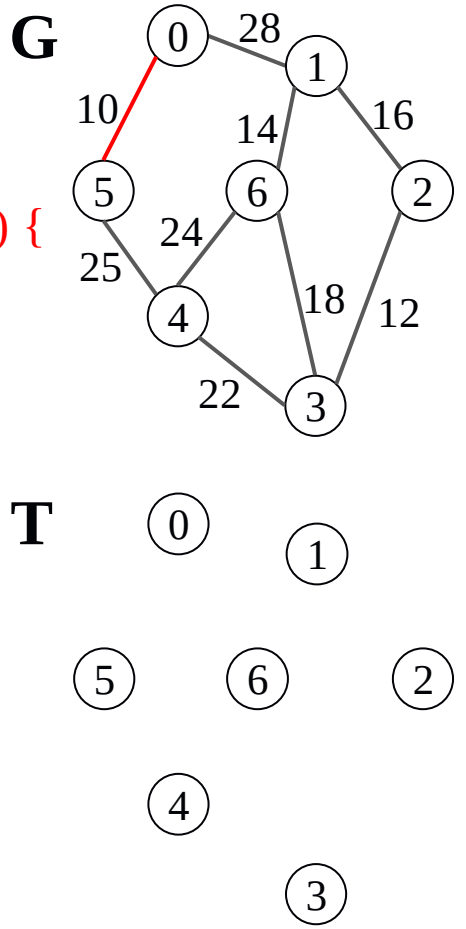
[Program 6.7] Kruskal's algorithm

```
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");
```

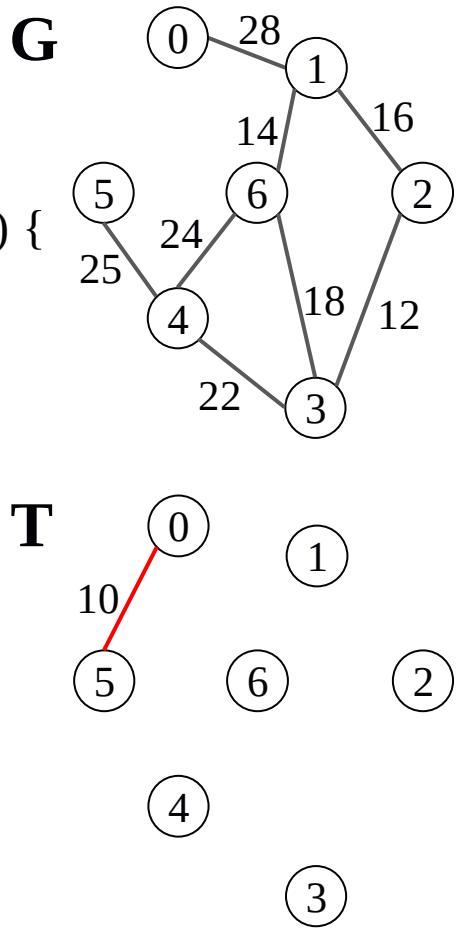
```
T = {};
while (T contains less than n-1 edges && E is not empty) {
    choose a least cost edge (v,w) from E;
    delete (v,w) from E;
    if ((v,w) does not create a cycle in T)
        add (v,w) to T;
    else
        discard (v,w);
}
if (T contains fewer than n-1 edges)
    printf("No spanning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



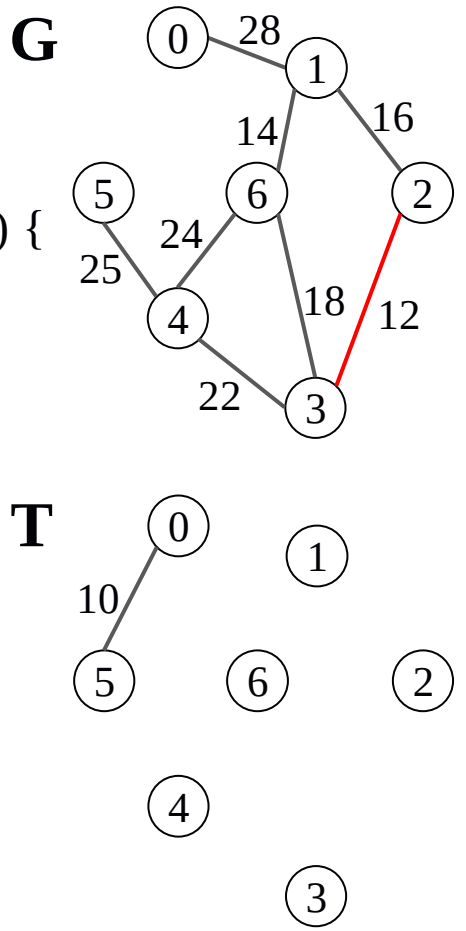
```
T = {};
while (T contains less than n-1 edges && E is not empty) {
    choose a least cost edge (v,w) from E;
    delete (v,w) from E;
    if ((v,w) does not create a cycle in T)
        add (v,w) to T;
    else
        discard (v,w);
}
if (T contains fewer than n-1 edges)
    printf("No spanning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



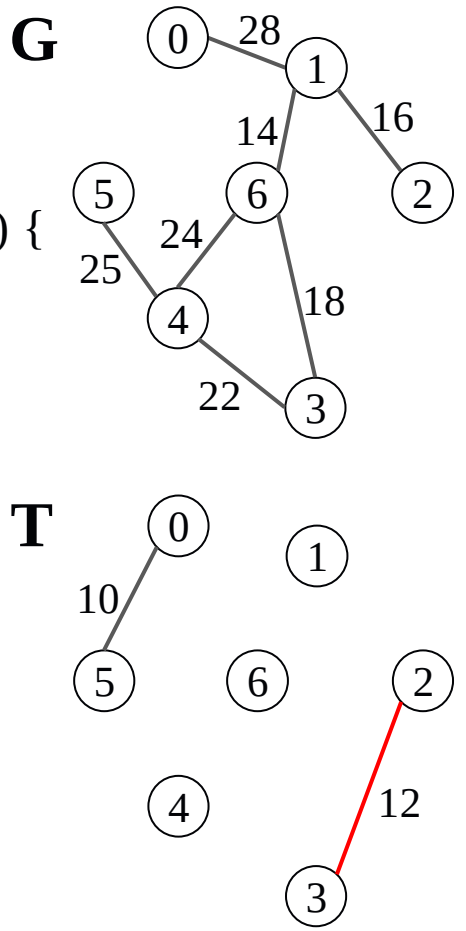
```
T = {};
while (T contains less than n-1 edges && E is not empty) {
    choose a least cost edge (v,w) from E;
    delete (v,w) from E;
    if ((v,w) does not create a cycle in T)
        add (v,w) to T;
    else
        discard (v,w);
}
if (T contains fewer than n-1 edges)
    printf("No spanning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



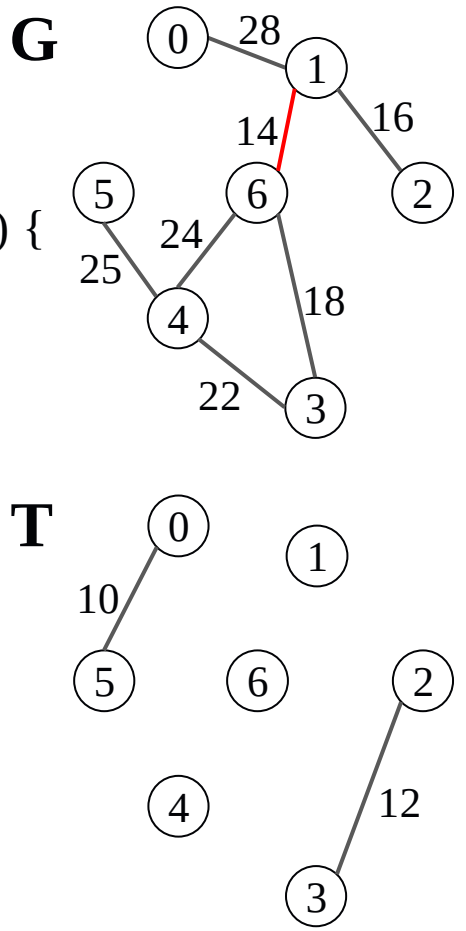
```
T = {};
while (T contains less than n-1 edges && E is not empty) {
    choose a least cost edge (v,w) from E;
    delete (v,w) from E;
    if ((v,w) does not create a cycle in T)
        add (v,w) to T;
    else
        discard (v,w);
}
if (T contains fewer than n-1 edges)
    printf("No spanning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



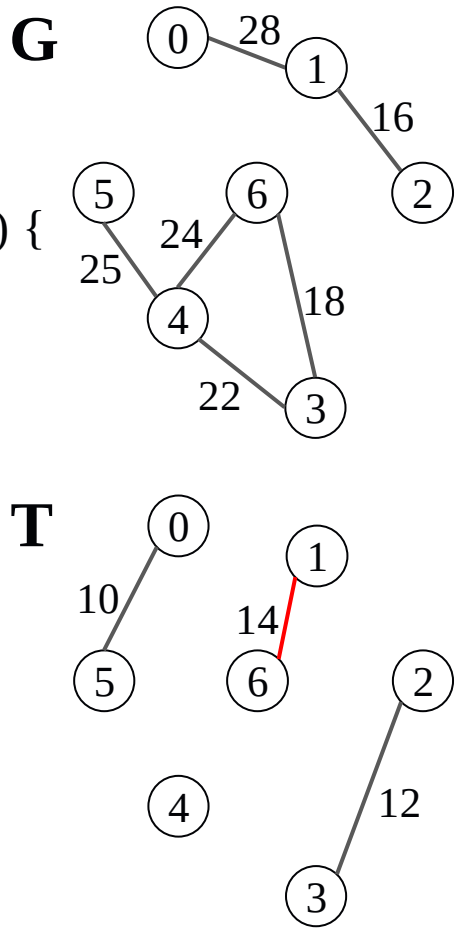
```
T = {};
while (T contains less than n-1 edges && E is not empty) {
    choose a least cost edge (v,w) from E;
    delete (v,w) from E;
    if ((v,w) does not create a cycle in T)
        add (v,w) to T;
    else
        discard (v,w);
}
if (T contains fewer than n-1 edges)
    printf("No spanning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



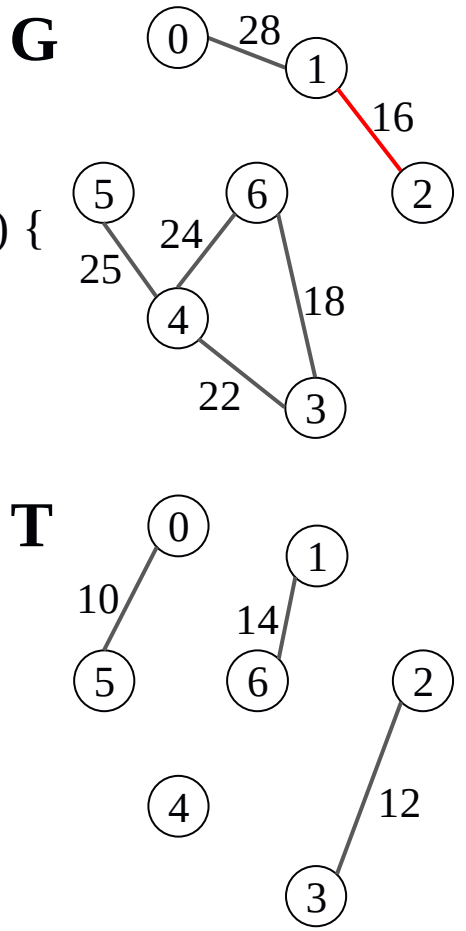
```
T = {};
while (T contains less than n-1 edges && E is not empty) {
    choose a least cost edge (v,w) from E;
    delete (v,w) from E;
    if ((v,w) does not create a cycle in T)
        add (v,w) to T;
    else
        discard (v,w);
}
if (T contains fewer than n-1 edges)
    printf("No spanning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



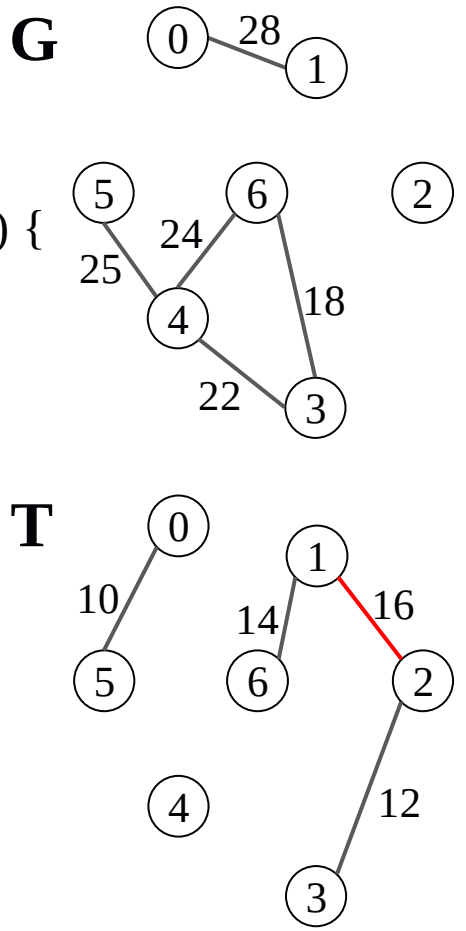
```
T = {};
while (T contains less than n-1 edges && E is not empty) {
    choose a least cost edge (v,w) from E;
    delete (v,w) from E;
    if ((v,w) does not create a cycle in T)
        add (v,w) to T;
    else
        discard (v,w);
}
if (T contains fewer than n-1 edges)
    printf("No spanning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



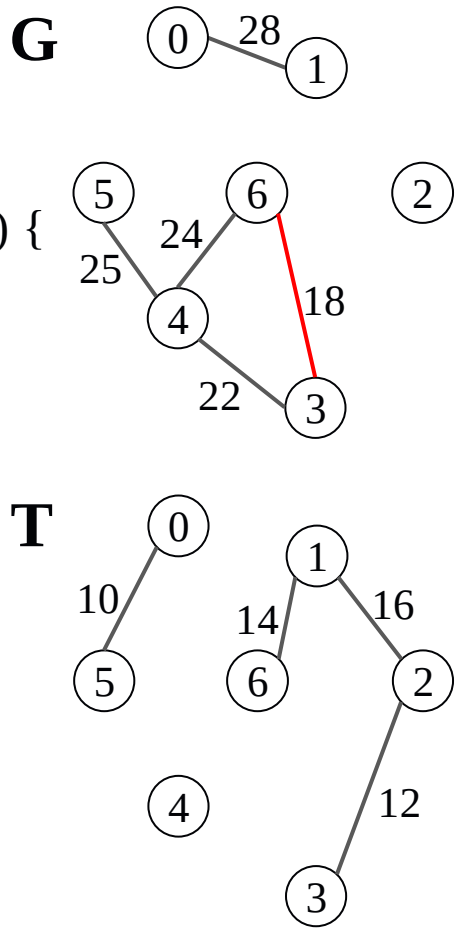
```
T = {};
while (T contains less than n-1 edges && E is not empty) {
    choose a least cost edge (v,w) from E;
    delete (v,w) from E;
    if ((v,w) does not create a cycle in T)
        add (v,w) to T;
    else
        discard (v,w);
}
if (T contains fewer than n-1 edges)
    printf("No spanning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



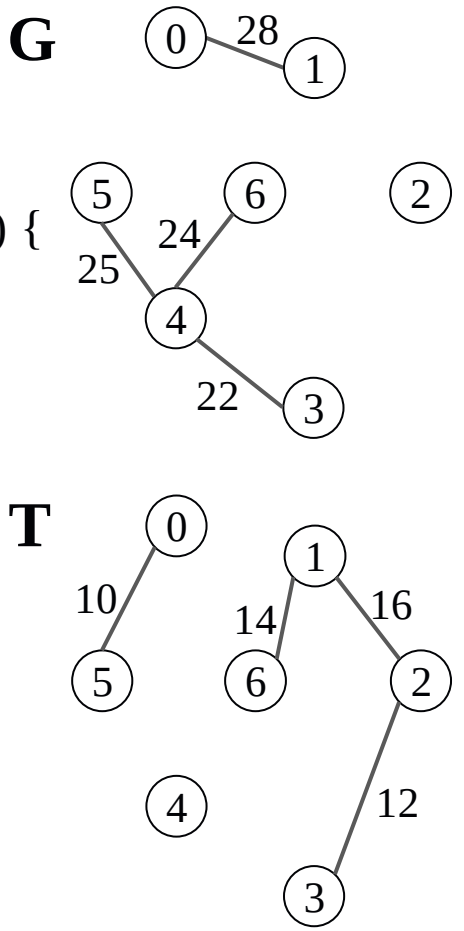
```
T = {};
while (T contains less than n-1 edges && E is not empty) {
    choose a least cost edge (v,w) from E;
    delete (v,w) from E;
    if ((v,w) does not create a cycle in T)
        add (v,w) to T;
    else
        discard (v,w);
}
if (T contains fewer than n-1 edges)
    printf("No spanning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



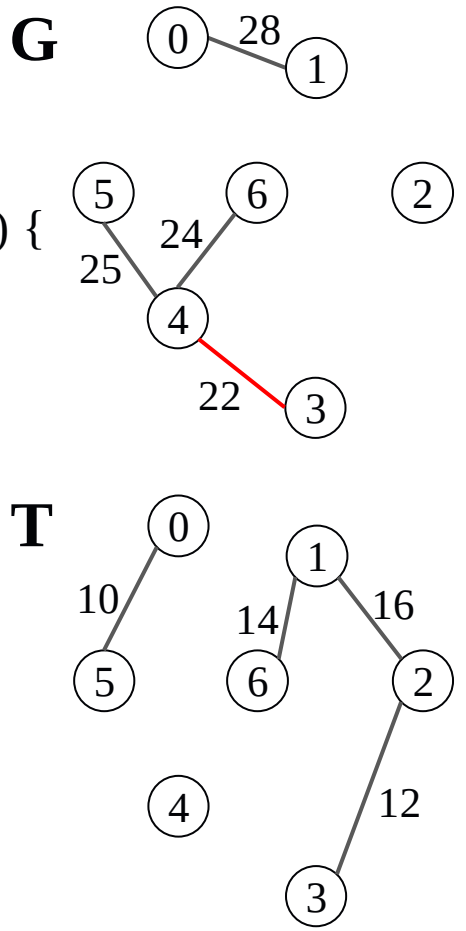
```
T = {};
while (T contains less than n-1 edges && E is not empty) {
    choose a least cost edge (v,w) from E;
    delete (v,w) from E;
    if ((v,w) does not create a cycle in T)
        add (v,w) to T;
    else
        discard (v,w);
}
if (T contains fewer than n-1 edges)
    printf("No spanning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



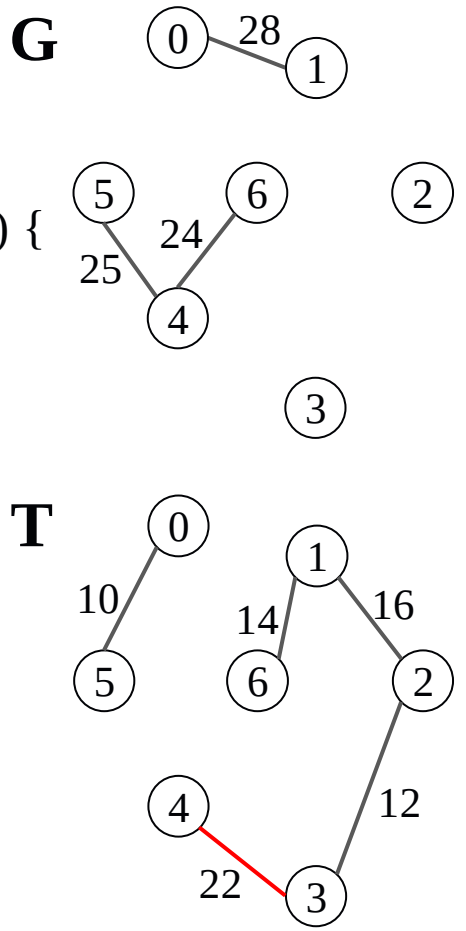
```
T = {};
while (T contains less than n-1 edges && E is not empty) {
    choose a least cost edge (v,w) from E;
    delete (v,w) from E;
    if ((v,w) does not create a cycle in T)
        add (v,w) to T;
    else
        discard (v,w);
}
if (T contains fewer than n-1 edges)
    printf("No spanning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



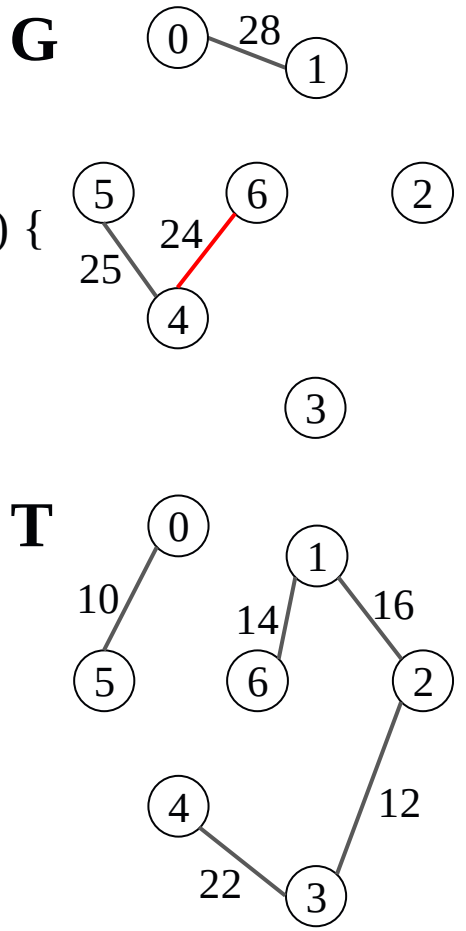
```
T = {};
while (T contains less than n-1 edges && E is not empty) {
    choose a least cost edge (v,w) from E;
    delete (v,w) from E;
    if ((v,w) does not create a cycle in T)
        add (v,w) to T;
    else
        discard (v,w);
}
if (T contains fewer than n-1 edges)
    printf("No spanning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



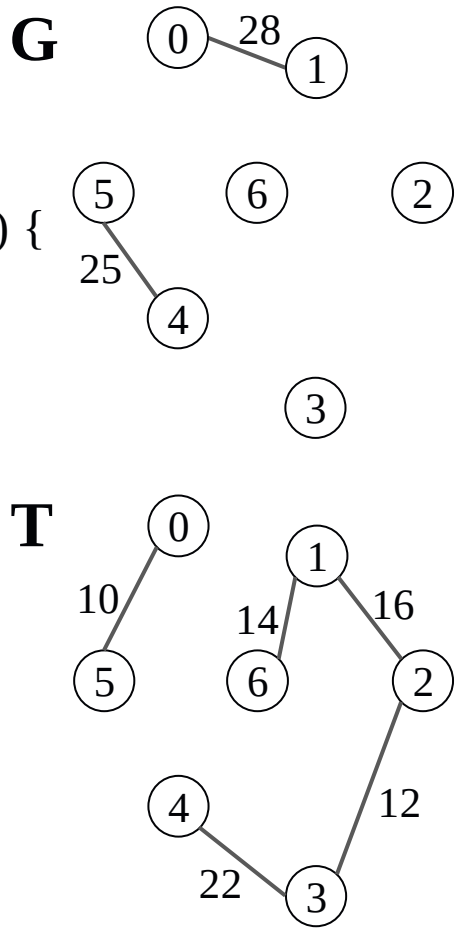
```
T = {};
while (T contains less than n-1 edges && E is not empty) {
    choose a least cost edge (v,w) from E;
    delete (v,w) from E;
    if ((v,w) does not create a cycle in T)
        add (v,w) to T;
    else
        discard (v,w);
}
if (T contains fewer than n-1 edges)
    printf("No spanning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



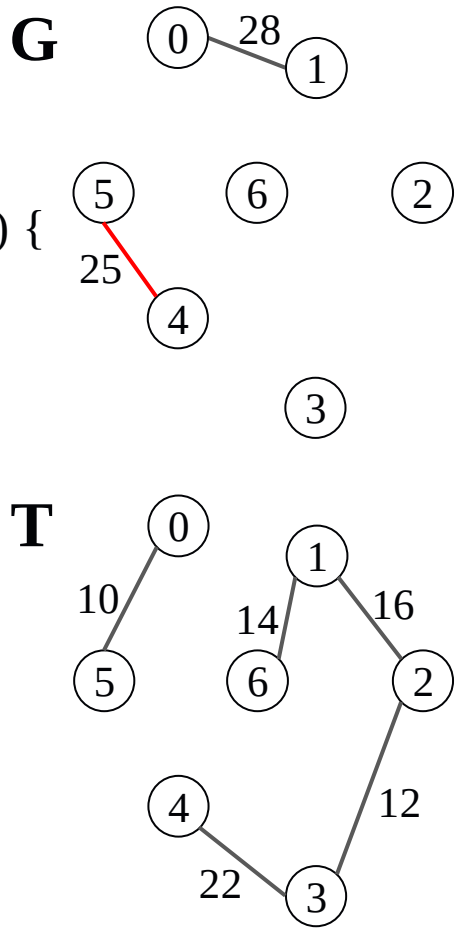
```
T = {};
while (T contains less than n-1 edges && E is not empty) {
    choose a least cost edge (v,w) from E;
    delete (v,w) from E;
    if ((v,w) does not create a cycle in T)
        add (v,w) to T;
    else
        discard (v,w);
}
if (T contains fewer than n-1 edges)
    printf("No spanning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



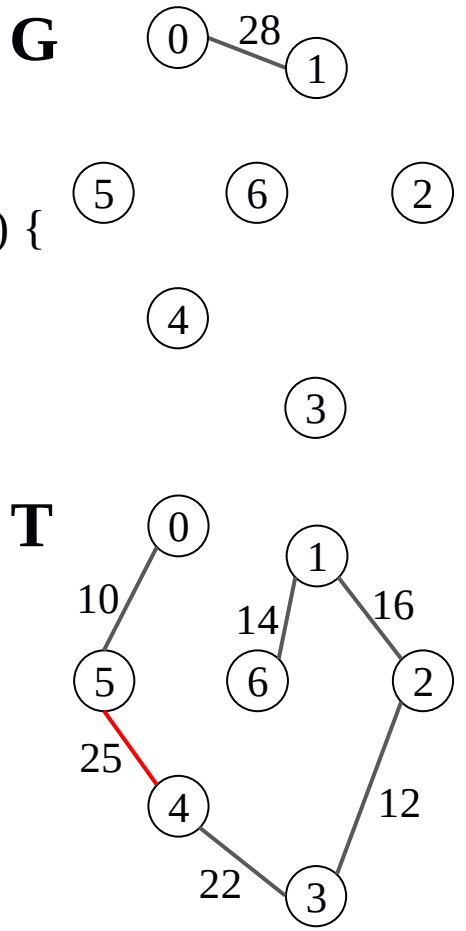
```
T = {};
while (T contains less than n-1 edges && E is not empty) {
    choose a least cost edge (v,w) from E;
    delete (v,w) from E;
    if ((v,w) does not create a cycle in T)
        add (v,w) to T;
    else
        discard (v,w);
}
if (T contains fewer than n-1 edges)
    printf("No spanning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



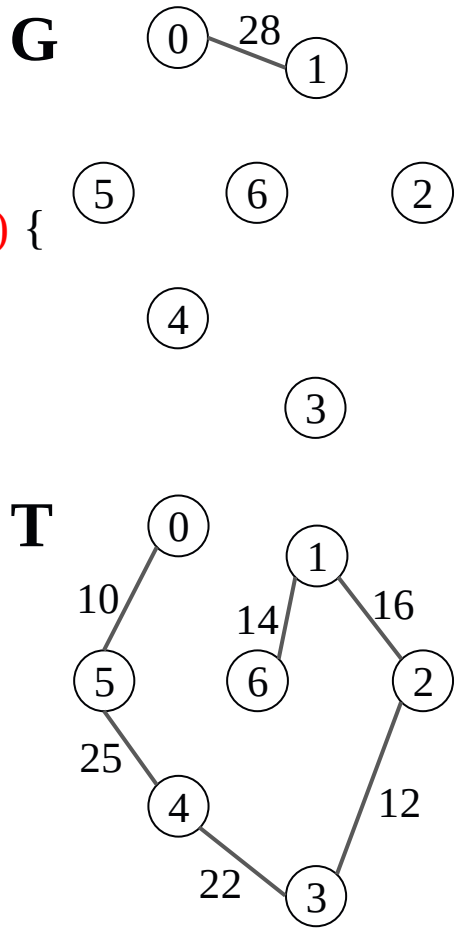
```
T = {};
while (T contains less than n-1 edges && E is not empty) {
    choose a least cost edge (v,w) from E;
    delete (v,w) from E;
    if ((v,w) does not create a cycle in T)
        add (v,w) to T;
    else
        discard (v,w);
}
if (T contains fewer than n-1 edges)
    printf("No spanning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



```
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");
```



```
T = {};                                                    → break
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");
```



Implementation :

How to determine an edge with minimum cost and delete that edge?
→ Sorting or using a min heap.

How to check that the new edge, (v, w), does not form a cycle in T?
→ We may use the union-find operations discussed in Section 5.10.

The computing time of Kruskal's algorithm is $O(e \log e)$.

[Theorem 6.1] : Let G be an undirected connected graph.
Kruskal's algorithm generates a minimum cost spanning tree.

6.3.2 Prim's Algorithm

Prim's algorithm, like Kruskal's, constructs the minimum cost spanning tree one edge at a time.

However, at each stage, the set of selected edges forms a tree.

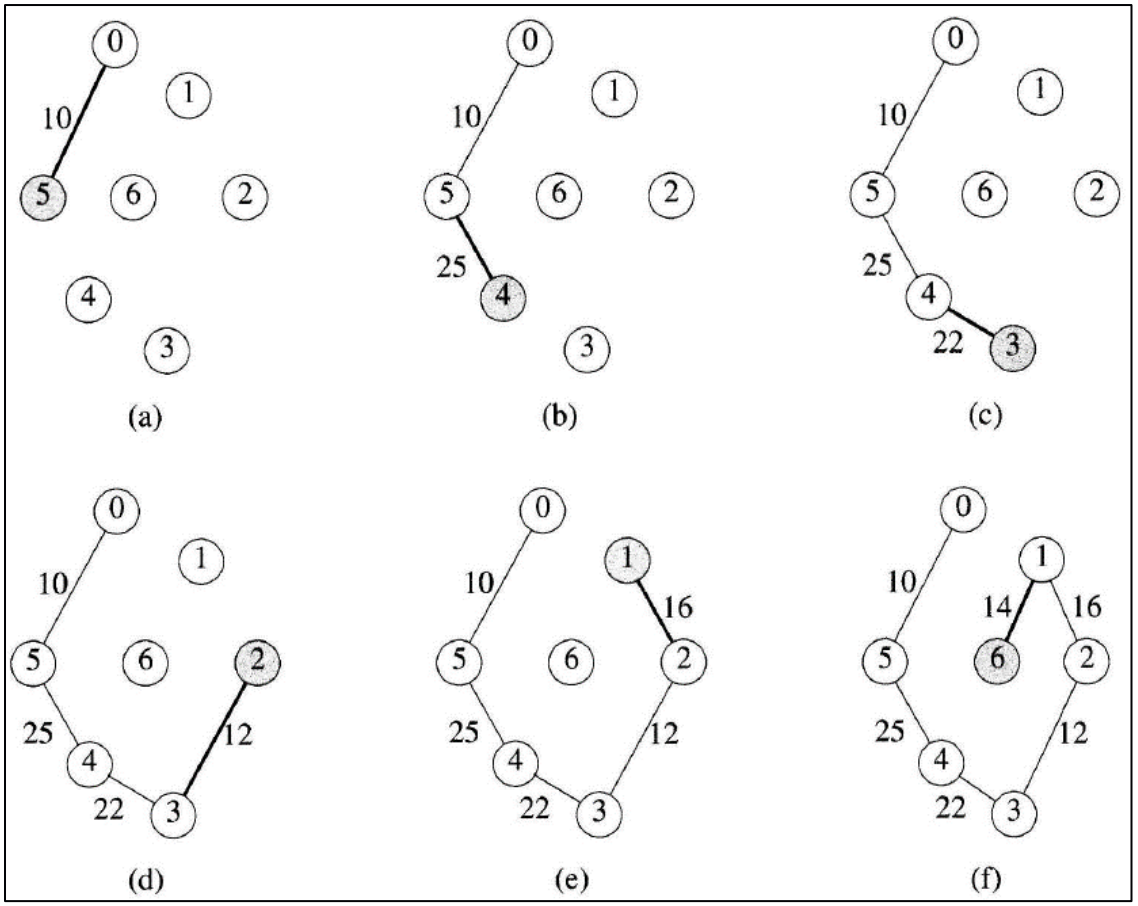
Prim's algorithm begins with a tree, T, that contains a single vertex.
This may be any of the vertices in the original graph.

Next, we add a least cost edge (u, v) to T such that $T \cup \{(u, v)\}$ is also a tree.

We repeat this edge addition step until T contains $n - 1$ edges.

To make sure that the added edge does not form a cycle,
at each step we choose the edge (u, v) such that exactly one of u or v in T.

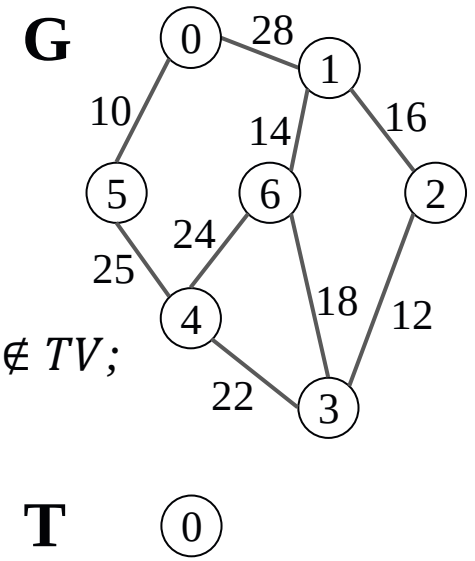
[Figure 6.24] Stages in Prim's algorithm



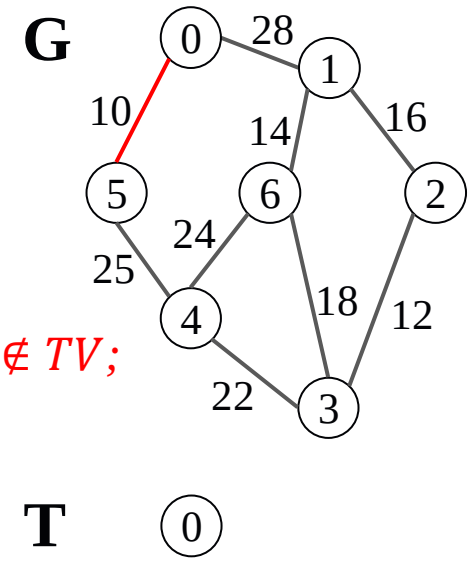
[Program 6.8] Prim's algorithm

```
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");
```

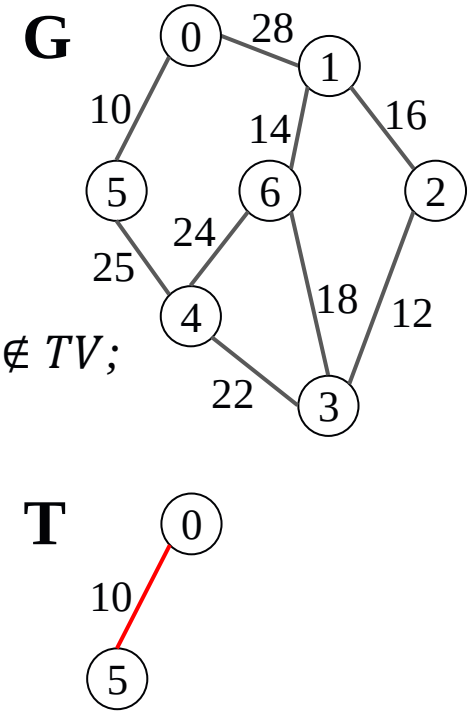
```
T = {};
TV = {0};   /* start with vertex 0 and no edges */
while (T contains fewer than n-1 edges) {
    let (u,v) be a least cost edge such that u ∈ TV and v ∉ TV;
    if (there is no such edge)
        break;
    add v to TV;
    add (u,v) to T;
}
if (T contains fewer than n-1 edges)
    printf("No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



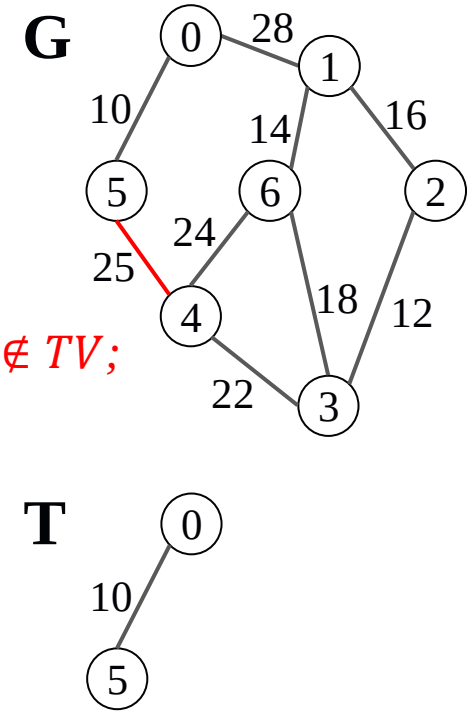
```
T = {};
TV = {0};   /* start with vertex 0 and no edges */
while (T contains fewer than n-1 edges) {
     let (u,v) be a least cost edge such that u ∈ TV and v ∉ TV;
     if (there is no such edge)
          break;
     add v to TV;
     add (u,v) to T;
}
if (T contains fewer than n-1 edges)
   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



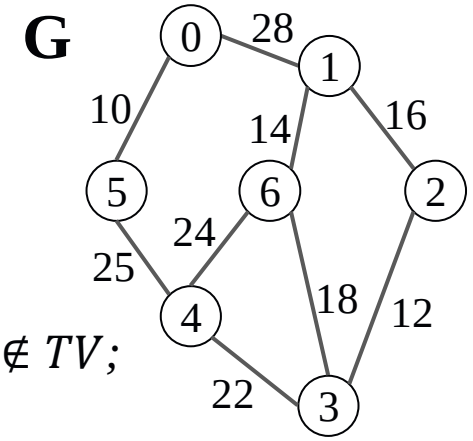
```
T = {};
TV = {0};   /* start with vertex 0 and no edges */
while (T contains fewer than n-1 edges) {
     let (u,v) be a least cost edge such that u ∈ TV and v ∉ TV;
     if (there is no such edge)
          break;
     add v to TV;
     add (u,v) to T;
}
if (T contains fewer than n-1 edges)
   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

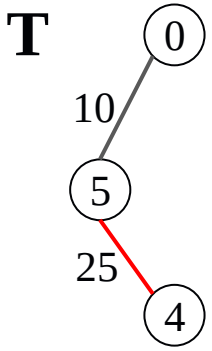


```
T = {};
TV = {0};   /* start with vertex 0 and no edges */
while (T contains fewer than n-1 edges) {
     let (u,v) be a least cost edge such that u ∈ TV and v ∉ TV;
     if (there is no such edge)
          break;
     add v to TV;
     add (u,v) to T;
}
if (T contains fewer than n-1 edges)
   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

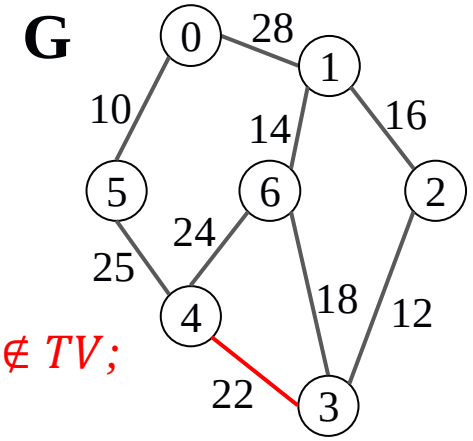


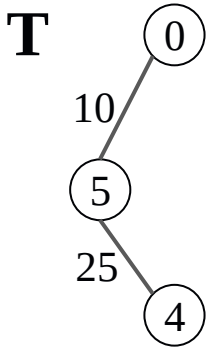
```
T = {};
TV = {0};   /* start with vertex 0 and no edges */
while (T contains fewer than n-1 edges) {
     let (u,v) be a least cost edge such that u ∈ TV and v ∉ TV;
     if (there is no such edge)
          break;
     add v to TV;
     add (u,v) to T;
}
if (T contains fewer than n-1 edges)
   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



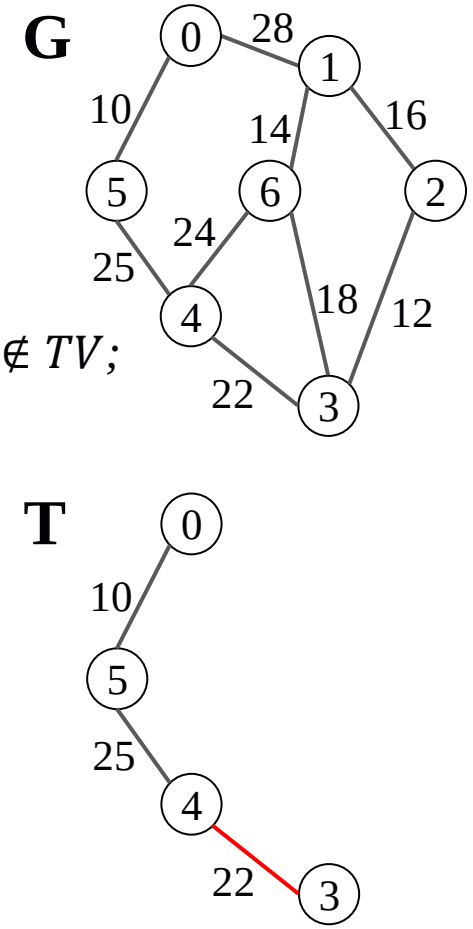


```
T = {};
TV = {0};   /* start with vertex 0 and no edges */
while (T contains fewer than n-1 edges) {
     let (u,v) be a least cost edge such that u ∈ TV and v ∉ TV;
     if (there is no such edge)
          break;
     add v to TV;
     add (u,v) to T;
}
if (T contains fewer than n-1 edges)
   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

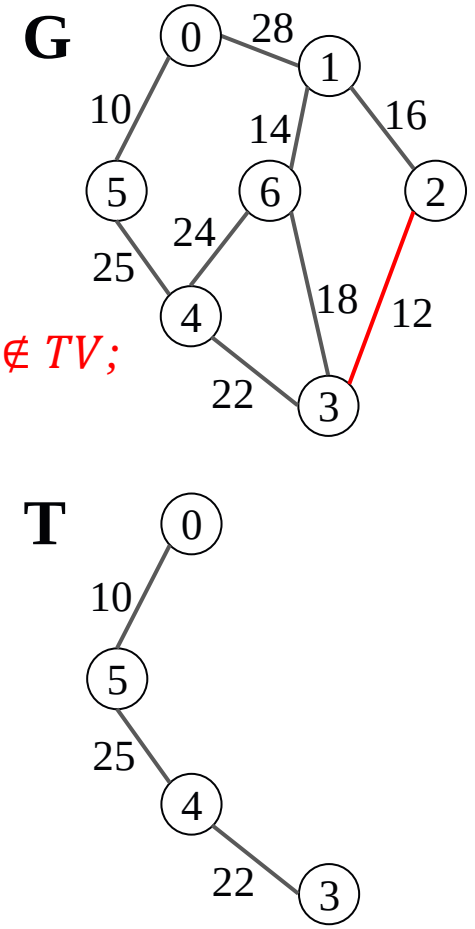




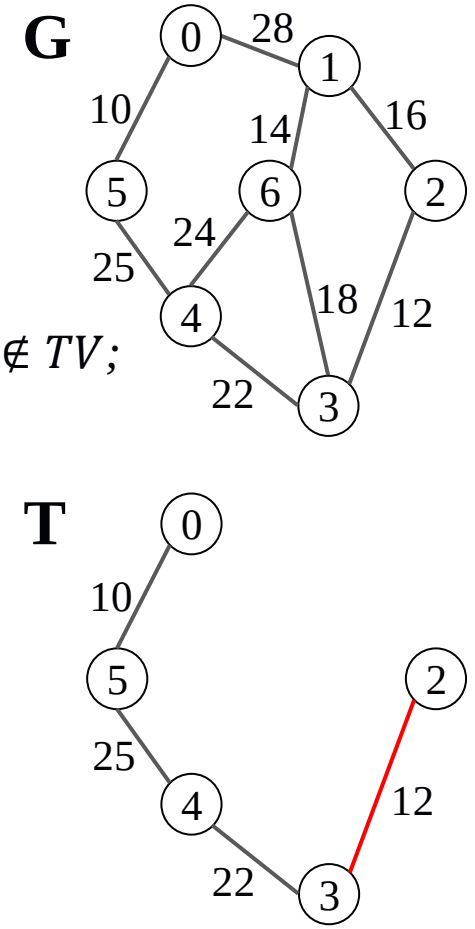
```
T = {};
TV = {0};   /* start with vertex 0 and no edges */
while (T contains fewer than n-1 edges) {
      let (u,v) be a least cost edge such that u ∈ TV and v ∉ TV;
      if (there is no such edge)
            break;
      add v to TV;
      add (u,v) to T;
}
if (T contains fewer than n-1 edges)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



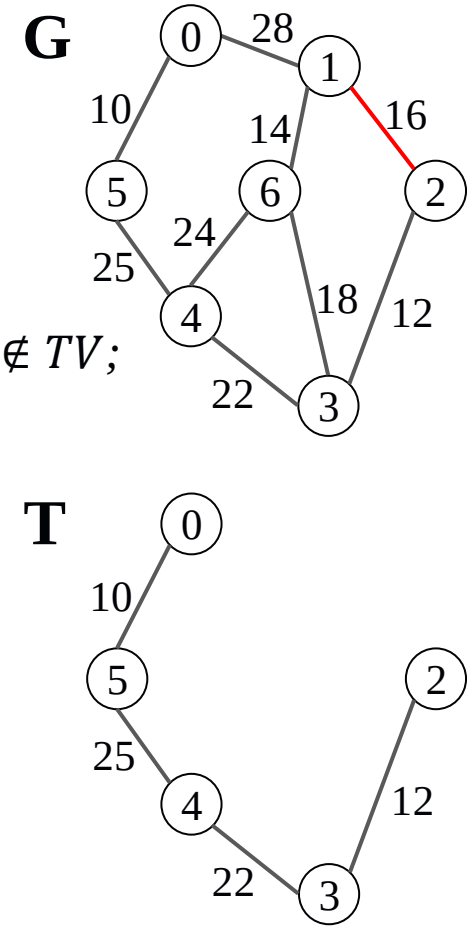
```
T = {};
TV = {0};   /* start with vertex 0 and no edges */
while (T contains fewer than n-1 edges) {
    let (u,v) be a least cost edge such that u ∈ TV and v ∉ TV;
    if (there is no such edge)
        break;
    add v to TV;
    add (u,v) to T;
}
if (T contains fewer than n-1 edges)
    printf("No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



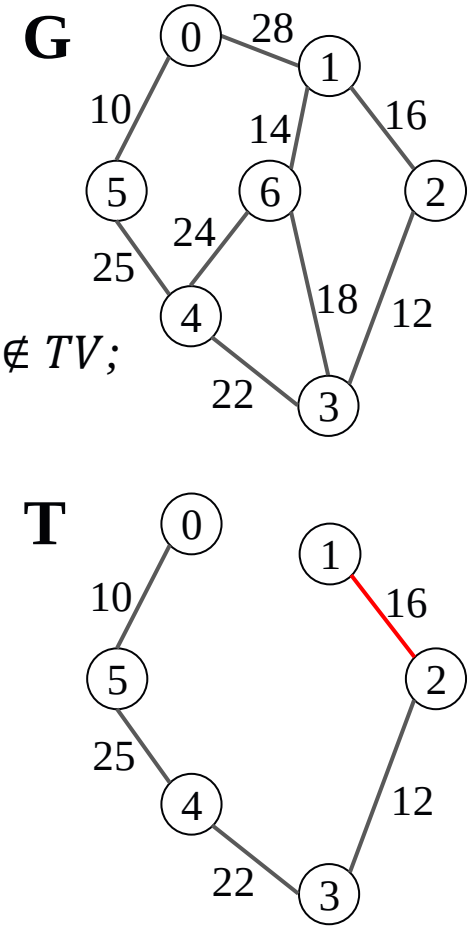
```
T = {};
TV = {0};   /* start with vertex 0 and no edges */
while (T contains fewer than n-1 edges) {
     let (u,v) be a least cost edge such that u ∈ TV and v ∉ TV;
     if (there is no such edge)
          break;
     add v to TV;
     add (u,v) to T;
}
if (T contains fewer than n-1 edges)
   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



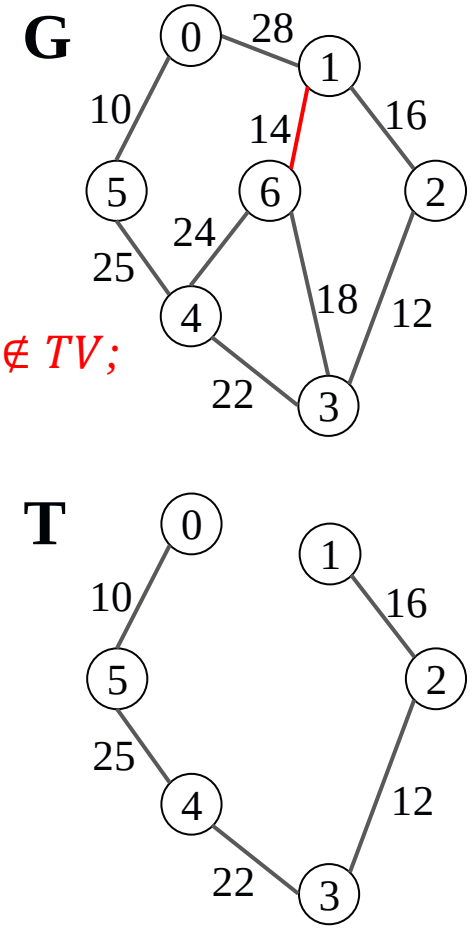
```
T = {};
TV = {0};   /* start with vertex 0 and no edges */
while (T contains fewer than n-1 edges) {
     let (u,v) be a least cost edge such that u ∈ TV and v ∉ TV;
     if (there is no such edge)
          break;
     add v to TV;
     add (u,v) to T;
}
if (T contains fewer than n-1 edges)
   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



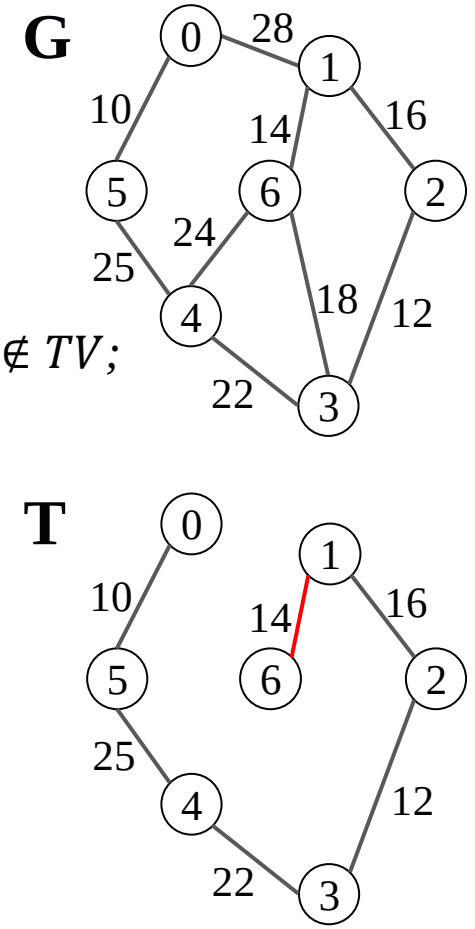
```
T = {};
TV = {0};   /* start with vertex 0 and no edges */
while (T contains fewer than n-1 edges) {
     let (u,v) be a least cost edge such that u ∈ TV and v ∉ TV;
     if (there is no such edge)
          break;
     add v to TV;
     add (u,v) to T;
}
if (T contains fewer than n-1 edges)
   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



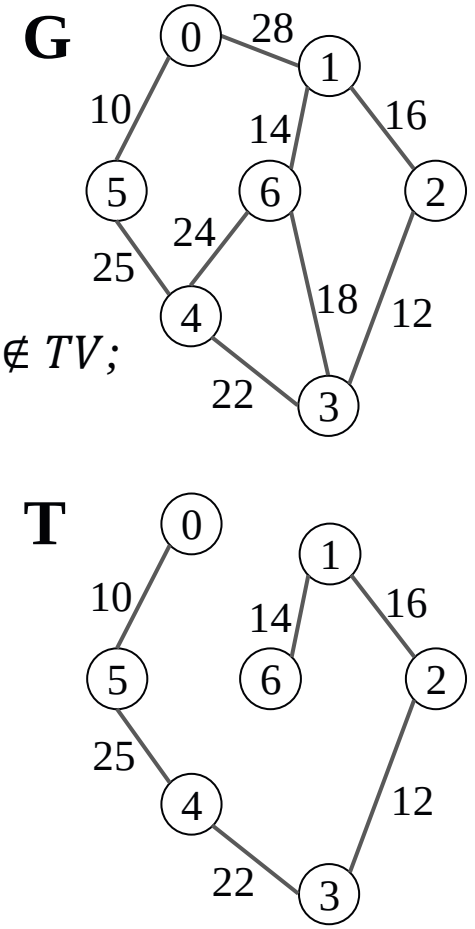
```
T = {};
TV = {0};   /* start with vertex 0 and no edges */
while (T contains fewer than n-1 edges) {
    let (u,v) be a least cost edge such that u ∈ TV and v ∉ TV;
    if (there is no such edge)
        break;
    add v to TV;
    add (u,v) to T;
}
if (T contains fewer than n-1 edges)
    printf("No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



```
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");
```



```
T = {};
TV = {0};   /* start with vertex 0 and no edges */
while (T contains fewer than n-1 edges) {   → break
     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");
```



Implementation :

We assume that each vertex v that is not in TV has a companion vertex, $near(v)$, such that $near(v) \in TV$ and $cost(near(v), v)$ is minimum over all such choices for $near(v)$.

At each stage we select v so that $cost(near(v), v)$ is minimum and $v \notin TV$.

Computing time is $O(n^2)$, where n is the number of vertices in G.

6.4 SHORTEST PATHS AND TRANSITIVE CLOSURE

Suppose we have a graph that represents the highway system.
In this graph, the vertices represent cities and
edges represent sections of the highway.
Each edge has a weight representing the distance between the two cities connected by the edge.

Questions (from a motorist wishing to drive from city A to B):

(1) Is there a path from A to B?
(2) If there is more than one path from A to B,
which path is the shortest?

We define the length of a path as the sum of the weights of the edges on that path.

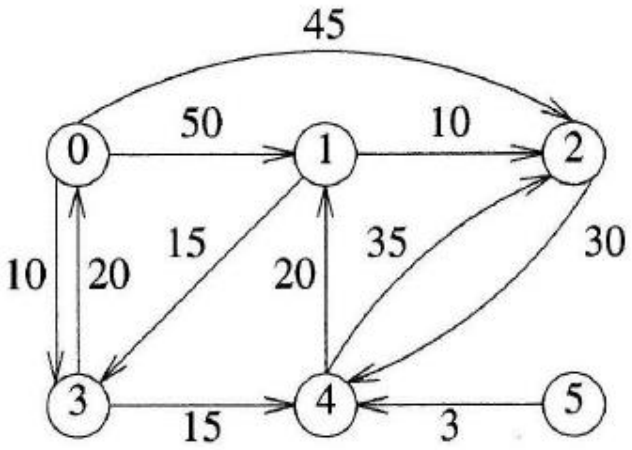
We assume a directed graph.

6.4.1 Single Source/All Destinations: Nonnegative Edge Costs

In this problem we are given a directed graph $G = (V, E)$, a weighting function, $w(e)$, $w(e) > 0$, for the edges of G, and a source vertex, v_0.

We wish to determine a shortest path from v_0 to each of the remaining vertices of G.

[Figure 6.26] Graph and shortest paths from vertex 0 to all destinations



(a) Graph

	Path	Length
1)	0, 3	10
2)	0, 3, 4	25
3)	0, 3, 4, 1	45
4)	0, 2	45

(b) Shortest paths from 0

We may use a greedy algorithm to generate the shortest paths in nondecreasing order of their lengths.

Let S be the set of vertices, including v_0, whose shortest paths have been found. For w not in S, let $distance[w]$ be the length of the shortest path starting from v_0, going through vertices only in S, and ending in w.

Observations:

(1) If the next shortest path is to vertex u, then the path from v_0 to u goes through only those vertices that are in S.

(2) Vertex u is chosen so that it has the minimum distance, $distance[u]$, among all the vertices not in S.

(3) Once we have selected u and generated the shortest path from v_0 to u, u becomes a member of S. Adding u to S can change the distance of shortest paths starting at v_0, going through vertices only in S, and ending at a vertex w, that is not currently in S.

(1) If the next shortest path is to vertex u, then the path from v_0 to u goes through only those vertices that are in S.

<proof>
Assume that there is a vertex w on this path that is not in S.

Then, the path from v_0 to u also contains a path from v_0 to w,
which has a length that is less than the length of the path from v_0 to u.

Since we assume that the shortest paths are generated in nondecreasing order of path length,
we must have previously generated the path from v_0 to w.
$\rightarrow$ This is obviously a contraction.

Therefore, there cannot be any intermediate vertex that is not in S.

(2) Vertex u is chosen so that it has the minimum distance, $distance[u]$, among all the vertices not in S.

- This follows from the definition of *distance* and observation (1).
- If there are several vertices not in S with the same distance, then we may select any one of them.

(3) Once we have selected u and generated the shortest path from v_0 to u, u becomes a member of S. Adding u to S can change the distance of shortest paths starting at v_0, going through vertices only in S, and ending at a vertex w, that is not currently in S.

- If the distance changes, we have found a shorter such path from v_0 to w. This path goes through u.
- The intermediate vertices on this path are in S and its subpath from u to w can be chosen so as to have no intermediate vertices.
- The length of the shorter path is $distance[u] + length(< u, w >)$.

Implementing Dijkstra's algorithm

We assume that the n vertices are numbered from 0 to $n-1$.

We maintain the set S as an array, *found*, with
found[i] = FALSE if vertex i is not in S and
found[i] = TRUE if vertex i is in S.

We represent the graph by its cost adjacency matrix, with cost[i][j] being the weight of edge <i, j>.

If the edge <i, j> is not in G, we set cost[i][j] to some large number.
The choice of this number is arbitrary, but we make two stipulations:

(1) The number must be larger than any of the value in the cost matrix.
(2) The number must be chosen so that distance[u] + cost[u][w] does not produce an overflow into the sign bit.

Declarations for the shortest path algorithm

```
#define MAX_VERTICES   6  /* maximum number of vertices*/
int cost[][MAX_VERTICES] =
                        {{    0,    50,    10,  1000,    45,  1000},
                         { 1000,     0,    15,  1000,    10,  1000},
                         {   20,  1000,     0,    15,  1000,  1000},
                         { 1000,    20,  1000,     0,    35,  1000},
                         { 1000,  1000,    30,  1000,     0,  1000},
                         { 1000,  1000,  1000,     3,  1000,     0}};
int distance[MAX_VERTICES];
short int found[MAX_VERTICES];
int n = MAX_VERTICES;
```

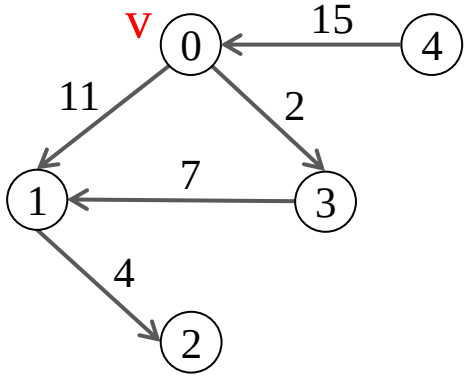
[Program 6.9] Single source shortest paths

```
void shortestpath(int v, int cost[][MAX_VERTICES], int distance[], int n, short  int found[])
{ /* distance[i] represents the shortest path from vertex v to i, found[i] holds 0 if the shortest path from
      vertex i has not been found and 1 if it has. cost is the adjacency matrix  */
    int i, u, w;
    for (i=0; i<n; i++)  {
        found[i] = FALSE;
        distance[i] = cost[v][i];
    }
    found[v] = TRUE;
    distance[v] = 0;
    for (i=0; i<n-2; i++)  {
        u = choose(distance, n, found);
        found[u] = TRUE;
        for (w=0; w<n; w++)
            if (!found[w])
                if (distance[u] + cost[u][w] < distance[w])
                    distance[w] = distance[u] + cost[u][w];
    }
}
```

[Program 6.10] Choosing the least cost edge

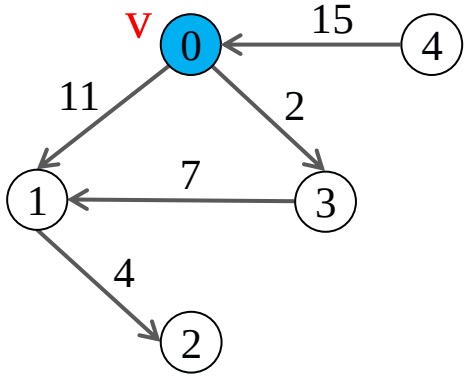
```
int choose(int distance[], int n, short int found[])
{ /* find the smallest distance not yet checked  */
     int i, min, minpos;
     min = INT_MAX;
     minpos = -1;
     for (i=0; i<n; i++)
          if (distance[i] < min && !found[i])  {
               min = distance[i];
               minpos = i;
          }
     return minpos;
}
```

```
void shortestpath(…)
{
    int i, u, w;
    for (i=0; i<n; i++)  {
        found[i] = FALSE;
        distance[i] = cost[v][i];
    }
    found[v] = TRUE;
    distance[v] = 0;
    for (i=0; i<n-2; i++)  {
        u = choose(distance, n, found);
        found[u] = TRUE;
        for (w=0; w<n; w++)
            if (!found[w])
                if (distance[u] + cost[u][w] < distance[w])
                    distance[w] = distance[u] + cost[u][w];
    }
}
```



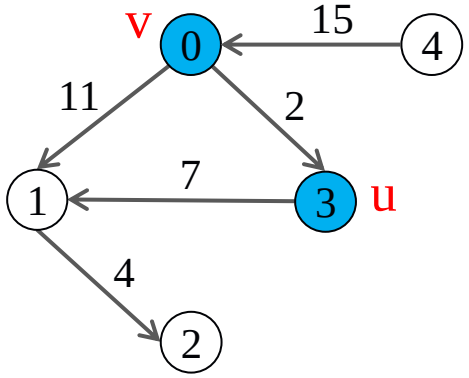
	[0]	[1]	[2]	[3]	[4]
found	FALSE	FALSE	FALSE	FALSE	FALSE
distance	0	11	∞	2	∞

```
void shortestpath(…)
{
    int i, u, w;
    for (i=0; i<n; i++)  {
        found[i] = FALSE;
        distance[i] = cost[v][i];
    }
    found[v] = TRUE;
    distance[v] = 0;
    for (i=0; i<n-2; i++)  {
        u = choose(distance, n, found);
        found[u] = TRUE;
        for (w=0; w<n; w++)
            if (!found[w])
                if (distance[u] + cost[u][w] < distance[w])
                    distance[w] = distance[u] + cost[u][w];
    }
}
```



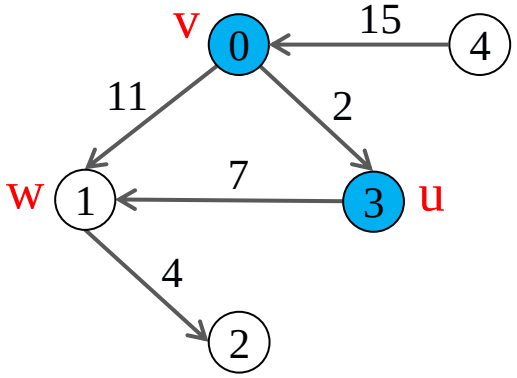
	[0]	[1]	[2]	[3]	[4]
found	TRUE	FALSE	FALSE	FALSE	FALSE
distance	0	11	∞	2	∞

```
void shortestpath(…)
{
    int i, u, w;
    for (i=0; i<n; i++)  {
        found[i] = FALSE;
        distance[i] = cost[v][i];
    }
    found[v] = TRUE;
    distance[v] = 0;
    for (i=0; i<n-2; i++)  {
        u = choose(distance, n, found);
        found[u] = TRUE;
        for (w=0; w<n; w++)
            if (!found[w])
                if (distance[u] + cost[u][w] < distance[w])
                    distance[w] = distance[u] + cost[u][w];
    }
}
```



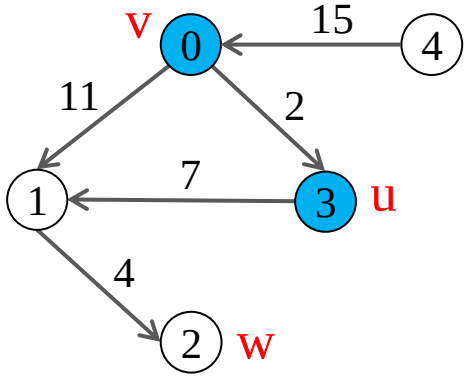
	[0]	[1]	[2]	[3]	[4]
found	TRUE	FALSE	FALSE	TRUE	FALSE
distance	0	11	∞	2	∞

```
void shortestpath(…)
{
    int i, u, w;
    for (i=0; i<n; i++)  {
        found[i] = FALSE;
        distance[i] = cost[v][i];
    }
    found[v] = TRUE;
    distance[v] = 0;
    for (i=0; i<n-2; i++)  {
        u = choose(distance, n, found);
        found[u] = TRUE;
        for (w=0; w<n; w++)
            if (!found[w])
                if (distance[u] + cost[u][w] < distance[w])
                    distance[w] = distance[u] + cost[u][w];
    }
}
```



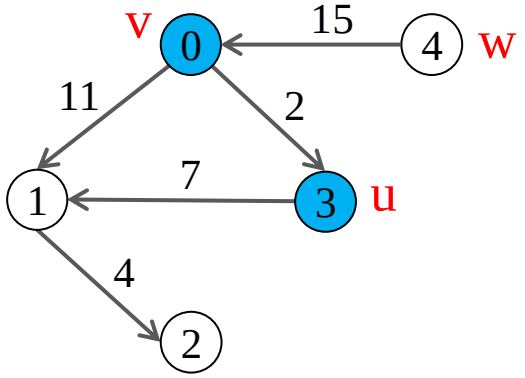
	[0]	[1]	[2]	[3]	[4]
found	TRUE	FALSE	FALSE	TRUE	FALSE
distance	0	9	∞	2	∞

```
void shortestpath(…)
{
    int i, u, w;
    for (i=0; i<n; i++)  {
        found[i] = FALSE;
        distance[i] = cost[v][i];
    }
    found[v] = TRUE;
    distance[v] = 0;
    for (i=0; i<n-2; i++)  {
        u = choose(distance, n, found);
        found[u] = TRUE;
        for (w=0; w<n; w++)
            if (!found[w])
                if (distance[u] + cost[u][w] < distance[w])
                    distance[w] = distance[u] + cost[u][w];
    }
}
```



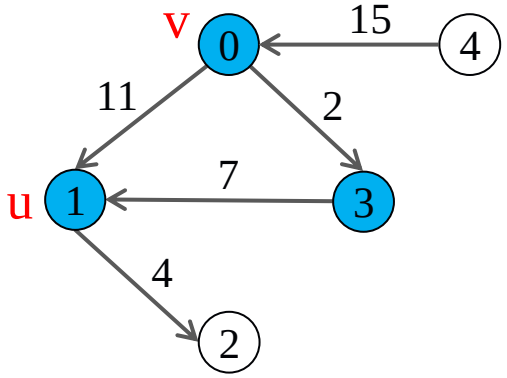
	[0]	[1]	[2]	[3]	[4]
found	TRUE	FALSE	FALSE	TRUE	FALSE
distance	0	9	∞	2	∞

```
void shortestpath(…)
{
    int i, u, w;
    for (i=0; i<n; i++)  {
        found[i] = FALSE;
        distance[i] = cost[v][i];
    }
    found[v] = TRUE;
    distance[v] = 0;
    for (i=0; i<n-2; i++)  {
        u = choose(distance, n, found);
        found[u] = TRUE;
        for (w=0; w<n; w++)
            if (!found[w])
                if (distance[u] + cost[u][w] < distance[w])
                    distance[w] = distance[u] + cost[u][w];
    }
}
```



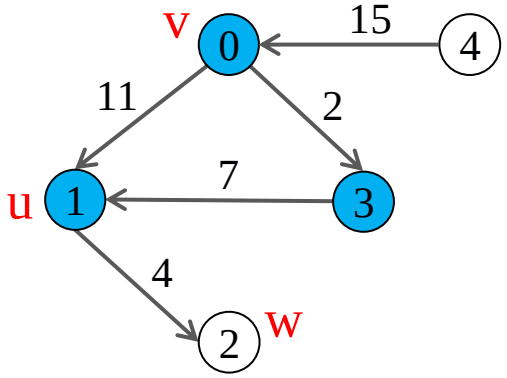
	[0]	[1]	[2]	[3]	[4]
found	TRUE	FALSE	FALSE	TRUE	FALSE
distance	0	9	∞	2	∞

```
void shortestpath(…)
{
    int i, u, w;
    for (i=0; i<n; i++)  {
        found[i] = FALSE;
        distance[i] = cost[v][i];
    }
    found[v] = TRUE;
    distance[v] = 0;
    for (i=0; i<n-2; i++)  {
        u = choose(distance, n, found);
        found[u] = TRUE;
        for (w=0; w<n; w++)
            if (!found[w])
                if (distance[u] + cost[u][w] < distance[w])
                    distance[w] = distance[u] + cost[u][w];
    }
}
```



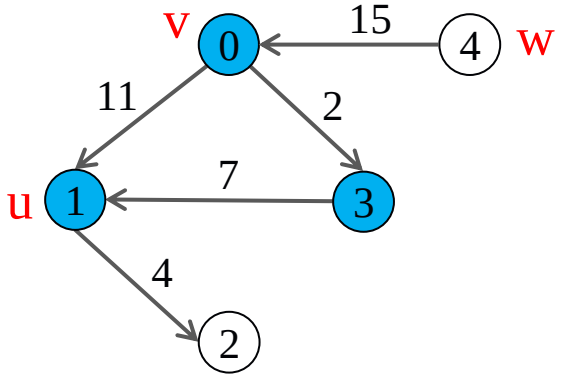
	[0]	[1]	[2]	[3]	[4]
found	TRUE	TRUE	FALSE	TRUE	FALSE
distance	0	9	∞	2	∞

```
void shortestpath(…)
{
    int i, u, w;
    for (i=0; i<n; i++)  {
        found[i] = FALSE;
        distance[i] = cost[v][i];
    }
    found[v] = TRUE;
    distance[v] = 0;
    for (i=0; i<n-2; i++)  {
        u = choose(distance, n, found);
        found[u] = TRUE;
        for (w=0; w<n; w++)
            if (!found[w])
                if (distance[u] + cost[u][w] < distance[w])
                    distance[w] = distance[u] + cost[u][w];
    }
}
```



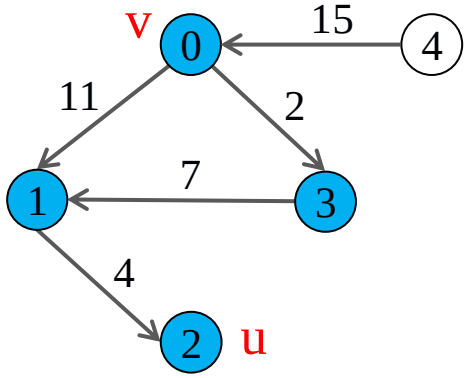
	[0]	[1]	[2]	[3]	[4]
found	TRUE	TRUE	FALSE	TRUE	FALSE
distance	0	9	13	2	∞

```
void shortestpath(…)
{
    int i, u, w;
    for (i=0; i<n; i++)  {
        found[i] = FALSE;
        distance[i] = cost[v][i];
    }
    found[v] = TRUE;
    distance[v] = 0;
    for (i=0; i<n-2; i++)  {
        u = choose(distance, n, found);
        found[u] = TRUE;
        for (w=0; w<n; w++)
            if (!found[w])
                if (distance[u] + cost[u][w] < distance[w])
                    distance[w] = distance[u] + cost[u][w];
    }
}
```



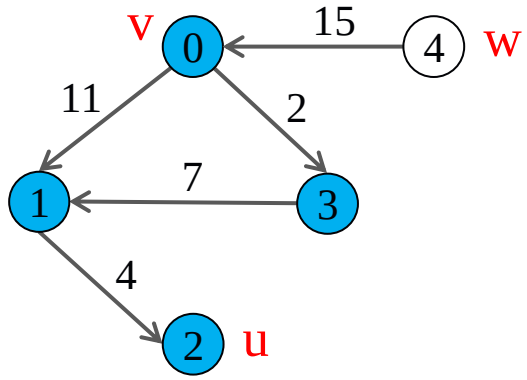
	[0]	[1]	[2]	[3]	[4]
found	TRUE	TRUE	FALSE	TRUE	FALSE
distance	0	9	13	2	∞

```
void shortestpath(…)
{
    int i, u, w;
    for (i=0; i<n; i++)  {
        found[i] = FALSE;
        distance[i] = cost[v][i];
    }
    found[v] = TRUE;
    distance[v] = 0;
    for (i=0; i<n-2; i++)  {
        u = choose(distance, n, found);
        found[u] = TRUE;
        for (w=0; w<n; w++)
            if (!found[w])
                if (distance[u] + cost[u][w] < distance[w])
                    distance[w] = distance[u] + cost[u][w];
    }
}
```



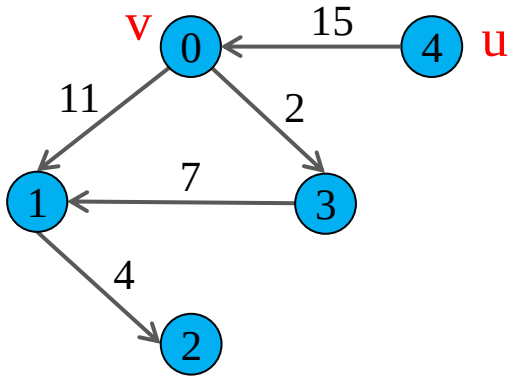
	[0]	[1]	[2]	[3]	[4]
found	TRUE	TRUE	TRUE	TRUE	FALSE
distance	0	9	13	2	∞

```
void shortestpath(…)
{
    int i, u, w;
    for (i=0; i<n; i++)  {
        found[i] = FALSE;
        distance[i] = cost[v][i];
    }
    found[v] = TRUE;
    distance[v] = 0;
    for (i=0; i<n-2; i++)  {
        u = choose(distance, n, found);
        found[u] = TRUE;
        for (w=0; w<n; w++)
            if (!found[w])
                if (distance[u] + cost[u][w] < distance[w])
                    distance[w] = distance[u] + cost[u][w];
    }
}
```



	[0]	[1]	[2]	[3]	[4]
found	TRUE	TRUE	TRUE	TRUE	FALSE
distance	0	9	13	2	∞

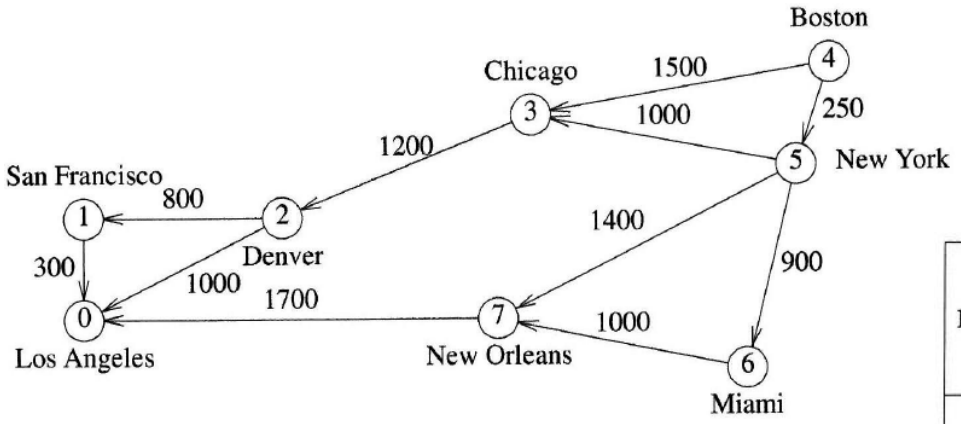
```
void shortestpath(…)
{
    int i, u, w;
    for (i=0; i<n; i++)  {
        found[i] = FALSE;
        distance[i] = cost[v][i];
    }
    found[v] = TRUE;
    distance[v] = 0;
    for (i=0; i<n-2; i++)  {
        u = choose(distance, n, found);
        found[u] = TRUE;
        for (w=0; w<n; w++)
            if (!found[w])
                if (distance[u] + cost[u][w] < distance[w])
                    distance[w] = distance[u] + cost[u][w];
    }
}
```



	[0]	[1]	[2]	[3]	[4]
found	TRUE	TRUE	TRUE	TRUE	TRUE
distance	0	9	13	2	∞

Analysis of *shortestpath* : $O(n^2)$

Example 6.4:



(a) Digraph

	0	1	2	3	4	5	6	7
0	0							
1	300	0						
2	1000	800	0					
3			1200	0				
4				1500	0	250		
5				1000		0	900	1400
6							0	1000
7	1700							0

(b) Length-adjacency matrix

Iteration	Vertex selected	Distance							
		LA	SF	DEN	CHI	BOST	NY	MIA	NO
		[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
Initial	----	∞	∞	∞	1500	0	250	∞	∞
1	5	∞	∞	∞	1250	0	250	1150	1650
2	6	∞	∞	∞	1250	0	250	1150	1650
3	3	∞	∞	2450	1250	0	250	1150	1650
4	7	3350	∞	2450	1250	0	250	1150	1650
5	2	3350	3250	2450	1250	0	250	1150	1650
6	1	3350	3250	2450	1250	0	250	1150	1650

6.4.2 Single Source/All Destinations: General Weights

Dijkstra's algorithm does not work for graphs with negative weights

Figure 6.29

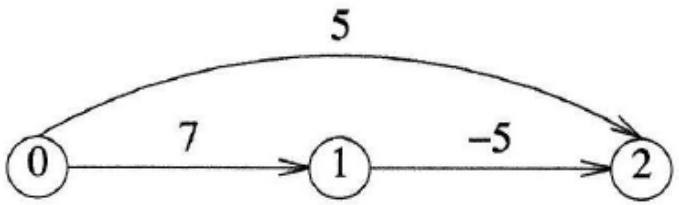
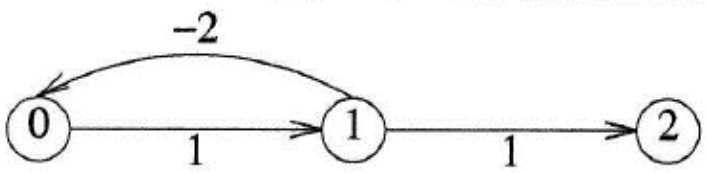


Figure 6.30



Bellman-Ford algorithm solves this problem without cycles of negative length.

$dist^{l}[u]$: length of a shortest path from the source vertex v to vertex u under the constraint that the shortest path contains at most l edges

Goal : compute $dist^{n-1}[u]$ for all u

Observations:

1. If the shortest path from v to u with at most k, $k > 1$, edges has no more than $k-1$ edges, then $dist^k[u] = dist^{k-1}[u]$.

2. If the shortest path from v to u with at most k, $k > 1$, edges has exactly k edges, then it is comprised of a shortest path from v to some vertex j followed by the edge $< j, u >$. The path from v to j has $k-1$ edges, and its length is $dist^{k-1}[j]$. All vertices i such that the edge $< i, u >$ is in the graph are candidates for j. Since we are interested in a shortest path, the i that minimizes $dist^{k-1}[i] + length[i][u]$ is the correct value for j.

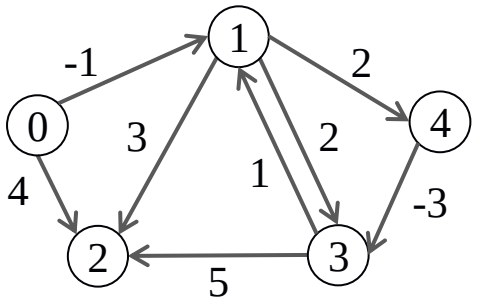
$$\rightarrow \quad dist^k[u] = \min\{dist^{k-1}[u], \min_i\{dist^{k-1}[i] + length[i][u]\}\}$$

[Program 6.11] Bellman and Ford algorithm to compute shortest paths

```
void BellmanFord(int n, int v)
{ /* Single source all destination shortest paths
      with negative edge lengths */
     for (int i=0; i<n; i++)
          dist[i] = length[v][i];  /* initialize dist */
     for (int k=2; k<=n-1; k++)
          for (each u such that u != v and u has at least one incoming edge)
               for (each <i, u> in the graph)
                    if (dist[u] > dist[i] + length[i][u])
                         dist[u] = dist[i] + length[i][u];
}
```

Time complexity: $O(n^3)$ with adjacency matrix, $O(ne)$ with list

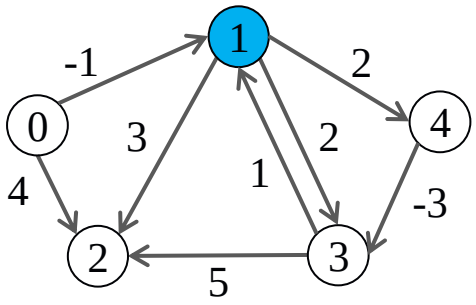
```
void BellmanFord(int n, int v)
{
    for (int i=0; i<n; i++)
        dist[i] = length[v][i];
    for (int k=2; k<=n-1; k++)
        for (each u such that u != v and u has at least one incoming edge)
            for (each <i, u> in the graph)
                if (dist[u] > dist[i] + length[i][u])
                    dist[u] = dist[i] + length[i][u];
}
```



n	5
v	0
k	
u	
i	

	[0]	[1]	[2]	[3]	[4]
dist	0	-1	4	∞	∞

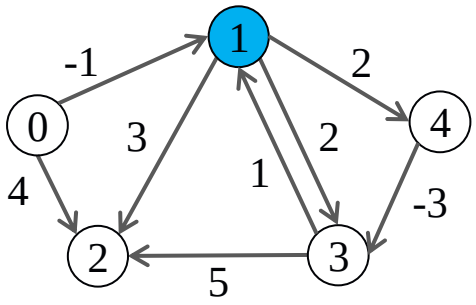
```
void BellmanFord(int n, int v)
{
    for (int i=0; i<n; i++)
        dist[i] = length[v][i];
    for (int k=2; k<=n-1; k++)
        for (each u such that u != v and u has at least one incoming edge)
            for (each <i, u> in the graph)
                if (dist[u] > dist[i] + length[i][u])
                    dist[u] = dist[i] + length[i][u];
}
```



n	5
v	0
k	2
u	1
i	

	[0]	[1]	[2]	[3]	[4]
dist	0	-1	4	∞	∞

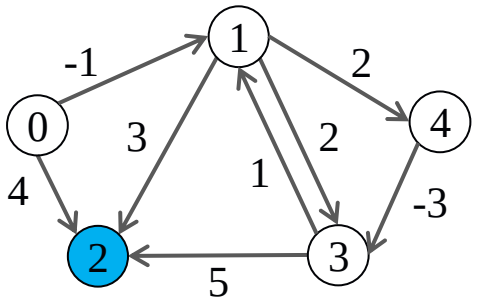
```
void BellmanFord(int n, int v)
{
    for (int i=0; i<n; i++)
        dist[i] = length[v][i];
    for (int k=2; k<=n-1; k++)
        for (each u such that u != v and u has at least one incoming edge)
            for (each <i, u> in the graph)
                if (dist[u] > dist[i] + length[i][u])
                    dist[u] = dist[i] + length[i][u];
}
```



n	5
v	0
k	2
u	1
i	0

	[0]	[1]	[2]	[3]	[4]
dist	0	-1	4	∞	∞

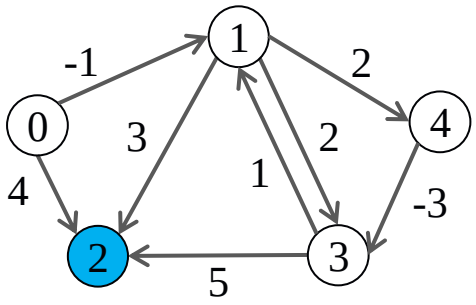
```
void BellmanFord(int n, int v)
{
    for (int i=0; i<n; i++)
        dist[i] = length[v][i];
    for (int k=2; k<=n-1; k++)
        for (each u such that u != v and u has at least one incoming edge)
            for (each <i, u> in the graph)
                if (dist[u] > dist[i] + length[i][u])
                    dist[u] = dist[i] + length[i][u];
}
```



n	5
v	0
k	2
u	2
i	

	[0]	[1]	[2]	[3]	[4]
dist	0	-1	4	∞	∞

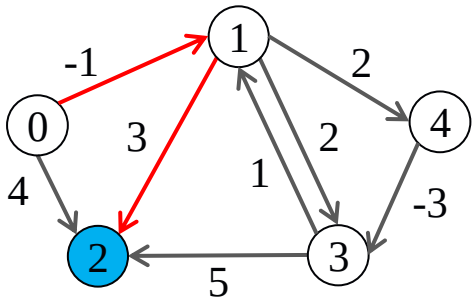
```
void BellmanFord(int n, int v)
{
    for (int i=0; i<n; i++)
        dist[i] = length[v][i];
    for (int k=2; k<=n-1; k++)
        for (each u such that u != v and u has at least one incoming edge)
            for (each <i, u> in the graph)
                if (dist[u] > dist[i] + length[i][u])
                    dist[u] = dist[i] + length[i][u];
}
```



n	5
v	0
k	2
u	2
i	0

	[0]	[1]	[2]	[3]	[4]
dist	0	-1	4	∞	∞

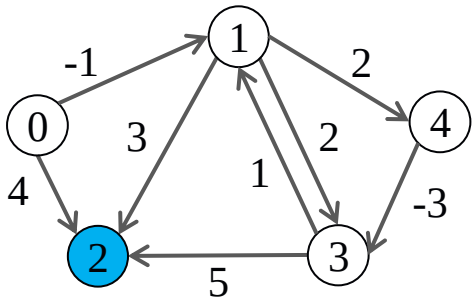
```
void BellmanFord(int n, int v)
{
    for (int i=0; i<n; i++)
        dist[i] = length[v][i];
    for (int k=2; k<=n-1; k++)
        for (each u such that u != v and u has at least one incoming edge)
            for (each <i, u> in the graph)
                if (dist[u] > dist[i] + length[i][u])
                    dist[u] = dist[i] + length[i][u];
}
```



n	5
v	0
k	2
u	2
i	1

	[0]	[1]	[2]	[3]	[4]
dist	0	-1	2	∞	∞

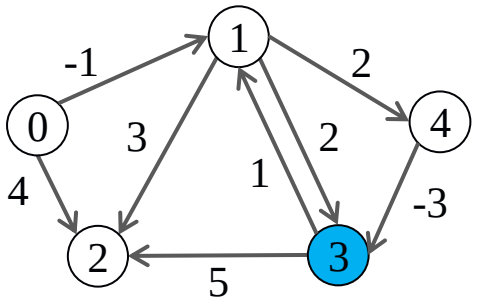
```
void BellmanFord(int n, int v)
{
    for (int i=0; i<n; i++)
        dist[i] = length[v][i];
    for (int k=2; k<=n-1; k++)
        for (each u such that u != v and u has at least one incoming edge)
            for (each <i, u> in the graph)
                if (dist[u] > dist[i] + length[i][u])
                    dist[u] = dist[i] + length[i][u];
}
```



n	5
v	0
k	2
u	2
i	3

	[0]	[1]	[2]	[3]	[4]
dist	0	-1	2	∞	∞

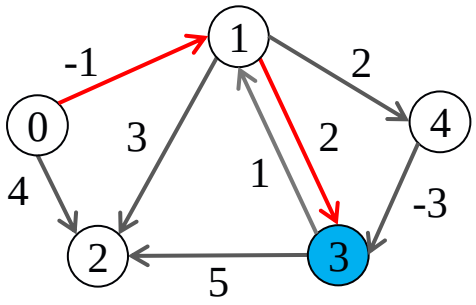
```
void BellmanFord(int n, int v)
{
    for (int i=0; i<n; i++)
        dist[i] = length[v][i];
    for (int k=2; k<=n-1; k++)
        for (each u such that u != v and u has at least one incoming edge)
            for (each <i, u> in the graph)
                if (dist[u] > dist[i] + length[i][u])
                    dist[u] = dist[i] + length[i][u];
}
```



n	5
v	0
k	2
u	3
i	

	[0]	[1]	[2]	[3]	[4]
dist	0	-1	2	∞	∞

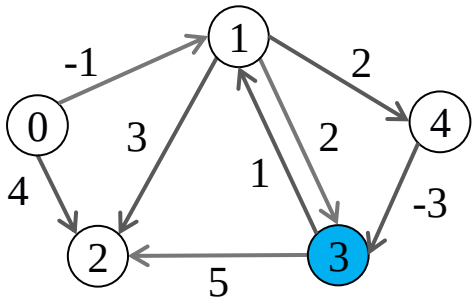
```
void BellmanFord(int n, int v)
{
    for (int i=0; i<n; i++)
        dist[i] = length[v][i];
    for (int k=2; k<=n-1; k++)
        for (each u such that u != v and u has at least one incoming edge)
            for (each <i, u> in the graph)
                if (dist[u] > dist[i] + length[i][u])
                    dist[u] = dist[i] + length[i][u];
}
```



n	5
v	0
k	2
u	3
i	1

	[0]	[1]	[2]	[3]	[4]
dist	0	-1	2	1	∞

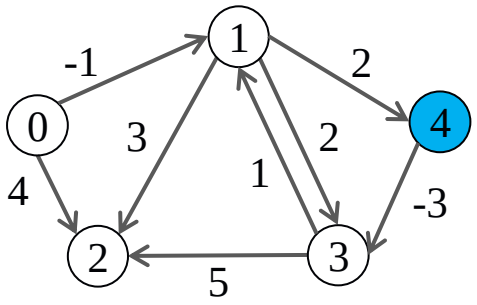
```
void BellmanFord(int n, int v)
{
    for (int i=0; i<n; i++)
        dist[i] = length[v][i];
    for (int k=2; k<=n-1; k++)
        for (each u such that u != v and u has at least one incoming edge)
            for (each <i, u> in the graph)
                if (dist[u] > dist[i] + length[i][u])
                    dist[u] = dist[i] + length[i][u];
}
```



n	5
v	0
k	2
u	3
i	4

	[0]	[1]	[2]	[3]	[4]
dist	0	-1	2	1	∞

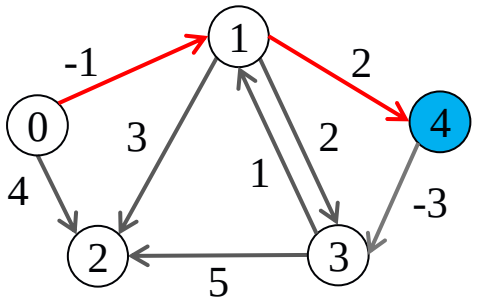
```
void BellmanFord(int n, int v)
{
    for (int i=0; i<n; i++)
        dist[i] = length[v][i];
    for (int k=2; k<=n-1; k++)
        for (each u such that u != v and u has at least one incoming edge)
            for (each <i, u> in the graph)
                if (dist[u] > dist[i] + length[i][u])
                    dist[u] = dist[i] + length[i][u];
}
```



n	5
v	0
k	2
u	4
i	

	[0]	[1]	[2]	[3]	[4]
dist	0	-1	2	1	∞

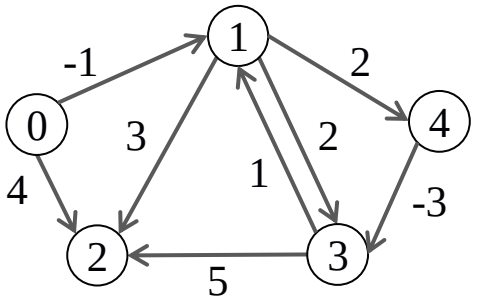
```
void BellmanFord(int n, int v)
{
    for (int i=0; i<n; i++)
        dist[i] = length[v][i];
    for (int k=2; k<=n-1; k++)
        for (each u such that u != v and u has at least one incoming edge)
            for (each <i, u> in the graph)
                if (dist[u] > dist[i] + length[i][u])
                    dist[u] = dist[i] + length[i][u];
}
```



n	5
v	0
k	2
u	4
i	1

	[0]	[1]	[2]	[3]	[4]
dist	0	-1	2	1	1

After all iterations :



	[0]	[1]	[2]	[3]	[4]
dist	0	-1	2	-2	1