# Chapter 22: Graphs

## GRAPHS

### Introduction

*Graphs* are natural models that are used to represent arbitrary relationships among data objects. We often need to represent such arbitrary relationships among the data objects while dealing with problems in computer science, engineering, and many other disciplines. Therefore, the study of graphs as one of the basic data structures is important.

### Basic Definitions and Terminology

A graph is a structure made of two components: a set of vertices V, and a set of edges E. Therefore, a graph is G = (V,E), where G is a graph. The graph may be directed or undirected. In a *directed graph*, every edge of the graph is an ordered pair of vertices connected by the edge, whereas in an *undirected graph*, every edge is an unordered pair of vertices connected by the edge. Figure 22.1 shows an undirected and a directed graph.

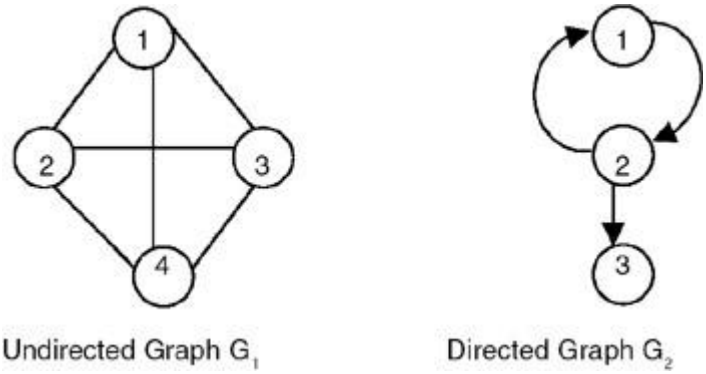



Figure 22.1: Graphs.

**Incident edge:** $(v_i,v_j)$ is an edge, then edge$(v_i,v_j)$ is said to be incident to vertices $v_i$ and $v_j$. For example, in graph $G_1$ shown in Figure 22.1, the edges incident on vertex 1 are (1,2), (1,4), and (1,3), whereas in $G_2$, the edges incident on vertex 1 are (1,2).

**Degree of vertex:** The number of edges incident onto the vertex. For example, in graph $G_1$, the degree of vertex 1 is 3, because 3 edges are incident onto it. For a directed graph, we need to define indegree and outdegree. *Indegree* of a vertex vi is the number of edges incident onto vi, with vi as the head. *Outdegree* of vertex vi is the number of edges incident onto vi, with vi as the tail. For graph $G_2$, the indegree of vertex 2 is 1, whereas the outdegree of vertex 2 is 2.

**Directed edge:** A directed edge between the vertices vi and vj is an ordered pair. It is denoted by <vi,vj>.

**Undirected edge:** An undirected edge between the vertices $v_i$ and $v_j$ is an unordered pair. It is denoted by $(v_i,v_j)$.

**Path:** A path between vertices $v_p$ and $v_q$ is a sequence of vertices $v_p$, $v_{i1}$, $v_{i2}$,…, $v_{in}$,$v_q$ such that there exists a sequence of edges $(v_p, v_{i1})$, $(v_{i1}, v_{i2})$, …, $(v_{in}, v_q)$. In case of a directed graph, a path between the vertices $v_p$ and $v_q$ is a sequence of vertices $v_p$, $v_{i1}$, $v_{i2}$,…, $v_{in}$, $v_q$ such that there exists a sequence of edges $<v_p, v_{i1}>$, $< v_{i1}, v_{i2}>$, …, $<v_{in}, v_q>$. If there exists a path from vertex $v_p$ to $v_q$ in an undirected graph, then there always exists a path from $v_q$ to $v_p$ also. But, in the case of a directed graph, if there exists a path from vertex $v_p$ to $v_q$, then it does not necessarily imply that there exists a path from $v_q$ to $v_p$ also.

**Simple path:** A simple path is a path given by a sequence of vertices in which all vertices are distinct except the first and the last vertices. If the first and the last vertices are same, the path will be a cycle.

**Maximum number of edges:** The maximum number of edges in an undirected graph with *n* vertices is $n(n-1)/2$. In a directed graph, it is $n(n-1)$.

**Subgraph:** A *subgraph* of a graph G = (V,E) is a graph G where V(G) is a subset of V(G). E(G) consists of edges (v1,v2) in E(G), such that both v1 and v2 are in V(G). [Note: If G = (V,E) is a graph, then V(G) is a set of vertices of G and E(G) is a set of edges of G.]

If E(G) consists of all edges (v1,v2) in E(G), such that both v1 and v2 are in V(G), then G is called an induced subgraph of G. For example, the graph shown in Figure 22.2 is a subgraph of the graph G2.

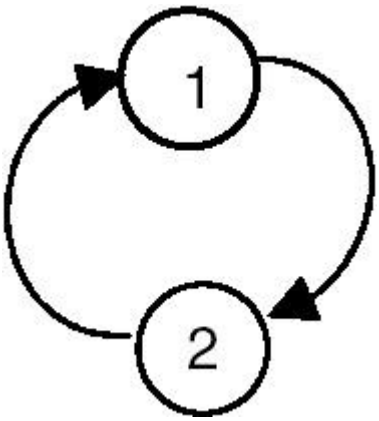


Figure 22.2: The subgraph of graph G2.

For the graph shown in Figure 22.3, one of the induced subgraphs is shown in Figure 22.4.

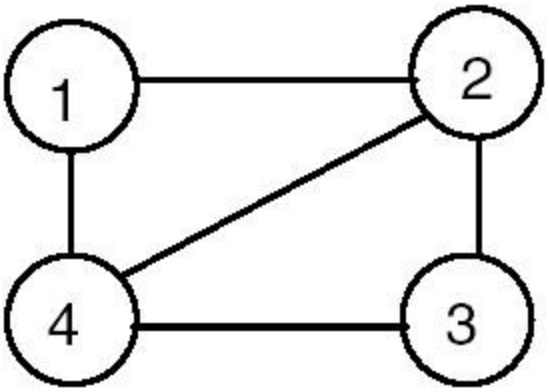


Figure 22.3: Graph G.

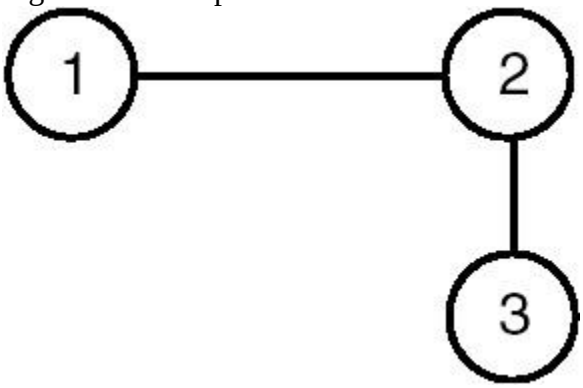


Figure 22.4: Induced subgraph of Graph G of Figure 22.3.

In the undirected graph G, the two vertices $v_1$ and $v_2$ are said to be connected if there exists a path in G from $v_1$ to $v_2$ (being an undirected graph, there exists a path from $v_2$ to $v_1$ also).

**Connected graph:** A graph G is said to be connected if for every pair of distinct vertices $(v_i,v_j)$, there is a path from $v_i$ to $v_j$. A connected graph is shown in Figure 22.5.

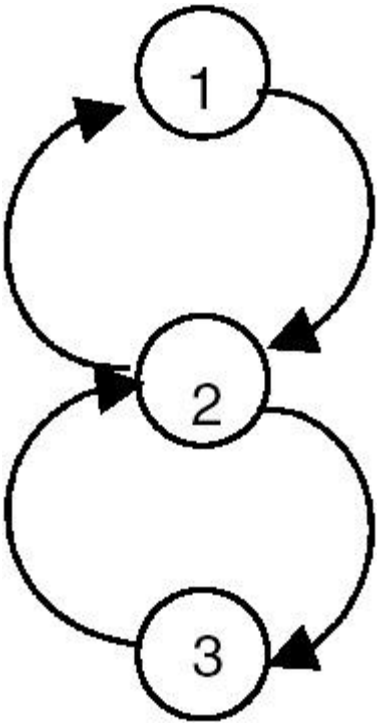


Figure 22.5: A connected graph.

**Completely connected graph:** A graph G is completely connected if, for every pair of distinct vertices ($v_i$,$v_j$), there exists an edge. A completely connected graph is shown in Figure 22.6.

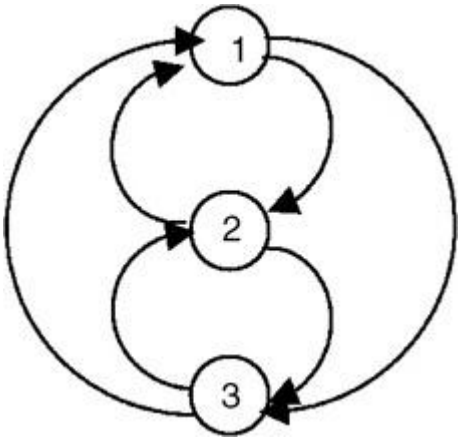


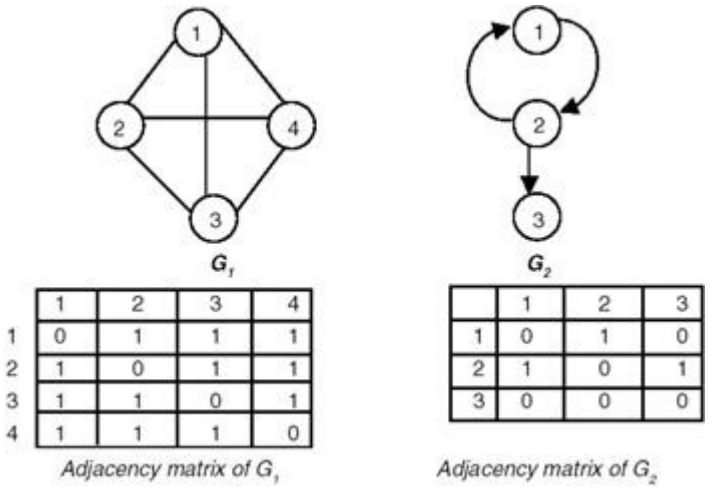
Figure 22.6: A completely connected graph.

PREV < Day Day Up > NEXT
PREV < Day Day Up > NEXT

# REPRESENTATIONS OF A GRAPH

## Array Representation

One way of representing a graph with n vertices is to use an $n^2$ matrix (that is, a matrix with *n* rows and *n* columns—that means there is a row as well as a column corresponding to every vertex of the graph). If there is an edge from $v_i$ to $v_j$ then the entry in the matrix with row index as $v_i$ and column index as $v_j$ is set to 1 (adj[$v_i$, $v_j$] = 1, if ($v_i$, $v_j$) is an edge of graph G). If e is the total number of edges in the graph, then there will 2e entries which will be set to 1, as long as G is an undirected graph. Whereas if G were a directed graph, only e entries would have been set to 1 in the adjacency matrix. The adjacency matrix representation of an undirected as well as a directed graph is show in Figure 22.7.




| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 |
| 2 | 1 | 0 | 1 | 1 |
| 3 | 1 | 1 | 0 | 1 |
| 4 | 1 | 1 | 1 | 0 |

*Adjacency matrix of $G_1$*

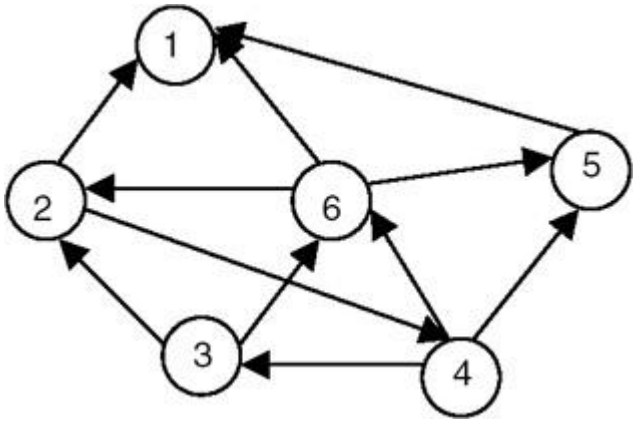
| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 1 | 0 |
| 2 | 1 | 0 | 1 |
| 3 | 0 | 0 | 0 |

*Adjacency matrix of $G_2$*

Figure 22.7: Adjacency matrices.

## Example

The adjacency matrix representation of the following diagraph(directed graph), along with the indegree and outdegree of each node is shown here:




The adjacency matrix representation of the above diagraph is shown here:

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 0 | 1 | 0 | 0 |
| 3 | 0 | 1 | 0 | 0 | 0 | 1 |
| 4 | 0 | 0 | 1 | 0 | 1 | 1 |
| 5 | 1 | 0 | 0 | 0 | 0 | 0 |
| 6 | 1 | 1 | 0 | 0 | 1 | 0 |

The indegree and outdegree of each node is shown here:

| | Indegree | Outdegree |
|---|---|---|
| 1 | 3 | 0 |
| 2 | 2 | 2 |
| 3 | 1 | 2 |
| 4 | 1 | 3 |
| 5 | 2 | 1 |
| 6 | 2 | 3 |

## Linked List Representation

Another way of representing a graph G is to maintain a list for every vertex containing all vertices adjacent to that vertex, as shown in Figure 22.8.

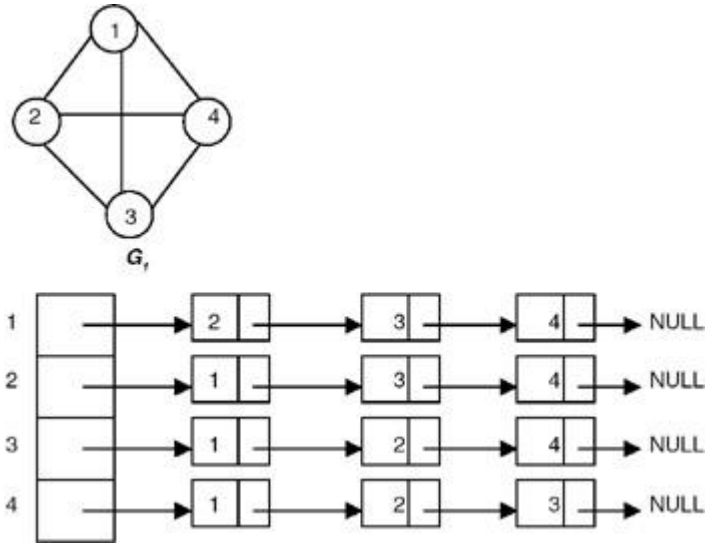



Figure 22.8: Adjacency list of $G_1$.

# COMPUTING INDEGREE AND OUTDEGREE OF A NODE OF A GRAPH USING ADJACENCY MATRIX REPRESENTATION

## Introduction

To compute the indegree of a node n by using the adjacency matrix representation of a graph, use the node number *n* as a column index in the adjacency matrix and count the number of 1's in that column of the adjacency matrix. This count is the indegree of node *n*. Similarly, to compute the outdegree of a node *n* of a graph, use the node number *n* as the row index in the adjacency matrix and count the number of 1's in that row of the adjacency matrix. This is the outdegree of the node *n*. A complete C program to compute the indegree and outdegree of each node of a graph using the adjacency matrix representation of a graph follows.

## Program: Computing the indegree and outdegree

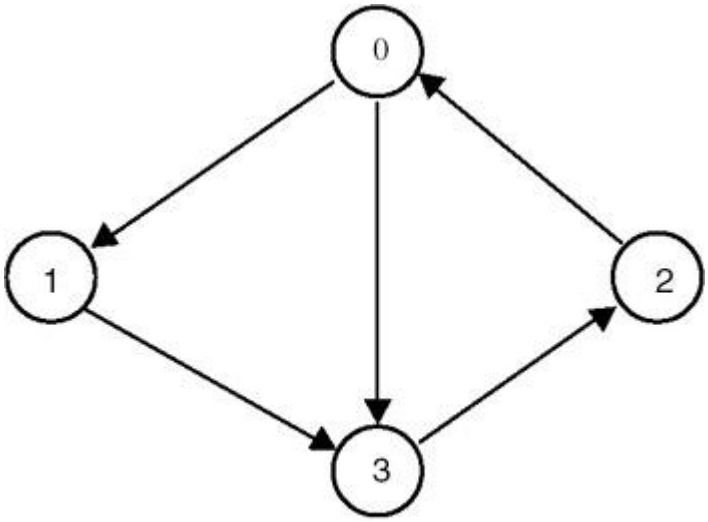
```
#include <stdio.h>
#define MAX 10
/* a function to build an adjacency matrix of the graph*/
void buildadjm(int adj[][MAX], int n)
   {
     int i,j;
     for(i=0;i<n;i++)
         for(j=0;j<n;j++)
         {
         printf("Enter 1 if there is an edge from %d to %d, otherwise
enter 0 \n",
i,j);

         scanf("%d",&adj[i][j]);
         }
   }

/* a function to compute outdegree of a node*/
int outdegree(int adj[][MAX],int x,int n)
    {
       int i, count =0;
       for(i=0;i<n;i++)
           if( adj[x][i] ==1) count++;
      return(count);
    }
   /* a function to compute indegree of a node*/
   int indegree(int adj[][MAX],int x,int n)
       {
          int i, count =0;
          for(i=0;i<n;i++)
```

```
            if( adj[i][x] ==1) count++;
        return(count);
       }
  void main()
     {
       int adj[MAX][MAX],node,n,i;
       printf("Enter the number of nodes in graph maximum = %d\n",MAX);
        scanf("%d",&n);
       buildadjm(adj,n);
       for(i=0;i<n;i++)
       {
          printf("The indegree of the node %d is
%d\n",i,indegree(adj,i,n));
        printf("The outdegree of the node %d is %d\n",
i,outdegree(adj,i,n));
   }
}
```

## Explanation

1. This program uses the adjacency matrix representation of a directed graph to compute the indegree and outdegree of each node of the graph.
2. It first builds an adjacency matrix of the graph by calling a `buildadjm` function, then goes in a loop to compute the indegree and outdegree of each node by calling the `indegree` and `outdegree` functions, respectively.
3. The indegree function counts the number of 1's in a column of an adjacency matrix using the node number whose indegree is to be computed as a column index.
4. The `outdegree` function counts the number of 1's in a row of an adjacency matrix by using the node number whose outdegree is to be computed as a row index.
    - Input: 1. The number of nodes in a graph
      2. Information about edges, in the form of values, to be stored in adjacency matrix 1, if there is an edge from node i to node j; 0 otherwise.
    - Output: The indegree and outdegree of each node.

## Example




Graph $G_1$

The adjacency matrix for graph $G_1$ is:

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 |
| 2 | 1 | 0 | 0 | 0 |
| 3 | 0 | 0 | 1 | 0 |

For this graph as the input, the output is:

- The indegree of node 0 is 1
- The outdgree of node 0 is 2
- The indegree of node 1 is 1
- The outdgree of node 1 is 1
- The indegree of node 2 is 1
- The outdgree of node 2 is 1
- The indegree of node 3 is 2
- The outdgree of node 3 is 1


PREV < Day Day Up > NEXT

# DEPTH-FIRST TRAVERSAL

## Introduction

A graph can be traversed either by using the *depth-first traversal* or *breadth-first traversal*. When a graph is traversed by visiting the nodes in the forward (deeper) direction as long as possible, the traversal is called depth-first traversal. For example, for the graph shown in Figure 22.9, the depth-first traversal starting at the vertex 0 visits the node in the orders:

i. 0 1 2 6 7 8 5 3 4
ii. 0 4 3 5 8 6 7 2 1

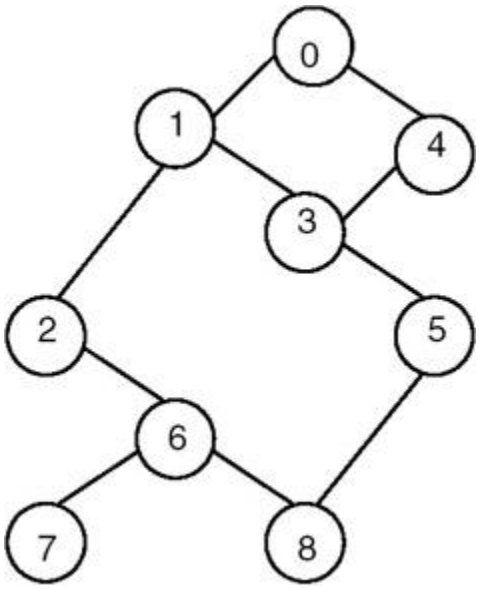


Figure 22.9: Graph G and its depth first traversals starting at vertex 0.

A complete C program for depth-first traversal of a graph follows. It makes use of an array visited of *n* elements where *n* is the number of vertices of the graph, and the elements are Boolean. If visited[i] = 1 then it means that the $i^{th}$ vertex is visited. Initially we set visited[i] = 0.

## Program

```
#include <stdio.h>
#define max 10

/* a function to build adjacency matrix of a graph */
void buildadjm(int adj[][max], int n)
   {
     int i,j;
     for(i=0;i<n;i++)
         for(j=0;j<n;j++)
          {
     printf("enter 1 if there is an edge from %d to %d, otherwise enter
```

```
0 \n",
   i,j);

            scanf("%d",&adj[i][j]);
            }
      }

/* a function to visit the nodes in a depth-first order */
void dfs(int x,int visited[],int adj[][max],int n)
{
   int j;
   visited[x] = 1;
   printf("The node visited id %d\n",x);
   for(j=0;j<n;j++)
      if(adj[x][j] ==1 && visited[j] ==0)
           dfs(j,visited,adj,n);
}
void main()
   {
      int adj[max][max],node,n;
      int i, visited[max];
      printf("enter the number of nodes in graph maximum = %d\n",max);
      scanf("%d",&n);
      buildadjm(adj,n);
      for(i=0; i<n; i++)
        visited[i] =0;
      for(i=0; i<n; i++)
       if(visited[i] ==0)
            dfs(i,visited,adj,n);
      }
```

## Explanation

1. Initially, all the elements of an array named `visited` are set to 0 to indicate that all the vertices are unvisited.
2. The traversal starts with the first vertex (that is, vertex 0), and marks it visited by setting `visited[0]` to 1. It then considers one of the unvisited vertices adjacent to it and marks it `visited`, then repeats the process by considering one of its unvisited adjacent vertices.
3. Therefore, if the following adjacency matrix that represents the graph of Figure 22.9 is given as input, the order in which the nodes are visited is given here:
    - Input: 1. The number of nodes in a graph
      2. Information about edges, in the form of values to be stored in adjacency matrix 1 if there is an edge from node i to node j, 0 otherwise
    - Output: Depth-first ordering of the nodes of the graph starting from the initial vertex, which is vertex 0, in our case.

## Example

### *Input*

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 3 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 4 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 6 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 8 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

## Output

0, 1, 2, 6, 8, 5, 3, 4, 7

## Analysis

1. If the graph G to which the depth-first search (dfs) is applied is represented using adjacency lists, then the vertices y adjacent to x can be determined by following the list of adjacent vertices for each vertex.
2. Therefore, the `for` loop searching for adjacent vertices has the total cost of $d_1 + d_2 + \ldots + d_n$, where $d_i$ is the degree of vertex $v_i$, because the number of nodes in the adjacency list of vertex $v_i$ are $d_i$.
3. If the graph G has n vertices and e edges, then the sum of the degree of each vertex ($d_1 + d_2 + \ldots + d_n$) is 2e. Therefore, there are total of 2e list nodes in the adjacency lists of G. If G is a directed graph, then there are a total of e list nodes only.
4. The algorithm examines each node in the adjacency lists once, at most. So the time required to complete the search is O(e), provided $n <= e$. Instead of using adjacency lists, if an adjacency matrix is used to represent a graph G, then the time required to determine all adjacent vertices of a vertex is O($n$), and since at most $n$ vertices are visited, the total time required is O($n^2$).

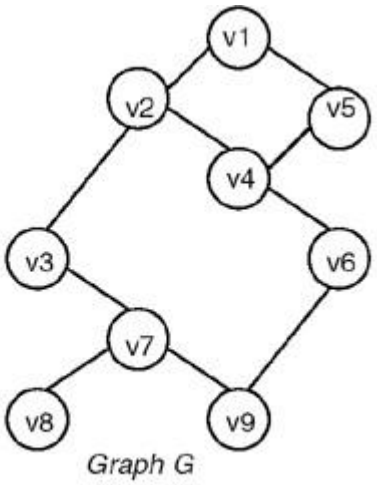
PREV
< Day Day Up >
NEXT
PREV
< Day Day Up >
NEXT

# BREADTH-FIRST TRAVERSAL

## Introduction

When a graph is traversed by visiting all the adjacent nodes/vertices of a node/vertex first, the traversal is called breadth-first traversal. For example, for a graph in which the

breadth-first traversal starts at vertex $v_1$, visits to the nodes take place in the order shown in Figure 22.10.




breadth-first traversal order = v1 v2 v5 v3 v4 v7 v6 v8 v9

Figure 22.10: Breadth-first traversal of graph G starting at vertex v1.

## Program

A complete C program for breadth-first traversal of a graph appears next. The program makes use of an array of *n* visited elements where *n* is the number of vertices of the graph. If `visited[i] = 1`, it means that the $i^{th}$ vertex is visited. The program also makes use of a queue and the procedures `addqueue` and `deletequeue` for adding a vertex to the queue and for deleting the vertex from the queue, respectively. Initially, we set `visited[i] = 0`.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10
struct node
   {
     int data;
     struct node *link;
   };
void buildadjm(int adj[][MAX], int n)
   {
     int i,j;
     printf("enter adjacency matrix \n",i,j);
     for(i=0;i<n;i++)
         for(j=0;j<n;j++)
                 scanf("%d",&adj[i][j]);
   }

/* A function to insert a new node in queue*/
struct node *addqueue(struct node *p,int val)
{
   struct node *temp;
   if(p == NULL)
```

```
   {
      p = (struct node *) malloc(sizeof(struct node)); /* insert the
new node first node*/
      if(p == NULL)
         {
              printf("Cannot allocate\n");
              exit(0);
         }
     p->data = val;
     p->link=NULL;
  }
 else
 {
 temp= p;
while(temp->link != NULL)
{
 temp = temp->link;
}
   temp->link = (struct node*)malloc(sizeof(struct node));
   temp = temp->link;
   if(temp == NULL)
        {
              printf("Cannot allocate\n");
              exit(0);
        }
   temp->data = val;
   temp->link = NULL;
}
return(p);
}
struct node *deleteq(struct node *p,int *val)
{  struct node *temp;
   if(p == NULL)
   {
        printf("queue is empty\n");
               return(NULL);
   }
    *val = p->data;
    temp = p;
    p = p->link;
    free(temp);
    return(p);
  }

void bfs(int adj[][MAX], int x,int visited[], int n, struct node **p)
{
    int y,j,k;
    *p = addqueue(*p,x);
    do{

            *p = deleteq(*p,&y);
            if(visited[y] == 0)
                 {
                      printf("\nnode visited = %d\t",y);
              visited[y] = 1;
                  for(j=0;j<n;j++)
              if((adj[y][j] ==1) && (visited[j] == 0))
```

```
                        *p = addqueue(*p,j);
                }

       }while((*p) != NULL);
}
void main()
   {
     int adj[MAX][MAX];

     int n;
     struct node *start=NULL;
     int i, visited[MAX];
     printf("enter the number of nodes in graph maximum = %d\n",MAX);
     scanf("%d",&n);
     buildadjm(adj,n);
     for(i=0; i<n; i++)
        visited[i] =0;
     for(i=0; i<n; i++)
       if(visited[i] ==0)
            bfs(adj,i,visited,n,&start);
    }
```

## Example

### *Input and Output*

Enter the number of nodes in graph maximum = 10 9

Enter adjacency matrix

```
0 1 0 0 1 0 0 0 0
1 0 1 1 0 0 0 0 0
0 1 0 0 0 0 1 0 0
0 1 0 0 1 1 0 0 0
1 0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0 1
0 0 1 0 0 0 0 1 1
0 0 0 0 0 0 1 0 0
0 0 0 0 0 1 1 0 0
node visited = 0
node visited = 1
node visited = 4
node visited = 2
node visited = 3
node visited = 6
node visited = 5
node visited = 7
node visited = 8
```


PREV < Day Day Up > NEXT

# DEPTH-FIRST SPANNING TREE AND BREADTH-FIRST SPANNING TREE

## Introduction

If graph G is connected, the edges of G can be partitioned into two disjointed sets. One is a set of tree edges, which we denote by set T, and the other is a set of back edges, which we denote by B. The tree edges are precisely those edges that are followed during the depth-first traversal or during the breadth-first traversal of graph G. If we consider only the tree edges, we get a subgraph of G containing all the vertices of G, and this subgraph is a tree called *spanning tree* of the graph G. For example, consider the graph shown in Figure 22.14.

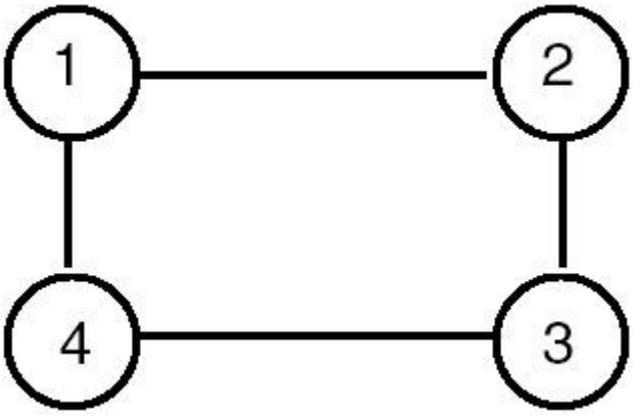


Figure 22.14: Graph G.

One of the depth-first traversal orders for this tree is 1-2-3-4, so the tree edges are (1,2), (2,3) and (3,4). Therefore, one of the spanning trees obtained by using depth-first traversal of the graph of Figure 22.14 is shown in Figure 22.15.

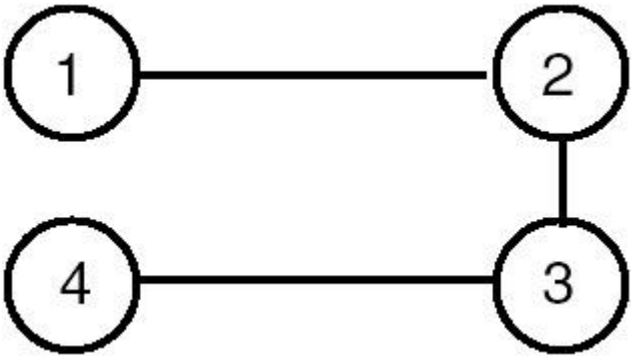


Figure 22.15: Depth first spanning tree of the graph of Figure 22.14.

Similarly, one of the breadth-first traversal orders for this tree is 1-2-4-3, so the tree edges are (1,2), (1,4) and (4,3). Therefore, one of the spanning trees obtained using breadth-first traversal of the graph of Figure 22.14 is shown in Figure 22.16.

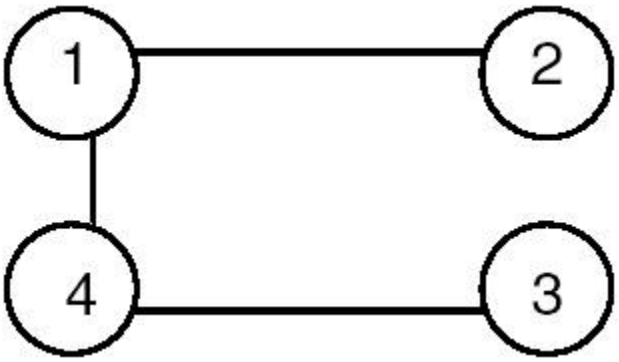


Figure 22.16: Breadth-first spanning tree of the graph of Figure 22.14.

The algorithm for obtaining the depth-first spanning tree (dfst) appears next.

```
T = f; {initially set of tree nodes is empty}
dfst( v : node);
{
   if (visited[v] = false)
   {
      visited[v] = true;
      for every adjacent i of v do
      {
            T = T È {(v,i)}
            dfst(i);
      }
   }
}
```

If a graph G is not connected, the tree edges, which are precisely those edges followed during the depth-first, traversal of the graph G, constitute the depth-first *spanning forest*. The depth-first spanning forest will be made of trees, each of which is one of the connected components of graph G.

When a graph G is directed, the tree edges, which are precisely those edges followed during the depth-first traversal of the graph G, form a depth-first spanning forest for G. In addition to this, there are three other types of edges. These are called *back edges, forward edges*, and *cross edges*. An edge A →B is called a back edge, if B is an ancestor of A in the spanning forest. A non-tree edge that goes from a vertex to a proper descendant is called a forward edge. An edge which goes from a vertex to another vertex that is neither an ancestor nor a descendant is called a cross edge. An edge from a vertex to itself is a back edge. For example, consider the directed graph G shown in Figure 22.17.

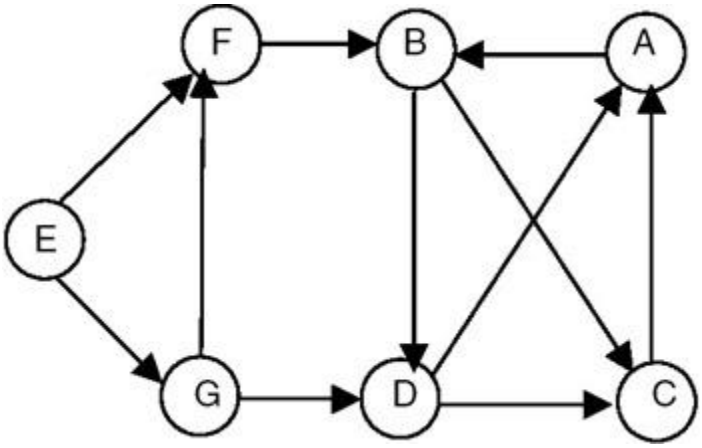


Figure 22.17: A directed graph G.

The depth-first spanning forest for graph G of Figure 22.17 is shown in Figure 22.18.

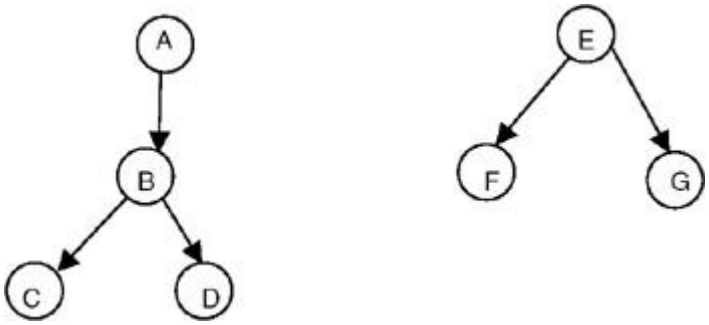


Figure 22.18: Depth-first spanning forest for the graph G of Figure 22.17.

In graph G of Figure 22.17, the edges such as C → A and D → A are the back edges, the edges such as D → C and G → D are cross edges.

## Example

Consider the graph shown in Figure 22.19.

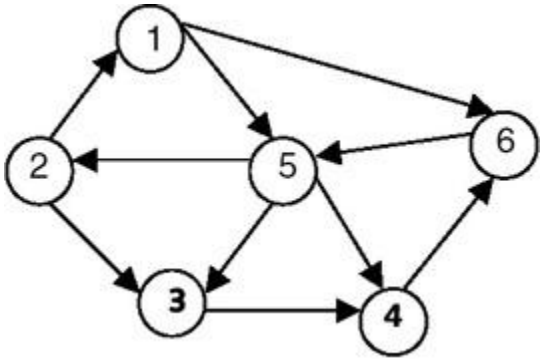


Figure 22.19: A graph G.

If we apply the procedure dfst to this graph, one of the depth-first spanning trees that we get by starting with vertex 1 is shown in Figure 22.20.

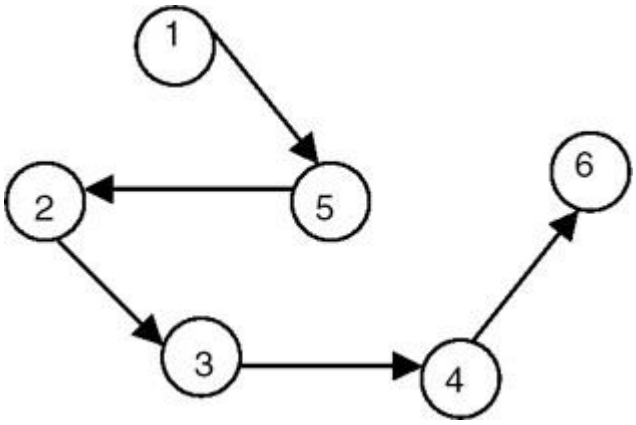



Figure 22.20: Depth-first spanning tree of the graph G of Figure 22.19.



# MINIMUM-COST SPANNING TREE

## Introduction

When the edges of the graph have weights representing the cost in some suitable terms, we can obtain that spanning tree of a graph whose cost is minimum in terms of the weights of the edges. For this, we start with the edge with the minimum-cost/weight, add it to set T, and mark it as visited. We next consider the edge with minimum-cost that is not yet visited, add it to T, and mark it as visited. While adding an edge to the set T, we first check whether both the vertices of the edge are visited; if they are, we do not add to the set T, because it will form a cycle. For example, consider the graph shown in Figure 22.21.

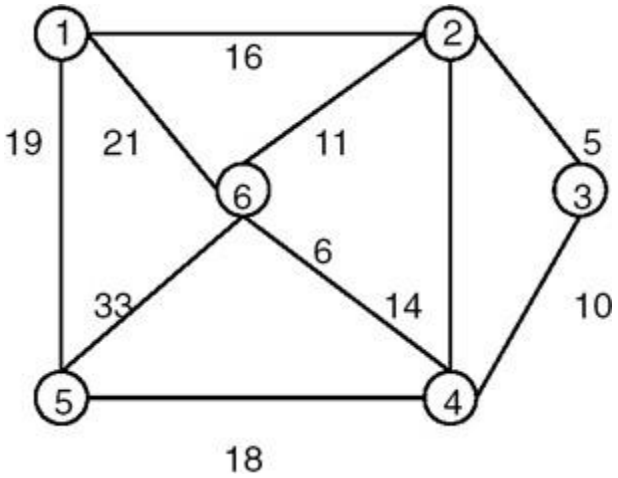


Figure 22.21: A graph G.

The *minimum-cost spanning tree* of the graph of Figure 22.21 is shown in Figure 22.22.

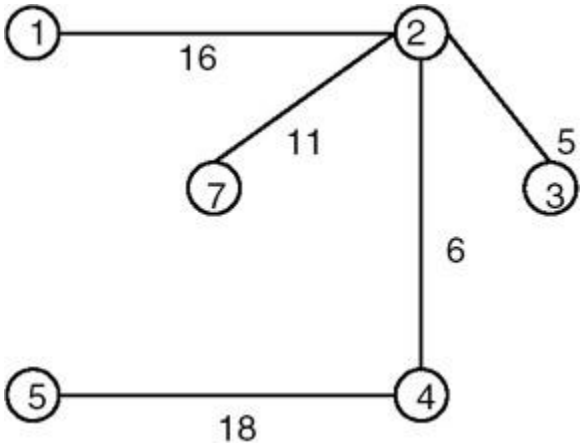


Figure 22.22: The minimum-cost spanning tree of graph G of Figure 22.21.

## MST Property

Let G = (V,E) be a connected graph with a cost function defined on the edges. Let U be some proper subset of the set of vertices V. If (u,v) is an edge of lowest cost such that u is in U, and v is in V-U, there is a minimum-cost spanning tree that includes edge (u,v). Many of the methods of constructing a minimum-cost spanning tree use the following properties.

## Prim's Algorithm

Let G = (V,E) be a weighted graph, and suppose V = {1,2,…,n}. The Prim's algorithm begins with a set U initialized to {1}, and at each stage finds the shortest edge (u,v) that connects u in U and v in V-U, and then adds v to U. It repeats this step until U = V.

```
mcost(G is a graph; T is a set of edges)
   U is a set of vertices
```

```
   u,v are vertices;
{
   T= ô
   U = {1}
   while U '" V
   {
```

Find the lowest-cost edge (u,v) such that u is in U and v is in V-U

```
      add (u,v) to T
      add v to U
   }
}
```

## Program

The following program can be used to find the minimum spanning tree of a graph.

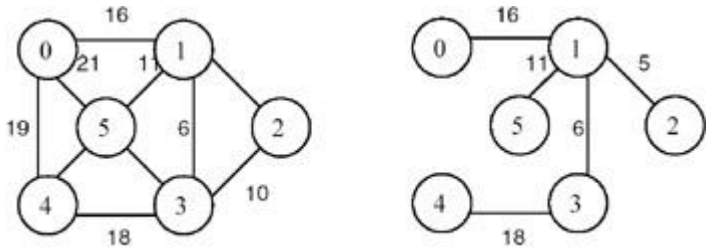
```
#include <stdio.h>

#define MAXVERTICES 10
#define MAXEDGES 20
typedef enum {FALSE, TRUE} bool;

int getNVert(int edges[][3], int nedges) {
   /*
    * returns no of vertices = maxvertex + 1;
    */
   int nvert = -1;
   int j;

   for( j=0; j<nedges; ++j ) {
      if( edges[j][0] > nvert )
             nvert = edges[j][0];

      if( edges[j][1] > nvert )
             nvert = edges[j][1];
   }
   return ++nvert;         // no of vertices = maxvertex + 1;
}

bool isPresent(int edges[][3], int nedges, int v) {
   /*
    * checks whether v has been included in the spanning tree.
    * thus we see whether there is an edge incident on v which has
    * a negative cost. negative cost signifies that the edge has been
    * included in the spanning tree.
    */

   int j;
   for(j=0; j<nedges; ++j)
      if(edges[j][2] < 0 && (edges[j][0] == v || edges[j][1] == v))
               return TRUE;

   return FALSE;
```

```
}

void spanning(int edges[][3], int nedges) {
   /*
    * finds a spanning tree of the graph having edges.
    * uses kruskal's method.
    * assumes all costs to be positive.
    */
   int i, j;
   int tv1, tv2, tcost;
   int nspanedges = 0;
   int nvert = getNVert(edges, nedges);

   // sort edges on cost.
   for(i=0; i<nedges-1; ++i)
      for(j=i; j<nedges; ++j)
             if(edges[i][2] > edges[j][2]) {
                    tv1 = edges[i][0]; tv2 = edges[i][1]; tcost =
edges[i][2];
                    edges[i][0] = edges[j][0]; edges[i][1] =
edges[j][1]; edges[i][2] = edges[j][2];
                    edges[j][0] = tv1; edges[j][1] = tv2; edges[j][2] =
tcost;
                  }
    for(j=0; j<nedges-1; ++j) {
        // consider edge j connecting vertices v1 and v2.
        int v1 = edges[j][0];
        int v2 = edges[j][1];

         // check whether it forms a cycle in the up until now formed
spanning tree.
         // checking can be done easily by checking whether both v1 and
v2 are in
        // the current spanning tree!
        if(isPresent(edges, nedges, v1) && isPresent(edges, nedges,
v2)) // cycle.
                 printf("rejecting: %d %d %d...\n", edges[j][0],
edges[j][1], edges[j][2]);
   else {
           edges[j][2] = -edges[j][2];
           printf("%d %d %d.\n", edges[j][0], edges[j][1], -
edges[j][2]);
           if(++nspanedges == nvert-1)
                   return;
         }
   }
   printf("No spanning tree exists for the graph.\n");
}

main() {
    int edges[][3] = {
                                    {0,1,16},
                                    {0,4,19},
                                    {0,5,21},
                                    {1,2,5},
                                    {1,3,6},
                                    {1,5,11},
```

```
                                    {2,3,10},
                                    {3,4,18},
                                    {3,5,14},
                                    {4,5,33}
                                };
 int nedges = sizeof(edges)/3/sizeof(int);
 spanning(edges, nedges);

 return 0;
}
```

## Explanation

1. A tree consisting solely of edges in a graph G, and including all vertices in G, is called a spanning tree. A minimum spanning tree of a weighted graph is the spanning tree with minimum total cost of its edges.




**An example graph and its minimum spanning tree.**

2. The graph is represented as an array of edges. Each entry in the array is a triplet representing an edge consisting of source vertex, destination vertex, and the cost associated with the edge. The method used in finding a minimum spanning tree is that given by Kruskal. In this approach, a minimum spanning tree T is built edge by edge. Edges are considered for inclusion in T in non- decreasing order of their costs. An edge is included if it does not form a cycle with the edges that are already in T. Since graph G is connected and has $n > 0$ vertices, exactly $n - 1$ edges will be selected for inclusion in T.
3. Kruskal's algorithm is as follows:

```
4.   T={};       // empty set.
5.   while T contains less than n-1 edges and E not empty do
6.      choose an edge (v, w) from E of lowest cost.
7.      delete (v, w) from E.
8.      if (v, w) does NOT create a cycle in T
9.             add (v, w) to T.
10.     else
11.            discard (v, w).
12.  endwhile.
13.  if T contains less than n-1 edges
14.     print("no spanning tree exists for this graph.");
```

15. In order for the choice of the lowest-cost edge from E to become efficient, we sort the edge array over the cost of edge. To check whether an edge (v, w) forms a cycle, we simply need to check whether both v and w appear in any of the

previously added edges in T. We assume that all the costs are positive and we make them negative to signify that the edge has been included in T.

16. Example:

    For the example graph in item 1, the run of the algorithm goes as follows:

| STEP | EDGE | COST | ACTION | SPANNING-TREE |
|---|---|---|---|---|
| 0 | — | — | — | {}. |
| 1 | (1, 2) | 5 | accept | {(1, 2)}. |
| 2 | (1, 3) | 6 | accept | {(1, 2), (1, 3)}. |
| 3 | (2, 3) | 10 | reject | {(1, 2), (1, 3)}. |
| 4 | (1, 5) | 11 | accept | {(1, 2), (1, 3), (1, 5)}. |
| 5 | (3, 5) | 14 | reject | {(1, 2), (1, 3), (1, 5)}. |
| 6 | (0, 1) | 16 | accept | {(1, 2), (1, 3), (1, 5), (0, 1)}. |
| 7 | (3, 4) | 18 | accept | {(1, 2), (1, 3), (1, 5), (0, 1), (3, 4)}. |

## Points to Remember

1. A minimum spanning tree of a weighted graph G is a tree that consists of edges solely from the edges of G, which covers all the vertices in G, and which has the minimum combined cost of its edges.
2. The complexity of Kruskal's method used for finding the minimum spanning tree of a graph G is O(eloge) where e is the number of edges in G.
3. Note that the union and find algorithms for set representation can be used for checking for cycle and inclusion of an edge in a set.
4. There can be multiple minimum spanning trees in a graph.

## Application of Minimum-Cost Spanning Tree

A property of a spanning tree of a graph G is that a spanning tree is a minimal connected subgraph of G (by minimal, we mean the one with the fewest number of edges). Therefore, if nodes of G represent cities and the edges represent possible communication links connecting two cities, then the spanning trees of graph G represent all feasible choices of the communication network. If each edge has weight representing cost measured in some suitable terms (such as cost of construction or distance etc.), then the minimum-cost spanning tree of G is the selection of the required communication network.