Depth First Search

Depth first search of an undirected graph proceeds as follows. The start vertex **v** is visited. Next an unvisited vertex **w** adjacent to **v** is selected and a depth first search from **w** is initiated. When a vertex **u** is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited which has an unvisited vertex **w** adjacent to it and initiate a depth first search from **w**. The search terminates when no unvisited vertex can be reached from any of the visited ones. This procedure is best-described recursively and has been implemented in the program given below.

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
#include <windows.h>

#define TRUE 1
#define FALSE 0
#define MAX 8

struct node
{
    int data ;
    struct node *next ;
} ;

int visited[MAX] ;

void dfs ( int, struct node ** ) ;
struct node * getnode_write ( int ) ;
void del ( struct node * ) ;
```

```
int main( )
{
    struct node *arr[MAX] ;
    struct node *v1, *v2, *v3, *v4 ;
    int i ;

    system ( "cls" ) ;

    v1 = getnode_write ( 2 ) ;
    arr[0] = v1 ;
    v1 -> next = v2 = getnode_write ( 3 ) ;
    v2 -> next = NULL ;

    v1 = getnode_write ( 1 ) ;
    arr[1] = v1 ;
    v1 -> next = v2 = getnode_write ( 4 ) ;
    v2 -> next = v3 = getnode_write ( 5 ) ;
    v3 -> next = NULL ;

    v1 = getnode_write ( 1 ) ;
    arr[2] = v1 ;
    v1 -> next = v2 = getnode_write ( 6 ) ;
    v2 -> next = v3 = getnode_write ( 7 ) ;
    v3 -> next = NULL ;

    v1 = getnode_write ( 2 ) ;
    arr[3] = v1 ;
    v1 -> next = v2 = getnode_write ( 8 ) ;
    v2 -> next = NULL ;

    v1 = getnode_write ( 2 ) ;
    arr[4] = v1 ;
    v1 -> next = v2 = getnode_write ( 8 ) ;
    v2 -> next = NULL ;

    v1 = getnode_write ( 3 ) ;
    arr[5] = v1 ;
```

```
    v1 -> next = v2 = getnode_write ( 8 ) ;
    v2 -> next = NULL ;

    v1 = getnode_write ( 3 ) ;
    arr[6] = v1 ;
    v1 -> next = v2 = getnode_write ( 8 ) ;
    v2 -> next = NULL ;

    v1 = getnode_write ( 4 ) ;
    arr[7] = v1 ;
    v1 -> next = v2 = getnode_write ( 5 ) ;
    v2 -> next = v3 = getnode_write ( 6 ) ;
    v3 -> next = v4 = getnode_write ( 7 ) ;
    v4 -> next = NULL ;

    dfs ( 1, arr ) ;

    for ( i = 0 ; i < MAX ; i++ )
        del ( arr[i] ) ;

    return 0 ;
}

void dfs ( int v, struct node **p )
{
    struct node *q ;
    visited[v - 1] = TRUE ;

    printf ( "%d\t", v ) ;

    q = * ( p + v - 1 ) ;

    while ( q != NULL )
    {
        if ( visited[q -> data - 1] == FALSE )
            dfs ( q -> data, p ) ;
        else
            q = q -> next ;
```

Scanned by CamScanner

```
    }
}

struct node * getnode_write ( int val )
{
    struct node *newnode ;
    newnode = ( struct node * ) malloc ( sizeof ( struct node ) ) ;
    newnode -> data = val ;
    return newnode ;
}

void del ( struct node *n )
{
    struct node *temp ;

    while ( n != NULL )
    {
        temp = n -> next ;
        free ( n ) ;
        n = temp ;
    }
}
```

Breadth First Search

Starting at vertex **v** and marking it as visited, breadth first search differs from depth first search in that all unvisited vertices adjacent to **v**, are visited next. Then unvisited vertices adjacent to these vertices are visited and so on. A breadth first search beginning at vertex $\mathbf{v_1}$ of Figure 11-3 would first visit $\mathbf{v_1}$ and then $\mathbf{v_2}$ and $\mathbf{v_3}$. Next vertices $\mathbf{v_4}$, $\mathbf{v_5}$, $\mathbf{v_6}$ and $\mathbf{v_7}$ will be visited and finally $\mathbf{v_8}$. The following program implements this algorithm.

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
#include <windows.h>

#define TRUE 1
#define FALSE 0
#define MAX 8

struct node
{
    int data ;
    struct node *next ;
} ;

int visited[MAX] ;
int q[8] ;
int front, rear ;

void bfs ( int, struct node ** ) ;
struct node * getnode_write ( int ) ;
void addqueue ( int ) ;
int deletequeue( ) ;
int isempty( ) ;
void del ( struct node * ) ;

int main( )
```

```
{
    struct node *arr[MAX] ;
    struct node *v1, *v2, *v3, *v4 ;
    int i ;

    system ( "cls" ) ;

    v1 = getnode_write ( 2 ) ;
    arr[0] = v1 ;
    v1 -> next = v2 = getnode_write ( 3 ) ;
    v2 -> next = NULL ;

    v1 = getnode_write ( 1 ) ;
    arr[1] = v1 ;
    v1 -> next = v2 = getnode_write ( 4 ) ;
    v2 -> next = v3 = getnode_write ( 5 ) ;
    v3 -> next = NULL ;

    v1 = getnode_write ( 1 ) ;
    arr[2] = v1 ;
    v1 -> next = v2 = getnode_write ( 6 ) ;
    v2 -> next = v3 = getnode_write ( 7 ) ;
    v3 -> next = NULL ;

    v1 = getnode_write ( 2 ) ;
    arr[3] = v1 ;
    v1 -> next = v2 = getnode_write ( 8 ) ;
    v2 -> next = NULL ;

    v1 = getnode_write ( 2 ) ;
    arr[4] = v1 ;
    v1 -> next = v2 = getnode_write ( 8 ) ;
    v2 -> next = NULL ;

    v1 = getnode_write ( 3 ) ;
    arr[5] = v1 ;
    v1 -> next = v2 = getnode_write ( 8 ) ;
    v2 -> next = NULL ;
```

```
v1 = getnode_write ( 3 ) ;
arr[6] = v1 ;
v1 -> next = v2 = getnode_write ( 8 ) ;
v2 -> next = NULL ;

v1 = getnode_write ( 4 ) ;
arr[7] = v1 ;
v1 -> next = v2 = getnode_write ( 5 ) ;
v2 -> next = v3 = getnode_write ( 6 ) ;
v3 -> next = v4 = getnode_write ( 7 ) ;
v4 -> next = NULL ;

front = rear = -1 ;
bfs ( 1, arr ) ;

for ( i = 0 ; i < MAX ; i++ )
    del ( arr[i] ) ;

return 0 ;
}

void bfs ( int v, struct node **p )
{
    struct node *u ;

    visited[v - 1] = TRUE ;
    printf ( "%d\t", v ) ;
    addqueue ( v ) ;

    while ( isempty( ) == FALSE )
    {
        v = deletequeue( ) ;
        u = * ( p + v - 1 ) ;

        while ( u != NULL )
        {
            if ( visited [u -> data - 1] == FALSE )
```

Scanned by CamScanner

```
                {
                        addqueue ( u -> data ) ;
                        visited [u -> data - 1] = TRUE ;
                        printf ( "%d\t", u -> data ) ;
                }
                u = u -> next ;
            }
        }
    }

    struct node * getnode_write ( int val )
    {
        struct node *newnode ;
        newnode = ( struct node * ) malloc ( sizeof ( struct node ) ) ;
        newnode -> data = val ;
        return newnode ;
    }

    void addqueue ( int vertex )
    {
        if ( rear == MAX - 1 )
        {
            printf ( "Queue Overflow.\n" ) ;
            exit ( 0 ) ;
        }

        rear++ ;
        q[rear] = vertex ;

        if ( front == -1 )
            front = 0 ;
}

int deletequeue( )
{
    int data ;

    if ( front == -1 )
```

```
    {
        printf ( "Queue Underflow.\n" ) ;
        exit ( 0 ) ;
    }

    data = q[front] ;

    if ( front == rear )
        front = rear = -1 ;
    else
        front++ ;

    return data ;
}

int isempty( )
{
    if ( front == -1 )
        return TRUE ;
    return FALSE ;
}

void del ( struct node *n )
{
    struct node *temp ;

    while ( n != NULL )
    {
        temp = n -> next ;
        free ( n ) ;
        n = temp ;
    }
}
```

The working of functions **getnode_write( )** & **del( )** and arrays **arr[ ]** & **visited[ ]** is exactly same as in the previous program.

The function **bfs( )** visits each vertex and marks it visited. The functions **isempty( )**, **addqueue( )** and **deletequeue( )** are called while maintaining the queue of vertices.

Spanning tree

A spanning tree of a graph is an undirected tree consisting of only those edges that are necessary to connect all the vertices in the original graph. A spanning tree has a property that for any pair of vertices there exists only one path between them, and the insertion of any edge to a spanning tree form a unique cycle.

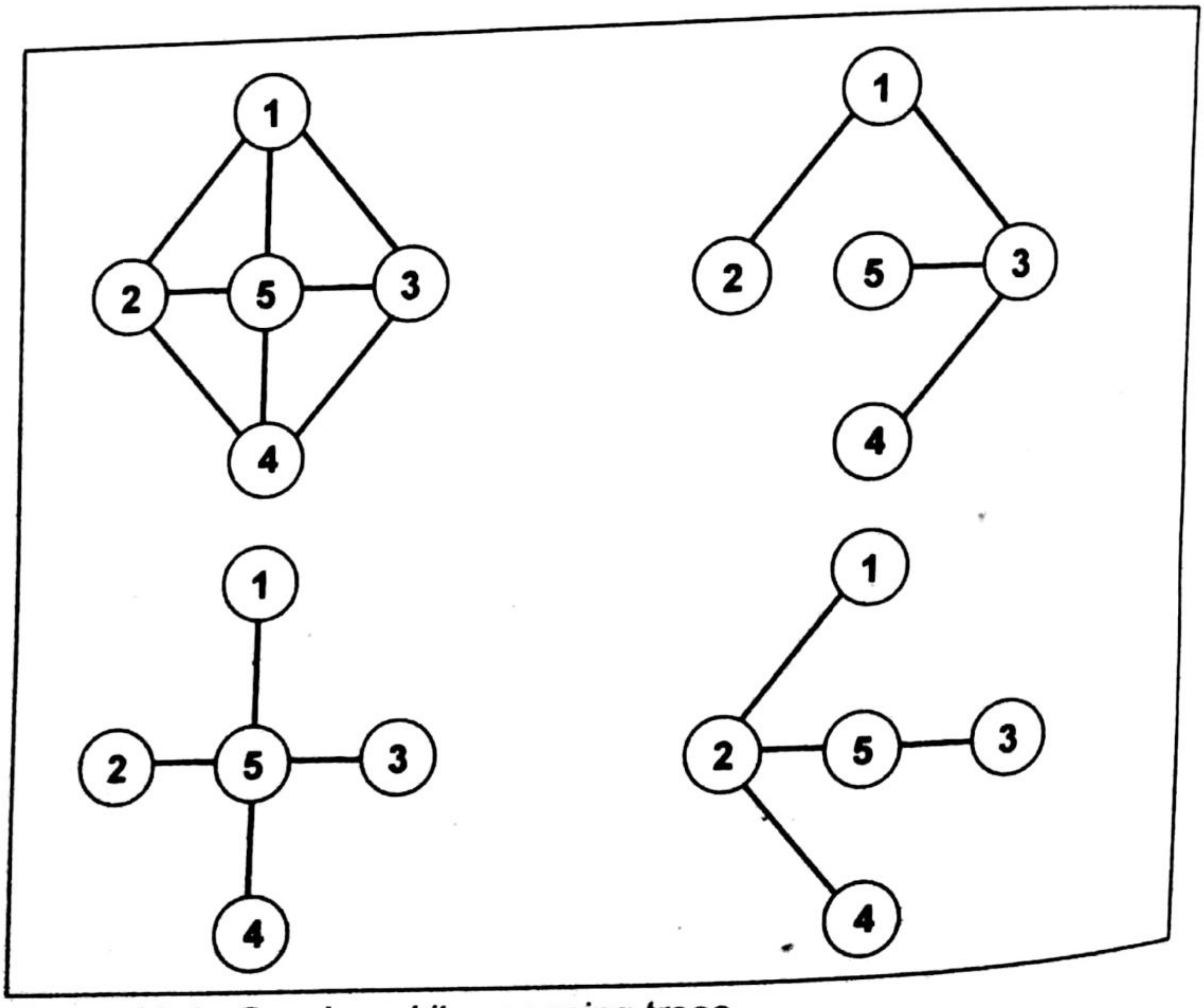


Figure 11-4. *Graph and its spanning trees.*

Kruskal's Algorithm

In this algorithm a minimum cost spanning tree T is built edge by edge. Edges are considered for inclusion in T in increasing order of their costs. An edge is included in T if it does not form a cycle with edges already in T. Let us understand this with the help of an example. Consider Figure 11-6.

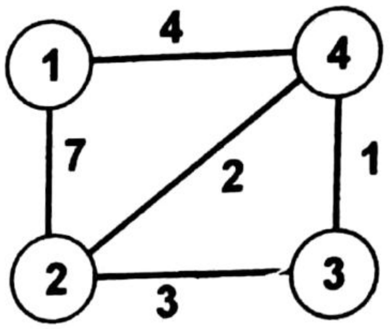


Figure 11-6. *Graph with the respective costs of edges.*

To find the minimum cost of spanning tree the edges are inserted to tree in increasing order of their costs. Figure 11-7 shows insertion or rejection of each edge. The meanings of symbols I, R, C, etc. used in the figure are as follows:

E - Edge
C - Cost
A - Action

T - Tree
I - Inclusion
R - Rejection

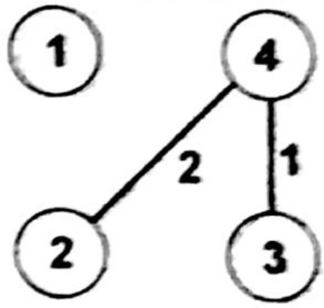
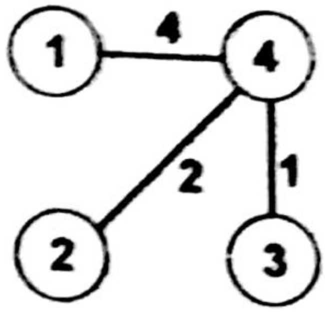
E	C	A	T
			
4-3	1	I	
4-2	2	I	
3-2		R	
4-1	4	I	

Figure 11-7. *Creating minimum cost spanning tree.*

To begin with edge 4-3 is inserted as it has the lowest cost 1. The the edge 4-2 is inserted which has a cost 2. The next edge in th order of cost is 3-2, but it is rejected as it forms a cyclic pat between the existing vertices. Then the edge 4-1 is inserted and it i accepted as it forms a non-cyclic path.

The minimum cost of spanning tree is given by the sum of costs c the existing edges, i.e. the edges that are inserted while building th spanning tree of minimum cost. In our case it is found to be 7 as

The following program implements the Kruskal's algorithm.

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
#include <windows.h>

struct lledge
{
    int v1, v2 ;
    float cost ;
    struct lledge *next ;
} ;

int stree[5] ;
int count[5] ;
```

```
float mincost ;

struct lledge * kminstree ( struct lledge *, int ) ;
int getrval ( int ) ;
void combine ( int, int ) ;
void del ( struct lledge * ) ;

int main( )
{
    struct lledge *temp, *root ;
    int i ;

    system ( "cls" ) ;

    root = ( struct lledge * ) malloc ( sizeof ( struct lledge ) ) ;

    root -> v1 = 4 ;
    root -> v2 = 3 ;
    root -> cost = 1 ;
    temp = root -> next = ( struct lledge * ) malloc ( sizeof ( struct lledge ) ) ;

    temp -> v1 = 4 ;
    temp -> v2 = 2 ;
    temp -> cost = 2 ;
    temp -> next = ( struct lledge * ) malloc ( sizeof ( struct lledge ) ) ;

    temp = temp -> next ;
    temp -> v1 = 3 ;
    temp -> v2 = 2 ;
    temp -> cost = 3 ;
    temp -> next = ( struct lledge * ) malloc ( sizeof ( struct lledge ) ) ;

    temp = temp -> next ;
    temp -> v1 = 4 ;
    temp -> v2 = 1 ;
    temp -> cost = 4 ;
    temp -> next = NULL ;
```

```
    root = kminstree ( root, 5 ) ;

    for ( i = 1 ; i <= 4 ; i++ )
        printf ( "stree[%d] -> %d\n", i, stree[i] ) ;
    printf ( "The minimum cost of spanning tree is %d\n", mincost ) ;
    del ( root ) ;

    return 0 ;
}

struct lledge * kminstree ( struct lledge *root, int n )
{
    struct lledge *temp = NULL ;
    struct lledge *p, *q ;
    int noofedges = 0 ;
    int i, p1, p2 ;

    for ( i = 0 ; i < n ; i++ )
        stree[i] = i ;
    for ( i = 0 ; i < n ; i++ )
        count[i] = 0 ;

    while ( ( noofedges < ( n - 1 ) ) && ( root != NULL ) )
    {
        p = root ;

        root = root -> next ;

        p1 = getrval ( p -> v1 ) ;
        p2 = getrval ( p -> v2 ) ;

        if ( p1 != p2 )
        {
            combine ( p -> v1, p -> v2 ) ;
            noofedges++ ;
            mincost += p -> cost ;
            if ( temp == NULL )
            {
```

```
                temp = p ;
                q = temp ;
            }
            else
            {
                q -> next = p ;
                q = q -> next ;
            }
            q -> next = NULL ;
        }
    }
    return temp ;
}

int getrval ( int i )
{
    int j, k, temp ;
    k = i ;
    while ( stree[k] != k )
        k = stree[k] ;
    j = i ;
    while ( j != k )
    {
        temp = stree[j] ;
        stree[j] = k ;
        j = temp ;
    }
    return k ;
}

void combine ( int i, int j )
{
    if ( count[i] < count[j] )
        stree[i] = j ;
    else
    {
        stree[j] = i ;
        if ( count[i] == count[j] )
```

```
                count[j]++ ;
        }
}

void del ( struct lledge *root )
{
        struct lledge *temp ;

        while ( root != NULL )
        {
                temp = root -> next ;
                free ( root ) ;
                root = temp ;
        }
}
```

Output:

stree[1] -> 4
stree[2] -> 4
stree[3] -> 4
stree[4] -> 4
The minimum cost of spanning tree is 7

compared to earlier ones.

Finally, the vertex 4 is considered, which results into the adjacency matrix that holds the shortest path between any two vertices.

$$
\begin{array}{c c}
 & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} &
\begin{bmatrix}
7 & 5 & \boxed{8} & 7 \\
\boxed{6} & \boxed{6} & \boxed{3} & 2 \\
\boxed{9} & 3 & \boxed{6} & 5 \\
4 & 4 & 1 & 6
\end{bmatrix}
\end{array}
\begin{bmatrix}
11 & 12 & \boxed{1243} & 124 \\
\boxed{241} & \boxed{2432} & \boxed{243} & 24 \\
\boxed{3241} & 32 & \boxed{3243} & 324 \\
41 & 432 & 43 & 4324
\end{bmatrix}
$$

The following program shows how to find the shortest the path between any two vertices.

```c
#include <stdio.h>
#include <conio.h>
#include <windows.h>

#define INF 9999

int main( )
{
	int arr[4][4] ;
	int cost[4][4] = {
				7, 5, 0, 0,
				7, 0, 0, 2,
				0, 3, 0, 0,
				4, 0, 1, 0
			} ;
	int i, j, k, n = 4 ;

	system ( "cls" ) ;

	for ( i = 0 ; i < n ; i++ )
	{
		for ( j = 0; j < n ; j++ )
		{
			if ( cost[i][j] == 0 )
				arr[i][j] = INF ;
			else
				arr[i][j] = cost[i][j] ;
		}
	}

	printf ( "Adjacency matrix of cost of edges:\n" ) ;
	for ( i = 0 ; i < n ; i++ )
	{
		for ( j = 0; j < n ; j++ )
			printf ( "%d\t", arr[i][j] ) ;
		printf ( "\n" ) ;
	}
```

```
for ( k = 0 ; k < n ; k++ )
{
    for ( i = 0 ; i < n ; i++ )
    {
        for ( j = 0 ; j < n ; j++ )
        {
            if ( arr[i][j] > arr[i][k] + arr[k][j] )
                arr[i][j] = arr[i][k] + arr[k][j];
        }
    }
}

 printf ( "\n" ) ;
 printf ( "Adjacency matrix of lowest cost between the vertices:\n" ) ;
 for ( i = 0 ; i < n ; i++ )
 {
     for ( j = 0; j < n ; j++ )
         printf ( "%d\t", arr[i][j] ) ;
     printf ( "\n" ) ;
 }

 return 0 ;
}
```

Output:

```
Adjacency matrix of cost of edges:
7     5     9999  9999
7     9999  9999  2
9999  3     9999  9999
4     9999  1     9999

Adjacency matrix of lowest cost between the vertices:
7     5     8     7
6     6     3     2
9     3     6     5
4     4     1     6
```