Program No:8

SET OPERATIONS USING BIT STRING

Aim :- To perform different bit string operations on 2 different sets

Algorithm :-

1. Initialize arrays u, a, and b with set elements.
2. Create bitstrings us, sa, and sb to represent sets u, a, and b.
3. Iterate through the elements of set u and set corresponding bits in us to 1
4. Print the bitstring representation of set u.
5. For sets a and b, create bitstring representations sa and sb by setting bits to
6. 1 if the element is present in the respective sets.
7. Print bitstring representations of sets a and b.
8. Initialize arrays cs and ds to represent the union and intersection bitsets.
9. Perform union operation: set the bit in cs to 1 if it's set in either sa or sb.
10. Print the bitstring representation of the union bitset and the corresponding number set c.
11. Perform intersection operation: set the bit in ds to 1 if it's set in both sa and sb.
12. Print the bitstring representation of the intersection bitset and the corresponding number set d.
13. Return 0 to end the program

Program :-

```
#include<stdio.h>
#include<stdlib.h>
#define N 10
int main()
{
	int i,j,k=0,t;
	int u[N]={1,2,3,4,5,6,7,8,9,10};
	int a[N]={1,3,4,5,7};
	int b[N]={1,2,4,6,8,10};
	int sa[N], sb[N], us[N], cs[N], ds[N], c[N], d[2];
	for(i=0;i<N;i++)
	{
		if(u[i]!=-1)
			us[i]=1;
	}
	printf("bitstring\n");
	for(i=0;i<N;i++)
	{
		printf("%d\t",us[i]);
	}
	for(i=0;i<N;i++)
	{
		t=0;
		for(j=0;j<N;j++)
		{
			if(a[j]==u[i])
			{
				t=1;
				break;
			}
		}
		if(t==1)
			sa[i]=1;
		else
			sa[i]=0;
	}
	printf("\nbitstring ofA\n");
	for(i=0;i<N;i++)
	{
		printf("%d\t",sa[i]);
	}
	for(i=0;i<N;i++)
	{
		t=0;
		for(j=0;j<N;j++)
		{
			if(b[j]==u[i])
			{
				t=1;
				break;
			}
		}
		if(t==1)
```

```
            sb[i]=1;
        else
            sb[i]=0;
}
printf("\nbitstring ofB\n");
for(i=0;i<N;i++)
{
        printf("%d\t",sb[i]);
}
printf("\n-------------------------------------------------------------------------------
-------");
for(i=0;i<10;i++)
{
        if(sa[i]==1||sb[i]==1)
            cs[i]=1;
        else
            cs[i]=0;
}
printf("\nunion bit set\n");
printf("\n");
for(i=0;i<10;i++)
{
        printf("%d\t",cs[i]);
}
printf(" nunion number set\n");
for(i=0;i<10;i++)
{
        if(cs[i]==1)
```



```
        {
            c[k]=u[i];
            printf("%d\t",c[k]);
            k++;
        }
}
printf("\n-------------------------------------------------------------------------------
-------");
for(i=0;i<10;i++)
{
        if(sa[i]==1 && sb[i]==1)
            ds[i]=1;
        else
            ds[i]=0;
}
printf("\nintersection bit set\n");
printf("\n");
for(i=0;i<10;i++)
{
        printf("%d\t",ds[i]);
}
printf("\nintersection number set\n");
for(i=0;i<10;i++)
{
        if(ds[i]==1)
 {
        d[k]=u[i];
        printf("%d\t", d[k]);
```

```
            k++;
            }
        }
        printf("\n");
        /*for(i=0;i<k;i++)
        {
            printf("%d\t",d[i]);
        }*/
        return 0;
}
```

Output :-

bitstring

1 1 1 1 1 1 1 1 1 1

bitstring of A

1 0 1 1 1 0 1 0 0 0

bitstring of B

1 1 0 1 0 1 0 1 0 1

---

union bit set

1 1 1 1 1 1 1 1 0 1

union number set

1 2 3 4 5 6 7 8 10

---

intersection bit set

1 0 0 1 0 0 0 0 0 0

intersection number set

1 4

Result :-Program executed successfully and output verified.

Program No:9

DISJOINT SET OPERATIONS

Aim :- To find a graph contain cycle or not

Algorithm :-

1. Define Node Structure:
   - Define a structure node with members rep (representative), next (pointer to the next node), and data (integer value).

2. Declare Global Variables:
   - Declare arrays heads and tails to keep track of set representatives.
   - Initialize a static variable countRoot to 0 to keep track of the number of sets.

3. makeSet(int x):
   - Allocate memory for a new node.
   - Set its representative (rep) to itself, next to NULL, and data to the given value x.
   - Add the new node to the array of heads (heads) and tails (tails), and increment countRoot.

4. find(int a):
   - Iterate through the array of heads and search for the node with data equal to a.
   - Return the representative (rep) of the set containing element a.

5. unionSets(int a, int b):
   - Find representatives (rep1 and rep2) of sets containing elements a and b.
   - If one of them is not found, print an error message.
   - If the representatives are different, merge the sets:
     - Update the tail of the first set to point to the representative of the second set.
     - Update the tails array and reduce the count of sets.
     - Update the representative of all nodes in the second set to be the representative of the first set.

6. search(int x):
   - Iterate through the array of heads and search for the element x in each set.
   - Return 1 if found, 0 otherwise.

7. main():
   - Display a menu with options to:
   - Make a set
   - Display set representatives
   - Union sets
   - Find set
   - Exit
   - Take user input for the chosen option and perform the corresponding operation in a loop until the user chooses to exit.

Program :-

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct node{
struct node *rep;
struct node *next;
  int data;
}*heads[50],*tails[50];
static int countRoot=0;
 void makeSet(int x)
{
        struct node *new=(struct node *)malloc(sizeof(struct node));
        new->rep=new;
        new->next=NULL;
        new->data=x;
        heads[countRoot]=new;
        tails[countRoot++]=new;
}
struct node* find(int a){
        int i;
        struct node *tmp=(struct node *)malloc(sizeof(struct node));
        for(i=0;i<countRoot;i++){
                tmp=heads[i];
                while(tmp!=NULL){
                        if(tmp->data==a)
                                return tmp->rep;
                        tmp=tmp->next;
                }
        }
        return NULL;
}
void unionSets(int a,int b){
        int i,pos,flag=0,j;
        struct node *tail2=(struct node *)malloc(sizeof(struct node));
        struct node *rep1=find(a);
        struct node *rep2=find(b);
        if(rep1==NULL||rep2==NULL){
```

```
        printf("\nElement not present in the DS\n");
        return;
}
if(rep1!=rep2){
        for(j=0;j<countRoot;j++){
                if(heads[j]==rep2){
                        pos=j;
                        flag=1;
                        countRoot-=1;
                        tail2=tails[j];
                for(i=pos;i<countRoot;i++){
                        heads[i]=heads[i+1];
                        tails[i]=tails[i+1];
                }
                }
                if(flag==1)
                break;
        }
        for(j=0;j<countRoot;j++){
                if(heads[j]==rep1){
                        tails[j]->next=rep2;
                        tails[j]=tail2;
                        break;
                }
                while(rep2!=NULL){
                        rep2->rep=rep1;
                        rep2=rep2->next;
                }
        }
}
int search(int x){
        int i;
        struct node *tmp=(struct node *)malloc(sizeof(struct node));
        for(i=0;i<countRoot;i++){
                tmp=heads[i];
                if(heads[i]->data==x)
                        return 1;
                while(tmp!=NULL){
                        if(tmp->data==x)
                                return 1;
                        tmp=tmp->next;
                }
        }
        return 0;
}
void main(){
        int choice,x,i,j,y,flag=0;
```

```
do{
        printf("\n||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||\n");
        printf("\n.....MENU.....\n\n1.Make Set\n2.Display set
        representatives\n3.Union\n4.Find Set\n5.Exit\n");

        printf("Enter your choice :  ");
        scanf("%d",&choice);
        printf("\n||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||\n");
        switch(choice){
                case 1:
                        printf("\nEnter new element : ");
                        scanf("%d",&x);
                        if(search(x)==1)
                                printf("\nElement already present in the disjoint set
                                DS\n");
                        else{
                        makeSet(x);}
                        break;
                case 2:
                        printf("\n");
                        for(i=0;i<countRoot;i++)
                        printf("%d ",heads[i]->data);
                        printf("\n");

                        break;
                case 3:
                        printf("\nEnter first element : ");
                        scanf("%d",&x);
                        printf("\nEnter second element : ");
                        scanf("%d",&y);
                        unionSets(x,y);
                        break;
                case 4:
                        printf("\nEnter the element");
                        scanf("%d",&x);
                        struct node *rep=(struct node *)malloc(sizeof(struct node));
                        rep=find(x);
                        if(rep==NULL)
                                printf("\nElement not present in the DS\n");
                        else
                                printf("\nThe representative of %d is
%d\n",x,rep->data);
                        break;
                        case 5:
                        exit(0);
                default:
                printf("\nWrong choice\n");
                break;
        }
```

```
    }
    while(1);
};
```

Output :-

```
||||||||||||||||||||||||||||||||||||||||||||||||
.....MENU.....
1.Make Set
2.Display set representatives
3.Union
4.Find Set 5.Exit
Enter your choice :  1
||||||||||||||||||||||||||||||||||||||||||||||||
Enter new element : 15
||||||||||||||||||||||||||||||||||||||||||||||||
.....MENU.....
1.Make Set
2.Display set representatives
3.Union
4.Find Set 5.Exit
Enter your choice :  1
||||||||||||||||||||||||||||||||||||||||||||||||
Enter new element : 32
||||||||||||||||||||||||||||||||||||||||||||||||
.....MENU.....
```



```
1.Make Set
2.Display set representatives
3.Union
4.Find Set
5.Exit
Enter your choice :  2
||||||||||||||||||||||||||||||||||||||||
15 32
||||||||||||||||||||||||||||||||||||||||
.....MENU.....
1.Make Set
2.Display set representatives
3.Union
4.Find Set
5.Exit
Enter your choice :  3
||||||||||||||||||||||||||||||||||||||||
Enter first element : 15
Enter second element : 32
||||||||||||||||||||||||||||||||||||||||
```

Result :- Program executed successfully and output verified.

Program No:10

BFS GRAPH TRAVERSAL

Aim :- To create a program to implement BFS traversal

Algorithm :-

1. Create a queue data structure to store the nodes that will be visited.
2. Create a boolean array to mark the visited nodes.
3. Start with a source node and mark it as visited.
4. Enqueue the source node into the queue.
5. While the queue is not empty, do the following:
    a. Dequeue the front node from the queue.
    b. For each of the adjacent nodes of the dequeued node, do the following:
        i. If the adjacent node has not been visited, mark it as visited and enqueue it into the queue.
6. Repeat step 5 until the queue is empty.

Program :-

```
#include<stdio.h>
#include<stdlib.h>
```

```
#define MAX 100
#define initial 1
#define waiting 2
#define visited 3

int n;
int adj[MAX][MAX];
int state[MAX];
void create_graph();
void BF_Traversal();
 void BFS(int v);
 int queue[MAX], front = -1,rear = -1;
 void insert_queue(int vertex);
int delete_queue();
int isEmpty_queue();

int main()
{
        create_graph();
        BF_Traversal();
        return 0;
}

void BF_Traversal()
{
        int v;

        for(v=0; v<n; v++)
                state[v] = initial;

        printf("Enter Start Vertex for BFS: \n");

         scanf("%d", &v);

        BFS(v);
}

void BFS(int v)
{
         int i;
         printf("BFS traversal");
         insert_queue(v);
         state[v] = waiting;
         while(!isEmpty_queue())
         {
                 v = delete_queue();
                 printf("%d ",v);
                  state[v] = visited;
                   for(i=0;i<n;i++)
                    {
```

```
                if(adj[v][i] == 1 && state[i] == initial)
                {
                        insert_queue(i);
                        state[i] = waiting;
                }
        }
    }
    printf("\n");
}
void insert_queue(int vertex)
{
    if(rear == MAX-1)
        printf("Queue Overflow\n");
    else
    {
        if(front == -1)
        front = 0;
        rear = rear+1;
        queue[rear] = vertex;
    }
}


int isEmpty_queue() {
    if(front == -1 || front > rear)
        return 1;
    else
        return 0;
}


int delete_queue()
{
    int delete_item;
    if(front == -1 || front > rear)
    {
        printf("Queue Underflow\n");
        exit(1);
    }
    delete_item =
    queue[front];
    front = front+1;
    return delete_item;
}
void create_graph()
{
    int count,max_edge,origin,destin;
    printf("Enter number of vertices : ");
```

```
    scanf("%d",&n);    max_edge = n*(n-1);
    for(count=1;count<=max_edge;count++)
    {
        printf("Enter edge %d( -1 -1 to quit ) :",count);
        scanf("%d %d",&origin,&destin);
        if((origin == -1) && (destin == -1))
            break;
        if(origin>=n || destin>=n || origin<0 || destin<0)
        {
            printf("Invalid edge!\n");
            count--;
        }
    else
        {
            adj[origin][destin] = 1;
        }
    }
}
```

Output :-

Enter number of vertices : 5

Enter edge 1(-1 -1 to quit) : 0 2

Enter edge 2(-1 -1 to quit) : 2 0

Enter edge 3(-1 -1 to quit) : 0 3

Enter edge 4(-1 -1 to quit) : 3 0

Enter edge 5(-1 -1 to quit) : 0 1

Enter edge 6(-1 -1 to quit) : 1 0

Enter edge 7(-1 -1 to quit) : 2 4

Enter edge 8(-1 -1 to quit) : 4 2

Enter edge 9(-1 -1 to quit) : 1 2

Enter edge 10(-1 -1 to quit) : 2 1 Enter

edge 11(-1 -1 to quit) : -1 -1 Enter

Start Vertex for BFS:

0

BFS traversal

0 1 2 3 4

Result :- Program executed successfully and output verified.

Program No:11

DFS GRAPH TRAVERSAL

Aim :- To create a program to implement DFS traversal

Algorithm :-

1. Create a stack to store the vertices to be visited.
2. Initialize all vertices as not visited.
3. Pick a starting vertex and mark it as visited and push it to the stack.
4. While the stack is not empty, do the following:
    a. Pop a vertex from the stack and print it.
    b. Get all adjacent vertices of the popped vertex.
    c. For each adjacent vertex, if it has not been visited, mark it as visited and push it to the stack.
5. Repeat step 4 until the stack is empty.

Program :-

```
#include<stdio.h>
#include<stdlib.h>
#define MAX 100
#define initial 1
#define waiting 2
#define visited 3

int n;
int adj[MAX][MAX];
```

```
int state[MAX];
void create_graph();
void DFS_Traversal();
void DFS(int v);
int stack[MAX], top = -1;
void insert(int vertex);
int delete();
int isEmpty();

int main()
{
        create_graph();
        DFS_Traversal();
        return 0;
}

void DFS_Traversal()
{
        int v;
        for(v=0; v<n; v++)
        state[v] = initial;
        printf("Enter
StartVertex for DFS: \n");
        scanf("%d", &v);
        DFS(v);
}

void DFS(int v)
{
        int i;
        printf("DFS traversal");
        insert(v);
        state[v] = waiting;
        while(!isEmpty())
        {
                v = delete();
                printf("%d ",v);
                state[v] = visited;
                for(i=0;i<n;i++)
                {
                        if(adj[v][i] == 1 && state[i] == initial)
                        {
                                insert(i);
                                state[i] = waiting;
                        }
                }
        }
        printf("\n");
```

```
}

void insert(int vertex)
{
        if(top == MAX-1)
                printf("stack Overflow\n");
        else
        {
                top++;
                stack[top] = vertex ;
        }
}

int isEmpty()
{
        if(top==-1)
                return 1;
        else
                return 0;
}

int delete()
{
        int delete_item;
        if(top == -1)
        {
                printf("stack empty\n");
                exit(1);
        }
        delete_item = stack[top];
        top--;
        return delete_item;
}
void create_graph()
{
        int count,max_edge,origin,destin;
        printf("Enter number of vertices : ");
        scanf("%d",&n);
        max_edge = n*(n-1);
        for(count=1;count<=max_edge;count++)
        {
                printf("Enter edge %d( -1 -1 to quit ) : ",count);
                scanf("%d %d",&origin,&destin);
                if((origin == -1) && (destin == -1))
                        break;
                if(origin>=n || destin>=n || origin<0 || destin<0)
                {
                        printf("Invalid edge!\n");
```

```
                count--;
            }
            else
            {
                adj[origin][destin] = 1;
            }
        }
}
```

Output :-

```
Enter number of vertices : 5
Enter edge 1( -1 -1 to quit ) : 0 1
Enter edge 2( -1 -1 to quit ) : 1 0
Enter edge 3( -1 -1 to quit ) : 0 2
Enter edge 4( -1 -1 to quit ) : 2 0
Enter edge 5( -1 -1 to quit ) : 0 3
Enter edge 6( -1 -1 to quit ) : 3 0
Enter edge 7( -1 -1 to quit ) : 1 4
Enter edge 8( -1 -1 to quit ) : 4 1
Enter edge 9( -1 -1 to quit ) : 2 4
Enter edge 10( -1 -1 to quit ) : 4 2
Enter edge 11( -1 -1 to quit ) : 3 4
Enter edge 12( -1 -1 to quit ) : 4 3
Enter edge 13( -1 -1 to quit ) : -1 -1
Enter Start Vertex for DFS:
0
DFS traversal: 0 3 4 2 1
```

Result :- Program executed successfully and output verified.

Program No:12

Topological Sorting

Aim :- To obtain the Topological ordering of vertices in a given digraph.

Algorithm :-

1. Initialize variables:
   - n (Number of vertices)
   - adj[MAX][MAX] (Adjacency Matrix)
   - queue[MAX], front, and rear for queue operations
2. Create the graph using create_graph function:
   - Prompt the user to enter the number of vertices (n).
   - Use a loop to input edges until the user enters -1 -1.
   - Validate input edges and populate the adjacency matrix.
3. Initialize arrays:
   - indeg[MAX] to store the indegree of each vertex.
   - topo_order[MAX] to store the topological ordering.
4. Calculate the indegree of each vertex using indegree function:
   - Iterate through vertices and count incoming edges for each vertex.
5. Enqueue vertices with indegree 0 into the queue:
   - Iterate through vertices and enqueue those with indegree 0.
6. Initialize count to 0:
   - count will track the number of vertices processed.
7. Topological Sorting:

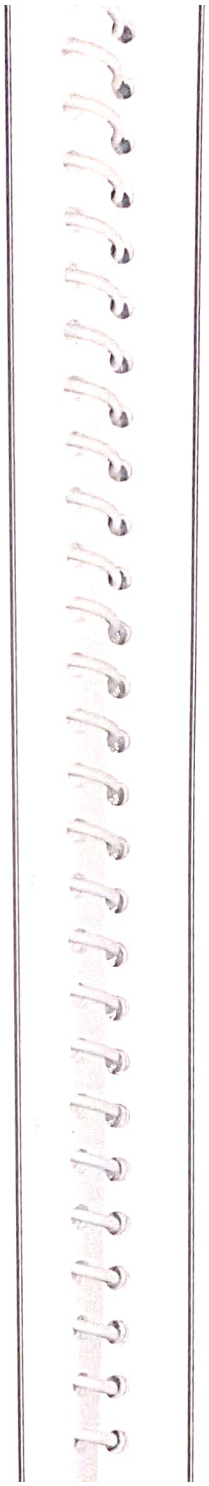
- While the queue is not empty and count < n:
- Dequeue a vertex v.
- Add v to topo_order.
- Remove all outgoing edges from v and update the indegree of adjacent vertices.
- Enqueue vertices with updated indegree 0.

8. Check for cycles:
   - If count < n, print an error message indicating a cycle in the graph.
9. Print the vertices in topological order:
   - Print the contents of the topo_order array.
10. End the program.

Program :-

```
#include<stdio.h>
#include<stdlib.h>

#define MAX 100

int n;    /Number of vertices in the graph/
int adj[MAX][MAX]; /Adjacency Matrix/
void create_graph();

int queue[MAX], front = -1,rear = -1;
void insert_queue(int v);
int delete_queue();
int isEmpty_queue();

int indegree(int v);

int main()
{
    int i,v,count,topo_order[MAX],indeg[MAX];

    create_graph();

    /Find the indegree of each vertex/
    for(i=0;i<n;i++)
    {
        indeg[i] = indegree(i);
        if( indeg[i] == 0 )
            insert_queue(i);
    }

    count = 0;
```

```
while(  !isEmpty_queue( ) && count < n )
{
      v = delete_queue();
topo_order[++count] = v; /Add vertex v to topo_order array/
      /*Delete all edges going fron vertex v */
      for(i=0; i<n; i++)
      {
            if(adj[v][i] == 1)
            {
                  adj[v][i] = 0;
                  indeg[i] = indeg[i]-1;
                  if(indeg[i] == 0)
                        insert_queue(i);
            }
      }
}

if( count < n )
{
      printf("\nNo topological ordering possible, graph contains cycle\n");
      exit(1);
}
```



```
      printf("\nVertices in topological order are :\n");
      for(i=1; i<=count; i++)
            printf( "%d ",topo_order[i] );
      printf("\n");

      return 0;
}/End of main()/

void insert_queue(int vertex)
{
      if (rear == MAX-1)
            printf("\nQueue Overflow\n");
      else
      {
            if (front == -1)  /*If queue is initially empty */
                  front = 0;
            rear = rear+1;
            queue[rear] = vertex ;
      }
}/End of insert_queue()/

int isEmpty_queue()
```

```
{
    if(front == -1 || front > rear )
        return 1;
    else
        return 0;
}/End of isEmpty_queue()/

int delete_queue()
{
    int del_item;
    if (front == -1 || front > rear)
    {
        printf("\nQueue Underflow\n");
        exit(1);
    }
    else
    {
        del_item = queue[front];
        front = front+1;
        return del_item;
    }
}/*End of delete_queue() */
```

```
int indegree(int v)
{
    int i,in_deg = 0;
    for(i=0; i<n; i++)
        if(adj[i][v] == 1)
            in_deg++;
    return in_deg;
}/*End of indegree() */

void create_graph()
{
    int i,max_edges,origin,destin;

    printf("\nEnter number of vertices : ");
    scanf("%d",&n);
    max_edges = n*(n-1);

    for(i=1; i<=max_edges; i++)
    {
        printf("\nEnter edge %d(-1 -1 to quit): ",i);
        scanf("%d %d",&origin,&destin);
```

```
        if((origin == -1) && (destin == -1))
            break;

        if( origin >= n || destin >= n || origin<0 || destin<0)
        {
            printf("\nInvalid edge!\n");
            i--;
        }
        else
            adj[origin][destin] = 1;
    }
}
```

Output :-

Enter number of vertices : 4

Enter edge 1(-1 -1 to quit): 0 1

Enter edge 2(-1 -1 to quit): 1 2

Enter edge 3(-1 -1 to quit): 3 0

Enter edge 4(-1 -1 to quit): 3 2

Enter edge 5(-1 -1 to quit): -1 -1

Vertices in topological order are :

3 0 1 2

Result :- Program executed successfully and output verified.

Program No:13

PRIMS ALGORITHM

Aim :- To find minimum cost spanning tree using Prim's algorithm

Algorithm :-

1. Start with a random vertex, say vertex V, from the given graph.
2. Initialize an empty set called the MST (minimum spanning tree) that will eventually contain the edges of the MST.
3. Initialize a priority queue called PQ that will contain pairs of vertices and their respective distances. The distance between vertices is the weight of the edge connecting them.
4. For every vertex adjacent to V, add the pair (V, adjacent vertex) to the PQ with its respective distance.
5. While the PQ is not empty, do the following:
    a. Remove the vertex-pair with the smallest distance from the PQ.
    b. If both vertices of the removed pair are already in the MST, continue to the next iteration.
    c. Otherwise, add the edge connecting the two vertices to the MST and add all the pairs of vertices adjacent to the newly added vertex to the PQ with their respective distances.
6. Continue iterating until all vertices are in the MST.
7. Return the MST.

Program :-

```
#include<stdio.h>

int a,b,u,v,n,i,j,ne=1;

int visited[10]={0},min,mincost=0,cost[10][10];

void main()

{
```

```
printf("\nEnter the number of nodes:");
scanf("%d",&n);
printf("\nEnter the adjacency matrix:\n");
for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
        {
                scanf("%d",&cost[i][j]);
                if(cost[i][j]==0)
                cost[i][j]=999;
        }
        visited[1]=1;
        printf("\n");
        while(ne < n)
        {
        for(i=1,min=999;i<=n;i++)
        for(j=1;j<=n;j++)
                if(cost[i][j]< min)
        if(visited[i]!=0)
        {
                min=cost[i][j];
                a=u=i;

                b=v=j;
        }
```



```
        if(visited[u]==0 || visited[v]==0)
        {
                printf("\n Edge %d:(%d %d) cost:%d",ne++,a,b,min);
                mincost+=min;
                visited[b]=1;

        }
        cost[a][b]=cost[b][a]=999;
    }
    printf("\n Minimun cost=%d",mincost);
}
```

Output :-

Enter the number of nodes:5 Enter

the adjacency matrix:

0	2	1	0	0
2	0	0	4	5
1	0	0	0	2
0	4	0	0	0
0	0	2	0	0

Edge 1:(1 3) cost:1

Edge 2:(1 2) cost:2

Edge 3:(3 5) cost:2

Edge 4:(2 4) cost:4

Minimun cost=9

Result :- Program executed successfully and output verified.

Program No:14

KRUSKALS ALGORITHM

Aim :- To find minimum cost spanning tree using Kruskal's algorithm

Algorithm :-

1. Prompt the user to enter the number of vertices (n).
2. Accept the cost adjacency matrix for the graph, where cost[i][j] represents the cost of the edge between vertices i and j.
3. If an edge has a cost of 0, replace it with a large value (e.g., 999) to represent infinity.
4. Initialize variables:
   - ne (number of edges) to 1.
   - mincost (total cost of MST) to 0.
   - Arrays:
     - parent[9] to store parent vertices for each vertex.
     - i, j, k, a, b, u, v for loop control and temporary storage.
5. Output a header indicating the implementation of Kruskal's algorithm.
6. Print the cost adjacency matrix for reference.
7. Execute the Kruskal's algorithm loop until the number of edges (ne) is less than the number of vertices (n):
   - Nested loop to find the minimum cost edge (min) in the cost matrix.
   - Identify the vertices (a and b) corresponding to the minimum cost edge.
   - Find the parent vertices of a and b.
   - If the vertices are not in the same set, perform the union operation, and add the edge to the MST.
   - Update the cost matrix by setting the cost of the chosen edge to a large value (999).
   - Increment the number of edges (ne) and update the total cost (mincost).
8. Print the edges of the Minimum Cost Spanning Tree along with their costs.
9. Print the total minimum cost of the MST.

Program :-

```
#include<stdio.h>

#include<stdlib.h>
```

```
int i,j,k,a,b,u,v,n,ne=1;
int min,mincost=0,cost[9][9],parent[9];
int find(int);
int uni(int,int);
void main()
{
        printf("\nImplementation of Kruskal's algorithm\n");
        printf("\nEnter the no. of vertices:");
        scanf("%d",&n);
        printf("\nEnter the cost adjacency matrix:\n");
        for(i=1;i<=n;i++)
        {
                for(j=1;j<=n;j++)
                {
                        scanf("%d",&cost[i][j]);
                        if(cost[i][j]==0)
                                cost[i][j]=999;
                }
        }
        printf("The edges of Minimum Cost Spanning Tree are\n");
        while(ne < n)
        {
                for(i=1,min=999;i<=n;i++)
                {
```



```
                        for(j=1;j <= n;j++)
                        {
                                if(cost[i][j] < min)
                                {
                                        min=cost[i][j];
                                        a=u=i;
                                        b=v=j;
                                }
                        }
                }
                u=find(u);
                v=find(v);
                if(uni(u,v))
                {
                        printf("%d edge (%d,%d) =%d\n",ne++,a,b,min);
                        mincost +=min;
                }
                cost[a][b]=cost[b][a]=999;
        }
        printf("\nMinimum cost = %d\n",mincost);
}
int find(int i)
{
```

```
        while(parent[i])
                i=parent[i];
        return i;
}
int uni(int i,int j)
{
        if(i!=j)
        {
                parent[j]=i;
                return 1;
        }
        return 0;
}
```

Output :-

Implementation of Kruskal's algorithm

Enter the no. of vertices:5

Enter the cost adjacency matrix:

0	2	1	0	0
2	0	0	4	5
1	0	0	0	2
0	4	0	0	0
0	0	2	0	0

The edges of Minimum Cost Spanning Tree are

1 edge (1,3) =1

2 edge (1,2) =2

3 edge (3,5) =2

4 edge (2,4) =4

Minimum cost = 9

Result :- Program executed successfully and output verified.

Program No:15

DIJIKSTRA'S ALGORITHM

Aim :- To find the shortest path of a graph using Dijkstra's algorithm.

Algorithm :-

1: Read Input

- Read the number of vertices n.
- Read the adjacency matrix G representing the weighted graph.
- Read the starting node u.

2: Initialization

- Create a cost matrix cost[MAX][MAX].
- Create arrays distance[MAX], pred[MAX], and visited[MAX].
- Set count to 1.

3: Initialize Cost Matrix

- For each pair of vertices (i, j):
  - If there is no edge between i and j, set cost[i][j] to INFINITY.
  - Otherwise, set cost[i][j] to the weight of the edge.

4: Initialize Arrays

- For each vertex i:
  - Set distance[i] to the weight of the edge from the starting node to i.
  - Set pred[i] to the starting node.
  - Set visited[i] to 0.

5: Set Starting Node

- Set distance[startnode] to 0.

- Set visited[startnode] to 1.

6: Dijkstra's Algorithm Loop

- While count is less than n - 1:
  - Find the node nextnode with the minimum distance that has not been visited.
  - Mark nextnode as visited.
  - Update the distance and predecessor arrays for the neighboring nodes.

7: Print Results

- For each node i (excluding the starting node):
  - Print the distance of node i.
  - Print the path from the starting node to node i using the predecessor array.

8: End

Program :-

```
#include<stdio.h>
#include<conio.h>
#define INFINITY 9999
#define MAX 10

void dijkstra(int G[MAX][MAX],int n,int startnode);

int main()
{
int G[MAX][MAX],i,j,n,u;
        printf("Enter no. of vertices:");
        scanf("%d",&n);
```



```
        printf("\nEnter the adjacency matrix:\n");
        for(i=0;i<n;i++)
                for(j=0;j<n;j++)
                        scanf("%d",&G[i][j]);
        printf("\nEnter the starting node:");
        scanf("%d",&u);
        dijkstra(G,n,u);
        return 0;
}
void dijkstra(int G[MAX][MAX],int n,int startnode)
{
        int cost[MAX][MAX],distance[MAX],pred[MAX];
        int visited[MAX],count,mindistance,nextnode,i,j;
        //pred[] stores the predecessor of each node
        //count gives the number of nodes seen so far
        //create the cost matrix
        for(i=0;i<n;i++)
                for(j=0;j<n;j++)
                        if(G[i][j]==0)
                                cost[i][j]=INFINITY;
                        else
                                cost[i][j]=G[i][j];
                //initialize pred[],distance[] and visited[]
        for(i=0;i<n;i++)
```

```
{
        distance[i]=cost[startnode][i];
        pred[i]=startnode;
        visited[i]=0;
}
distance[startnode]=0;
visited[startnode]=1;
count=1;
while(count<n-1)
{
        mindistance=INFINITY;
        //nextnode gives the node at minimum distance
        for(i=0;i<n;i++)
                if(distance[i]<mindistance&&!visited[i])
                {
                        mindistance=distance[i];
                        nextnode=i;
                }
        //check if a better path exists through nextnode
        visited[nextnode]=1;
        for(i=0;i<n;i++)
        if(!visited[i])
                if(mindistance+cost[nextnode][i]<distance[i])
                {
```



```
                        distance[i]=mindistance+cost[nextnode][i];
                        pred[i]=nextnode;
                }
        count++;
}

//print the path and distance of each node
for(i=0;i<n;i++)
        if(i!=startnode)
                {
                        printf("\nDistance of node%d=%d",i,distance[i]);
                        printf("\nPath=%d",i);
                        j=i;
                        do
                        {
                                j=pred[j];
                                        printf("<-%d",j);
                        }while(j!=startnode);
                }
}
```

Output :-

Enter no. of vertices:6

Enter the adjacency matrix:

0 1 1 0 0 0

1 0 1 1 0 0

1 1 0 0 1 0

0 1 0 0 1 1

0 0 1 1 0 1

0 0 0 1 1 0

Enter the starting node:0

Distance of node1=1

Path=1<-0

Distance of node2=1

Path=2<-0

Distance of node3=2

Path=3<-1<-0

Distance of node4=2

Path=4<-2<-0

Distance of node5=3

Path=5<-3<-1<-0

Result :- Program executed successfully and output verified.

Certified by
Aswathy Pavappuram