**Practical 6**

**Aim: - Graph**

**6.1 Print all the nodes reachable from a given starting node in a digraph using BFS method.**

**6.2 Check whether a given graph is connected or not using DFS method.**

**6.3 Find Minimum Cost spanning tree of a given undirected graph using Kruksal’s algorithm**

**6.4 Find Minimum Cost spanning tree of a given undirected graph using Prim’s algiorithm.**

**6.5 From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra’s algorithm**

**6.1 Print all the nodes reachable from a given starting node in a digraph using BFS method.**

**Theory:-**

* Breadth-first search (BFS) is an [algorithm](https://en.wikipedia.org/wiki/Algorithm) for traversing or searching [tree](https://en.wikipedia.org/wiki/Tree_data_structure) or [graph](https://en.wikipedia.org/wiki/Graph_(data_structure)) data structures.
* It starts at the root and explores the neighbor nodes first, before moving to the next level neighbors.
* The time complexity can be expressed as O (|V|+|E|), whereas the space complexity is O (|V|).
* If the graph is represented by an [adjacency list](https://en.wikipedia.org/wiki/Adjacency_list) it occupies O (|V|+|E|) space in memory, while an [adjacency matrix](https://en.wikipedia.org/wiki/Adjacency_matrix) representation occupies O (|V2|).

**Algorithm: -**

Algorithm BFS (G)

{

1. While ( G has an unvisited node) do
2. v  <-  an unvisited node;
3. visit [v] <- 1
4. en\_queue (v, Q);
5. While ( Q is not empty) do
6. x <- del\_queue ( Q)
7. For ( unvisited neighbor y of x ) do
8. visit [y] <- 1;
9. en\_queue ( v, Q) ;

**Program: -**

**Code: -**

#include<iostream>

using namespace std;

int a[20][20],q[20],visited[20],n,i,j,f=0,r=-1;

void BFS(int v)

{

for(i=1;i<=n;i++)

if(a[v][i] && !visited[i])

q[++r]=i;

if(f<=r)

{

visited[q[f]]=1;

BFS(q[f++]);

}

}

main()

{

int v;

cout<<"\n Enter the number of vertices:";

cin>>n;

for(i=1;i<=n;i++)

{

q[i]=0;

visited[i]=0;

}

cout<<"\n Enter adjacency Matrix:\n\n";

for(i=1;i<=n;i++)

{

cout<<"\t";

for(j=1;j<=n;j++)

cin>>a[i][j];

}

cout<<"\n Enter the starting vertex:";

cin>>v;

BFS(v);

cout<<"\n The sequence is:\n";

for(i=1;i<=n;i++)

if(visited[i])

cout<<i;

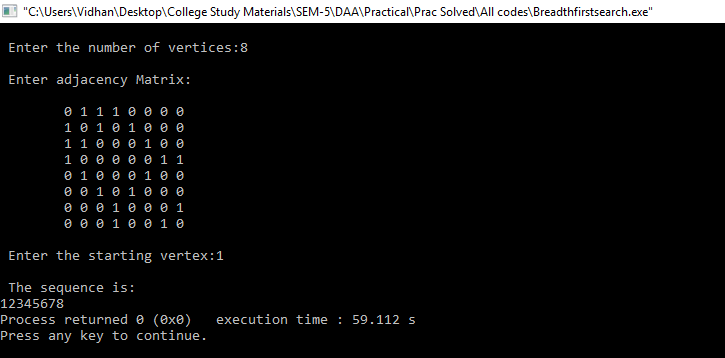
else

cout<<"\n BFS is not possible";

return 0;

}

**Output: -**



**Conclusion: -**

Breadth first search is a graph traversal approach having the time complexity of O (V + E). The algorithm gets complex as the number of vertices and edges increases.

**6.2 Check whether a given graph is connected or not using DFS method.**

**Theory:-**

* Depth-first search (DFS) is an [algorithm](https://en.wikipedia.org/wiki/Algorithm) for traversing or searching [tree](https://en.wikipedia.org/wiki/Tree_data_structure) or [graph](https://en.wikipedia.org/wiki/Graph_(data_structure)) data structures. One starts at the [root](https://en.wikipedia.org/wiki/Tree_(data_structure)#Terminology) (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before [backtracking](https://en.wikipedia.org/wiki/Backtracking).
* DFS is typically used to traverse an entire graph, and takes time O (|V| + |E|), linear in the size of the graph. In the worst case to store the stack of vertices on the current search path as well as the applications it also uses space O (|V|) in set of already-visited vertices.
* DFS may suffer from non-termination; in this case search is performed for a limited depth.

**Algorithm: -**

DFS-iterative (G, s):

let S be stack

S.push(s) //Inserting s in stack

mark s as visited.

while ( S is not empty):

//Pop a vertex from stack to visit next

v = S.top( )

S.pop( )

//Push all the neighbours of v in stack that are not visited

for all neighbours w of v in Graph G:

if w is not visited :

S.push( w )

mark w as visited

DFS-recursive(G, s):

mark s as visited

for all neighbours w of s in Graph G:

if w is not visited:

DFS-recursive(G, w)

**Program: -**

**Code: -**

#include<iostream>

using namespace std;

int TOP=-1;

int stack[10];

void push(int a);

int pop();

main()

{

int n,in[10][10],i,j,x=0,q,flag=0,start,current,visit[10]={0},connected[10]={0},count=0,path[10];

cout<<"\n Enter The No Of Nodes:";

cin>>n;

cout<<"\n Enter The Start Node:";

cin>>start;

cout<<"\n Enter The Adjacency Matrix :\n\n";

for(i=0;i<n;i++)

{

cout<<("\t");

for(j=0;j<n;j++)

{

cin>>in[i][j];

}

}

q=n;

push(start-1);

visit[start-1]=1;

connected[start-1]=1;

while(q>0)

{

current=pop();

path[x]=current;

x++;

for(i=n-1;i>=0;i--)

{

if(in[current][i]==1 && visit[i]==0)

{

visit[i]=1;

connected[i]=1;

push(i);

}

}

q--;

}

while(TOP>=0)

{

current=pop();

visit[current]=1;

x--;

path[x]=current;

x++;

}

if(TOP==-1)

{

for(i=0;i<n;i++)

{

if(connected[i]==1)

{

count++;

}

}

}

if(count==n)

{

cout<<"\n Graph is Connected.\n\n The Sequence is :";

for(i=0;i<n;i++) cout<<path[i];

}

else

{

cout<<"\n Graph is Not Connected.";

}

}

void push(int a)

{

TOP++;

stack[TOP]=a;

}

int pop()

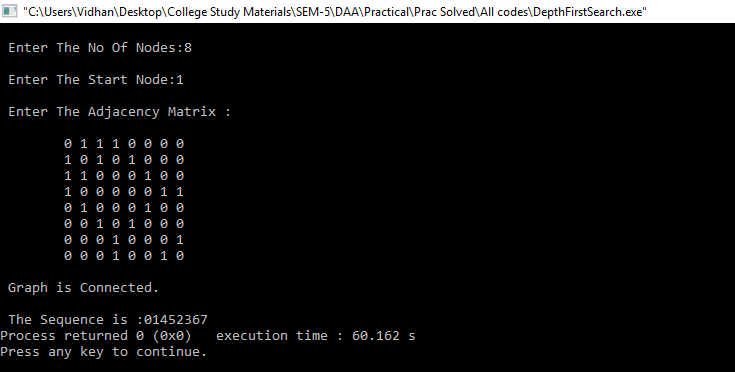
{

TOP--;

return stack[TOP+1];

}

**Output: -**



**Conclusion: -**

Depth First search is a graph traversal algorithm, having the time complexity of O (V+E). As the number of vertices and edges increases, the complexity of the algorithm directly increases.

**6.3 Find Minimum Cost spanning tree of a given undirected graph using Kruksal’s algorithm**

**Theory:-**

Kruskal's algorithm is a [minimum-spanning-tree algorithm](https://en.wikipedia.org/wiki/Minimum_spanning_tree#Algorithms) which finds an edge of the least possible weight that connects any two trees in the forest.

It is a [greedy algorithm](https://en.wikipedia.org/wiki/Greedy_algorithm) in [graph theory](https://en.wikipedia.org/wiki/Graph_theory) as it finds a [minimum spanning tree](https://en.wikipedia.org/wiki/Minimum_spanning_tree) for a [connected](https://en.wikipedia.org/wiki/Connectivity_(graph_theory)) graph adding increasing cost arcs at each step. This means it finds a subset of the [edges](https://en.wikipedia.org/wiki/Edge_(graph_theory)) that forms a tree that includes every [vertex](https://en.wikipedia.org/wiki/Vertex_(graph_theory)), where the total weight of all the edges in the tree is minimized.

**Algorithm: -**

**MST-KRUSKAL *(G,w)***

1. *A* ← ∅
2. **for** each vertex *v* ∈*V*[*G*]
3. **do** MAKE-SET*(v)*
4. sort the edges of *E* into non-decreasing order by weight *w*
5. **for** each edge *(u, v)* ∈*E*, taken in non-decreasing order by weight
6. **do if** FIND-SET*(u)* ≠ FIND-SET*(v)*
7. **then** *A* ← *A* ∪ {*(u, v)*}
8. UNION*(u, v)*
9. **return** *A*

**Program: -**

**Code: -**

#include<iostream>

#include<conio.h>

#include<stdlib.h>

using namespace std;

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);

main()

{

cout<<"\n\tImplementation of Kruskal's algorithm\n";

cout<<"\nEnter the no. of vertices:";

cin>>n;

cout<<"\nEnter the cost adjacency matrix:\n";

for(i=1;i<=n;i++)

{

for(j=1;j<=n;j++)

{

cin>>cost[i][j];

if(cost[i][j]==0)

cost[i][j]=999;

}

}

cout<<"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))

{

cout<<"edge\n"<<ne++,a,b,min;

mincost +=min;

}

cost[a][b]=cost[b][a]=999;

}

cout<<"\n\tMinimum cost = %d\n",mincost;

return 0;

}

int find(int i)

{

while(parent[i])

i=parent[i];

return i;

}

int uni(int i,int j){

if(i!=j){

parent[j]=i;

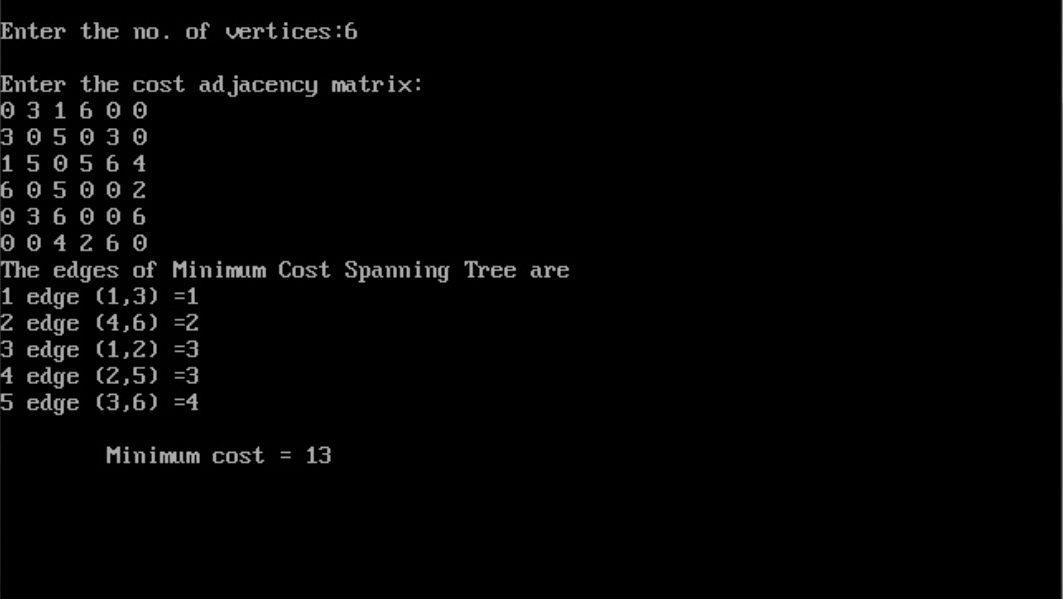
return 1;

}

return 0;

}

**Output: -**



**Conclusion: -**

Kruskal’s algorithm is an approach to obtain the minimum spanning tree. It has time complexity of O (E log V), where E: edges and V: Vertices.

**6.4 Find Minimum Cost spanning tree of a given undirected graph using Prim’s algorithm.**

**Theory:-**

* Prim's algorithm is a [greedy algorithm](https://en.wikipedia.org/wiki/Greedy_algorithm) that finds a [minimum spanning tree](https://en.wikipedia.org/wiki/Minimum_spanning_tree) for a [weighted](https://en.wikipedia.org/wiki/Weighted_graph) [undirected graph](https://en.wikipedia.org/wiki/Undirected_graph).
* This means it finds a subset of the [edges](https://en.wikipedia.org/wiki/Edge_(graph_theory)) that forms a [tree](https://en.wikipedia.org/wiki/Tree_(graph_theory)) that includes every [vertex](https://en.wikipedia.org/wiki/Vertex_(graph_theory)), where the total weight of all the [edges](https://en.wikipedia.org/wiki/Graph_theory) in the tree is minimized.
* The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

**Algorithm: -**

**MST-PRIM*( G, w, r)***

*1.A={}*

2. for each *u* ∈ *V*[*G*]

3. do *key*[*u*]←∞

4. *π*[*u*]← NIL

5. *key*[*r*] ← 0

6. *Q* ← *V*[*G*]

7. while *Q* = ∅

8. do *u* ← EXTRACT-MIN*(Q)*

*9. Q=Q-u*

*10. If(π*[*u*]!=NIL)

*11. A=AU(u, π*[*u*]*)*

12. for each *v* ∈ *Adj*[*u*]

13. do if *v* ∈ *Q* and *w(u, v) < key*[*v*]

14.  then *π*[*v*]← *u*

15. *key*[*v*] ← *w(u, v)*

**Program: -**

**Code: -**

#include<iostream>

#include<stdlib.h>

using namespace std;

#define infinity 9999

#define MAX 20

int G[MAX][MAX],spanning[MAX][MAX],n;

int prims();

int main()

{

int i,j,total\_cost;

cout<<"Enter no. of vertices:";

cin>>n;

cout<<"\nEnter the adjacency matrix:\n";

for(i=0;i<n;i++)

for(j=0;j<n;j++)

cin>>G[i][j];

total\_cost=prims();

cout<<"\nspanning tree matrix:\n";

for(i=0;i<n;i++)

{

cout<<("\n");

for(j=0;j<n;j++)

cout<<endl<<spanning[i][j];

}

cout<<"\n\nTotal cost of spanning tree=",total\_cost);

return 0;

}

int prims()

{

int cost[MAX][MAX];

int u,v,min\_distance,distance[MAX],from[MAX];

int visited[MAX],no\_of\_edges,i,min\_cost,

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];

spanning[i][j]=0;

}

distance[0]=0;

visited[0]=1;

for(i=1;i<n;i++)

{

distance[i]=cost[0][i];

from[i]=0;

visited[i]=0;

}

min\_cost=0;

no\_of\_edges=n-1;

while(no\_of\_edges>0)

{

min\_distance=infinity;

for(i=1;i<n;i++)

if(visited[i]==0&&distance[i]<min\_distance)

{

v=i;

min\_distance=distance[i];

}

u=from[v];

spanning[u][v]=distance[v];

spanning[v][u]=distance[v];

no\_of\_edges--;

visited[v]=1;

for(i=1;i<n;i++)

if(visited[i]==0&&cost[i][v]<distance[i])

{

distance[i]=cost[i][v];

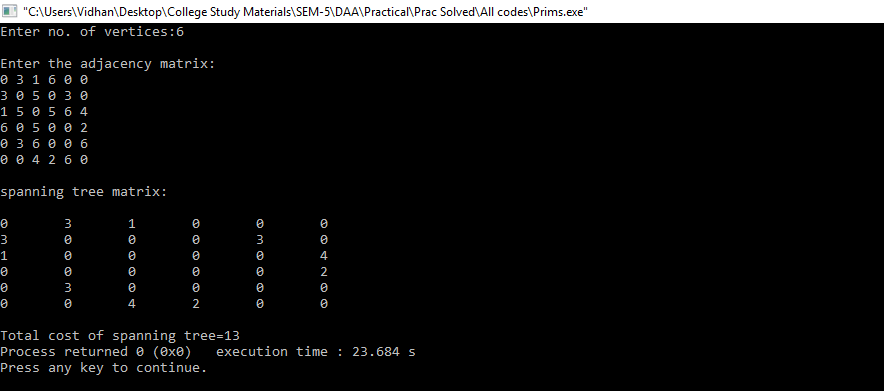
from[i]=v;

}

min\_cost=min\_cost+cost[u][v];

}

**Output: -**



**Conclusion: -**

Prim’s algorithm is an approach to obtain the minimum spanning tree. It has time complexity of O (E log V), where E: edges and V: Vertices.

**6.5 From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra’s algorithm.**

**Theory:-**

Dijkstra's algorithm has many variants but the most common one is to find the shortest paths from the source vertex to all other vertices in the graph.

The algorithm exists in many variants; Dijkstra's original variant found the shortest path between two nodes, but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a [shortest-path tree](https://en.wikipedia.org/wiki/Shortest-path_tree).

Time complexity is O (|V2|).

**Algorithm:**

Dijkstra (Graph, source)

step1: create vertex set Q

step2: for each vertex v in Graph:

dist[v] ← INFINITY

prev[v] ← UNDEFINED

add v to Q

step3: dist[source] ← 0

step4: while Q is not empty:

u ← vertex in Q with min dist[u]

remove u from Q

step5: for each neighbor v of u:

alt ← dist[u] + length (u, v)

if alt <dist[v]:

dist[v] ← alt

prev[v] ← u

step6: return dist[], prev[]

**Program: -**

**Code: -**

#include<iostream>

using namespace std;

main()

{

int in[10][10], i, j, k=0, n, src, q, select[10], ans[10], dist=0,

solution[10][10], l=0, min=10000, p, count=0, f, flag=0, temp, count1=0;

cout<<" Enter number Of Nodes:";

cin>>n;

src=0;

cout<<"\nEnter The Adjacency Matrix:\n\n";

for(i=0; i<n; i++){

cout<<"\t";

for(j=0; j<n; j++){

cin>>in[i][j];

solution[i][j]=-5;

}

}

q=n;

for(i=0; i<n; i++){

solution[l][i]=-1;

select[i]=0;

}

select[src]=1;

solution[l][src]=0;

l++;

ans[k]=src+1;

k++;

while(q>0){

for(i=0;i<n;i++){

if(in[src][i]>0){

if(select[i]==0){

temp = dist + in[src][i];

if(temp<min){

min=temp;

p=i;

}

solution[l][i]=temp;

}

}

else{

count++;

if(solution[l-1][i]==-1){

solution[l][i]=-1;

}

else if(select[i]==0){

solution[l][i]=solution[l-1][i];

}

}

}

ans[k]=p+1;

min=1000;

for(f=0;f<n;f++){

if(select[f]==1){

count1++;

}

}

if(count==n && count1==n-2){

src=ans[k-2]-1;

select[src]=0;

dist=dist-in[l-2][ans[k]-1];

count=0;

flag=1;

}

else{

src=p;

dist=solution[l][p];

k++;

l++;

}

if(flag==0){

select[p]=1;

flag=0;

}

else{

select[src]=1;

}

count=0;

count1=0;

q--;

}

cout<<"\n\nSolution:\n";

for(i=0;i<n;i++){

for(j=0;j<n;j++){

if(solution[i][j]!=-5){

cout<<"%4d"<<solution[i][j];

}

else{

cout<<" ";

}

}

cout<<"\n";

}

cout<<"\n\n Path:";

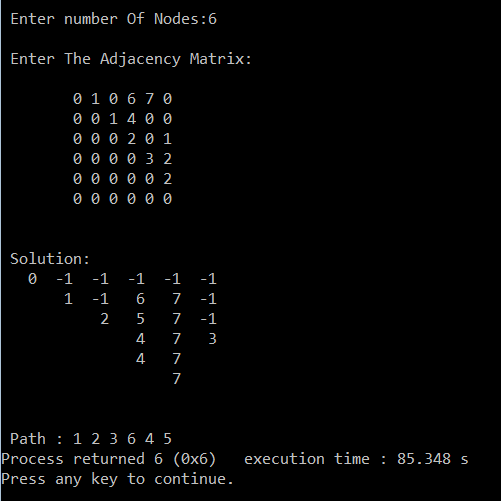
for(i=0;i<n;i++){

cout<<"%d ",ans[i];

}

}

**Output: -**



**Conclusion: -**

Dijkstra’s algorithm is an approach to obtain the minimum spanning tree. It has time complexity is O (|V2|), E: edges and V: Vertices.