**Date-**

**Assignment No. :**

**Problem Statement:**

Program in C to find the minimum spanning tree from a given graph G by Prim’s Algorithm.

**Theory:**

Like Kruskal’s algorithm, Prim’s algorithm is also a [Greedy algorithm](https://www.geeksforgeeks.org/archives/18528). It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.  
A group of edges that connects two set of vertices in a graph is called [cut in graph theory](http://en.wikipedia.org/wiki/Cut_%28graph_theory%29). **So, at every step of Prim’s algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the verices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).**

 The idea behind Prim’s algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets of vertices must be connected to make a Spanning Tree. And they must be connected with the minimum weight edge to make it a Minimum Spanning Tree.

The algorithm may informally be described as performing the following steps:

1. Initialize a tree with a single vertex, chosen arbitrarily from the graph.
2. Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.
3. Repeat step 2 (until all vertices are in the tree).

As described above, the starting vertex for the algorithm will be chosen arbitrarily, because the first iteration of the main loop of the algorithm will have a set of vertices in Q that all have equal weights, and the algorithm will automatically start a new tree in F when it completes a spanning tree of each connected component of the input graph. The algorithm may be modified to start with any particular vertex s by setting C[s] to be a number smaller than the other values of C (for instance, zero), and it may be modified to only find a single spanning tree rather than an entire spanning forest (matching more closely the informal description) by stopping whenever it encounters another vertex flagged as having no associated edge.

**Algorithm:**

**Input specification:**

1. gptr : The incidence matrix of dimension (n x n) of the given graph
2. v0 : The source vertex to start the search from

**Output specification:**

1. A two dimensional array tree[][] which will indicate the MST of the given graph.

**Steps:**

Algorithm for method main():

1. Print "Enter number of vertices : "
2. Input n
3. Set gptr = make\_2d(n) //make\_2d() is a function to make a 2d array //of order n and returns the address of the first element and here gptr is a //pointer to pointer of integer type
4. Set tree = NULL //tree is a pointer to pointer of integer type
5. Print "(If any two vertices is not connected by an edge, enter 0)"
6. Repeat through Step 7 to Step13 For i = 0 to n-1
7. Repeat through Step 8 to Step 12 For j = 0 to n-1
8. Print "Enter the weight between vertices"(i + 1)"and" j + 1": "
9. Input gptr[i][j]
10. If(gptr[i][j] = 0) Then
11. Set gptr[i][j] = INT\_MAX

[End of If structure]

1. Set j=j+1

[End of inner For loop]

1. Set i=i+1

[End of outer For loop]

1. Print "Enter the starting vertex : "
2. Input v0
3. If(v0 < 1 OR v0 > n) Then
4. Print "Invalid staring vertex"v0
5. goto Step 22

[End of If structure]

1. Print "The weighted adjacency matrix of the given graph is : "
2. print\_2d(gptr, n)//print2d is a function to print the elements of a 2d //array of order n
3. tree = prims(gptr, n, v0)
4. Print "The adjacency matrix of the minimal spanning tree of the given graph is : "
5. print\_2d(tree, n)
6. Repeat through Step 7 to Step 28 For i = 0 to n-1
7. free(gptr[i])//free() is a function to frees up allocated memory of the //given argument
8. If(tree != NULL) Then
9. free(tree[i])

[End of If structure]

1. i=i+1

[End of For loop]

1. free(gptr)
2. If(tree != NULL) Then
3. free(tree)

[End of If structure]

Algorithm for method Prims():

1. Repeat through Step 2 to Step For i=1 to N
2. Selected[i]=FALSE
3. Set i=i+1

[End of For loop]

1. Repeat through Step 2 to Step For i=1 to N
2. Repeat through Step 2 to Step For j=1 to N
3. Tree[i][j]=0
4. Set j=j+1

[End of inner For loop]

1. Set i=i+1

[End of outer For loop]

1. Selected[1]=TRUE, ne=1
2. Repeat through Step 11 to Step While(ne<N)
3. min=Infinite
4. Repeat through Step 2 to Step 20 For i=1 to N
5. If(Selected[i]=TRUE) Then
6. Repeat through Step 2 to Step 19 For j=1 to N
7. If(Selected[j]=FALSE) Then
8. If(min>Gptr[i][j]) Then
9. Set min=Gptr[i][j]
10. Set x=i ,y=j

[End of inner If structure]

[End of outer If structure]

1. Set j=j+1

[End of inner For loop]

[End of If structure]

1. Set i=i+1

[End of outer For loop]

1. Tree[x][y]=1
2. Selected[y]=TRUE
3. ne=ne+1

[End of While loop]

1. Return (TREE)

**Source Code:**

#include <stdio.h>

#include <stdbool.h>

#include <stdlib.h>

#include <limits.h>

#define inf INT\_MAX

int \*\* make\_2d(int n){//Making a 2d array by dynamic memory allocation

int \*\* matrix = (int \*\*)malloc(sizeof(int \*) \* n);

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

matrix[i] = (int \*)malloc(sizeof(int) \* n);

return matrix;

}

int \*\* prims(int \*\* gptr, int n, int v0){

bool selected[n];

int \*\* tree = make\_2d(n), i = 0, j = 0, ne = 0, min, x, y;

// Initializations

while(i < n)

selected[i++] = false;

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

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

tree[i][j] = 0;

selected[v0] = true, ne = 1;

// Finding the nearest neighbour of the selected vertex

while(ne < n){

min = inf;

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

if(selected[i] == true){

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

if(selected[j] == false){

if(min > gptr[i][j]){

min = gptr[i][j];

x = i, y = j;

}

}

}

}

}

tree[x][y] = 1;

selected[y] = true;

ne++;

}

return tree;

}

void print\_2d(int \*\* matrix, int n){//Printing the adjacency matrix

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

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

if(matrix[i][j] == INT\_MAX)

printf("-- ");

else

printf("%2d ", matrix[i][j]);

}

printf("\n");

}

}

int main(){

int n, v0;

printf("\nEnter number of vertices : ");

scanf("%d", &n);

int \*\* gptr = make\_2d(n), \*\* tree = NULL;

printf("\n(If any two vertices is not connected by an edge, enter 0)");

printf("\n\n");

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

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

printf("\b\rEnter the weight between vertices %d and %d : ", (i + 1), (j + 1));

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

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

gptr[i][j] = INT\_MAX;

}

}

printf("\nEnter the starting vertex : ");

scanf("%d", &v0);

if(v0 < 1 || v0 > n){

printf("\nInvalid staring vertex %d!", v0);

goto freeall;

}

printf("\nThe weighted adjacency matrix of the given graph is : \n");

print\_2d(gptr, n);

tree = prims(gptr, n, v0);

printf("\nThe adjacency matrix of the minimal spanning tree of the given graph is : \n");

print\_2d(tree, n);

freeall:

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

free(gptr[i]);

if(tree != NULL)

free(tree[i]);

}

free(gptr);

if(tree != NULL)

free(tree);

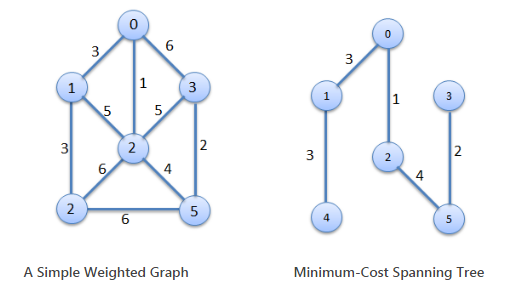
printf("\n");

return 0;

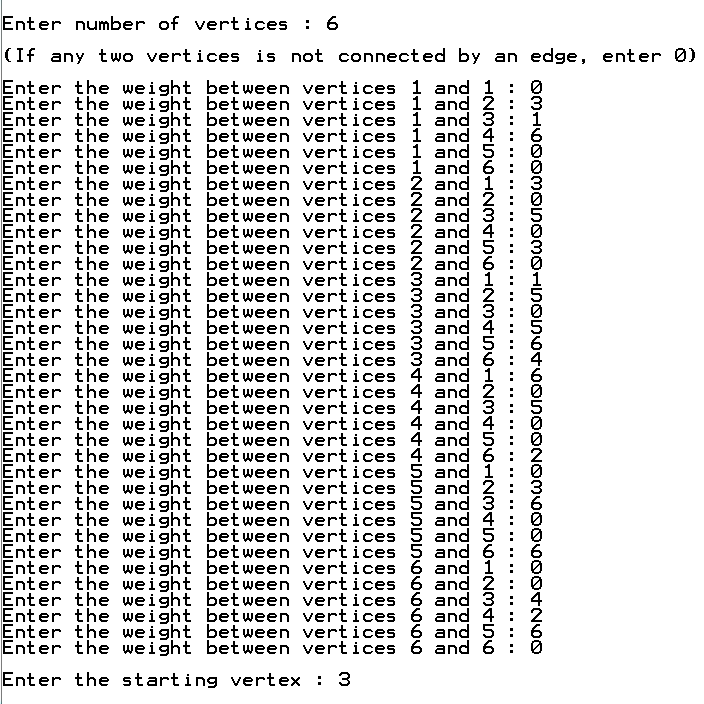
}

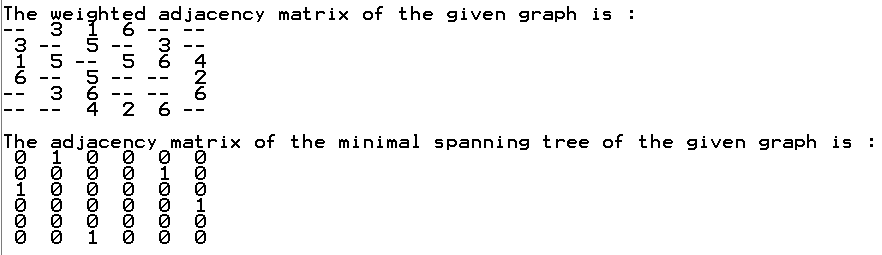
**Input & Output:**

Input graph:



Output of program:





**Discussion:**

1. In this algorithm an [adjacency matrix](https://en.wikipedia.org/wiki/Adjacency_matrix) or an [adjacency list](https://en.wikipedia.org/wiki/Adjacency_list) graph representation and linearly searching an array of weights to find the minimum weight edge to add, requires [O](https://en.wikipedia.org/wiki/Big-O_notation)(|V|2) running time. However, this running time can be greatly improved further by using [heaps](https://en.wikipedia.org/wiki/Heap_(data_structure)) to implement finding minimum weight edges in the algorithm's inner loop.
2. At every iteration of Prim's algorithm, an edge must be found that connects a vertex in a sub graph to a vertex outside the sub graph. Since P is connected, there will always be a path to every vertex. The output Y of Prim's algorithm is a [tree](https://en.wikipedia.org/wiki/Tree_(graph_theory)), because the edge and vertex added to tree Y are connected.
3. At the iteration when edge e was added to tree Y, edge f could also have been added and it would be added instead of edge e if its weight was less than e, and since edge f was not added, we conclude that{\displaystyle w(f)\geq w(e).},

w(f)≥w(e)

1. Using a simple [binary heap](https://en.wikipedia.org/wiki/Binary_heap) data structure, Prim's algorithm can now be shown to run in time [O](https://en.wikipedia.org/wiki/Big-O_notation)(|E| log |V|) where |E| is the number of edges and |V| is the number of vertices. Using a more sophisticated [Fibonacci heap](https://en.wikipedia.org/wiki/Fibonacci_heap), this can be brought down to [O](https://en.wikipedia.org/wiki/Big-O_notation)(|E| + |V| log |V|), which is [asymptotically faster](https://en.wikipedia.org/wiki/Asymptotic_computational_complexity) when the graph is [dense](https://en.wikipedia.org/wiki/Dense_graph) enough that |E| is [ω](https://en.wikipedia.org/wiki/Big-O_notation#Family_of_Bachmann.E2.80.93Landau_notations)(|V|), and [linear time](https://en.wikipedia.org/wiki/Linear_time) when |E| is at least |V| log |V|. For graphs of even greater density (having at least |V|*c* edges for some *c* > 1), Prim's algorithm can be made to run in linear time even more simply, by using a [*d*-ary heap](https://en.wikipedia.org/wiki/D-ary_heap) in place of a Fibonacci heap.