**II: The Greedy Method Date:**

**Aim:-**Write algorithm and C program to implement the following problems using the greedy method

a. Prim’s Algorithm

b. Kruskal’s Algorithm

c. Single Source Shortest paths

**THEORY:**

The greedy method suggests that one can devise an algorithm that works in stages,

considering one input at a time. At each stage, a decision is made regarding whether a

particular input is an optimal solution. This is done by considering the inputs in an order

determined by some selection procedure. If the inclusion of the next input into the partially

constructed optimal solution will result in an infeasible solution, then this input is not added

to the partial solution. Otherwise, it is added. The selection procedure itself is based on some

optimization measure. This measure may be the objective function. In fact, several different

optimization measures may be plausible for a given problem. Most of these, however, will

result in algorithms that generates unoptimal solutions. This version of the greedy techniques

called the subset paradigm

The function Select selects an input from a[ ] and removes it. The selected input's value is

assigned to x. Feasible is a Boolean-valued function that determines whether x can be

included into the solution vector. The function Union combines x with the solution and

updates the objective function. The function Greedy describes the essential way that a greedy

algorithm will look, once a particular problem is chosen and the functions Select, Feasible,

and Union are properly implemented.

1 **Algorithm** **Greedy**(a, n)

2 // a[l : n] contains the n inputs.

3 {

4 solution:=0; // Initialize the solution.

5 for i :=1to n do

6 {

7 x :=Select(a);

8 if Feasible(solution, x) then

9 solution:= Union (solution, x);

10 }

11 return solution;

12 }

**Greedy Algorithms:**

**Definition:**

A greedy algorithm is a problem-solving approach that makes locally optimal choices at each step with the aim of finding a global optimum solution. It selects the best immediate option without considering future consequences. Greedy algorithms are characterized by their greedy choice property, optimal substructure, and lack of backtracking.

**Features:**

1. Greedy Choice Property: Locally optimal choices are made at each step without considering future consequences.

2. Optimal Substructure: The optimal solution to the problem can be constructed from optimal solutions to its subproblems.

3. No Backtracking: Decisions are made once and not revisited or revised during the algorithm's execution.

**Advantages:**

1. Efficiency: Greedy algorithms are simple and efficient to implement, often involving straightforward local decisions.

2. Quick Solutions: They can find solutions rapidly, particularly for problems with numerous potential solutions.

3. Space Efficiency: Greedy algorithms typically require less memory space compared to other techniques due to their localized decision-making.

**Disadvantages:**

1. Suboptimal Solutions: Greedy algorithms may not always produce the optimal solution, potentially leading to suboptimal outcomes.

2. Risk of Local Optima: They can get stuck in local optima, especially if the problem space lacks the greedy choice property or optimal substructure.

3. Correctness: Without careful analysis, there's no guarantee of correctness, as certain cases may lead to incorrect solutions.

**Applications:**

1. Minimum Spanning Trees: Prim's and Kruskal's algorithms use greedy strategies to find the minimum spanning tree of a graph by adding minimum-weight edges iteratively.

2. Shortest Path Algorithms: Dijkstra's algorithm, for example, employs a greedy approach by selecting the shortest path from the source node iteratively.

3. Job Scheduling: Greedy algorithms can prioritize tasks based on criteria like earliest deadline first (EDF) or shortest processing time (SPT).

4. Fractional Knapsack Problem: They are useful in solving the fractional knapsack problem, maximizing total value within weight constraints.

5. Huffman Coding: Used in lossless data compression, Huffman coding constructs a variable-length prefix coding scheme based on character frequencies, using a greedy algorithm to build an optimal prefix-free binary tree.

Greedy algorithms find applications in diverse problem domains. However, their suitability depends on the problem characteristics, and thorough analysis is essential to ensure correctness and optimality. While they offer efficiency and quick solutions, caution must be exercised to address their limitations and validate their outcomes carefully.

1. **Prim’s Algorithm**

**Date:**

**Code**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

int cost[20][20];

void mini(int \*\*cost, int n, int \*k, int \*l)

{

    int min = cost[0][0];

    for (int i = 0; i < n; i++)

    {

        for (int j = 0; j < n; j++)

        {

            if (cost[i][j] < min)

            {

                min = cost[i][j];

                \*k = i;

                \*l = j;

            }

        }

    }

}

int prim(int \*\*cost, int n, int \*\*t)

{

    int i, j, k, l,x,y, near[n], min, mincost;

    mini(cost, n, &k, &l);

    t[0][0] = k;

    t[0][1] = l;

    mincost = cost[k][l];

    for (i = 0; i < n; i++)

    {

        if (cost[i][k] < cost[i][l])

            near[i] = k;

        else

            near[i] = l;

    }

    near[k] = -1;

    near[l] = -1;

    printf("Table of spanning tree edges:\n");

    for (x = 0; x < 1; x++)

    {

        for (y = 0; y < 2; y++)

        {

            printf("%d ", t[x][y] + 1);

        }

        printf("\n");

    }

    printf("Mincost: %d\n", mincost);

        printf("\n");

    for (i = 1; i < n - 1; i++)

    {

        min = 0;

        for (k = 0; k < n; k++)

        {

            if (near[k] != -1 && min == 0)

            {

                min = cost[k][near[k]];

                j = k;

            }

            else if (near[k] != -1 && cost[k][near[k]] < min)

            {

                min = cost[k][near[k]];

                j = k;

            }

        }

        printf("\nstep %d: ", i + 1);

        for (k = 0; k < n; k++)

        {

            printf(" near[%d] ", k + 1);

        }

        printf("\n");

        printf("Near:\t  ");

        for (k = 0; k < n; k++)

        {

            if (near[k] == -1)

                printf("  0      ");

            else

                printf("  %d      ", near[k] + 1);

        }

        printf("\n");

        printf("Cost:\t  ");

        for (k = 0; k < n; k++)

        {

            if (cost[k][near[k]] == INT\_MAX)

                printf(" inf     ");

            else if(near[k]==-1)

                printf(" ---     ");

            else

                printf(" %2d      ", cost[k][near[k]]);

        }

        printf("\n");

        printf("j=%d\n", j + 1);

        if (cost[j][near[j]] != INT\_MAX)

        {

            t[i][0] = j;

            t[i][1] = near[j];

            mincost = mincost + cost[j][near[j]];

        }

        else

        {

            t[i][0] = j;

            t[i][1] = j;

        }

        printf("Table of spanning tree edges:\n");

    for (x = 0; x <= i; x++)

    {

        for (y = 0; y < 2; y++)

        {

            printf("%d ", t[x][y] + 1);

        }

        printf("\n");

    }

    printf("Mincost: %d\n", mincost);

        printf("\n");

        near[j] = -1;

        for (k = 0; k < n; k++)

        {

            if (near[k] != -1 && (cost[k][near[k]] > cost[k][j]))

            {

                near[k] = j;

            }

        }

    }

    return mincost;

}

int main()

{

    int n, maxedge, i, j, src, dst, prc;

    printf("\nEnter the no. of vertices in graph: ");

    scanf("%d", &n);

    maxedge = (n \* (n - 1)) / 2;

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

    for (i = 0; i < n; i++)

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

    int \*\*t = malloc((n - 1) \* sizeof(int \*));

    for (i = 0; i < n - 1; i++)

        t[i] = malloc(2 \* sizeof(int));

    for (i = 0; i < n; i++)

    {

        for (j = 0; j < n; j++)

        {

            cost[i][j] = INT\_MAX;

        }

    }

    printf("(enter -1,-1 to exit)\nEnter the source and destination vertices followed by cost: \n");

    for (i = 0; i < maxedge; i++)

    {

        scanf("%d %d %d", &src, &dst, &prc);

        if (src == -1 && dst == -1)

            break;

        else if (src - 1 < 0 || src - 1 >= n || dst - 1 < 0 || dst - 1 >= n)

        {

            printf("invalid edge\n");

            i--;

        }

        else

        {

            cost[src - 1][dst - 1] = prc;

            cost[dst - 1][src - 1] = prc;

        }

    }

    mini(cost, n, &src, &dst);

    printf("\n\nMinimum edge: %d %d\n", src + 1, dst + 1);

    prc = prim(cost, n, t);

    /\*printf("\nTable of spanning tree edges:\n");

    for (i = 0; i < n - 1; i++)

    {

        for (j = 0; j < 2; j++)

        {

            printf("%d ", t[i][j] + 1);

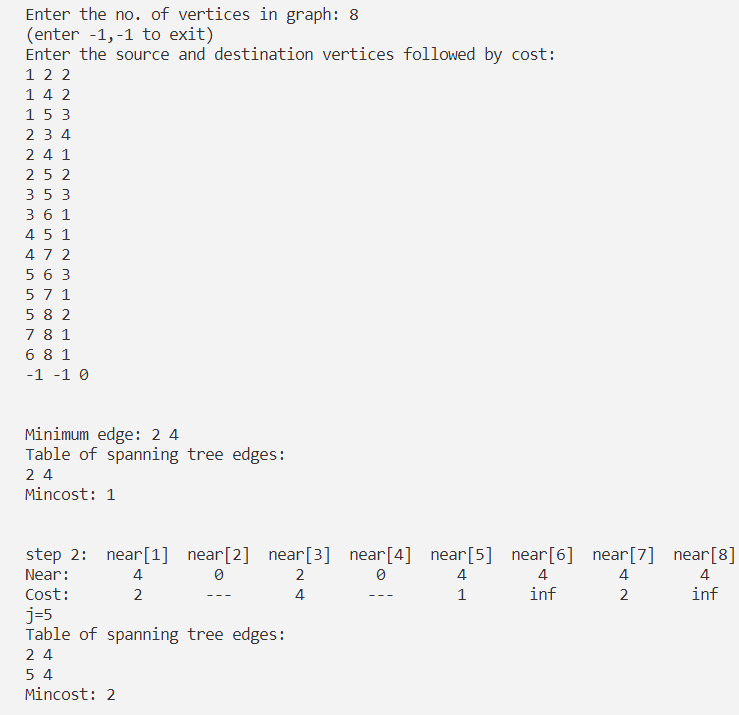
        }

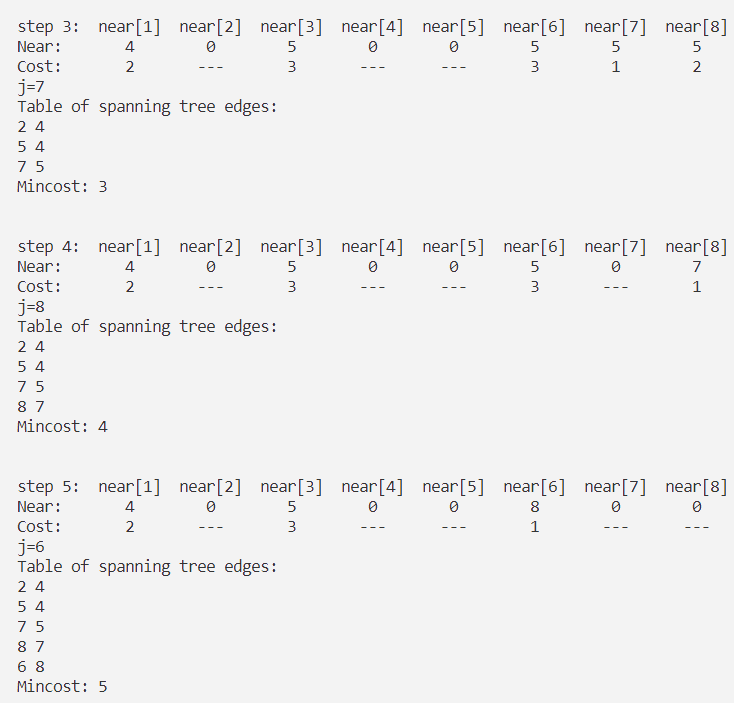
        printf("\n");

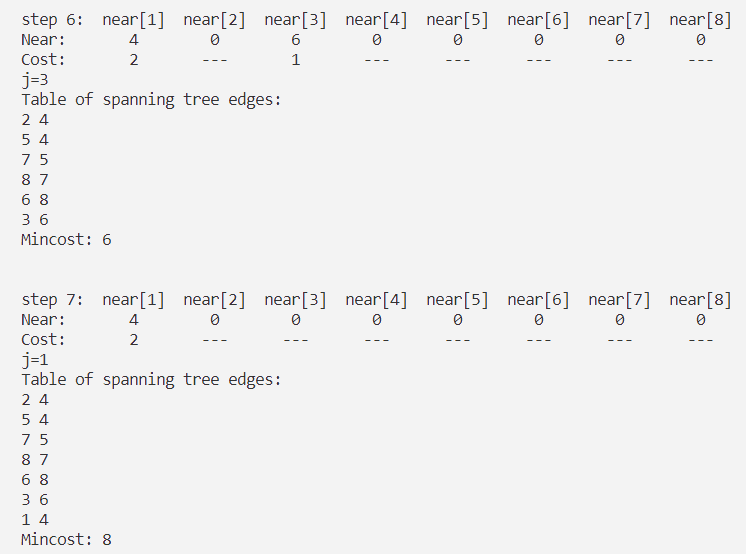
        }

    printf("Mincost: %d\n", prc);\*/

}

**Output**

****

****

1. **Kruskal’s Algorithm**

**Date:**

**Code**

#include<stdio.h>

#include<stdlib.h>

#define TRUE 1

#define FALSE 0

int parent[20];

struct edge{

    int src;

    int dst;

    int cost;

};

void assign(int \*p, int q){

    \*p = q;

}

void adjust(struct edge \*E,int i,int n){

    int j=2\*i, item=E[i].cost, items=E[i].src, itemd=E[i].dst ;

    while(j<=n){

        if(j<n && E[j].cost>E[j+1].cost)

            j=j+1;

        if(item<=E[j].cost)

            break;

        assign(&E[j/2].cost,E[j].cost);

        assign(&E[j/2].src,E[j].src);

        assign(&E[j/2].dst,E[j].dst);

        j = 2\*j;

    }

    assign(&E[j/2].cost,item);

    assign(&E[j/2].src,items);

    assign(&E[j/2].dst,itemd);

    E[j/2].cost=item;

}

void heapify(struct edge \*E,int n){

    for(int i=n/2;i>=1;i--)

        adjust(E,i,n);

}

int delmin(struct edge \*E,int n,struct edge \*x){

    if(n==0){

        printf("minheap is empty\n");

        return FALSE;

    }

    assign(&x->cost,E[1].cost);

    assign(&x->src,E[1].src);

    assign(&x->dst,E[1].dst);

    assign(&E[1].cost, E[n].cost);

    assign(&E[1].src, E[n].src);

    assign(&E[1].dst, E[n].dst);

    adjust(E,1,n-1);

    return TRUE;

}

int find(int i){

    extern int parent[20];

    while(parent[i]>=0)

        i=parent[i];

    return i;

}

void unio(int i, int j){

    parent[i]=j;

}

int kruskal(struct edge \*E, int n, int g, int \*\*t){

    int i,mincost=0,l,j,k,p,q;

    struct edge x;

    heapify(E,n);

    for(i=0;i<=g;i++)

        parent[i]=-1;

    i=0;

    l=n;

     while(i<g-1 && delmin(E,l--,&x)){

        j=find(x.src);

        k=find(x.dst);

        if(j!=k){

            i++;

            t[i][0]=x.src;

            t[i][1]=x.dst;

            mincost = mincost + x.cost;

            unio(j,k);

            printf("\nstep %d(%d,%d) -", i, x.src, x.dst);

            for(int e=1;e<=g;e++)

            printf("  %d", e);

            printf("  j k");

            printf("\nparent arr:  ");

                for(int e=1;e<=g;e++)

                    printf(" %c%d",parent[e]<0?'\0':' ', parent[e]);

            printf("  %d %d",j,k);

            printf("\nTable of spanning tree edges:\n");

    for (p = 1; p <= i; p++)

    {

        for ( q = 0; q < 2; q++)

        {

            printf("%d ", t[p][q]);

        }

        printf("\n");

}

printf("Mincost: %d\n", mincost);

        }

    }

    if(i!=g-1)

        printf("No Spanning tree\n");

    else

        return mincost;

}

int main(){

    struct edge \*E;

    int g,n,i,mincost=0;

    printf("\nEnter the no. of vertices in graph: ");

    scanf("%d", &g);

    printf("\nEnter the no. of edges to be entered: ");

    scanf("%d", &n);

    int \*\*t = malloc(2 \* sizeof(int \*));

    for (i = 0; i <= g; i++)

        t[i] = malloc((g) \* sizeof(int));

    E=(struct edge \*)malloc((n+1)\*sizeof(struct edge));

    E[0].cost = 0;

    E[0].src = 0;

    E[0].dst = 0;

    printf("Enter the source and destination vertices followed by cost: \n");

    for(i=1;i<=n;i++){

        scanf("%d %d %d", &E[i].src, &E[i].dst, &E[i].cost);

    }

    mincost=kruskal(E,n,g,t);

    printf("\n\nTable of spanning tree edges:\n");

    for (i = 1; i <= g-1; i++)

    {

        for (int j = 0; j < 2; j++)

        {

            printf("%d ", t[i][j]);

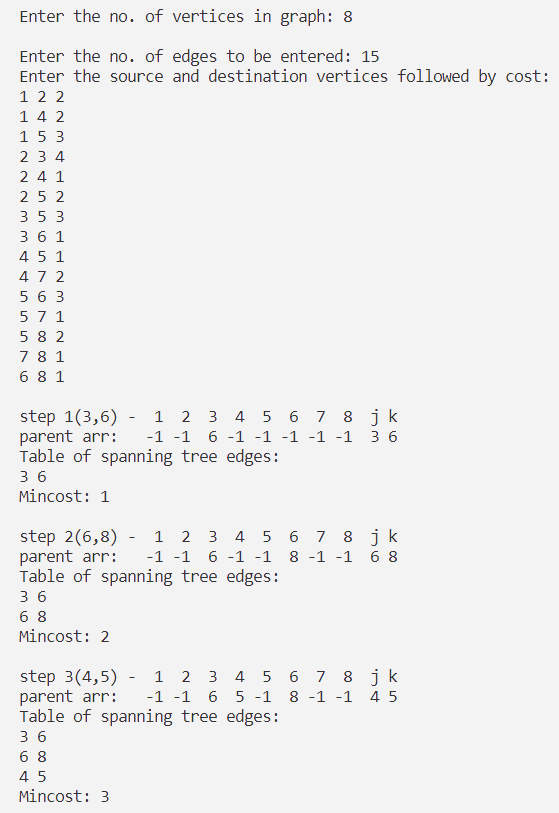
        }

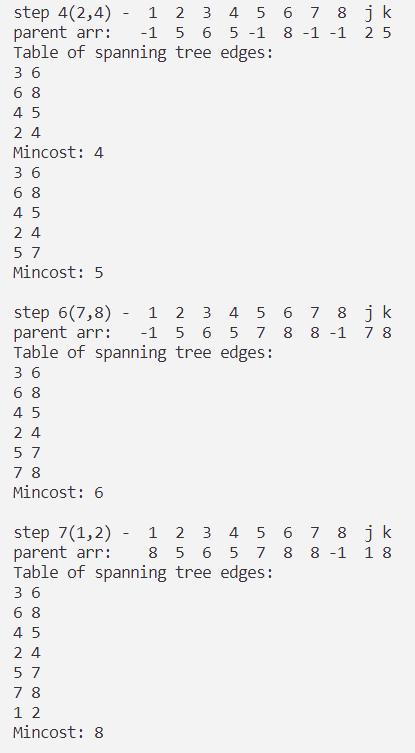
        printf("\n");

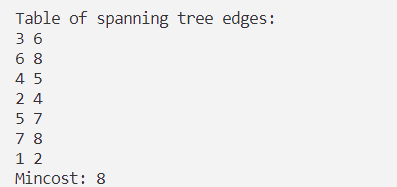
}

printf("Mincost: %d\n", mincost);

}

**Output**

****

****

1. **Single Source Shortest Paths**

**Date:**

**Code**

#include<stdio.h>

#include<stdlib.h>

#include<limits.h>

#include<stdbool.h>

#define MAX 100

int cost[MAX][MAX];

int dist[MAX];

bool s[MAX];

int n;

int edge;

void shortestPath(int v);

void displayDist();

int main(){

int u, v;

int wt;

int i, j;

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

scanf("%d", &n);

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

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

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

}

}

printf("\nEnter the number of Edges:" );

scanf("%d", &edge);

printf("Enter Edges in the format soruce-destination-weight\n");

for( i=1 ; i<=edge ; i++){

printf("\nEnter edge %d : ", i);

scanf("%d", &u);

scanf("%d", &v);

scanf("%d", &wt);

cost[u][v] = wt;

}

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

scanf("%d", &v);

shortestPath(v);

displayDist();

return 0;

}

void shortestPath(int v){

int i, j;

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

dist[i] = cost[v][i];

s[i] = false;

}

s[v] = true;

dist[v] = 0.0;

for( j=2 ; j<=n ; j++){

int u, k, w;

int min = INT\_MAX;

for( k=1 ; k<=n ; k++){

if(s[k]==false && dist[k]<min){

u = k;

min = dist[k];

}

}

s[u] = true;

for( w=1 ; w<=n ; w++){

if(cost[u][w]!=INT\_MAX && s[w]==false){

if(dist[w] > dist[u] + cost[u][w]){

dist[w] = dist[u] + cost[u][w];

}

}

}

}

}

void displayDist(){

int i;

printf("\nShortest distance of each vertex from initial vertex v is: \n");

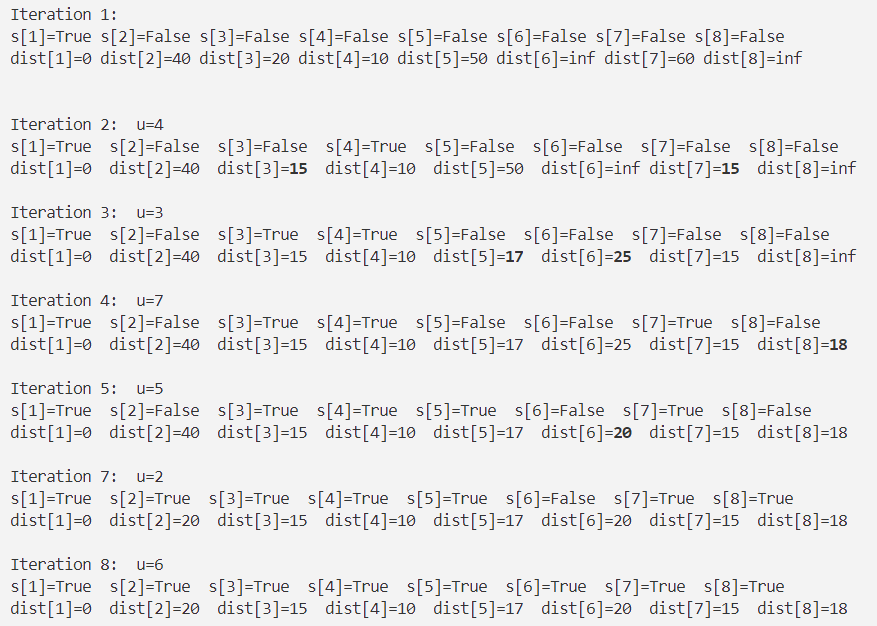
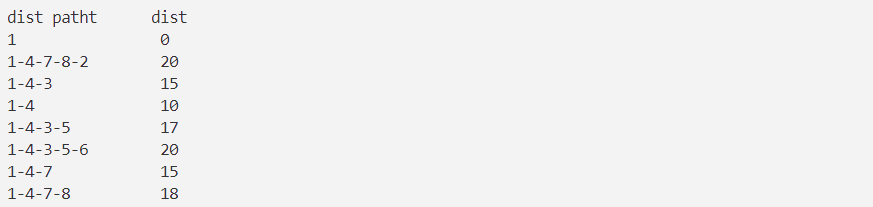
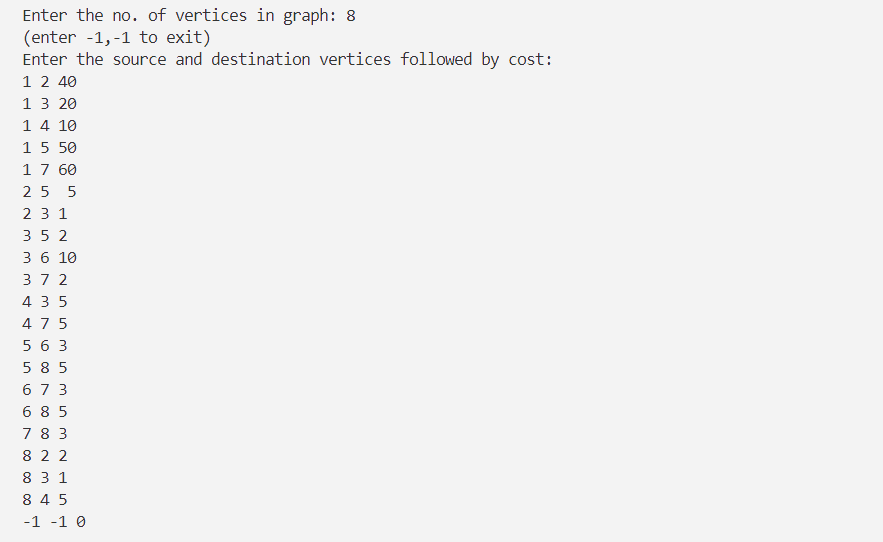
for( i=1 ; i<=n ; i++){

printf("%d - %d", i, dist[i]);

printf("\n");

}

}

**Output**

**CONCLUSION:**

The Greedy method strategy was studied. The programs for (a) Prim’s (b) Kruskal’s (c) Single source shortest paths algorithms were studied and implemented successfully.