The Greedy Method

The **Greedy Method** is an algorithmic paradigm that builds up a solution in **stages**, by considering **one input at a time** and making a series of **locally optimal choices**. At each stage, a **decision** is made about whether to include a given input element into the final solution, based on a specific **optimization criterion**.

In many greedy problems, the goal is to select a **subset** of the input elements that satisfies certain **constraints**. Any subset that satisfies these constraints is referred to as a **feasible solution**. Among all feasible solutions, the one that **maximizes or minimizes** the given **objective function** is termed as the **optimal solution**.

To arrive at an optimal solution, the greedy method employs a **selection strategy** based on an optimization metric, such as cost, weight, or value. This metric typically forms the **basis of comparison** for choosing the next input to consider.

.

Two Paradigms in the Greedy Approach

1. Subset Paradigm

In problems where the aim is to select an optimal subset of elements, the greedy method proceeds by evaluating one element at a time using a selection procedure, and checking whether its inclusion results in a feasible solution. If so, the element is added to the current solution. This continues until all inputs are considered.

Example Problems: 0/1 Knapsack (under certain constraints), Activity Selection Problem.

2. Ordering Paradigm

In this variation, the inputs are first arranged in a specific order, often based on some greedy criterion (e.g., minimum weight or maximum value). The algorithm then processes the elements one by one in that order, and decisions are made using partial solutions constructed so far.

Example Problems: Fractional Knapsack, Prim's Algorithm, Kruskal's Algorithm

```
Greedy (a,n)
//a[1:n] contains the n inputs
{
    solution :=0  //Initialise the solution.
    for i :=1 to n do
    {
        x:=Select(a);
        If Feasible(solution,x) then
            solution:=Union(solution,x);
    }
    return solution;
}
```

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.

EXPERIMENT NO. : 7 **DATE:23/01/2025**

FRACTIONAL KNAPSACK

Aim: To implement fractional knapsack for maximum profit.

Theory: Given the weights and profits of N items, in the form of {profit, weight} put these items in a knapsack of capacity W to get the maximum total profit in the knapsack. In Fractional Knapsack, we can break items for maximizing the total value of the knapsack.

Problem statement:- Given n objects and a knapsack bag. Object i has a weight wi and the knapsack has a capacity m. If a fraction xi, 0 <xi < 1,of object i is placed into the knapsack, then a profit of pixi is earned. The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is m, we require the total weight of all chosen objects to
be at most m.

ALGORITHM:

```
Algorithm GreedyKnapsack(m,n)
 {for i=1 to n
do
x[i]:=0.0;
U:=m;
for i=1 to n do
{
if(w[i] > u) then break;
x[i]:=1.0;
U:=U-w[i];
}
if(i<=n) then x[i]:=u/w[i];
}
}
```

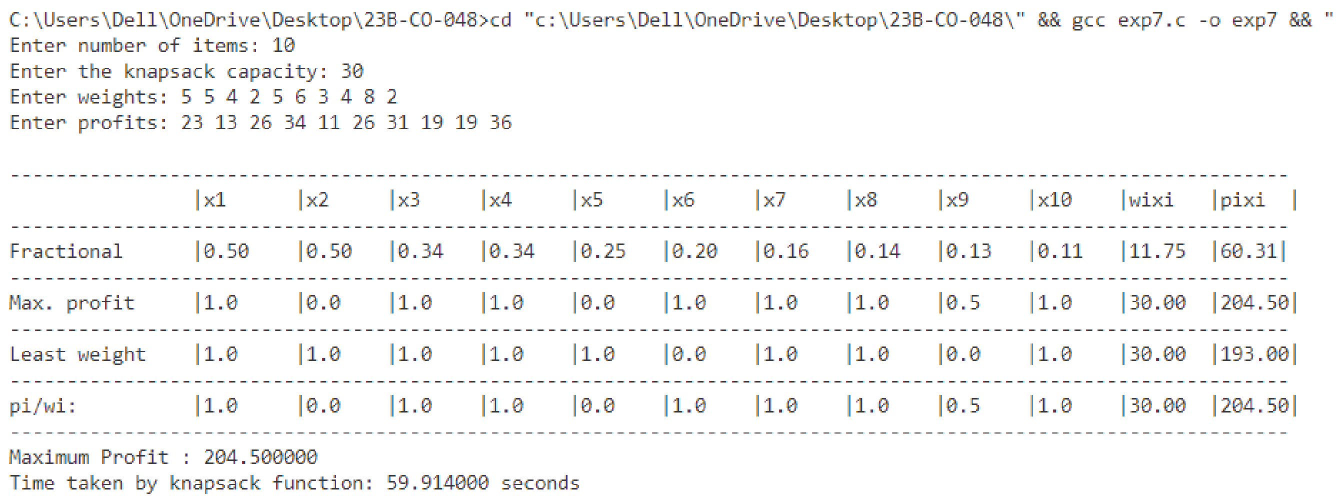
Program :

```
#include <stdio.h>
#include <limits.h>
#include <time.h>
void knapsack(int m,
int n) {
   clock_t start, end;
   double time_taken;
   start = clock();
   float x[n], wixi = 0.0,
pixi = 0.0, u;
   float w[n], p[n],
op[n];
   float f[10] = {0.5,
0.5, 0.34, 0.34, 0.25,
0.2, 0.16, 0.14, 0.125,
0.11};
int max, min, index, i,
j;
 printf("Enter weights:
");
   for (i = 0; i < n; i++)
      scanf("%f",
&w[i]);
 printf("Enter profits:
");
   for (i = 0; i < n; i++)
      scanf("%f", &p[i]);
   for (i = 0; i < n; i++)
{
      wixi += f[i] * w[i];
      pixi += f[i] * p[i];
   }
   printf("\n-------------
");
   for(i=0;i<n+1;i++)
      printf("---------");
   printf("\n");
   printf("           \t");
   for(i=0;i<n;i++)
      printf("|x%d\t
",i+1);
 printf("|wixi\t |pixi
|\n");
   printf("-------------");
   for(i=0;i<n+1;i++)
      printf("---------");
   printf("\n");
printf("Fractional\t|");
   for (i = 0; i < n; i++)
      printf("%.2f\t |",
f[i]);
   printf("%.2f\t
|%.2f|\n", wixi, pixi);
   printf("-------------");
   for(i=0;i<n+1;i++)
      printf("---------");
   printf("\n");
   for (i = 0; i < n; i++)
      x[i] = 0.0;
   u = m;
   for (i = 0; i < n; i++)
{
      max = INT_MIN;
      index = -1;
for (j = 0; j < n; j++) {
       if (p[j] > max &&
x[j] != 1) {
            max = p[j];
            index = j;
         }
      }
 if (index == -1 ||
w[index] > u)
         break;
      x[index] = 1.0;
      u -= w[index];
   }
 if (index != -1 && i <
n)
x[index] = u / w[index];
   printf("Max.
profit\t|");
   for (i = 0; i < n; i++)
  printf("%.1f\t |", x[i]);
   wixi = 0;
   pixi = 0;
   for (i = 0; i < n; i++)
{
      wixi += x[i] * w[i];
      pixi += x[i] * p[i];
   }
printf("%.2f\t |%.2f|\n",
wixi, pixi);
```

```
    printf("-------------");
    for(i=0;i<n+1;i++)
        printf("---------");
    printf("\n");
    for (i = 0; i < n; i++)
        x[i] = 0.0;
    u = m;
    for (i = 0; i < n; i++)
{
        min = INT_MAX;
        index = -1;
 for (j = 0; j < n; j++) {
            if (w[j] < min
&& x[j] != 1) {
                min = w[j];
                index = j;
            }
        }
        if (index == -1 ||
w[index] > u)
            break;
        x[index] = 1.0;
        u -= w[index];
    }
 if (index != -1 && i <
n)
        x[index] = u /
w[index];
 printf("Least
weight\t|");
    for (i = 0; i < n; i++)
  printf("%.1f\t |", x[i]);
    wixi = 0;
    pixi = 0;
    for (i = 0; i < n; i++)
{
        wixi += x[i] * w[i];
        pixi += x[i] * p[i];
    }
printf("%.2f\t |%.2f|\n",
wixi, pixi);
    printf("-------------");
    for(i=0;i<n+1;i++)
        printf("---------");
    printf("\n");
    for (i = 0; i < n; i++)
{
        op[i] = p[i] / w[i];
        x[i] = 0.0;
    }
    u = m;
    for (i = 0; i < n; i++)
{
        max = INT_MIN;
        index = -1;
for (j = 0; j < n; j++) {
        if (op[j] > max &&
x[j] != 1) {
                max = op[j];
                index = j;
            }
        }
 if (index == -1 ||
w[index] > u)
            break;
        x[index] = 1.0;
        u -= w[index];    }
if (index != -1 && i < n)
        x[index] = u /
w[index];
    printf("pi/wi:\t\t|");
    for (i = 0; i < n; i++)
  printf("%.1f\t |", x[i]);
    wixi = 0;
    pixi = 0;
    for (i = 0; i < n; i++)
{
        wixi += x[i] * w[i];
        pixi += x[i] * p[i];
    }
    printf("%.2f\t
|%.2f|\n", wixi, pixi);
    printf("-------------");
    for(i=0;i<n+1;i++)
        printf("---------");
    printf("\n");
    printf("Maximum
Profit : %f\n", pixi);
    end = clock();
    time_taken =
((double)(end - start))
/
CLOCKS_PER_SEC;
    printf("Time taken
by knapsack function:
%f seconds\n",
time_taken);
}
```

```
int main() {
   int m, n;
   printf("Enter
number of items: ");
   scanf("%d", &n);
   printf("Enter the
knapsack capacity: ");
   scanf("%d", &m);
   knapsack(m, n);
   return 0;
}
```

OUTPUT:



```
C:\Users\Dell\OneDrive\Desktop\23B-CO-048>cd "c:\Users\Dell\OneDrive\Desktop\23B-CO-048\" && gcc exp7.c -o exp7 && "
Enter number of items: 10
Enter the knapsack capacity: 30
Enter weights: 5 5 4 2 5 6 3 4 8 2
Enter profits: 23 13 26 34 11 26 31 19 19 36

-------------------------------------------------------------------------------------------------------------------
               |x1      |x2     |x3     |x4     |x5     |x6     |x7     |x8     |x9     |x10    |wixi   |pixi  |
-------------------------------------------------------------------------------------------------------------------
Fractional     |0.50    |0.50   |0.34   |0.34   |0.25   |0.20   |0.16   |0.14   |0.13   |0.11   |11.75  |60.31|
-------------------------------------------------------------------------------------------------------------------
Max. profit    |1.0     |0.0    |1.0    |1.0    |0.0    |1.0    |1.0    |1.0    |0.5    |1.0    |30.00  |204.50|
-------------------------------------------------------------------------------------------------------------------
Least weight   |1.0     |1.0    |1.0    |1.0    |1.0    |0.0    |1.0    |1.0    |0.0    |1.0    |30.00  |193.00|
-------------------------------------------------------------------------------------------------------------------
pi/wi:         |1.0     |0.0    |1.0    |1.0    |0.0    |1.0    |1.0    |1.0    |0.5    |1.0    |30.00  |204.50|
-------------------------------------------------------------------------------------------------------------------
Maximum Profit : 204.500000
Time taken by knapsack function: 59.914000 seconds
```

Conclusion:- The program implements the Fractional Knapsack problem, where it calculates the optimal solution by maximizing profit, minimizing weight, and selecting items based on the profit-to-weight ratio. It demonstrates three strategies: maximizing profit, minimizing weight, and optimizing the profit-to-weight ratio.

EXPERIMENT NO. : 8 **DATE :23/01/2025**

PRIM’S ALGORITHM

Aim: To implement Prim’s Algorithm for minimum spanning tree.

Theory: Prim’s Algorithm is a greedy method to obtain a minimum cost spanning tree which build the tree edge by edge. The next edge to include is chosen according to some optimization criterion.

Problem statement: E is the set of edges in graph G. cost[1:n , 1:n] is the cost adjacency matrix of an n vertex graph such that cost[i,j] is either a positive real number or infinity if no edge (i,j) exists. A minimum spanning tree is computed and stored as a set of edges in the array t. The final cost is returned.

ALGORITHM :

```
AlgorithmPrim(E,cost,n,t)
// E is the set of edges in G,cost[1:n, 1:n] is the cost 3 adjacency matrix of an n vertex graph
such that cost[i,j] is either a positive real number or oo if no edge(i,j) exists.
//A minimum spanning tree is computed and stored as a set of edges in the
//array t[1:n-1,1:2]. (t[i,1],t[i,2]) is an edge in the minimum-cost spanning tree.
//The Final cost is returned.
{
Let (k,l) be an edge of minimum cost in E;
mincost:= cost[k,l];
t[1,1]:=k;  t[1,2]:=l;
for i :=1 to n do // Initialize near.
if (cost[i,l] < cost[i,k])) then near[i]:=l; Else
near[i] :=k;
near[k] := near[l]:=0;
for i :=2 to n-1 do
{   // Find n -2 additional edges for t.
Let j be an index such that near[j] != 0 and cost[j,near[j]] is minimum;
t[i,1]:=j;  t[i,2]:=near[j];
mincost:=mincost+cost[j,near[j]];
near[j]:=0;
for k :=1 to n do  // Updatenear[].
 if ((near[k] !=0) and (cost[k,near[k]]>cost[k,j])) then
near[k]:=j;
}
return mincost;
}
```

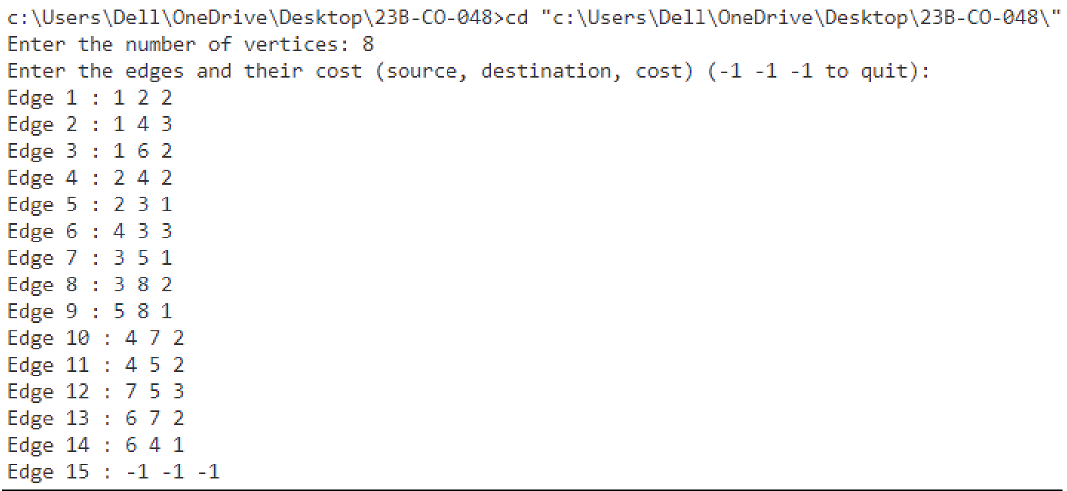
Program:

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

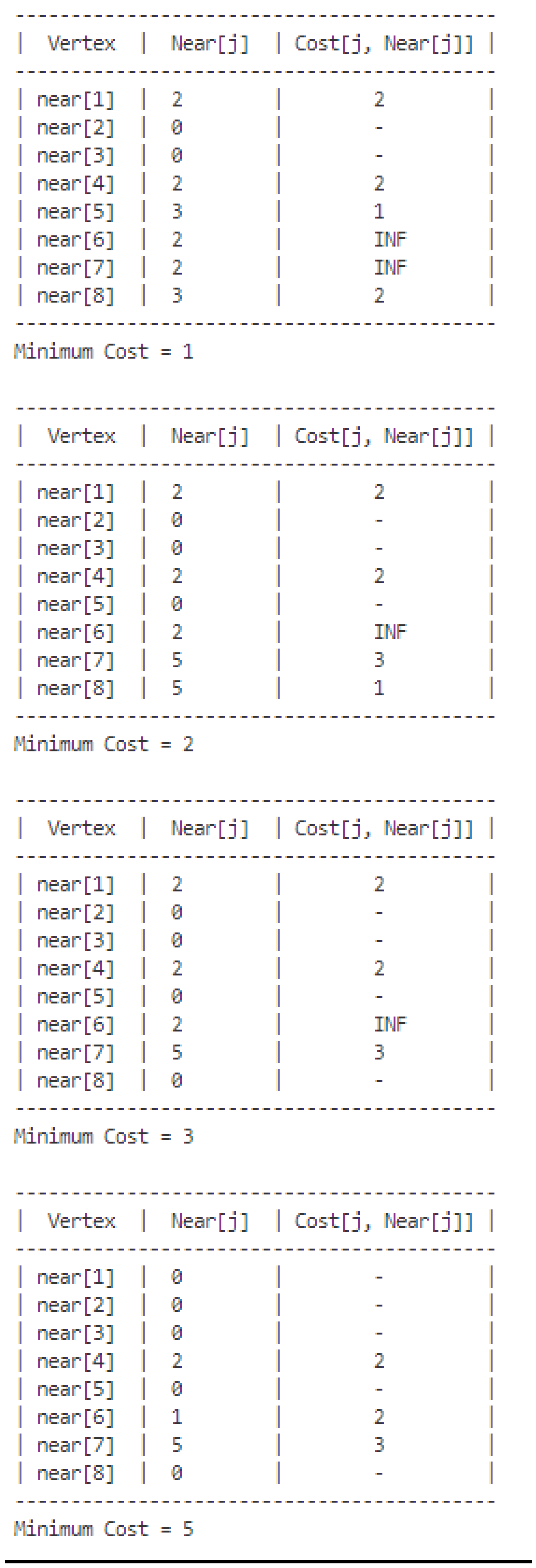
```
    #include <sys/time.h>
    #define MAX 30
    struct Edges {
       int ori, dest;  };
    int max_edges, n, count
    = 0;
    int t[MAX][2],
    near[MAX],
    cost[MAX][MAX];
    struct Edges E[MAX];
    int prims(struct Edges
    E[MAX], int
    cost[MAX][MAX], int n,
    int t[MAX][2]) {
       int i, j, k, l, m, index,
    o, d;
       int mincost =
    INT_MAX;
       if (count == 0) {
    printf("No valid edges
    entered.\n");
          return -1;   }
     for (i = 1; i <= count;
    i++) {
          o = E[i].ori;
          d = E[i].dest;
          if (cost[o][d] <
    mincost) {
             mincost =
    cost[o][d];
             k = o;
             l = d;  }
       }
       t[1][0] = k;
       t[1][1] = l;
       for (i = 1; i <= n; i++)
    near[i] = (cost[i][l] <
    cost[i][k]) ? l : k;
       near[k] = near[l] = 0;
    printf("\n---------------------
    ----------------------\n");
    printf("|  Vertex  |
    Near[j]  | Cost[j, Near[j]]
    |\n");
    printf("-----------------------
    -------------------\n");
       for (i = 1; i <= n; i++) {
          if (near[i] == 0)
    printf("| near[%d]  |  %-
    7d  |        -        |\n", i,
    near[i]);
else if (cost[i][near[i]] ==
INT_MAX)
printf("| near[%d]  |  %-7d  |
INF       |\n", i, near[i]);
else
printf("| near[%d]  |  %-7d  |
%-4d      |\n", i, near[i],
cost[i][near[i]]);
      }
printf("--------------------------------
-----------\n");
printf("Minimum Cost = %d\n",
mincost);
for (i = 2; i <= n - 1; i++) {
         m = INT_MAX;
         for (j = 1; j <= n; j++) {
if (near[j] != 0 &&
cost[j][near[j]] < m) {
                  index = j;
            m = cost[j][near[j]];   }
         }
         t[i][0] = index;
         t[i][1] = near[index];
         mincost += m;
         near[index] = 0;
         for (k = 1; k <= n; k++) {
if (near[k] != 0 &&
cost[k][near[k]] >
cost[k][index])
            near[k] = index;   }
printf("\n----------------------------
--------------\n");
printf("|  Vertex  |  Near[j]  |
Cost[j, Near[j]] |\n");
printf("--------------------------------
-----------\n");
  for (j = 1; j <= n; j++) {
           if (near[j] == 0)
printf("| near[%d]  |  %-7d  |
-         |\n", j, near[j]);
else if (cost[j][near[j]] ==
INT_MAX)
printf("| near[%d]  |  %-7d  |
INF       |\n", j, near[j]);
else
printf("| near[%d]  |  %-7d  |
%-4d      |\n", j, near[j],
cost[j][near[j]]);     }
printf("--------------------------------
-----------\n");
printf("Minimum Cost = %d\n",
mincost);    }
      return mincost;
  }
  void create_graph() {
      int origin, destin, weight;
printf("Enter the number of
vertices: ");
scanf("%d", &n);
max_edges = (n * (n - 1)) /
2;
for (int i = 0; i <= n; i++)
for (int j = 0; j <= n; j++)
cost[i][j] = INT_MAX;
printf("Enter the edges and
their cost (source,
destination, cost) (-1 -1 -1
to quit):\n");
for (int i = 1; i <=
max_edges; i++) {
printf("Edge %d : ", i);
scanf("%d%d%d", &origin,
&destin, &weight);
if (origin == -1 && destin
== -1 && weight == -1)
 break;
if (origin < 1 || destin < 1 ||
origin > n || destin > n) {
printf("Invalid vertex! Try
again.\n");
        i--;
        continue;
        }
cost[origin][destin] =
weight;
cost[destin][origin] =
weight;
        count++;
        E[count].ori = origin;
        E[count].dest =
  destin; }
  }
```

```
int main() {
   int m;
   struct timeval start, end;
   long s, us;
   double t_ms;
   create_graph();
gettimeofday(&start, NULL);
   m = prims(E, cost, n, t);
gettimeofday(&end, NULL);
   s = end.tv_sec -
start.tv_sec;
us = end.tv_usec -
start.tv_usec;
t_ms = (s * 1000000 + us) /
1000.0;
if (m != -1) {
printf("\nFinal Minimum
Spanning Tree Edges:\n");
printf("----------------------------
\n");
printf("|  T  | Vertex 1 | Vertex
2 |\n");
printf("----------------------------
\n");
for (int i = 1; i < n; i++)
printf("| %2d  |    %-4d  |    %-
4d  |\n",i, t[i][0], t[i][1]);
printf("----------------------------
\n");
printf("\nExecution time of
Prim's function: %f ms\n",
t_ms);
     }
     return 0;
}
```

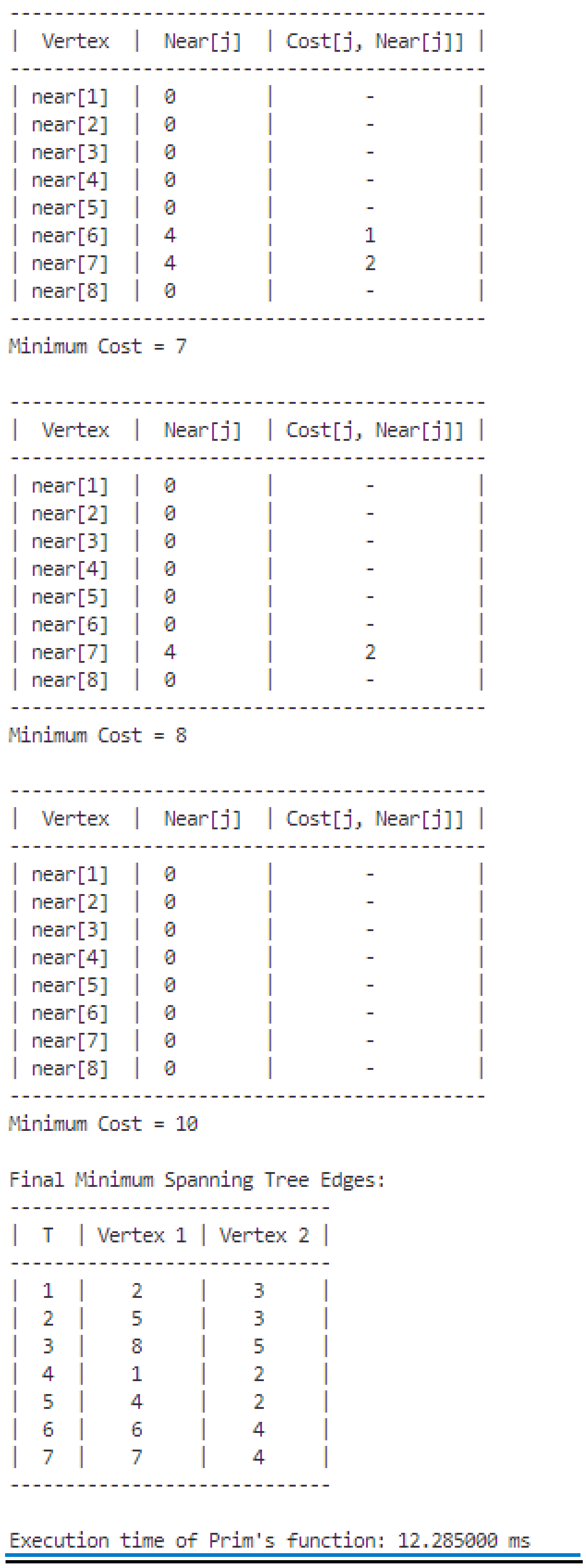
Output:



```
c:\Users\Dell\OneDrive\Desktop\23B-CO-048>cd "c:\Users\Dell\OneDrive\Desktop\23B-CO-048\"
Enter the number of vertices: 8
Enter the edges and their cost (source, destination, cost) (-1 -1 -1 to quit):
Edge 1 : 1 2 2
Edge 2 : 1 4 3
Edge 3 : 1 6 2
Edge 4 : 2 4 2
Edge 5 : 2 3 1
Edge 6 : 4 3 3
Edge 7 : 3 5 1
Edge 8 : 3 8 2
Edge 9 : 5 8 1
Edge 10 : 4 7 2
Edge 11 : 4 5 2
Edge 12 : 7 5 3
Edge 13 : 6 7 2
Edge 14 : 6 4 1
Edge 15 : -1 -1 -1
```



```
-------------------------------------------
|  Vertex  |  Near[j]  | Cost[j, Near[j]] |
-------------------------------------------
| near[1]  |  2        |        2         |
| near[2]  |  0        |        -         |
| near[3]  |  0        |        -         |
| near[4]  |  2        |        2         |
| near[5]  |  3        |        1         |
| near[6]  |  2        |        INF       |
| near[7]  |  2        |        INF       |
| near[8]  |  3        |        2         |
-------------------------------------------
Minimum Cost = 1

-------------------------------------------
|  Vertex  |  Near[j]  | Cost[j, Near[j]] |
-------------------------------------------
| near[1]  |  2        |        2         |
| near[2]  |  0        |        -         |
| near[3]  |  0        |        -         |
| near[4]  |  2        |        2         |
| near[5]  |  0        |        -         |
| near[6]  |  2        |        INF       |
| near[7]  |  5        |        3         |
| near[8]  |  5        |        1         |
-------------------------------------------
Minimum Cost = 2

-------------------------------------------
|  Vertex  |  Near[j]  | Cost[j, Near[j]] |
-------------------------------------------
| near[1]  |  2        |        2         |
| near[2]  |  0        |        -         |
| near[3]  |  0        |        -         |
| near[4]  |  2        |        2         |
| near[5]  |  0        |        -         |
| near[6]  |  2        |        INF       |
| near[7]  |  5        |        3         |
| near[8]  |  0        |        -         |
-------------------------------------------
Minimum Cost = 3

-------------------------------------------
|  Vertex  |  Near[j]  | Cost[j, Near[j]] |
-------------------------------------------
| near[1]  |  0        |        -         |
| near[2]  |  0        |        -         |
| near[3]  |  0        |        -         |
| near[4]  |  2        |        2         |
| near[5]  |  0        |        -         |
| near[6]  |  1        |        2         |
| near[7]  |  5        |        3         |
| near[8]  |  0        |        -         |
-------------------------------------------
Minimum Cost = 5
```



```
-------------------------------------------
|  Vertex  |  Near[j]  | Cost[j, Near[j]] |
-------------------------------------------
| near[1]  |  0        |        -         |
| near[2]  |  0        |        -         |
| near[3]  |  0        |        -         |
| near[4]  |  0        |        -         |
| near[5]  |  0        |        -         |
| near[6]  |  4        |        1         |
| near[7]  |  4        |        2         |
| near[8]  |  0        |        -         |
-------------------------------------------
Minimum Cost = 7

-------------------------------------------
|  Vertex  |  Near[j]  | Cost[j, Near[j]] |
-------------------------------------------
| near[1]  |  0        |        -         |
| near[2]  |  0        |        -         |
| near[3]  |  0        |        -         |
| near[4]  |  0        |        -         |
| near[5]  |  0        |        -         |
| near[6]  |  0        |        -         |
| near[7]  |  4        |        2         |
| near[8]  |  0        |        -         |
-------------------------------------------
Minimum Cost = 8

-------------------------------------------
|  Vertex  |  Near[j]  | Cost[j, Near[j]] |
-------------------------------------------
| near[1]  |  0        |        -         |
| near[2]  |  0        |        -         |
| near[3]  |  0        |        -         |
| near[4]  |  0        |        -         |
| near[5]  |  0        |        -         |
| near[6]  |  0        |        -         |
| near[7]  |  0        |        -         |
| near[8]  |  0        |        -         |
-------------------------------------------
Minimum Cost = 10

Final Minimum Spanning Tree Edges:
-----------------------------
|  T  | Vertex 1 | Vertex 2 |
-----------------------------
|  1  |    2     |    3     |
|  2  |    5     |    3     |
|  3  |    8     |    5     |
|  4  |    1     |    2     |
|  5  |    4     |    2     |
|  6  |    6     |    4     |
|  7  |    7     |    4     |
-----------------------------

Execution time of Prim's function: 12.285000 ms
```

Conclusion: The program implements Prim's Algorithm to find the Minimum Spanning Tree (MST) of a graph. It takes as input the number of vertices and the edges of the graph, including the cost of each edge. The algorithm progressively selects the minimum cost edges to connect all the vertices, ensuring no cycles are formed, and computes the MST.

EXPERIMENT NO. : 9 **DATE :30/01/2025**

KRUSKAL'S ALGORITHM

Aim: To implement Kruskal's Algorithm for minimum spanning tree.

Theory: In kruskal's algorithm edges of the graph are considered in nondecreasing order of cost. This interpretation is that the set t of edges so far selected for the spanning tree be such that it is possible to complete t into a tree. Thus t may not be a tree at all stages in the algorithm. In fact, it will generally only be a forest since the set of edges t can be completed into a tree iff there are no cycles in t. This interpretation of the greedy method also results in a minimum-cost spanning tree.

Problem statement:E is the set of edges in G. G has n vertices.cost[u, v] is the cost of edge(u,v). t is the set of edges in the minimum-cost spanning tree.The final cost is returned.

ALGORITHM:

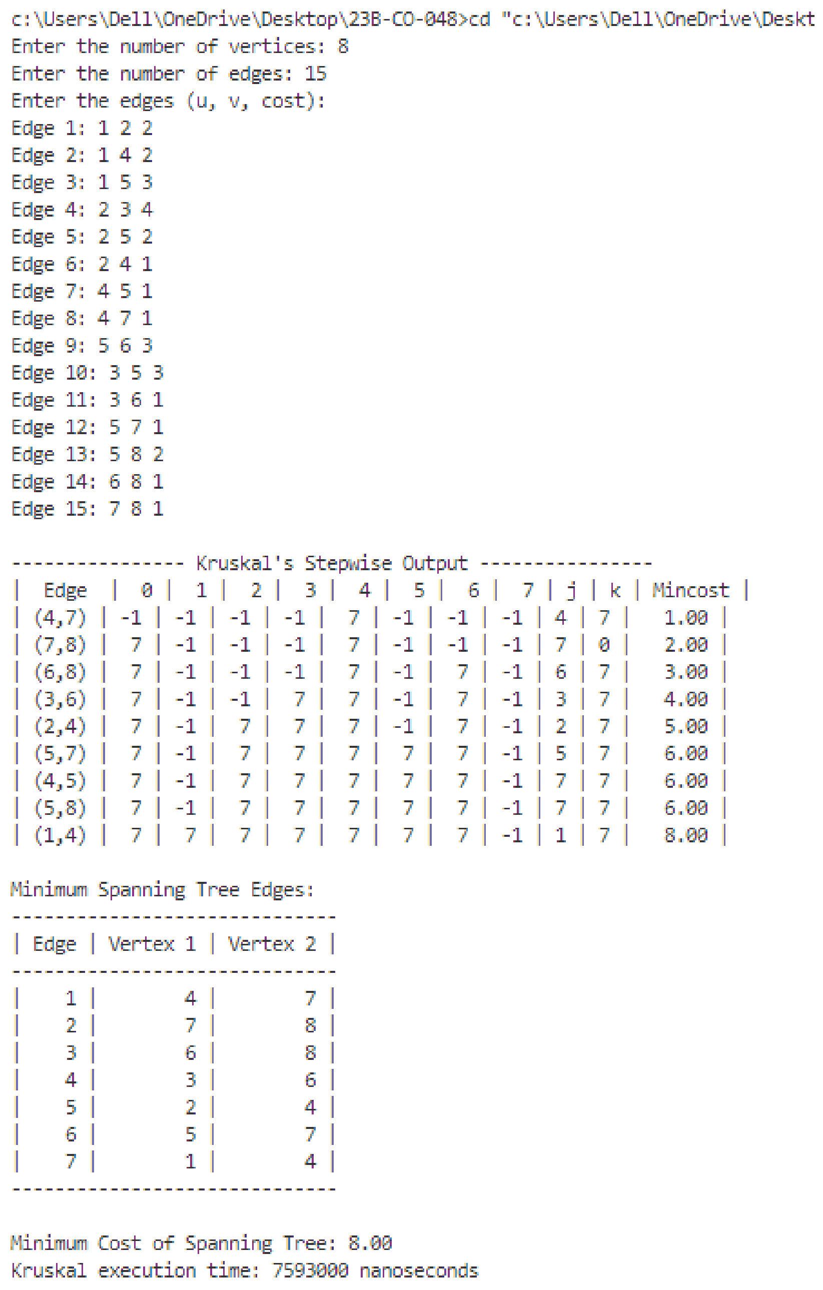
```
Algorithm Kruskal(E,cost, nt)
// E is the set of edges in G. G has n vertices.cost[u, v] is the cost of edge(u,v). t is the set of
edges in
//the minimum-cost spanning tree.The final cost is returned.
{
Construct A heap out of the edge costs using Heapify;
for i :=1 to n do parent[i]:= - 1;
i :=0;              mincost:=0.0;
while ((i <n -1) and(heap not empty)) do
{
Delete a minimum cost edge(u,v) from the heap and re heapify
usingAdjust; j :=Find(u); k :=Find(v);
if (j != k) then
{
i:=i+1; t[i,1]:=u;  t[i,2]:=v  mincost:=mincost+ cost[u, v]; Union(j,k);
}
}if (i!=n)then write ("No Spanning Tree"); else return mincost;
}

Algorithm Union(i,j)
{
parent[i]=j;
}
```

```
Algorithm Find(i)
{ while(parent[i]>=0) do
        i=parent[i];
      return i;
         while(j<=n) do{
          if((j<n) and (a[j] > a[j+1])
                 j=j+1;
        if (item <=a[j]) then
                 break;
        a[j/2]=a[j];
         j=2*j;
      }
      a[j/2]=item;
      }

Algorithm heapify(a,n)
{ for(i=n/2; i>=1; i–)
       Adjust(a,i,n);
      }

Algorithm Adjust(a,i,n)
{      j=2*i;
      item=a[i];
      }

Algorithm Delmin(a,n,x)
{ if(n=0) then {
     write("Heap Empty");
     return;
         }
       x=a[1];
       a[1]=a[n];
       Adjust(a,1,n-1);
       return true;
      }
```

PROGRAM :

```
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>
#include <sys/time.h>  // For
gettimeofday()

#define MAX 100

int minDistance(int dist[], bool s[], int n) {
    int min = INT_MAX, min_index;
    for (int v = 1; v <= n; v++)
        if (!s[v] && dist[v] <= min)
            min = dist[v], min_index = v;
    return min_index;
}

void single_shortest_path(int
graph[MAX][MAX], int v, int n) {
    int dist[MAX], parent[MAX];
    bool s[MAX];
    for (int i = 1; i <= n; i++) {
        dist[i] = INT_MAX;
        s[i] = false;
        parent[i] = -1;
    }
    dist[v] = 0;

    printf("\nIterations:\n");
    for (int count = 1; count < n; count++) {
        int u = minDistance(dist, s, n);
        s[u] = true;

        printf("Iteration %d: u = %d\n", count,
u);
        printf("S[");
        bool first = true;
        for (int i = 1; i <= n; i++) {
            if (s[i]) {
                if (!first) printf("=S[%d]", i);
                else { printf("%d]", i); first = false;
}
            }
        }
        printf(" = true\n");
        printf("S[");
        first = true;
        for (int i = 1; i <= n; i++) {
            if (!s[i]) {
                if (!first) printf("=S[%d]", i);
                else { printf("%d]", i); first = false;
}
            }
        }
        printf(" = false\n");

        for (int w = 1; w <= n; w++) {
            if (!s[w] && graph[u][w] && dist[u] !=
INT_MAX && dist[u] + graph[u][w] <
dist[w]) {
                dist[w] = dist[u] + graph[u][w];
                parent[w] = u;
            }
        }

        for (int i = 1; i <= n; i++) {
            printf("dist[%d] = ", i);
            if (dist[i] == INT_MAX)
                printf("INF\n");
            else
                printf("%d\n", dist[i]);
        }
        printf("\n");
    }

    printf("\nFinal Shortest Paths:\n");
    for (int i = 1; i <= n; i++) {
        printf("source:1, destination:%d, ", i);
        if (dist[i] == INT_MAX) {
            printf("No Path\n");
            continue;
        }
        printf("Length = %d, Path = ", dist[i]);
        int path[MAX], index = 0, temp = i;
        while (temp != -1) {
            path[index++] = temp;
            temp = parent[temp];
        }
        for (int j = index - 1; j >= 0; j--)
            printf("%d%s", path[j], (j > 0) ? " ->
" : "");
        printf("\n");
    }
}

int main() {
    int n, e, u, v, w, graph[MAX][MAX] = {0};
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter number of edges: ");
    scanf("%d", &e);
    printf("Enter edges (from to weight):\n");
    for (int i = 0; i < e; i++) {
        scanf("%d %d %d", &u, &v, &w);
        graph[u][v] = w;
    }
    int source;
    printf("Enter source vertex: ");
    scanf("%d", &source);

    // Time measurement
    struct timeval start, end;
    gettimeofday(&start, NULL);

    single_shortest_path(graph, source, n);

    gettimeofday(&end, NULL);
    long seconds = end.tv_sec -
start.tv_sec;
    long microseconds = end.tv_usec -
start.tv_usec;
    double elapsed = seconds +
microseconds * 1e-6;

    printf("\nExecution time: %.6f
seconds\n", elapsed);

    return 0;
}
```

OUTPUT :



```
c:\Users\Dell\OneDrive\Desktop\23B-CO-048>cd "c:\Users\Dell\OneDrive\Deskt
Enter the number of vertices: 8
Enter the number of edges: 15
Enter the edges (u, v, cost):
Edge 1: 1 2 2
Edge 2: 1 4 2
Edge 3: 1 5 3
Edge 4: 2 3 4
Edge 5: 2 5 2
Edge 6: 2 4 1
Edge 7: 4 5 1
Edge 8: 4 7 1
Edge 9: 5 6 3
Edge 10: 3 5 3
Edge 11: 3 6 1
Edge 12: 5 7 1
Edge 13: 5 8 2
Edge 14: 6 8 1
Edge 15: 7 8 1

---------------- Kruskal's Stepwise Output ----------------
|  Edge  |  0 |  1 |  2 |  3 |  4 |  5 |  6 |  7 | j | k | Mincost |
| (4,7) | -1 | -1 | -1 | -1 |  7 | -1 | -1 | -1 | 4 | 7 |    1.00 |
| (7,8) |  7 | -1 | -1 | -1 |  7 | -1 | -1 | -1 | 7 | 0 |    2.00 |
| (6,8) |  7 | -1 | -1 | -1 |  7 | -1 |  7 | -1 | 6 | 7 |    3.00 |
| (3,6) |  7 | -1 | -1 |  7 |  7 | -1 |  7 | -1 | 3 | 7 |    4.00 |
| (2,4) |  7 | -1 |  7 |  7 |  7 | -1 |  7 | -1 | 2 | 7 |    5.00 |
| (5,7) |  7 | -1 |  7 |  7 |  7 |  7 |  7 | -1 | 5 | 7 |    6.00 |
| (4,5) |  7 | -1 |  7 |  7 |  7 |  7 |  7 | -1 | 7 | 7 |    6.00 |
| (5,8) |  7 | -1 |  7 |  7 |  7 |  7 |  7 | -1 | 7 | 7 |    6.00 |
| (1,4) |  7 |  7 |  7 |  7 |  7 |  7 |  7 | -1 | 1 | 7 |    8.00 |

Minimum Spanning Tree Edges:
------------------------------
| Edge | Vertex 1 | Vertex 2 |
------------------------------
|    1 |        4 |        7 |
|    2 |        7 |        8 |
|    3 |        6 |        8 |
|    4 |        3 |        6 |
|    5 |        2 |        4 |
|    6 |        5 |        7 |
|    7 |        1 |        4 |
------------------------------

Minimum Cost of Spanning Tree: 8.00
Kruskal execution time: 7593000 nanoseconds
```

Conclusion: The provided code implements Kruskal’s algorithm using a min-heap to find the Minimum Spanning Tree (MST) of a weighted, undirected graph. The graph is represented by an edge list, and the algorithm processes these edges to construct the MST while keeping track of the minimum cost at each step.

EXPERIMENT NO. : 10 **DATE :30/01/2025**

SINGLE SOURCE SHORTEST PATH

Aim: To implement a single source shortest path to find the minimum distance.

Theory: The starting vertex of the path is referred to as the source,and the last vertex the destination. The graphs are digraphs.

Problem statement: Given a directed graph G = (V,E),a weighting function cost for the edges of G, and a source vertex q. The problem is to determine the shortest paths from v0 to all the remaining vertices of G. It is assumed that all the weights are positive. The shortest path between vo and some other node v is an ordering among subset of the edges.

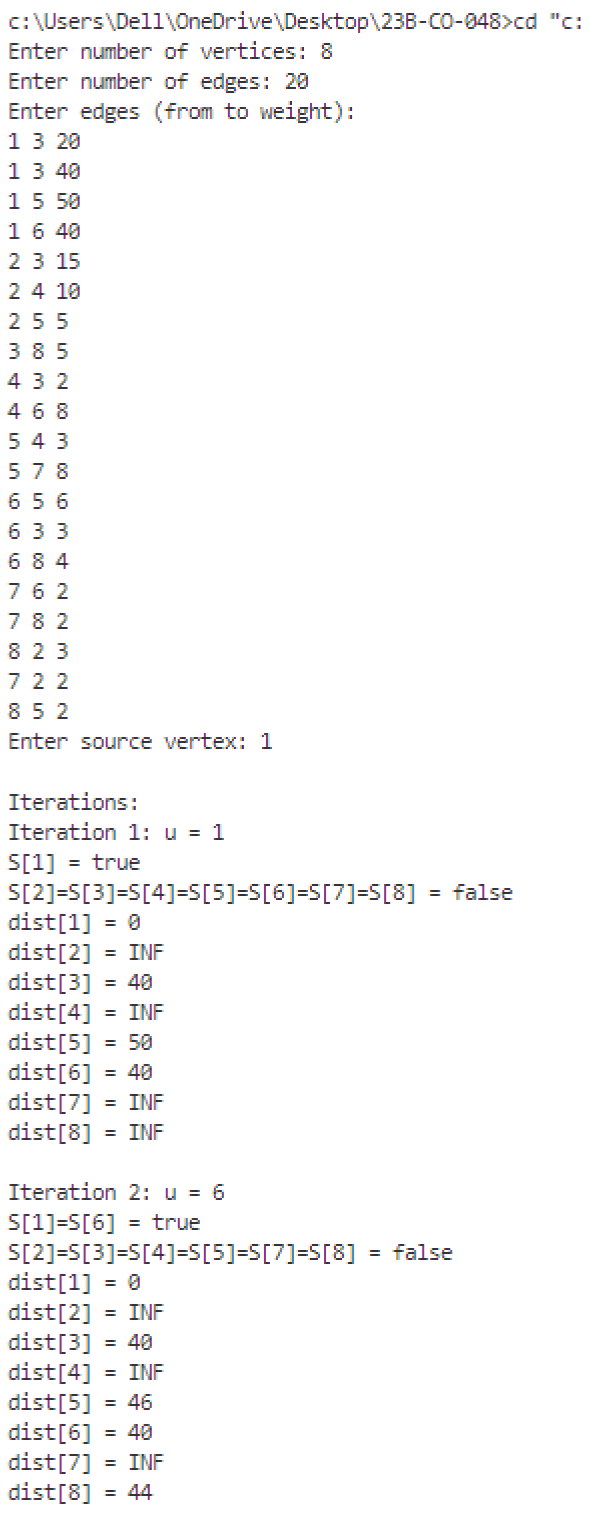
ALGORITHM:

```
Algorithm ShortestPaths(v,cost,dist,n)
// dist[j], 1,<j <n, is set to the length of the shortest path from vertex v to vertex j in a digraph G
with n
// vertices. dist[v] is set to zero. G is represented by its cost adjacency matrix cost[l :n, 1:n].
{
for i :=1to n do
{// InitializeS.
S[i]:=false;  dist[i] :=cost[v,i];
}
S[v]:=true;  dist[v] :=0.0;  // Put v in S.
for num :=2 to n do
{ // Determinen n-1 paths from v.
 Choose u from among those vertices not in S such that dist[u] is
minimum; S[u]:=true; // Put u in S.
for (each w adjacent to u with S[w]= false)do
// Updatedistances.
 if (dist[w]>dist[u]+cost[u,w])) then dist[w]
:=dist[u]+cost[u,w];
}
}
```

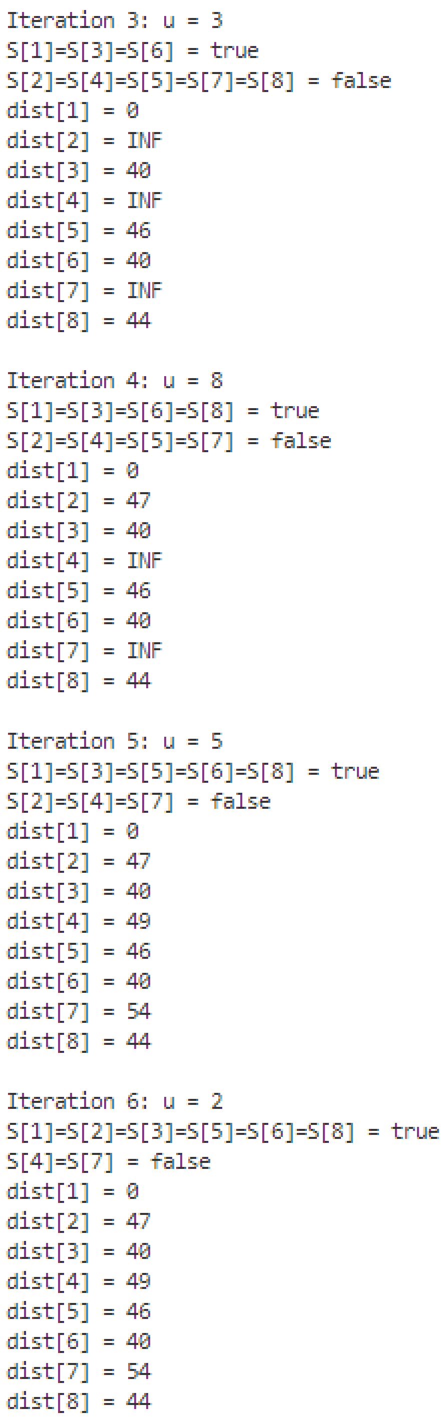
PROGRAM:

```
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>
#define MAX 100
int minDistance(int dist[], bool s[], int n) {
    int min = INT_MAX, min_index;
    for (int v = 1; v <= n; v++)
        if (!s[v] && dist[v] <= min)
            min = dist[v], min_index = v;
    return min_index;
}
void single_shortest_path(int
graph[MAX][MAX], int v, int n) {
    int dist[MAX], parent[MAX];
      bool s[MAX];
      for (int i = 1; i <= n; i++) {
          dist[i] = INT_MAX;
          s[i] = false;
          parent[i] = -1;    }
      dist[v] = 0;
      printf("\nIterations:\n");
      for (int count = 1; count < n; count++) {
          int u = minDistance(dist, s, n);
          s[u] = true;
  printf("Iteration %d: u = %d\n", count, u);
          printf("S[");
          bool first = true;
          for (int i = 1; i <= n; i++) {
              if (s[i]) {
          if (!first) printf("=S[%d]", i);
          else { printf("%d]", i); first = false;  }
               }
          }
          printf(" = true\n");
          printf("S[");
          first = true;
          for (int i = 1; i <= n; i++) {
             if (!s[i]) {
             if (!first) printf("=S[%d]", i);
           else { printf("%d]", i); first = false; }
              }
          }
          printf(" = false\n");
          for (int w = 1; w <= n; w++) {
  if (!s[w] && graph[u][w] && dist[u] !=
  INT_MAX && dist[u] + graph[u][w] <
dist[w])
{
       dist[w] = dist[u] + graph[u][w];
       parent[w] = u;       }
        }
        for (int i = 1; i <= n; i++) {
            printf("dist[%d] = ", i);
            if (dist[i] == INT_MAX)
                printf("INF\n");
            else
                printf("%d\n", dist[i]);   }
    printf("\n");  }
    printf("\nFinal Shortest Paths:\n");
    for (int i = 1; i <= n; i++) {
    printf("source:1, destination:%d, ", i);
        if (dist[i] == INT_MAX) {
            printf("No Path\n");
            continue;      }
printf("Length = %d, Path = ", dist[i]);
        int path[MAX], index = 0, temp = i;
        while (temp != -1) {
            path[index++] = temp;
            temp = parent[temp];       }
 for (int j = index - 1; j >= 0; j--)
 printf("%d%s", path[j], (j > 0) ? " -> " : "");
        printf("\n");    }
}

int main() {
int n, e, u, v, w, graph[MAX][MAX] = {0};
printf("Enter number of vertices: ");
    scanf("%d", &n);
   printf("Enter number of edges: ");
    scanf("%d", &e);
    printf("Enter edges (from to
weight):\n");
    for (int i = 0; i < e; i++) {
        scanf("%d %d %d", &u, &v, &w);
        graph[u][v] = w;
    }
    int source;
    printf("Enter source vertex: ");
    scanf("%d", &source);
 single_shortest_path(graph, source, n);
    return 0;
}
```

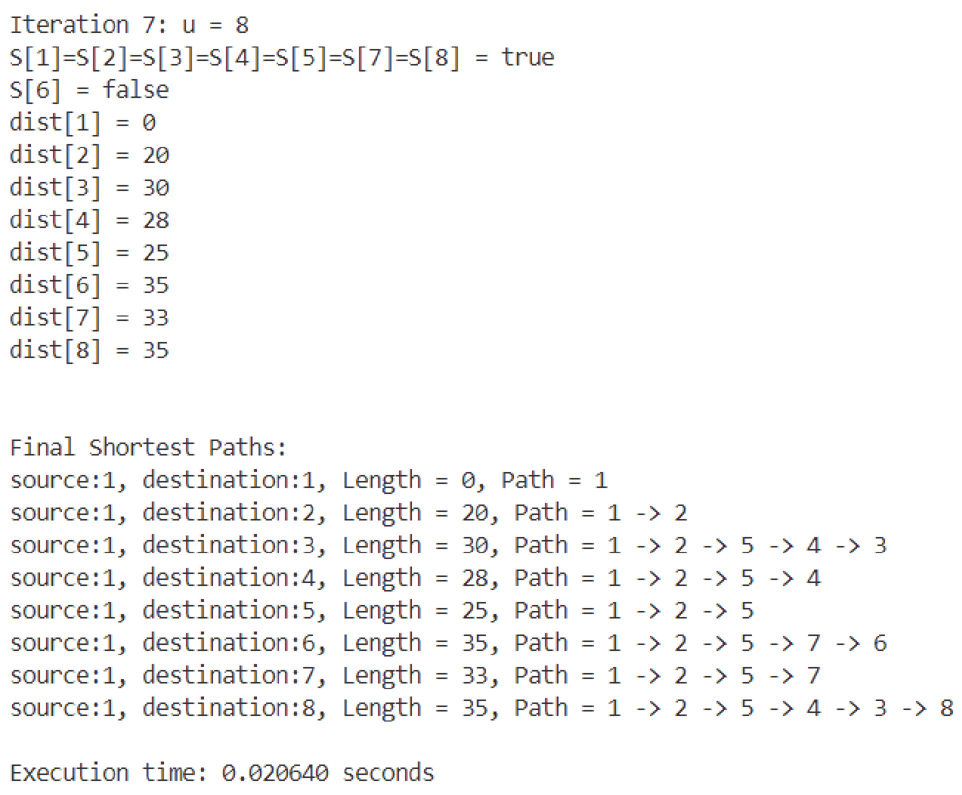
OUTPUT:



```
c:\Users\Dell\OneDrive\Desktop\23B-CO-048>cd "c:
Enter number of vertices: 8
Enter number of edges: 20
Enter edges (from to weight):
1 3 20
1 3 40
1 5 50
1 6 40
2 3 15
2 4 10
2 5 5
3 8 5
4 3 2
4 6 8
5 4 3
5 7 8
6 5 6
6 3 3
6 8 4
7 6 2
7 8 2
8 2 3
7 2 2
8 5 2
Enter source vertex: 1

Iterations:
Iteration 1: u = 1
S[1] = true
S[2]=S[3]=S[4]=S[5]=S[6]=S[7]=S[8] = false
dist[1] = 0
dist[2] = INF
dist[3] = 40
dist[4] = INF
dist[5] = 50
dist[6] = 40
dist[7] = INF
dist[8] = INF

Iteration 2: u = 6
S[1]=S[6] = true
S[2]=S[3]=S[4]=S[5]=S[7]=S[8] = false
dist[1] = 0
dist[2] = INF
dist[3] = 40
dist[4] = INF
dist[5] = 46
dist[6] = 40
dist[7] = INF
dist[8] = 44
```



```
Iteration 3: u = 3
S[1]=S[3]=S[6] = true
S[2]=S[4]=S[5]=S[7]=S[8] = false
dist[1] = 0
dist[2] = INF
dist[3] = 40
dist[4] = INF
dist[5] = 46
dist[6] = 40
dist[7] = INF
dist[8] = 44

Iteration 4: u = 8
S[1]=S[3]=S[6]=S[8] = true
S[2]=S[4]=S[5]=S[7] = false
dist[1] = 0
dist[2] = 47
dist[3] = 40
dist[4] = INF
dist[5] = 46
dist[6] = 40
dist[7] = INF
dist[8] = 44

Iteration 5: u = 5
S[1]=S[3]=S[5]=S[6]=S[8] = true
S[2]=S[4]=S[7] = false
dist[1] = 0
dist[2] = 47
dist[3] = 40
dist[4] = 49
dist[5] = 46
dist[6] = 40
dist[7] = 54
dist[8] = 44

Iteration 6: u = 2
S[1]=S[2]=S[3]=S[5]=S[6]=S[8] = true
S[4]=S[7] = false
dist[1] = 0
dist[2] = 47
dist[3] = 40
dist[4] = 49
dist[5] = 46
dist[6] = 40
dist[7] = 54
dist[8] = 44
```



```
Iteration 7: u = 8
S[1]=S[2]=S[3]=S[4]=S[5]=S[7]=S[8] = true
S[6] = false
dist[1] = 0
dist[2] = 20
dist[3] = 30
dist[4] = 28
dist[5] = 25
dist[6] = 35
dist[7] = 33
dist[8] = 35


Final Shortest Paths:
source:1, destination:1, Length = 0, Path = 1
source:1, destination:2, Length = 20, Path = 1 -> 2
source:1, destination:3, Length = 30, Path = 1 -> 2 -> 5 -> 4 -> 3
source:1, destination:4, Length = 28, Path = 1 -> 2 -> 5 -> 4
source:1, destination:5, Length = 25, Path = 1 -> 2 -> 5
source:1, destination:6, Length = 35, Path = 1 -> 2 -> 5 -> 7 -> 6
source:1, destination:7, Length = 33, Path = 1 -> 2 -> 5 -> 7
source:1, destination:8, Length = 35, Path = 1 -> 2 -> 5 -> 4 -> 3 -> 8

Execution time: 0.020640 seconds
```

Conclusion:- This program implements Dijkstra's Algorithm to find the shortest path from a source vertex to all other vertices in a weighted graph. The graph is represented by an adjacency matrix. The algorithm follows a greedy approach, selecting the nearest unvisited vertex and updating the shortest distances until all vertices have been processed