EXPERIMENT 2

THE GREEDY METHOD

The greedy method is an algorithmic approach that makes a series of choices, each of which looks best at the moment, in order to find an optimal solution. It follows the principle of local optimization, hoping that these local choices lead to a globally optimal solution.

Feasible Solution:

A feasible solution satisfies all the constraints of the problem.It may or may not be the best (optimal).Example: In the Knapsack Problem, a feasible solution is any valid selection of items that do not exceed the weight limit.

Optimal Solution:

An optimal solution is the best feasible solution according to the objective function.It maximizes or minimizes the given criteria (e.g., profit, cost, distance).Example: In the Knapsack Problem, an optimal solution is the selection of items that maximize the total value without exceeding the weight limit.

```
Algorithm Greedy(a, n)
// a[1:n] contains the n inputs.
{
   solution := ∅;  // initialize solution
   for i := 1 to n do
   {
      x := Select(a);
      if Feasible(solution, x) then
         solution := addition(solution, x);  // adding solution to the set
   }
   return solution;
}
```

FRACTIONAL KNAPSACK

DATE: 24-01-2025

AIM : Write a C program to implement Fractional knapsack problem using the greedy approach

PROBLEM STATEMENT

Consider 'n' objects & a knapsack bag of capacity 'm'. Object i has a weight wi. If a fraction xi, $0 \le x_i \le 1$ 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.

THEORY

Knapsack Algorithm uses a Greedy Approach because it involves finding the feasible solutions and the optimal solution to maximise the profit.In the knapsack approach, we are given n objects and a Knapsack (or bag) of capacity m.
Associated with each object i there is a weight w_i and profit p_i.
The objective function is to fill up the bag so as to maximize the profit earned.
The constraint is that the sum of the weights of the chosen objects put into the bag should not exceed the capacity of the bag.
If a fraction x_i ($0 <= x_i <= 1$) of an object i is chosen to include into the bag then a profit of p_ix_i is earned.
For this problem, we need to:

1. Maximize: $\sum_{1<=i<=n} p_ix_i$
2. Subject to the condition: $\sum_{1<=i<=n} w_ix_i <= m$
3. And $0 <= x_i <= 1$ and $1 <= i <= n$

The profits and weights are positive numbers. A feasible solution (or filling) is any set $(x_i,..., x_n)$ satisfying 2 and 3 above. An optimal solution is a feasible solution for which 1 is maximized.
In case the sum of all weights is <= m, then $x_i = 1$, $1 <= i <= n$ is an optimal solution.
All optimal solutions will fill the knapsack exactly.
In Knapsack, optimal solution is obtained when objects are selected in the decreasing order of p_i/w_i.

ALGORITHM

Algorithm GreedyKnapsack(m,n)

//p[1:n] and w[1:n] contain the profits and weights respectively

// of the n objects ordered such that p[i]/w[i] >= p[i+1]/[wi+1]

//m is the knapsack size and x[1 n] is the solution vector. for i:=1 to n do x[i]:= 0.0; // Initialize a

```
{
   for i:=1 to n do x[i] := 0.0;   //Initialize
   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];
}
```

FRACTIONAL KNAPSACK COMPLEXITY ANALYSIS

Recurrence Relation: Since the algorithm sorts items based on value per unit weight and picks items greedily, no recurrence relation is involved.

Time Complexity: O(nlogn) (due to sorting)

Space Complexity: O(1) (only a few extra variables are used)

PROGRAM

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 100
int sort[MAX];
void sortProfit(int n, float profit[]) {
   int i, j;
   for (i = 0; i < n; i++) {
      sort[i] = i; }

   for (i = 0; i < n - 1; i++) {
      for (j = 0; j < n - i - 1; j++) {
         if (profit[sort[j]] < profit[sort[j + 1]]) {
            int temp = sort[j];
            sort[j] = sort[j + 1];
            sort[j + 1] = temp;
 }}}}

void sortWeight(int n, float weight[]) {
   int i, j;
```

```
for (i = 0; i < n; i++) {
   sort[i] = i;
}

for (i = 0; i < n - 1; i++) {
   for (j = 0; j < n - i - 1; j++) {
      if (weight[sort[j]] > weight[sort[j + 1]]) {
         int temp = sort[j];
         sort[j] = sort[j + 1];
         sort[j + 1] = temp;
}}}}

void sortRatio(int n, float profit[], float weight[]) {
   int i, j;
   float pw[MAX];

   for (i = 0; i < n; i++) {
      sort[i] = i;
      pw[i] = profit[i] / weight[i];
   }

   for (i = 0; i < n - 1; i++) {
      for (j = 0; j < n - i - 1; j++) {
         if (pw[sort[j]] < pw[sort[j + 1]]) {
            int temp = sort[j];
            sort[j] = sort[j + 1];
            sort[j + 1] = temp;
}}}}
void maxProfit(int n, int m, float profit[], float weight[]) {
   float objx[MAX] = {0};
   float total = 0.0;
   int max = m, i;

   sortProfit(n, profit);

   for (i = 0; i < n && weight[sort[i]] <= max; i++) {
      total += profit[sort[i]];
      objx[sort[i]] = 1.0;
      max -= weight[sort[i]];
   }

   if (max > 0 && i < n) {
      objx[sort[i]] = (float)max / weight[sort[i]];
      total += profit[sort[i]] * objx[sort[i]];
   }
```

```
  printf("\nMAXIMUM PROFIT STRATEGY\n");
  for (i = 1; i <=n; i++) {
 printf ("\tx%d", i);
}
  printf("\t  WiXi\t\tPiXi\n");
  for (i = 0; i < n; i++) {
     printf("\t%.2f", objx[i]);
  }
  printf("\t  %d\t\t%.2f\n", m, total);
}

void minWeight(int n, int m, float profit[], float weight[]) {
  float objx[MAX] = {0};
  float total = 0.0;
  int max = m, i;

  sortWeight(n, weight);

  for (i = 0; i < n && weight[sort[i]] <= max; i++) {
     total += profit[sort[i]];
     objx[sort[i]] = 1.0;
     max -= weight[sort[i]];
  }

  if (max > 0 && i < n) {
     objx[sort[i]] = (float)max / weight[sort[i]];
     total += profit[sort[i]] * objx[sort[i]];
  }

  printf("\nMINIMUM WEIGHT STRATEGY\n");
     for (i = 1; i <=n; i++) {
 printf ("\tx%d", i);
}
  printf("\t  WiXi\t\tPiXi\n");
  for (i = 0; i < n; i++) {
     printf("\t%.2f", objx[i]);
  }
  printf("\t  %d\t\t%.2f\n", m, total);
}

void maxRatio(int n, int m, float profit[], float weight[]) {
  float objx[MAX] = {0};
  float total = 0.0;
  int max = m, i;

  sortRatio(n, profit, weight);

  for (i = 0; i < n && weight[sort[i]] <= max; i++) {
```

```
    total += profit[sort[i]];
    objx[sort[i]] = 1.0;
    max -= weight[sort[i]];
}

if (max > 0 && i < n) {
    objx[sort[i]] = (float)max / weight[sort[i]];
    total += profit[sort[i]] * objx[sort[i]];
}
printf("\nMAXIMUM PROFIT-TO-WEIGHT RATIO STRATEGY\n");
      for (i = 1; i <=n; i++) {
  printf ("\tx%d", i);
 }
    printf("\t  WiXi\t\tPiXi\n");
    for (i = 0; i < n; i++) {
       printf("\t%.2f", objx[i]);
    }
    printf("\t  %d\t\t%.2f\n", m, total);
}

void solve_multiple_strategies(int n, float weight[], float profit[], int m) {
    maxProfit(n, m, profit, weight);
    minWeight(n, m, profit, weight);
    maxRatio(n, m, profit, weight);
}

int main() {
    int i, n, m;
    printf("Enter number of elements (n): ");
    scanf("%d", &n);

    float profit[n], weight[n];

    printf("Enter weights (w1, w2, ...): ");
    for (i = 0; i < n; i++) {
       scanf("%f", &weight[i]);
    }

    printf("Enter value of m (capacity): ");
    scanf("%d", &m);

    printf("Enter profits (p1, p2, ...): ");
    for (i = 0; i < n; i++) {
       scanf("%f", &profit[i]);
    }

    clock_t start = clock();
```

```
    solve_multiple_strategies(n, weight, profit, m);
    clock_t end = clock();

    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("\nTime taken: %.6f seconds\n", time_taken);
    return 0;
}
```

OUTPUT

```
"C:\Users\vaidn\Or  ×
Enter number of elements (n): 10
Enter weights (w1, w2, ...): 4 3 2 3 3 8 4 5 6 3
Enter value of m (capacity): 30
Enter profits (p1, p2, ...): 34 22 32 12 18 35 32 22 16 24

MAXIMUM PROFIT STRATEGY
        x1      x2      x3      x4      x5      x6      x7      x8      x9      x10         WiXi            PiXi
        1.00    1.00    1.00    0.00    0.33    1.00    1.00    1.00    0.00    1.00        30              207.00

MINIMUM WEIGHT STRATEGY
        x1      x2      x3      x4      x5      x6      x7      x8      x9      x10         WiXi            PiXi
        1.00    1.00    1.00    1.00    1.00    0.00    1.00    1.00    0.50    1.00        30              204.00

MAXIMUM PROFIT-TO-WEIGHT RATIO STRATEGY
        x1      x2      x3      x4      x5      x6      x7      x8      x9      x10         WiXi            PiXi
        1.00    1.00    1.00    0.00    1.00    0.75    1.00    1.00    0.00    1.00        30              210.25

Time taken: 0.017000 seconds

Process returned 0 (0x0)   execution time : 37.600 s
Press any key to continue.
```

CONCLUSION

The Greedy Knapsack approach provides an efficient solution for the Fractional Knapsack Problem, where items can be divided to maximize the total value within a given weight limit. By sorting items based on their value-to-weight ratio and selecting them greedily, this method ensures an optimal solution in O(n log n) time. However, for the 0/1 Knapsack Problem, where items cannot be divided, the greedy approach fails to guarantee an optimal solution in all cases. Instead, dynamic programming or branch and bound techniques are required.

REFERENCES

1."Fundamentals of Computer Algorithms" by Ellis Horowitz, Sartaj Sahni, and Sanguthevar Rajasekaran: Page 197-199

2."Design and Analysis of Algorithms" by R. C. T. Lee, S. S. Tseng, R. C. Chang, and Y. T. Tsai: Knapsack Problem: Discussed on pages 45 and 46

PRIM'S ALGORITHM

DATE: 24-01-2025

AIM: Write a C program to find the Minimum Cost Spanning tree using Prim's algorithm.

PROBLEM STATEMENT

Given a connected, undirected, and weighted graph G=(V,E) find a minimum spanning tree (MST), which is a subset of edges that connects all vertices with the minimum possible total edge weight. The MST should not contain any cycles. Prim's algorithm grows the MST by adding the smallest available edge that connects an included vertex to an excluded vertex.

THEORY

It is a greedy method algorithm which finds a minimum spanning tree for an undirected weighted graph. It starts with an empty spanning tree. Two sets of vertices are maintained. The first set contains the vertices already included in the minimum spanning tree, the other set contains the vertices not yet included. Prim's Algorithm grows a solution from a random vertex by adding the next cheapest vertex to the existing tree. 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 minimum spanning tree.

ALGORITHM

```
Algorithm Prim(E, cost, n, t)
//E is the set of edges in 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[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 //Initialise 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 //Update near[].
      if(near[k]!=0) and (cost[k, near[k]] > cost[k,j])
         then near[k] := j;
}
   return mincost;
}
```

PRIMS ALGORITHM COMPLEXITY ANALYSIS

Recurrence Relation:

Using Priority Queue (Min-Heap) implementation: $T(n)=T(n-1)+O(\log n)$

Time Complexity:

- Using Adjacency Matrix + Simple Min Selection: $O(n^2)$
- Using Adjacency List + Min-Heap (Binary Heap): $O((n+E)\log n)$

Space Complexity: $O(n+E)$ (for adjacency list representation)

PROGRAM

```
#include <stdio.h>
#include <limits.h>
#include <time.h>
#define MAX 10
#define INF INT_MAX
int k = 1, l;
void displayNear(int* near, int n, int cost[MAX][MAX]) {
```

```
printf("\nnear: ");
for (int i = 1; i <= n; i++) {
   printf("[%02d] ", i);
}
printf("\n      ");
for (int i = 1; i <= n; i++) {
   printf("%2d   ", near[i]);
}
printf("\ncost:  ");
for (int i = 1; i <= n; i++) {
      if (cost[i][near[i]] == INF) {
         printf("inf  ");
         continue;
      }
      if (near[i] == 0) {
         printf("--   ");
         continue;
      }
      printf("%2d   ", cost[i][near[i]]);
   }
   printf("\n");
}

void showmst(int n, int t[][2]) {
   printf("\nThe edges of the Minimum Spanning Tree (MST) are:\n");
   for (int i = 0; i < n - 1; i++) {
      printf("%d - %d\n", t[i][0], t[i][1]);
   }
}

void showcost(int cost[MAX][MAX], int n) {
   printf("\nThe cost adjacency matrix is:\n");
   for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= n; j++) {
         if (cost[i][j] == INF) {
            printf("0 ");
         } else {
            printf("%d ", cost[i][j]);
         }
      }
      printf("\n");
   }
}

int prims(int t[][2], int size, int cost[MAX][MAX], int near[MAX]) {
   int mincost = cost[k][l];
   t[0][0] = k;
```

```
t[0][1] = l;

for (int i = 1; i <= size; i++) {
   if (cost[i][l] < cost[i][k]) {
      near[i] = l;
   } else {
      near[i] = k;
   }
}
near[k] = 0;
near[l] = 0;

displayNear(near, size, cost);

for (int j = 1; j < size - 1; j++) {
   int min = INF;
   int nextVertex = -1;
   for (int i = 1; i <= size; i++) {
      if (near[i] != 0 && cost[i][near[i]] < min) {
         min = cost[i][near[i]];
         nextVertex = i;
      }
   }
   t[j][0] = nextVertex;
   t[j][1] = near[nextVertex];
   mincost += cost[nextVertex][near[nextVertex]];
   near[nextVertex] = 0;

   for (int i = 1; i <= size; i++) {
      if (near[i] != 0 && cost[i][near[i]] > cost[i][nextVertex]) {
         near[i] = nextVertex;
      }
   }
   displayNear(near, size, cost);
}

showcost(cost, size);
showmst(size, t);
return mincost;
}

int main() {
   int n;
   int cost[MAX][MAX];
   int t[MAX][2];
   int near[MAX];
```

```
printf("Enter the number of vertices: ");
scanf("%d", &n);

printf("Enter the cost adjacency matrix (use 0 for no edge):\n");
for (int i = 1; i <= n; i++) {
   for (int j = 1; j <= n; j++) {
      cost[i][j] = INF;
   }
}
printf("Enter the edges and their costs (enter -1 -1 to stop):\n");
while (1) {
   int u, v, w;
   scanf("%d %d", &u, &v);
   if (u == -1 && v == -1) {
      break;
   }
   printf("Enter the cost of (%d, %d): ", u, v);
   scanf("%d", &w);
   cost[u][v] = w;
   cost[v][u] = w;
}

clock_t start = clock();

int min = INF;
for (int i = 1; i <= n; i++) {
   for (int j = 1; j <= n; j++) {
      if (cost[i][j] < min) {
         min = cost[i][j];
         k = i;
         l = j;
      }
   }
}

int mincost = prims(t, n, cost, near);

clock_t end = clock();
double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;

   printf("\nThe minimum cost is: %d\n", mincost);
   printf("Time taken: %.6f seconds\n", time_taken);
   return 0;
}
```

OUTPUT

```
C:\Users\vaidn\Do
Enter the number of vertices: 8
Enter the cost adjacency matrix (use 0 for no edge):
Enter the edges and their costs (enter -1 -1 to stop):
1 2
Enter the cost of (1, 2): 1
1 3
Enter the cost of (1, 3): 2
2 3
Enter the cost of (2, 3): 1
3 5
Enter the cost of (3, 5): 2
2 6
Enter the cost of (2, 6): 1
3 4
Enter the cost of (3, 4): 3
3 6
Enter the cost of (3, 6): 2
4 6
Enter the cost of (4, 6): 2
2 5
Enter the cost of (2, 5): 2
2 8
Enter the cost of (2, 8): 3
8 5
Enter the cost of (8, 5): 1
5 6
Enter the cost of (5, 6): 3
5 7
Enter the cost of (5, 7): 2
6 7
Enter the cost of (6, 7): 1
-1 -1

near: [01] [02] [03] [04] [05] [06] [07] [08]
        0    0    2    1    2    2    1    2
cost:  --   --    1   inf   2    1   inf   3

near: [01] [02] [03] [04] [05] [06] [07] [08]
        0    0    0    3    2    2    1    2
cost:  --   --   --    3    2    1   inf   3

near: [01] [02] [03] [04] [05] [06] [07] [08]
        0    0    0    6    2    0    6    2
cost:  --   --   --    2    2   --    1    3

near: [01] [02] [03] [04] [05] [06] [07] [08]
        0    0    0    6    2    0    0    2
cost:  --   --   --    2    2   --   --    3

near: [01] [02] [03] [04] [05] [06] [07] [08]
        0    0    0    0    2    0    0    2
cost:  --   --   --   --    2   --   --    3
```

```
near: [01] [02] [03] [04] [05] [06] [07] [08]
        0    0    0    0    0    0    0    5
cost:  --   --   --   --   --   --   --    1

near: [01] [02] [03] [04] [05] [06] [07] [08]
        0    0    0    0    0    0    0    0
cost:  --   --   --   --   --   --   --   --

The cost adjacency matrix is:
0 1 2 0 0 0 0 0
1 0 1 0 2 1 0 3
2 1 0 3 2 2 0 0
0 0 3 0 0 2 0 0
0 2 2 0 0 3 2 1
0 1 2 2 3 0 1 0
0 0 0 0 2 1 0 0
0 3 0 0 1 0 0 0

The edges of the Minimum Spanning Tree (MST) are:
1 - 2
3 - 2
6 - 2
7 - 6
4 - 6
5 - 2
8 - 5

The minimum cost is: 9
Time taken: 0.063000 seconds

Process returned 0 (0x0)   execution time : 124.109 s
Press any key to continue.
```

CONCLUSION

Prim's algorithm was successfully implemented using the greedy method to f ind the minimum – cost spanning tree. The program displays step – by – step the edges that constitute the MST, the minimum cost and the contents of the near array. The time complexity of the algorithm is calculated to be $O(n^2)$ if cost -adjacency matrix is used and O((n + |E|) logn) if red – black trees are used where n is the number of vertices and E is the set of edges.

REFERENCES

1."Fundamentals of Computer Algorithms" by Ellis Horowitz, Sartaj Sahni, and Sanguthevar Rajasekaran: Page 218-219

2."Design and Analysis of Algorithms" by R. C. T. Lee, S. S. Tseng, R. C. Chang, and Y. T. Tsai: Prim's Algorithm: Detailed explanation begins on page 27.

KRUSKAL'S ALGORITHM

DATE: 31-01-2025

AIM: Write a C program to find the Minimum Cost Spanning tree using Kruskal's algorithm.

PROBLEM STATEMENT

Given a connected, undirected, and weighted graph G=(V,E), find a minimum spanning tree (MST) by selecting edges in increasing order of weight while ensuring no cycles are formed. The algorithm stops when all vertices are included in the MST. Kruskal's algorithm uses the greedy approach and often employs the union-find data structure for cycle detection.

THEORY

It is a greedy method algorithm which finds a minimum spanning tree for an undirected weighted graph by selecting the edge with the least weight. If the graph is connected, it finds a minimum spanning tree. It is a greedy algorithm in graph theory as in each step it adds the next lowest-weight edge that will not form a cycle to the minimum spanning forest. Kruskal's Algorithm grows a solution from the cheapest edge by adding the next cheapest edge to the existing tree.

ALGORITHMS

Algorithm Kruskal (E, cost, n, t)

// 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;

// Each vertex is in a different set

i := 0, mincost := 0.0;

while ((i < n - 1) and (heap not empty)) do

{

Delete a min-cost edge (u, v) from the heap & reheapify using Adjust;

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 - 1) then write ("No Spanning tree");
   else return mincost;
}
```

Algorithm Heapify (a, n)

```
{
for i := ⌊ n/2 ⌋ to 1 step -1 do
 Adjust (a, i, n)
 }
```

Algorithm Adjust (a, i, n)

```
{
j := 2i;
item := a[i];
 while (j ≤ n) do
{
  if ((j < n) and (a[j] > a[j + 1])) then
  j := j + 1;
 if (item ≤ a[j]) then break;
 a[⌊ n/2 ⌋] := a[j];
  j := 2j;
 }
 a[⌊ j/2 ⌋] := item;
}
```

Algorithm DelMin (a, n, x)

```
{
 if (n = 0) then
 {
```

```
 write (“Heap Empty”);
 return;
}
x := a[i];
a[i] := a[n];
Adjust (a, 1, n – 1);
return true;
}
```

Algorithm Find (i)

```
{
 while (parent[i] ≥ 0) do i := parent[i];
 return i;
}
```

Algorithm Union (i, j)

```
{
 parent[i] := j;
}
```

KRUSKAL’S ALGORITHM COMPLEXITY ANALYSIS

Recurrence Relation:

Since Kruskal’s algorithm sorts the edges and processes them one by one, the recurrence is: T(E)=O(ElogE)+O(E)

Time Complexity:

Using Sorting + Union-Find (Path Compression & Rank): O(|E|log|E|)

Space Complexity: O(N+E) (for storing edges and Disjoint Set data structure)

PROGRAM

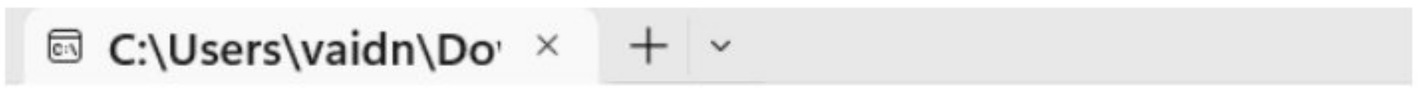
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 100
#define INF 99999
int parent[MAX], e = 0;
struct Edge {
   int o, d, weight;
};

void adjust(struct Edge cost[], int i, int n) {
   int j = 2 * i;
   struct Edge item = cost[i];
   while (j <= n) {
      if (j < n && (cost[j].weight > cost[j + 1].weight ||
```

```
            (cost[j].weight == cost[j + 1].weight && cost[j].o > cost[j + 1].o))) {
      j = j + 1;
    }
    if (item.weight <= cost[j].weight) {
      break;
    }
    cost[j / 2] = cost[j];
    j = 2 * j;
  }
  cost[j / 2] = item;
}
void heapify(struct Edge cost[], int n) {
  for (int i = n / 2; i >= 1; i--) {
    adjust(cost, i, n);
  }
}
int delMin(struct Edge cost[], int *n, struct Edge *x) {
  if (*n == 0) {
    printf("\nHeap Empty\n");
    return 0;
  }
  *x = cost[1];
  cost[1] = cost[*n];
  (*n)--;
  adjust(cost, 1, *n);
  return 1;
}

int find(int i) {
  if (parent[i] < 0) {
    return i;
  }
  return parent[i] = find(parent[i]);
}

void Union(int i, int j) {
  parent[i] = j;
}

void kruskal(struct Edge cost[MAX], int n, int e) {
  int i, mincost = 0, z;
  struct Edge x;
  int mst[MAX][2];
  heapify(cost, e);
  for (i = 1; i <= n; i++) {
    parent[i] = -1;
  }
```

```
i = 0;
while (i < n - 1 && e > 0) {
   if (delMin(cost, &e, &x)) {
      int j = find(x.o);
      int k = find(x.d);

      if (j != k) {
         mst[i][0] = x.o;
         mst[i][1] = x.d;
         mincost += x.weight;
         Union(j, k);
         i++;
         printf("\nEdge %d: (%d - %d) Cost: %d\n", i, x.o, x.d, x.weight);

         printf("Parent Array: [ ");
         for (z = 1; z <= n; z++) {
            printf("%d ", parent[z]);
         }
         printf("]\nMinimum Cost: %d\n", mincost);
      }
   }
}

   if (i != n - 1) {
      printf("\nNo Spanning Tree Possible\n");
   } else {
      printf("\nEdges in Minimum Spanning Tree:\n");
      for (i = 0; i < n - 1; i++) {
         printf("%d - %d\n", mst[i][0], mst[i][1]);
      }
      printf("Total Minimum Cost: %d\n", mincost);
   }
}

int main() {
   int n, i, ori, des, we;
   struct Edge cost[MAX];
   clock_t start, end;
   double time_taken;

   printf("\nEnter the number of vertices: ");
   scanf("%d", &n);

   for (i = 1; i <= n * (n - 1) / 2; i++) {
      printf("Enter edge %d (-1, -1 to quit): ", i);
      scanf("%d%d", &ori, &des);
```

```
if (ori == -1 && des == -1) {
   break;
}
if (ori > n || des > n || ori < 1 || des < 1) {
   printf("\nInvalid Input! Try Again.\n");
   i--;
} else {
   printf("Enter cost of edge: ");
   scanf("%d", &we);
   cost[i].o = ori;
        cost[i].d = des;
        cost[i].weight = we;
        e++;
      }
   }
   start = clock();
   kruskal(cost, n, e);
   end = clock();
   time_taken = (double)(end - start) / CLOCKS_PER_SEC;
   printf("\nTime Taken: %.6f seconds\n", time_taken);
   return 0;
}
```

OUTPUT




```
Enter the number of vertices: 8
Enter edge 1 (-1, -1 to quit): 1 2
Enter cost of edge: 5
Enter edge 2 (-1, -1 to quit): 1 3
Enter cost of edge: 2
Enter edge 3 (-1, -1 to quit): 2 3
Enter cost of edge: 3
Enter edge 4 (-1, -1 to quit): 3 8
Enter cost of edge: 3
Enter edge 5 (-1, -1 to quit): 5 8
Enter cost of edge: 2
Enter edge 6 (-1, -1 to quit): 3 5
Enter cost of edge: 4
Enter edge 7 (-1, -1 to quit): 2 5
Enter cost of edge: 1
Enter edge 8 (-1, -1 to quit): 2 4
Enter cost of edge: 2
Enter edge 9 (-1, -1 to quit): 4 5
Enter cost of edge: 3
Enter edge 10 (-1, -1 to quit): 5 6
Enter cost of edge: 1
Enter edge 11 (-1, -1 to quit): 5 7
Enter cost of edge: 3
Enter edge 12 (-1, -1 to quit): 6 7
Enter cost of edge: 4
Enter edge 13 (-1, -1 to quit): 4 6
Enter cost of edge: 2
Enter edge 14 (-1, -1 to quit): -1 -1
```

C:\Users\vaidn\Do

```
Enter cost of edge: 1
Enter edge 11 (-1, -1 to quit): 5 7
Enter cost of edge: 3
Enter edge 12 (-1, -1 to quit): 6 7
Enter cost of edge: 4
Enter edge 13 (-1, -1 to quit): 4 6
Enter cost of edge: 2
Enter edge 14 (-1, -1 to quit): -1 -1

Edge 1: (2 - 5) Cost: 1
Parent Array: [ -1 5 -1 -1 -1 -1 -1 -1 ]
Minimum Cost: 1

Edge 2: (5 - 6) Cost: 1
Parent Array: [ -1 5 -1 -1 6 -1 -1 -1 ]
Minimum Cost: 2

Edge 3: (1 - 3) Cost: 2
Parent Array: [ 3 5 -1 -1 6 -1 -1 -1 ]
Minimum Cost: 4

Edge 4: (2 - 4) Cost: 2
Parent Array: [ 3 6 -1 -1 6 4 -1 -1 ]
Minimum Cost: 6

Edge 5: (5 - 8) Cost: 2
Parent Array: [ 3 6 -1 8 4 4 -1 -1 ]
Minimum Cost: 8

Edge 6: (2 - 3) Cost: 3
Parent Array: [ 3 8 -1 8 4 8 -1 3 ]
Minimum Cost: 11

Edge 7: (5 - 7) Cost: 3
Parent Array: [ 3 8 7 3 3 8 -1 3 ]
Minimum Cost: 14

Edges in Minimum Spanning Tree:
2 - 5
5 - 6
1 - 3
2 - 4
5 - 8
2 - 3
5 - 7
Total Minimum Cost: 14

Time Taken: 0.014000 seconds

Process returned 0 (0x0)   execution time : 121.920 s
Press any key to continue.
```

CONCLUSION

Kruskal's algorithm was successfully implemented using the greedy method to find the minimum – cost spanning tree. The program displays step – by – step the edges that constitute the MST, the minimum cost and the contents of the parent array. The time complexity of the algorithm is calculated to be $O(|E| \log|E|)$ where E is the set of edges.

REFERENCES

1."Fundamentals of Computer Algorithms" by Ellis Horowitz, Sartaj Sahni, and Sanguthevar Rajasekaran: Page 220-224

2."Design and Analysis of Algorithms" by R. C. T. Lee, S. S. Tseng, R. C. Chang, and Y. T. Tsai: Kruskal's Algorithm: Covered on page 28.

SINGLE SOURCE SHORTEST PATH **DATE:** 31-01-2025

AIM: Write a C program to implement Single Source Shortest Path problem using the greedy method.

PROBLEM STATEMENT

Given a weighted graph G=(V,E) with non-negative edge weights and a source vertex s, find the shortest path from s to all other vertices in the graph. The shortest path is the path with the minimum sum of edge weights. Dijkstra's algorithm uses a priority queue to iteratively find the next nearest vertex and update distances.

THEORY

Also known as Single Source Shortest Path Algorithm, Dijkstra's Algorithm is a greedy method algorithm where, given a graph and a source vertex in the graph, shortest paths from source to all vertices in the given graph if found.

We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

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[1:n, 1:n].
{
   for i:=1 to n do
   {
      //Initialise S.
      S[i] := false; dist[i] := cost[v,i];
   }
   S[v] := true; dist[v] := 0.0; // Put v in S.
   for num := 2 to n-1 do
   {
```

```
//Determine 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 in S[w] = false) do
   //Update distances.
      if(dist[w] > dist[u] + cost[u,w]) then
         dist[w] = dist[u] + cost[u,w]);
   }
}
```

SHORTEST PATH COMPLEXITY ANALYSIS

Recurrence Relation:
If implemented using a Priority Queue (Min-Heap): $T(n)=T(n-1)+O(\log n)$

Time Complexity:

- Using Adjacency Matrix + Simple Min Selection: $O(n^2)$
- Using Adjacency List + Min-Heap (Binary Heap): $O((n+E)\log n)$

Space Complexity: O(n+E) (for adjacency list representation)

PROGRAM

```
#include <stdio.h>
#include <stdbool.h>
#include <time.h>
#define MAX 100
#define INF 99999
int dist[MAX], parent[MAX];
bool s[MAX];

int minDist(int n) {
   int min = INF, index = -1, i;
   for (i = 1; i <= n; i++) {
      if (!s[i] && dist[i] < min) {
         min = dist[i];
         index = i;
      }
   }
   return index;
```

```
}

void printPath(int j) {
   if (parent[j] == -1) {
      printf("%d ", j);
      return;
   }
   printPath(parent[j]);
   printf("-> %d ", j);
}

void shortestPath(int n, int cost[MAX][MAX], int v) {
   int i, j, w;
   for (i = 1; i <= n; i++) {
      s[i] = false;
      dist[i] = cost[v][i];
      parent[i] = (cost[v][i] != INF && v != i) ? v : -1;
   }
   s[v] = true;
   dist[v] = 0;

   printf("\n\nIteration 1\tv = %d\n", v);
   for (j = 1; j <= n; j++) {
      if (s[j] == true) printf("s[%d] = ", j);
   }
   printf("TRUE\n");
   for (j = 1; j <= n; j++) {
      if (s[j] == false) printf("s[%d] = ", j);
   }
   printf("FALSE\nDist Array: [ ");
   for (j = 1; j <= n; j++) {
      if (dist[j] == INF) printf("INF ");
      else printf("%d ", dist[j]);
   }
   printf("]\n");

   for (i = 2; i <= n; i++) {
      int u = minDist(n);
      if (u == -1) break;
      s[u] = true;

      printf("\n\nIteration %d\tu = %d\n", i, u);
      for (j = 1; j <= n; j++) {
         if (s[j] == true) printf("s[%d] = ", j);
      }
      printf("TRUE\n");
      for (j = 1; j <= n; j++) {
```

```
    if (s[j] == false) printf("s[%d] = ", j);
  }
  printf("FALSE\nDist Array: [ ");

  for (w = 1; w <= n; w++) {
    if (!s[w] && cost[u][w] && dist[u] != INF && dist[w] > dist[u] + cost[u][w]) {
      dist[w] = dist[u] + cost[u][w];
      parent[w] = u;
    }
  }

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

  int main() {
    int n, maxE, i, j, o, d, w, start;
    int cost[MAX][MAX];

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

  for (i = 1; i <= n; i++) {
    for (j = 1; j <= n; j++) {
      cost[i][j] = INF;
    }
  }

  for (i = 1; i <= maxE; i++) {
    printf("Enter edge %d (-1, -1 to quit): ", i);
    scanf("%d%d", &o, &d);
    if (o == -1 && d == -1) {
      break;
    }
  if (o > n || d > n || o < 0 || d < 0) {
    printf("\nInvalid Input");
    i--;
  } else {
    printf("Enter weight of edge: ");
    scanf("%d", &w);
    cost[o][d] = w;
  }
```

```
}

printf("\nEnter the starting vertex: ");
scanf("%d", &start);
clock_t start_time = clock();
shortestPath(n, cost, start);
clock_t end_time = clock();
double time_taken = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;

printf("\n\n%-10s %-12s %-16s %-30s\n", "Source", "Destination", "Path Length",
"Path");
  printf("-----------------------------------------------------------\n");
  for (i = 1; i <= n; i++) {
    if (i != start) {
      printf("%-10d %-12d ", start, i);
      if (dist[i] == INF) {
        printf("%-16s %-30s\n", "INF", "No path");
      } else {
        printf("%-16d ", dist[i]);
        printPath(i);
        printf("\n");
   }}}
  printf("\nTime taken: %.6f seconds\n", time_taken);
  return 0;
}
```

OUTPUT

C:\Users\vaidn\Do

```
Enter the number of vertices: 8
Enter edge 1 (-1, -1 to quit): 1 2
Enter weight of edge: 40
Enter edge 2 (-1, -1 to quit): 1 3
Enter weight of edge: 20
Enter edge 3 (-1, -1 to quit): 1 4
Enter weight of edge: 10
Enter edge 4 (-1, -1 to quit): 1 7
Enter weight of edge: 60
Enter edge 5 (-1, -1 to quit): 1 5
Enter weight of edge: 50
Enter edge 6 (-1, -1 to quit): 2 3
Enter weight of edge: 1
Enter edge 7 (-1, -1 to quit): 4 3
Enter weight of edge: 5
Enter edge 8 (-1, -1 to quit): 2 5
Enter weight of edge: 5
Enter edge 9 (-1, -1 to quit): 3 5
Enter weight of edge: 2
Enter edge 10 (-1, -1 to quit): 3 6
Enter weight of edge: 10
```

```
C:\Users\vaidn\Do
Enter weight of edge: 1
Enter edge 7 (-1, -1 to quit): 4 3
Enter weight of edge: 5
Enter edge 8 (-1, -1 to quit): 2 5
Enter weight of edge: 5
Enter edge 9 (-1, -1 to quit): 3 5
Enter weight of edge: 2
Enter edge 10 (-1, -1 to quit): 3 6
Enter weight of edge: 10
Enter edge 11 (-1, -1 to quit): 4 7
Enter weight of edge: 5
Enter edge 12 (-1, -1 to quit): 3 7
Enter weight of edge: 2
Enter edge 13 (-1, -1 to quit): 7 8
Enter weight of edge: 3
Enter edge 14 (-1, -1 to quit): 6 7
Enter weight of edge: 3
Enter edge 15 (-1, -1 to quit): 8 3
Enter weight of edge: 1
Enter edge 16 (-1, -1 to quit): 6 8
Enter weight of edge: 5
Enter edge 17 (-1, -1 to quit): 5 6
Enter weight of edge: 3
Enter edge 18 (-1, -1 to quit): 2 8
Enter weight of edge: 2
Enter edge 19 (-1, -1 to quit): 5 8
Enter weight of edge: 5
Enter edge 20 (-1, -1 to quit): 8 4
Enter weight of edge: 5
Enter edge 21 (-1, -1 to quit): -1 -1

Enter the starting vertex: 1


Iteration 1     v = 1
s[1] = TRUE
s[2] = s[3] = s[4] = s[5] = s[6] = s[7] = s[8] = FALSE
Dist Array: [ 0 40 20 10 50 INF 60 INF ]


Iteration 2     u = 4
s[1] = s[4] = TRUE
s[2] = s[3] = s[5] = s[6] = s[7] = s[8] = FALSE
Dist Array: [ 0 40 15 10 50 INF 15 INF ]


Iteration 3     u = 3
s[1] = s[3] = s[4] = TRUE
s[2] = s[5] = s[6] = s[7] = s[8] = FALSE
Dist Array: [ 0 40 15 10 17 25 15 INF ]


Iteration 4     u = 7
s[1] = s[3] = s[4] = s[7] = TRUE
s[2] = s[5] = s[6] = s[8] = FALSE
Dist Array: [ 0 40 15 10 17 25 15 18 ]
```

```
C:\Users\vaidn\Do
s[2] = s[3] = s[5] = s[6] = s[7] = s[8] = FALSE
Dist Array: [ 0 40 15 10 50 INF 15 INF ]


Iteration 3     u = 3
s[1] = s[3] = s[4] = TRUE
s[2] = s[5] = s[6] = s[7] = s[8] = FALSE
Dist Array: [ 0 40 15 10 17 25 15 INF ]


Iteration 4     u = 7
s[1] = s[3] = s[4] = s[7] = TRUE
s[2] = s[5] = s[6] = s[8] = FALSE
Dist Array: [ 0 40 15 10 17 25 15 18 ]


Iteration 5     u = 5
s[1] = s[3] = s[4] = s[5] = s[7] = TRUE
s[2] = s[6] = s[8] = FALSE
Dist Array: [ 0 40 15 10 17 20 15 18 ]


Iteration 6     u = 8
s[1] = s[3] = s[4] = s[5] = s[7] = s[8] = TRUE
s[2] = s[6] = FALSE
Dist Array: [ 0 40 15 10 17 20 15 18 ]


Iteration 7     u = 6
s[1] = s[3] = s[4] = s[5] = s[6] = s[7] = s[8] = TRUE
s[2] = FALSE
Dist Array: [ 0 40 15 10 17 20 15 18 ]


Iteration 8     u = 2
s[1] = s[2] = s[3] = s[4] = s[5] = s[6] = s[7] = s[8] = TRUE
FALSE
Dist Array: [ 0 40 15 10 17 20 15 18 ]


Source     Destination  Path Length      Path
-------------------------------------------------------------
1          2            40               1 -> 2
1          3            15               1 -> 4 -> 3
1          4            10               1 -> 4
1          5            17               1 -> 4 -> 3 -> 5
1          6            20               1 -> 4 -> 3 -> 5 -> 6
1          7            15               1 -> 4 -> 7
1          8            18               1 -> 4 -> 7 -> 8

Time taken: 0.095000 seconds

Process returned 0 (0x0)   execution time : 294.173 s
Press any key to continue.
```

CONCLUSION

Single Source Shortest Path problem was successfully implemented using the greedy method to find the shortest distances between a source vertex and all other vertices. The program displays step – by – step the changing shortest distances between the source vertex and all other vertices, The time complexity of the algorithm is calculated to be $O(n^2)$ when cost adjacency matrix is used and $O((n + |E|) \log n)$ where n is the number of vertices and E is the set of edges.

REFERENCES

1."Fundamentals of Computer Algorithms" by Ellis Horowitz, Sartaj Sahni, and Sanguthevar Rajasekaran: Page 241-248

2."Design and Analysis of Algorithms" by R. C. T. Lee, S. S. Tseng, R. C. Chang, and Y. T. Tsai: Single Source Shortest Path Problem (Dijkstra's Algorithm): Explained on page 27.