Experiment No. 7
Kruskal's Algorithm
Date of Performance:
Date of Submission:

Experiment No. 7

Title: Kruskal’s Algorithm.

Aim: To study and implement Kruskal’s Minimum Cost Spanning Tree Algorithm.

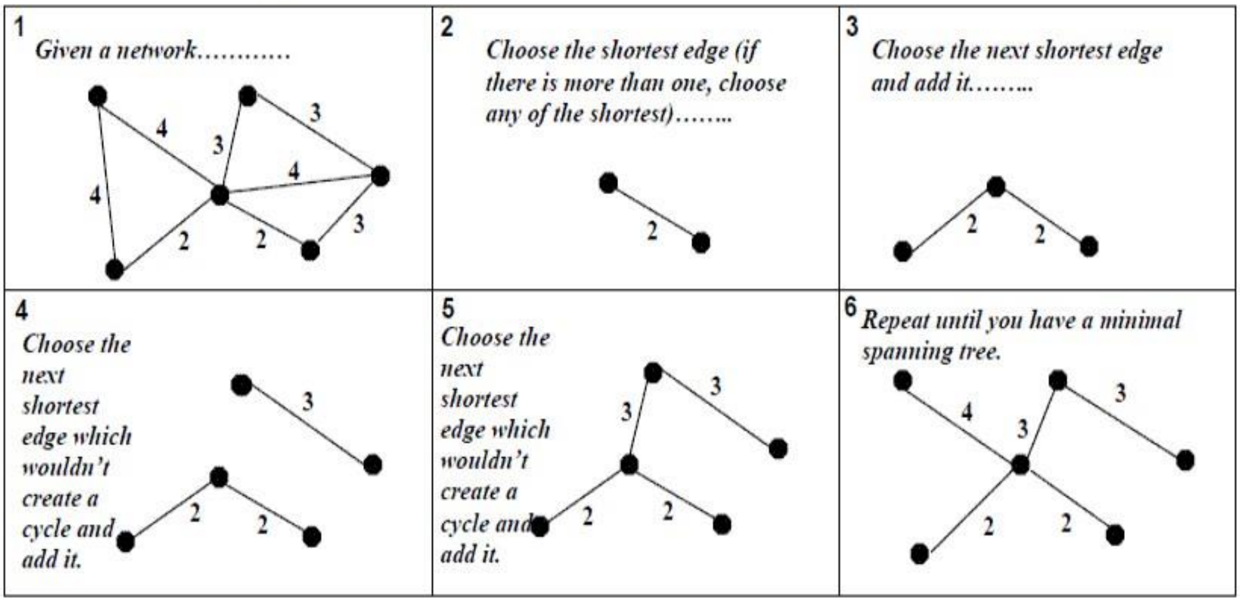
Objective: To introduce Greedy based algorithms.

Theory:

Kruskal's algorithm finds a minimum spanning forest of an undirected edge-weighted graph. If the graph is connected, it finds a minimum spanning tree. (A minimum spanning tree of a connected graph is a subset of the edges that forms a tree that includes every vertex, where the sum of the weights of all the edges in the tree is minimized. For a disconnected graph, a minimum spanning forest is composed of a minimum spanning tree for each connected component.) 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.

Example:

Kruskal's Algorithm



Algorithm and Complexity:

```
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 minimum cost edge (u, v) from the heap
        and 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;
}
```

Time Complexity is O(nlog n), Where, n = number of Edges

Code :

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

// Structure to represent an edge in the graph
struct Edge {
   int src, dest, weight;
};

// Structure to represent a subset for union-find
struct Subset {
   int parent;
   int rank;
};
```

```
// Function to find set of an element i
int find(struct Subset subsets[], int i) {
   if (subsets[i].parent != i)
      subsets[i].parent = find(subsets, subsets[i].parent);
   return subsets[i].parent;
}

// Function that does union of two sets of x and y
void Union(struct Subset subsets[], int x, int y) {
   int xroot = find(subsets, x);
   int yroot = find(subsets, y);

   // Attach smaller rank tree under root of high rank tree (Union by Rank)
   if (subsets[xroot].rank < subsets[yroot].rank)
      subsets[xroot].parent = yroot;
   else if (subsets[xroot].rank > subsets[yroot].rank)
      subsets[yroot].parent = xroot;
   else {
      subsets[yroot].parent = xroot;
      subsets[xroot].rank++;
   }
}

// Comparator function to sort edges based on weight
int compare(const void *a, const void *b) {
   struct Edge *a1 = (struct Edge *)a;
   struct Edge *b1 = (struct Edge *)b;
   return a1->weight > b1->weight;
}

// Function to construct and print MST using Kruskal's algorithm
void KruskalMST(struct Edge edges[], int V, int E) {
   struct Edge result[V];  // Store the result MST
   int e = 0;  // An index variable used for result[]
   int i = 0;  // An index variable used for sorted edges

   // Step 1: Sort all the edges in non-decreasing order of their weight
   qsort(edges, E, sizeof(edges[0]), compare);
```

```
    // Allocate memory for creating V subsets
    struct Subset *subsets = (struct Subset *)malloc(V * sizeof(struct Subset));

    // Create V subsets with single elements
    for (int v = 0; v < V; v++) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Number of edges to be taken is equal to V-1
    while (e < V - 1 && i < E) {
        // Step 2: Pick the smallest edge. And increment the index for next iteration
        struct Edge next_edge = edges[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        // If including this edge doesn't cause cycle, include it in result and increment
the index
        // of result for next edge
        if (x != y) {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
    }

    // Print the constructed MST
    printf("Following are the edges in the constructed MST:\n");
    for (i = 0; i < e; ++i)
        printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);
}

int main() {
    /* Let us create the following graph
          10
        0--------1
        |  \     |
       6|   5\   |15
        |      \ |
```

```
     2--------3
        4       */
   int V = 4;  // Number of vertices in graph
   int E = 5;  // Number of edges in graph
   struct Edge edges[] = {{0, 1, 10}, {0, 2, 6}, {0, 3, 5}, {1, 3, 15}, {2, 3, 4}};

   // Function call to construct and print MST
   KruskalMST(edges, V, E);

   return 0;
}
```

Output :

```
PS E:\Testing_Lang> cd "e:\Testing_Lang\" ; if ($?) { gcc test.c -o test } ; if ($?) { .\test }
Following are the edges in the constructed MST:
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
PS E:\Testing_Lang>
```

Conclusion:

This program implements Kruskal's MST algorithm using the Greedy method for a given graph represented using an array of edges. It constructs and prints the Minimum Spanning Tree (MST) of the graph.

Kruskal's algorithm works by sorting all the edges in non-decreasing order of their weight and selecting edges one by one in this sorted order, adding the edge to the MST if it doesn't create a cycle. It continues this process until all vertices are included in the MST.