Kruskal's Algorithm for Minimum Spanning Tree

Aim: The aim of the experiment is to implement Kruskal's algorithm for finding the Minimum Spanning Tree (MST) of a given graph.

Theory:
Kruskal's algorithm finds the Minimum Spanning Tree (MST) of a graph by sorting edges, using a union-find data structure to detect cycles, and iteratively adding edges with minimum weight that don't form cycles. It prioritizes the smallest-weight edges, ensuring an optimal tree connecting all vertices without cycles. This greedy approach guarantees the MST with minimal total edge weight. Kruskal's algorithm efficiently handles connected, undirected graphs, making it a fundamental tool for network optimization and spanning tree generation.k

Code:

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

struct Edge
{
   int src, dest, weight;
};

struct Graph
{
   int V, E;
   struct Edge *edge;
};

struct Subset
{
   int parent;
   int rank;
};

struct Graph *createGraph(int V, int E)
{
   struct Graph *graph = (struct Graph *)malloc(sizeof(struct Graph));
   graph->V = V;
   graph->E = E;
   graph->edge = (struct Edge *)malloc(graph->E * sizeof(struct Edge));
   return graph;
}

int find(struct Subset subsets[], int i)
{
   if (subsets[i].parent != i)
   {
      subsets[i].parent = find(subsets, subsets[i].parent);
   }
   return subsets[i].parent;
}
```

```
void Union(struct Subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);
    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++;
    }
}

int myComp(const void *a, const void *b)
{
    struct Edge *a1 = (struct Edge *)a;
    struct Edge *b1 = (struct Edge *)b;
    return a1->weight > b1->weight;
}

void KruskalMST(struct Graph *graph)
{
    int V = graph->V;
    struct Edge result[V];
    int e = 0;
    int i = 0;
    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);
    struct Subset *subsets = (struct Subset *)malloc(V * sizeof(struct Subset));
    for (int v = 0; v < V; v++)
    {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }
    while (e < V - 1 && i < graph->E)
    {
        struct Edge next_edge = graph->edge[i++];
        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);
        if (x != y)
        {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
    }
    printf("Following are the edges in the constructed MST\n");
    int minimumCost = 0;
```

```
    for (i = 0; i < e; ++i)
    {
        printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);
        minimumCost += result[i].weight;
    }
    printf("Minimum Cost Spanning Tree: %d\n", minimumCost);
}

int main()
{
    int V, E;
    printf("Enter the number of vertices: ");
    scanf("%d", &V);
    printf("Enter the number of edges: ");
    scanf("%d", &E);
    struct Graph *graph = createGraph(V, E);
    for (int i = 0; i < E; ++i)
    {
        printf("Enter the source, destination, and weight of edge %d: ", i + 1);
        scanf("%d %d %d", &graph->edge[i].src, &graph->edge[i].dest, &graph->edge[i].weight);
    }
    KruskalMST(graph);
    return 0;
}
```

Output:

```
Enter the number of vertices
: 5
Enter the number of edges: 6
Enter the source, destination, and weight of edge 1: 0 1 8
Enter the source, destination, and weight of edge 2: 0 2 9
Enter the source, destination, and weight of edge 3: 1 2 7
Enter the source, destination, and weight of edge 4: 1 3 5
Enter the source, destination, and weight of edge 5: 2 3 4
Enter the source, destination, and weight of edge 6: 3 4 6
Following are the edges in the constructed MST
2 -- 3 == 4
1 -- 3 == 5
3 -- 4 == 6
0 -- 1 == 8
Minimum Cost Spanning Tree: 23
```

Algorithm Analysis

Algorithm steps:

1. Sort Edges: Sort all the edges of the graph in non-decreasing order of their weights.

2. Initialize MST and Subsets: Initialize an empty set to store the edges of the MST. Also, initialize subsets for each vertex, initially pointing to themselves.

3. Iterate Through Sorted Edges:
- Start iterating through the sorted edges.
- For each edge:
  - Check if including the edge forms a cycle in the MST. This can be done by finding the subsets of the vertices connected by the edge and checking if they belong to the same subset.
  - If including the edge does not form a cycle, add it to the MST.

4. Union-Find Operations:
- Whenever adding an edge to the MST, perform union-find operations to merge the subsets of the vertices connected by the edge.

5. Stop Condition:
- Stop the iteration when the number of edges in the MST becomes equal to the number of vertices minus 1 (V-1), where V is the number of vertices in the graph.

6. Output MST:
- Once the MST is constructed, output the set of edges forming the MST along with their weights.

7. Minimum Spanning Tree:
- The resulting set of edges forms the Minimum Spanning Tree of the graph.

Time complexity:

1. Sorting Edges: Sorting all the edges of the graph takes O(E log E) time, where E is the number of edges in the graph.

2. Union-Find Operations:
- Each edge is considered exactly once.
- For each edge, two `find` operations and one `union` operation are performed.
- In the worst-case scenario, each `find` operation may traverse the entire height of the subset tree, which has a maximum height of log(V) (where V is the number of vertices).
- Therefore, for E edges, the total time complexity for union-find operations is O(E log V).

3. Overall Time Complexity:
- The dominant term in the overall time complexity is O(E log E) due to the sorting of edges.
- Therefore, the overall time complexity of Kruskal's algorithm is O(E log E), where E is the number of edges in the graph.

Note: In a dense graph where E approaches V^2, the sorting step becomes the dominant factor in the time complexity. However, in a sparse graph where E is much smaller than V^2, the union-find operations become more significant. Therefore, the actual runtime of Kruskal's algorithm can vary based on the characteristics of the input graph.

Space Complexity:

1. Edge List: The space required to store the edge list depends on the number of edges in the graph. Since each edge is represented by a structure containing source, destination, and weight, the space complexity for storing the edge list is O(E), where E is the number of edges.

2. Subsets Array: The array used to represent subsets for each vertex requires space proportional to the number of vertices in the graph. This array typically has a size of V, where V is the number of vertices.

3. Additional Variables: Other variables such as loop counters, temporary variables, and function parameters require constant space and do not contribute significantly to the overall space complexity.

4. Total Space Complexity:

- The dominant contributors to space complexity are the edge list (O(E)) and the subsets array (O(V)).
- Therefore, the overall space complexity of Kruskal's algorithm is O(E + V).