| |
|---|
| 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.

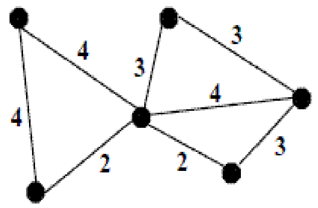
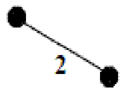
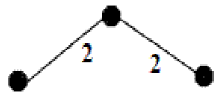
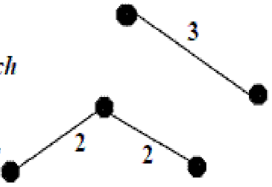
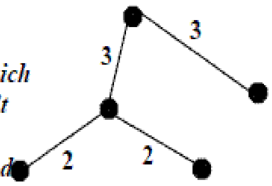
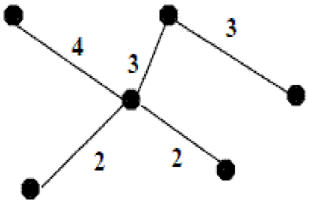
**Objective:** To introduce Greedy based algorithms

**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

| | | |
|---|---|---|
| 1 *Given a network…………*  | 2 *Choose the shortest edge (if there is more than one, choose any of the shortest)……..*  | 3 *Choose the next shortest edge and add it……..*  |
| 4 *Choose the next shortest edge which wouldn't create a cycle and add it.*  | 5 *Choose the next shortest edge which wouldn't create a cycle and add it.*  | 6 *Repeat until you have a minimal spanning tree.*  |

**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

**Implemenation:**

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

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

// Structure to represent a connected, undirected graph
struct Graph {
   int V, E;
   struct Edge* edge;
```

```
};

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

// Create a graph with V vertices and E edges
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(E * sizeof(struct Edge));
   return graph;
}

// Find set of an element i (uses path compression technique)
int find(struct Subset subsets[], int i) {
   if (subsets[i].parent != i)
      subsets[i].parent = find(subsets, subsets[i].parent);
   return subsets[i].parent;
}

// Union of two sets of x and y (uses union by rank)
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++;
   }
```

```
}

// Compare function for qsort
int compare(const void* a, const void* b) {
   struct Edge* aEdge = (struct Edge*)a;
   struct Edge* bEdge = (struct Edge*)b;
   return aEdge->weight - bEdge->weight;
}

// Kruskal's algorithm function
void Kruskal(struct Graph* graph) {
   int V = graph->V;
   struct Edge result[V];  // Array to store the result MST
   int e = 0;  // Index variable for result array
   int i = 0;  // Index variable for sorted edges array

   // Step 1: Sort all the edges in non-decreasing order of their weight
   qsort(graph->edge, graph->E, sizeof(graph->edge[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 < graph->E) {
      // Step 2: Pick the smallest edge
      struct Edge next_edge = graph->edge[i++];

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

```
        // If including this edge does not 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 edges of MST
    printf("Edges of Minimum Spanning Tree:\n");
    for (i = 0; i < e; ++i)
        printf("%d -- %d == %d\n", result[i].src, result[i].dest,
result[i].weight);

    free(subsets);
}

int main() {
    int V, E;
    printf("Enter number of vertices and edges: ");
    scanf("%d %d", &V, &E);

    struct Graph* graph = createGraph(V, E);

    printf("Enter edge details (source destination weight):\n");
    for (int i = 0; i < E; i++)
   scanf("%d %d %d", &graph->edge[i].src, &graph->edge[i].dest,
&graph->edge[i].weight);
    Kruskal(graph);

    return 0;
}
```

**Conclusion:** Kruskal's algorithm has been successfully implemented