ALGORITHM LABORATORY

ASSIGNMENT-4

PROBLEM STATEMENT: Implement Prim's and Kruskal's algorithm to find the minimum spanning tree of a weighted graph of n vertices.

ALGORITHM (Kruskal's):

a. Input Handling
   i. Read the number of vertices.
   ii. Provide options to either manually enter edges or generate a random graph.
b. Graph Representation
   i. Store edges as a list, where each edge consists of two endpoints and a weight.
c. Edge Input / Random Generation
   i. If manual input is selected, read **m** edges and store them.
   ii. If random generation is selected, create a fully connected graph with random weights.
d. Sorting Edges
   i. Arrange all edges in ascending order based on weight.
e. Disjoint Set Data Structure (Union-Find)
   i. Initialize a disjoint set with each vertex as its own parent.
   ii. Define functions to find the root of a set and to merge two sets.
f. Constructing MST Using Kruskal's Algorithm
   i. Initialize an empty list for the MST.
   ii. Traverse the sorted edge list and pick the smallest edge that does not form a cycle (i.e., belongs to different sets in the disjoint set).
   iii. Merge the sets and add the edge to the MST.
   iv. Keep track of the total weight of the MST.

g. Displaying Results

   i. Print the edges included in the MST along with their weights.

   ii. Display the total weight of the spanning tree.

PROGRAM CODE:

```
#include <bits/stdc++.h>
using namespace std;

struct Connection {
  int start, end, cost;
  bool operator<(const Connection &other) const { return cost < other.cost; }
};

class UnionFind {
 public:
  vector<int> leader, depth;
  UnionFind(int size) {
    leader.resize(size);
    depth.resize(size, 0);
    for (int i = 0; i < size; i++) leader[i] = i;
  }
  int locate(int node) {
    if (leader[node] != node) leader[node] = locate(leader[node]);
    return leader[node];
  }
  void merge(int a, int b) {
    int rootA = locate(a);
    int rootB = locate(b);
    if (rootA != rootB) {
      if (depth[rootA] > depth[rootB])
        leader[rootB] = rootA;
      else if (depth[rootA] < depth[rootB])
        leader[rootA] = rootB;
      else {
        leader[rootB] = rootA;
        depth[rootA]++;
      }
    }
  }
};

void displayLinks(vector<Connection> &links) {
  cout << "\nAvailable Links in Network:\n";
  for (auto &link : links) {
    cout << link.start << " - " << link.end << " : " << link.cost << endl;
  }
}

void mstKruskal(int nodes, vector<Connection> &links) {
  sort(links.begin(), links.end());
```

```
  UnionFind uf(nodes);
  vector<Connection> network;
  int netCost = 0;

  cout << "\nConnecting Nodes in MST:\n";
  for (auto &link : links) {
    if (uf.locate(link.start) != uf.locate(link.end)) {
      uf.merge(link.start, link.end);
      network.push_back(link);
      netCost += link.cost;
      cout << "Step " << network.size() << ": " << link.start << " - "
           << link.end << " (Cost: " << link.cost << ")\n";
    }
  }

  cout << "\nFinal Network Links:\n";
  for (auto &link : network) {
    cout << link.start << " - " << link.end << " : " << link.cost << endl;
  }
  cout << "Total Network Cost: " << netCost << endl;
}

void autoGenerate(int nodes, vector<Connection> &links) {
  srand(time(0));
  for (int i = 0; i < nodes; i++) {
    for (int j = i + 1; j < nodes; j++) {
      int cost = 1 + rand() % 100;
      links.push_back({i, j, cost});
    }
  }
  displayLinks(links);
}

int main() {
  int nodes, selection;
  cout << "Enter count of nodes: ";
  cin >> nodes;

  vector<Connection> links;

  cout << "Choose an input method:\n";
  cout << "1. Manual entry\n";
  cout << "2. Auto-generate links\n";
  cin >> selection;

  if (selection == 1) {
    int linkCount;
    cout << "Enter number of links: ";
    cin >> linkCount;
    cout << "Provide links (start end cost):\n";
    for (int i = 0; i < linkCount; i++) {
      int start, end, cost;
```

```
      cin >> start >> end >> cost;
      links.push_back({start, end, cost});
    }
  } else if (selection == 2) {
    autoGenerate(nodes, links);
  } else {
    cout << "Invalid input!\n";
    return 0;
  }

  mstKruskal(nodes, links);
  return 0;
}
```

TIME COMPLEXITY:

- Best Case: O(E logE)
- Average Case: O(E logE)
- Worst Case: O(E logE)

OUTPUT and PLOT:

Enter count of nodes: 5

Available Links in Network:

0 - 1 : 5

0 - 2 : 89

0 - 3 : 52

0 - 4 : 71

1 - 2 : 96

1 - 3 : 77

1 - 4 : 60

2 - 3 : 55

2 - 4 : 12

3 - 4 : 53

Connecting Nodes in MST:

Step 1: 0 - 1 (Cost: 5)

Step 2: 2 - 4 (Cost: 12)

Step 3: 0 - 3 (Cost: 52)

Step 4: 3 - 4 (Cost: 53)

Final Network Links:

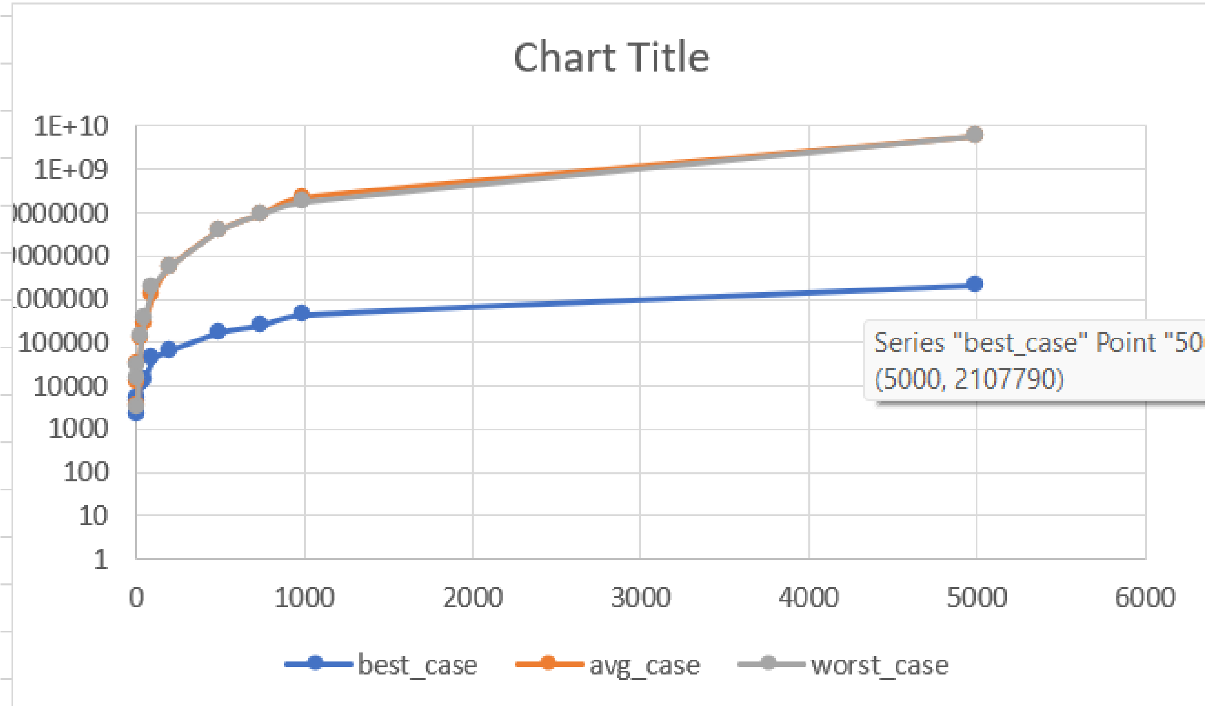
0 - 1 : 5

2 - 4 : 12

0 - 3 : 52

3 - 4 : 53

Total Network Cost: 122



Algorithm that builds multiple trees before forming a single spanning tree Krusikal's Algorithm

Algorithm that requires edge sorting as a necessary step : Krusikal's Algorithm

ALGORITHM (Prim's):

a. **Initialize Structures**:
   i. Create arrays to store the parent of each node , the minimum key value , and whether a node is included in MST .
   ii. Use a priority queue to always select the minimum edge.

b. **Start from an Initial Node**:
   i. Set the key value of the starting node (usually node 0) to 0.
   ii. Push (0, 0) into the priority queue (weight, node).

c. **Iterate Until MST is Formed**:
   i. Extract the minimum-weight node (u) from the priority queue.
   ii. If it is already in MST, continue to the next iteration.
   iii. Mark u as part of the MST.

d. **Update Neighboring Nodes**:
   i. For every adjacent node v of u:
      I. If v is not yet included in MST and adjMatrix[u][v] is less than its current key value, update:
         1. key[v] = adjMatrix[u][v]
         2. parent[v] = u
         3. Push (key[v], v) into the priority queue.

e. **Continue Until All Nodes Are Processed**:
   i. Repeat the above steps until all nodes are included in MST.

f. **Output the MST**:
   i. The parent array represents the edges in the MST.
   ii. The sum of key values gives the total weight of MST.

PROGRAM CODE:

```
#include <bits/stdc++.h>
using namespace std;

struct Connection {
    int start, end, cost;
    bool operator<(const Connection &other) const { return cost < other.cost; }
};

void displayLinks(vector<Connection> &links) {
    cout << "\nAvailable Links in Network:\n";
```

```
    for (auto &link : links) {
        cout << link.start << " - " << link.end << " : " << link.cost << endl;
    }
}

void mstPrim(int nodes, vector<vector<int>> &matrix) {
    vector<int> parent(nodes, -1);
    vector<int> minValue(nodes, INT_MAX);
    vector<bool> visited(nodes, false);
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>>
minHeap;

    minValue[0] = 0;
    minHeap.push({0, 0});

    vector<Connection> mstEdges;
    int totalCost = 0;

    cout << "\nConnecting Nodes in MST:\n";

    while (!minHeap.empty()) {
        int vertex = minHeap.top().second;
        int cost = minHeap.top().first;
        minHeap.pop();

        if (visited[vertex]) continue;
        visited[vertex] = true;
        totalCost += cost;

        if (parent[vertex] != -1) {
            mstEdges.push_back({parent[vertex], vertex, cost});
            cout << "Step " << mstEdges.size() << ": " << parent[vertex] << " - " << vertex
<< " (Cost: " << cost << ")\n";
        }

        for (int adj = 0; adj < nodes; adj++) {
            if (matrix[vertex][adj] && !visited[adj] && matrix[vertex][adj] <
minValue[adj]) {
                minValue[adj] = matrix[vertex][adj];
                parent[adj] = vertex;
                minHeap.push({minValue[adj], adj});
            }
        }
    }

    cout << "\nFinal Network Links:\n";
    for (auto &link : mstEdges) {
        cout << link.start << " - " << link.end << " : " << link.cost << endl;
    }
    cout << "Total Network Cost: " << totalCost << endl;
}
```

```cpp
void autoGenerate(int nodes, vector<vector<int>> &matrix) {
    srand(time(0));

    for (int i = 0; i < nodes; i++) {
        for (int j = i + 1; j < nodes; j++) {
            int cost = 1 + rand() % 100;
            matrix[i][j] = cost;
            matrix[j][i] = cost;
        }
    }
}

int main() {
    int nodes, selection;
    cout << "Enter count of nodes: ";
    cin >> nodes;

    vector<vector<int>> matrix(nodes, vector<int>(nodes, 0));

    cout << "Choose an input method:\n";
    cout << "1. Manual entry\n";
    cout << "2. Auto-generate links\n";
    cin >> selection;

    if (selection == 1) {
        int linkCount;
        cout << "Enter number of links: ";
        cin >> linkCount;
        cout << "Provide links (start end cost):\n";
        for (int i = 0; i < linkCount; i++) {
            int start, end, cost;
            cin >> start >> end >> cost;
            matrix[start][end] = cost;
            matrix[end][start] = cost;
        }
    } else if (selection == 2) {
        autoGenerate(nodes, matrix);
    } else {
        cout << "Invalid input!\n";
        return 0;
    }

    mstPrim(nodes, matrix);
    return 0;
}
```

TIME COMPLEXITY:

- Best Case: O(E logV)

- Average Case: O(E logV)
- Worst Case: O(E logV)

OUTPUT and PLOT:

Enter count of nodes: 5

Choose an input method:

1. Manual entry

2. Auto-generate links

Connecting Nodes in MST:

Step 1: 0 - 3 (Cost: 9)

Step 2: 3 - 1 (Cost: 9)

Step 3: 1 - 2 (Cost: 18)

Step 4: 3 - 4 (Cost: 39)

Final Network Links:

0 - 3 : 9

3 - 1 : 9

1 - 2 : 18

3 - 4 : 39

Total Network Cost: 75

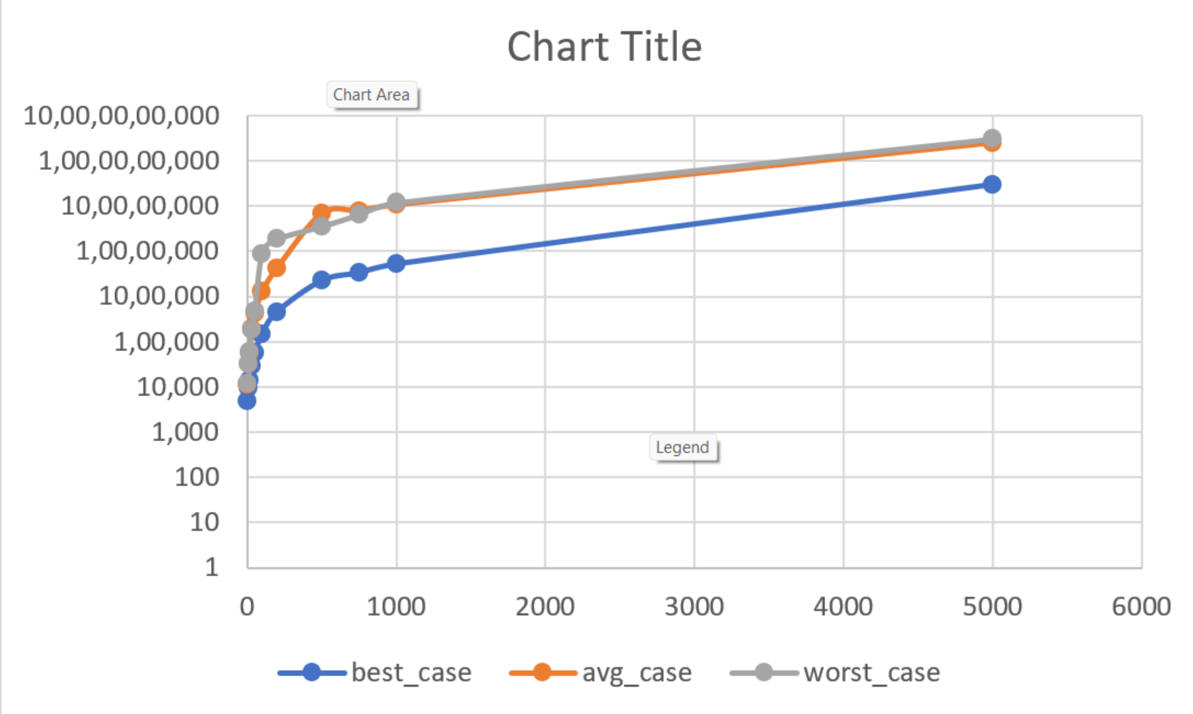


Chart Title
Chart Area
Legend
10,00,00,00,000
1,00,00,00,000
10,00,00,000
1,00,00,000
10,00,000
1,00,000
10,000
1,000
100
10
1
0
1000
2000
3000
4000
5000
6000
best_case
avg_case
worst_case

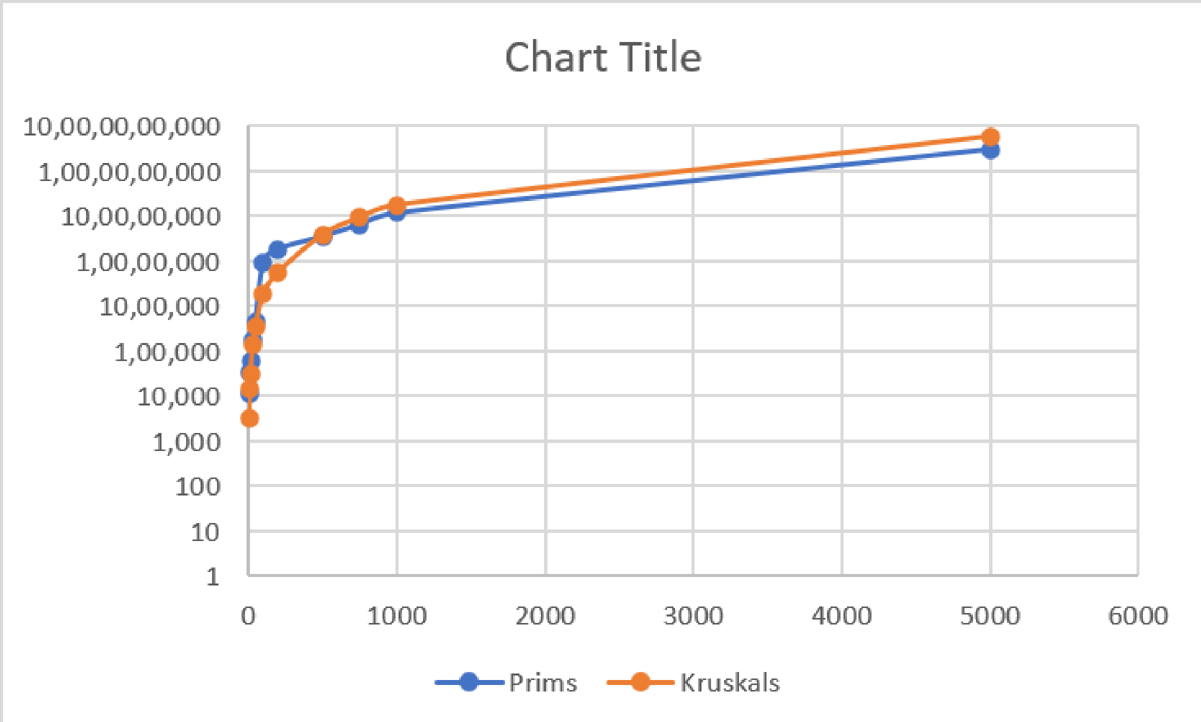


Chart Title
10,00,00,00,000
1,00,00,00,000
10,00,00,000
1,00,00,000
10,00,000
1,00,000
10,000
1,000
100
10
1
0
1000
2000
3000
4000
5000
6000
Prims
Kruskals