BHARATIYA VIDYA BHAVAN'S
SARDAR PATEL INSTITUTE OF TECHNOLOGY
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai – 400058-India
Department of Computer Engineering

Name	Balla Mahadev Shrikrishna
UID no.	2023300010
Experiment No.	4

AIM:	To implement Greedy Technique for MST algorithms using DSUF (Disjoint Set Union Find) Data Structures.
Program 1	
PROBLEM STATEMENT :	**Problem Definition & Assumptions –** Consider a connected, undirected graph G D (V,E), where V is the set of Vertices, E is the set of possible interconnections between edges, and for each edge (u,v) ∈ E, we have a weight w(u,v) specifying the cost to connect u and v. We then wish to find an acyclic subset T ⊆ E that connects all of the vertices and whose total weight w(T) is minimized [Read Chapter 23 of Coremen et al.] Since T is acyclic and connects all of the vertices, it must form a tree, which we call a spanning tree since it "spans" the graph G. We call the problem of determining the tree T the minimum-spanning-tree problem. In this experiment, two algorithms for solving the minimum spanning tree problem: Kruskal's algorithm and Prim's algorithm using greedy approach are considered. **Input –** 1) Fix random graphs of three types of sizes (e.g. V=8, V=15, V=20). input to two MST. **Output –** 1) Minimum spanning tree for all cases and both algorithms 2) Draw a plot of time required over three types of graphs for all both MST algorithms.
PROGRAM (mst.cpp):	#include <bits/stdc++.h> using namespace std; using namespace chrono; // Structure to represent an edge

```
struct Edge {
        int u, v, weight;
};

// Comparator to sort edges by weight
bool compareEdges(Edge a, Edge b) {
        return a.weight < b.weight;
}

// Disjoint Set Union-Find (DSUF) Data Structure
class DSUF {
private:
        vector<int> parent; // Stores the parent of each element
        vector<int> rank;   // Stores the rank of each set (for union by rank)

public:
        // Constructor to initialize the DSUF structure
        DSUF(int n) {
        parent.resize(n);
        rank.resize(n, 1); // Initialize rank of each set to 1
        for (int i = 0; i < n; ++i) {
        parent[i] = i; // Each element is its own parent initially
        }
        }

        // Find operation with path compression
        int find(int x) {
        if (parent[x] != x) {
        parent[x] = find(parent[x]); // Path compression
        }
        return parent[x];
        }

        // Union operation with union by rank
        void unionSets(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
```

```
        if (rootX != rootY) {
        // Union by rank: attach the smaller tree to the larger tree
        if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
        }
        else if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
        }
        else {
                parent[rootY] = rootX;
                rank[rootX]++; // Increase rank if ranks are equal
        }
        }
        }

        // Utility function to check if two elements are in the same set
        bool isConnected(int x, int y) {
        return find(x) == find(y);
        }
};

// Kruskal's Algorithm using DSUF
vector<Edge> kruskalMST(vector<Edge>& edges, int V) {
        vector<Edge> MST; // To store the MST
        DSUF dsu(V); // Initialize DSUF

        // Sort edges by weight
        sort(edges.begin(), edges.end(), compareEdges);

        // Iterate through sorted edges
        for (Edge e : edges) {
        int u = e.u;
        int v = e.v;
        if (!dsu.isConnected(u, v)) { // If u and v are not in the same set
        MST.push_back(e); // Add edge to MST
        dsu.unionSets(u, v); // Merge sets
        }
        }
```

```
    return MST;
}

// Prim's Algorithm using Array for extractMin
vector<Edge> primArray(vector<vector<pair<int, int>>>& graph, int V) {
    vector<Edge> MST; // To store the MST
    vector<bool> inMST(V, false); // Track vertices in MST
    vector<int> key(V, INT_MAX); // Key values for each vertex
    vector<int> parent(V, -1); // To store the MST

    key[0] = 0; // Start with vertex 0

    for (int count = 0; count < V - 1; count++) {
    // Find the vertex with the minimum key (brute-force)
    int u = -1;
    for (int v = 0; v < V; v++) {
    if (!inMST[v] && (u == -1 || key[v] < key[u])) {
        u = v;
    }
    }

    inMST[u] = true; // Add u to MST

    // Update keys of adjacent vertices
    for (auto& neighbor : graph[u]) {
    int v = neighbor.first;
    int weight = neighbor.second;
    if (!inMST[v] && weight < key[v]) {
        key[v] = weight;
        parent[v] = u;
    }
    }
    }

    // Construct MST edges
    for (int i = 1; i < V; i++) {
    MST.push_back({ parent[i], i, key[i] });
```

```
    }

    return MST;
}

// Min-Heap implementation for extractMin in Prim's Algorithm
class MinHeap {
private:
    vector<pair<int, int>> heap; // (key, vertex)

    void heapify(int i) {
    int smallest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < heap.size() && heap[left].first < heap[smallest].first) {
    smallest = left;
    }
    if (right < heap.size() && heap[right].first < heap[smallest].first) {
    smallest = right;
    }

    if (smallest != i) {
    swap(heap[i], heap[smallest]);
    heapify(smallest);
    }
    }

public:
    void push(pair<int, int> p) {
    heap.push_back(p);
    int i = heap.size() - 1;
    while (i > 0 && heap[(i - 1) / 2].first > heap[i].first) {
    swap(heap[i], heap[(i - 1) / 2]);
    i = (i - 1) / 2;
    }
    }
```

```
        pair<int, int> pop() {
        pair<int, int> root = heap[0];
        heap[0] = heap.back();
        heap.pop_back();
        heapify(0);
        return root;
        }

        bool empty() {
        return heap.empty();
        }
};

// Prim's Algorithm using Min-Heap for extractMin
vector<Edge> primHeap(vector<vector<pair<int, int>>>& graph, int V) {
        vector<Edge> MST; // To store the MST
        vector<bool> inMST(V, false); // Track vertices in MST
        vector<int> key(V, INT_MAX); // Key values for each vertex
        vector<int> parent(V, -1); // To store the MST

        MinHeap pq;
        pq.push({ 0, 0 }); // Start with vertex 0
        key[0] = 0;

        while (!pq.empty()) {
        int u = pq.pop().second; // Extract vertex with minimum key
        inMST[u] = true; // Add u to MST

        // Update keys of adjacent vertices
        for (auto& neighbor : graph[u]) {
        int v = neighbor.first;
        int weight = neighbor.second;
        if (!inMST[v] && weight < key[v]) {
                key[v] = weight;
                parent[v] = u;
                pq.push({ key[v], v });
        }
        }
```

```
    }

    // Construct MST edges
    for (int i = 1; i < V; i++) {
    MST.push_back({ parent[i], i, key[i] });
    }

    return MST;
}

int main() {
    // Hard-coded graphs

    // Graph 1 (V=8)
    int V1 = 8;
    vector<vector<pair<int, int>>> graph1 = {
    {{1, 4}, {7, 8}},        // 0
    {{0, 4}, {2, 8}, {7, 11}}, // 1
    {{1, 8}, {3, 7}, {8, 2}},  // 2
    {{2, 7}, {4, 9}, {5, 14}}, // 3
    {{3, 9}, {5, 10}},       // 4
    {{3, 14}, {4, 10}, {6, 2}},// 5
    {{5, 2}, {7, 1}, {8, 6}},  // 6
    {{0, 8}, {1, 11}, {6, 1}}, // 7
    {{2, 2}, {6, 6}}         // 8
    };

    // Graph 2 (V=15)
    int V2 = 15;
    vector<vector<pair<int, int>>> graph2 = {
    {{1, 2}, {2, 4}},        // 0
    {{0, 2}, {2, 1}, {3, 7}},  // 1
    {{0, 4}, {1, 1}, {3, 3}},  // 2
    {{1, 7}, {2, 3}, {4, 5}},  // 3
    {{3, 5}, {5, 2}},        // 4
    {{4, 2}, {6, 3}},        // 5
    {{5, 3}, {7, 1}},        // 6
    {{6, 1}, {8, 4}},        // 7
```

```
{{7, 4}, {9, 6}},        // 8
{{8, 6}, {10, 2}},       // 9
{{9, 2}, {11, 5}},       // 10
{{10, 5}, {12, 3}},      // 11
{{11, 3}, {13, 4}},      // 12
{{12, 4}, {14, 1}},      // 13
{{13, 1}}                // 14
};

// Graph 3 (V=20)
int V3 = 20;
vector<vector<pair<int, int>>> graph3 = {
{{1, 1}, {2, 3}},        // 0
{{0, 1}, {2, 2}, {3, 4}},  // 1
{{0, 3}, {1, 2}, {3, 5}},  // 2
{{1, 4}, {2, 5}, {4, 2}},  // 3
{{3, 2}, {5, 1}},        // 4
{{4, 1}, {6, 3}},        // 5
{{5, 3}, {7, 4}},        // 6
{{6, 4}, {8, 2}},        // 7
{{7, 2}, {9, 5}},        // 8
{{8, 5}, {10, 1}},       // 9
{{9, 1}, {11, 3}},       // 10
{{10, 3}, {12, 4}},      // 11
{{11, 4}, {13, 2}},      // 12
{{12, 2}, {14, 5}},      // 13
{{13, 5}, {15, 1}},      // 14
{{14, 1}, {16, 3}},      // 15
{{15, 3}, {17, 4}},      // 16
{{16, 4}, {18, 2}},      // 17
{{17, 2}, {19, 5}},      // 18
{{18, 5}}                // 19
};

// Output file for time results (CSV)
ofstream timeFile("time_results.csv");
timeFile << "Graph Size,Kruskal Time (ms),Prim Array Time (ms),Prim Min-Heap Time (ms)\n";
```

```
        // Run Kruskal's and Prim's Algorithms for each graph
        vector<vector<vector<pair<int, int>>>> graphs = { graph1, graph2,
graph3 };
        vector<int> sizes = { V1, V2, V3 };

        for (int i = 0; i < graphs.size(); i++) {
        auto graph = graphs[i];
        int V = sizes[i];

        // Convert graph to edge list for Kruskal's Algorithm
        vector<Edge> edges;
        for (int u = 0; u < V; u++) {
        for (auto& neighbor : graph[u]) {
                int v = neighbor.first;
                int weight = neighbor.second;
                edges.push_back({ u, v, weight });
        }
        }

        // Kruskal's Algorithm
        auto start = high_resolution_clock::now();
        auto mstKruskal = kruskalMST(edges, V);
        auto stop = high_resolution_clock::now();
        double durationKruskal = duration_cast<microseconds>(stop -
start).count() / 1000.0;

        // Prim's Algorithm using Array
        start = high_resolution_clock::now();
        auto mstPrimArray = primArray(graph, V);
        stop = high_resolution_clock::now();
        double durationPrimArray = duration_cast<microseconds>(stop -
start).count() / 1000.0;

        // Prim's Algorithm using Min-Heap
        start = high_resolution_clock::now();
        auto mstPrimHeap = primHeap(graph, V);
        stop = high_resolution_clock::now();
```

```
        double durationPrimHeap = duration_cast<microseconds>(stop -
start).count() / 1000.0;

        // Write time results to CSV file
        timeFile << V << "," << durationKruskal << "," <<
durationPrimArray << "," << durationPrimHeap << "\n";

        // Print MST results to terminal
        cout << "Graph Size: " << V << "\n";
        cout << "Kruskal's Algorithm:\n";
        for (Edge e : mstKruskal) {
        cout << e.u << " - " << e.v << " : " << e.weight << "\n";
        }
        cout << "Time: " << durationKruskal << " ms\n\n";

        cout << "Prim's Algorithm (Array):\n";
        for (Edge e : mstPrimArray) {
        cout << e.u << " - " << e.v << " : " << e.weight << "\n";
        }
        cout << "Time: " << durationPrimArray << " ms\n\n";

        cout << "Prim's Algorithm (Min-Heap):\n";
        for (Edge e : mstPrimHeap) {
        cout << e.u << " - " << e.v << " : " << e.weight << "\n";
        }
        cout << "Time: " << durationPrimHeap << " ms\n\n";
        }

        timeFile.close();

        return 0;
}
```

plot.ipynb:	import pandas as pd
	import matplotlib.pyplot as plt
	
	# Read the CSV file
	df = pd.read_csv("time_results.csv")
	
	# Extract data
	graph_sizes = df["Graph Size"]
	kruskal_times = df["Kruskal Time (ms)"]
	prim_array_times = df["Prim Array Time (ms)"]
	prim_heap_times = df["Prim Min-Heap Time (ms)"]
	
	# Plot the data
	plt.figure(figsize=(10, 6))
	plt.plot(graph_sizes, kruskal_times, marker='o', label="Kruskal's Algorithm")
	plt.plot(graph_sizes, prim_array_times, marker='s', label="Prim's Algorithm (Array)")
	plt.plot(graph_sizes, prim_heap_times, marker='^', label="Prim's Algorithm (Min-Heap)")
	
	# Add labels and title
	plt.xlabel("Graph Size (Number of Vertices)")
	plt.ylabel("Time (ms)")
	plt.title("Time Complexity Comparison of MST Algorithms")
	plt.legend()
	plt.grid(True)
	
	# Show the plot
	plt.show()

RESULT:

```
mahadev@mahadev-Inspiron-15-3520:~/Desktop/Mahadev/SE/Sem4/DAA/Lab/Lab Sessions/exp4$ g++ mst.cpp
^[[Amahadev@mahadev-Inspiron-15-3520:~/Desktop/Mahadev/SE/Sem4/DAA/Lab/Lab Sessions/exp4$ ./a.out
Graph Size: 8
Kruskal's Algorithm:
7 - 6 : 1
2 - 8 : 2
6 - 5 : 2
0 - 1 : 4
6 - 8 : 6
2 - 3 : 7
4 - 3 : 9
Time: 0.017 ms

Prim's Algorithm (Array):
0 - 1 : 4
1 - 2 : 8
2 - 3 : 7
3 - 4 : 9
6 - 5 : 2
7 - 6 : 1
0 - 7 : 8
Time: 0.012 ms

Prim's Algorithm (Min-Heap):
0 - 1 : 4
1 - 2 : 8
2 - 3 : 7
3 - 4 : 9
6 - 5 : 2
7 - 6 : 1
0 - 7 : 8
Time: 0.013 ms

Graph Size: 15
Kruskal's Algorithm:
6 - 7 : 1
14 - 13 : 1
2 - 1 : 1
5 - 4 : 2
9 - 10 : 2
0 - 1 : 2
5 - 6 : 3
3 - 2 : 3
12 - 11 : 3
12 - 13 : 4
7 - 8 : 4
10 - 11 : 5
4 - 3 : 5
9 - 8 : 6
Time: 0.026 ms

Prim's Algorithm (Array):
0 - 1 : 2
1 - 2 : 1
2 - 3 : 3
```

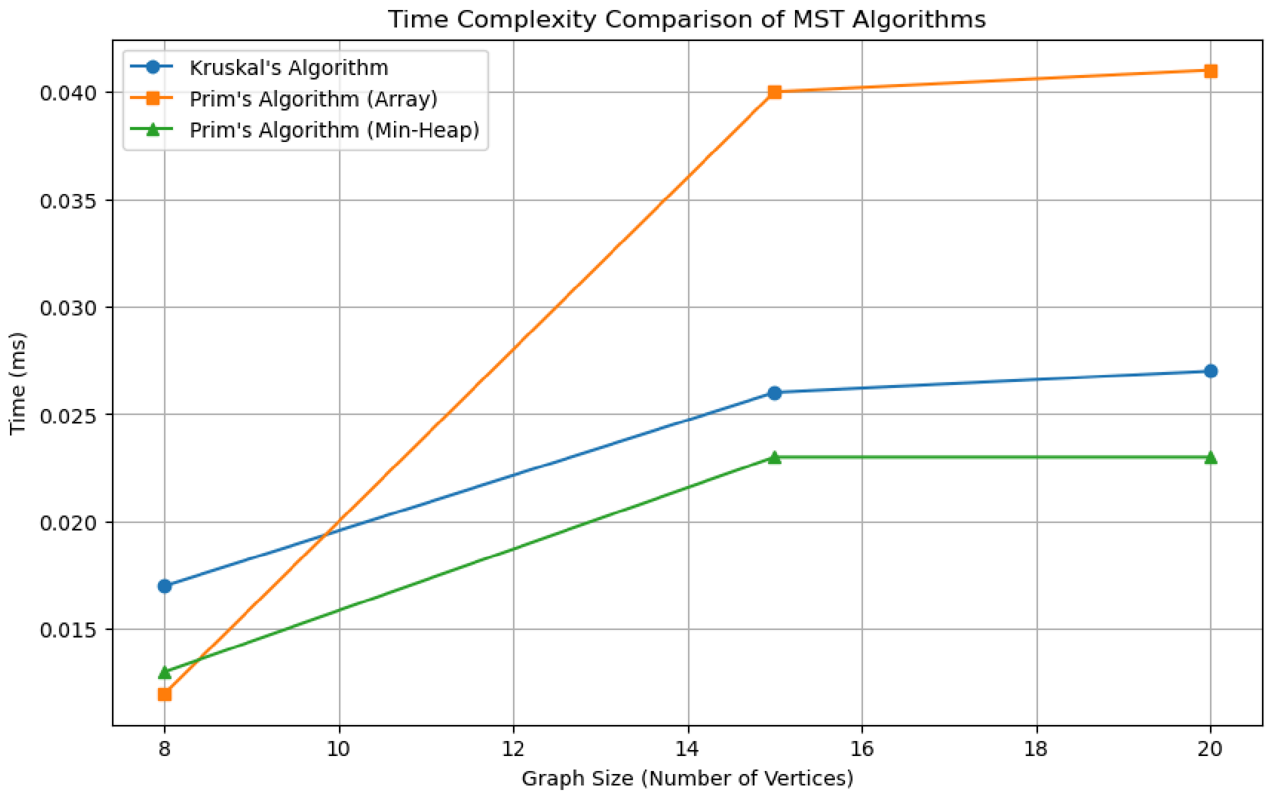
```
5 - 4 : 1
4 - 3 : 2
17 - 18 : 2
8 - 7 : 2
12 - 13 : 2
2 - 1 : 2
16 - 15 : 3
11 - 10 : 3
6 - 5 : 3
7 - 6 : 4
11 - 12 : 4
3 - 1 : 4
17 - 16 : 4
18 - 19 : 5
14 - 13 : 5
9 - 8 : 5
Time: 0.027 ms

Prim's Algorithm (Array):
0 - 1 : 1
1 - 2 : 2
1 - 3 : 4
3 - 4 : 2
4 - 5 : 1
5 - 6 : 3
6 - 7 : 4
7 - 8 : 2
8 - 9 : 5
9 - 10 : 1
10 - 11 : 3
11 - 12 : 4
12 - 13 : 2
13 - 14 : 5
14 - 15 : 1
15 - 16 : 3
16 - 17 : 4
17 - 18 : 2
18 - 19 : 5
Time: 0.041 ms

Prim's Algorithm (Min-Heap):
0 - 1 : 1
1 - 2 : 2
1 - 3 : 4
3 - 4 : 2
4 - 5 : 1
5 - 6 : 3
6 - 7 : 4
7 - 8 : 2
8 - 9 : 5
9 - 10 : 1
10 - 11 : 3
11 - 12 : 4
12 - 13 : 2
13 - 14 : 5
14 - 15 : 1
15 - 16 : 3
16 - 17 : 4
17 - 18 : 2
18 - 19 : 5
Time: 0.023 ms

mahadev@mahadev-Inspiron-15-3520:~/Desktop/Mahadev/SE/Sem4/DAA/Lab/Lab Sessions/exp4$
```

OUTPUT:

csv -

Graph Size	Kruskal Time (ms)	Prim Array Time (ms)	Prim Min-Heap Time (ms)
8	0.017	0.012	0.013
15	0.026	0.04	0.023
20	0.027	0.041	0.023

Plot -



CONCLUSION:

Name: Balla Mahadev Shrikrishna
UID: 2023300010
Div.: A
Batch: A

Exp - 4

* Analysis of Time & Space Complexities

1) Kruskal's Algo.

- Sorting edges: $O(|E| \lg(|E|))$
- DSUF operations: $O(|V|)$ - makeSet, $O(|E|)$ - find, $O(|V|)$ - union

→ $O(|V| + |E|)$ operations $\Rightarrow O((|V|+|E|)\alpha(|V|))$ time
$|E| \geq |V| - 1 \Rightarrow O(|E|\,\alpha(|V|))$ time
$\because \alpha(n)$ can be upper bounded by ht. of tree,
$\alpha(|V|) = O(\lg|V|) = O(\lg|E|)$

- Total Running Time: $O(E \lg(E))$ or $O(E \lg(V))$
- Space Complexity: $O(V+E)$ for storing edges & DSUF data structure.

2) Prim's Algorithm

i) Array implementatn:

- $O(V)$ - setting keys to ∞
- $O(V^2)$ - extractMin (Iterates over all vertices to find min. key in each step
- $O(E)$ - weight updatn
- Space Complexity - $O(V)$ for storing keys, parents & inMST arr.

ii) Min-Heap implementatn:

- $O(V)$ - setting keys to ∞
- $O(V \lg(V))$ - extractMin
- $O(E \lg(V))$ - wt. updatn & $O(E)$ times decreaseKey.
- Space Complexity - $O(V+E)$ for storing the graph & min-heap.

	* Conclusion: · Kruskal's algo. is simple & eff. for sparse graphs but becomes slower for dense graphs due to the sorting step. · Prim's algo (Min-Heap) is the most eff. for larger graphs due to its logarithmic time complexity for extractMin op.