**Report**

**Summary:**

**A Minimum Spanning Tree (MST):**

An MST refers to a subset of edges within an undirected, connected graph, which links all the vertices while keeping the total edge weight as low as possible. Essentially, it's a tree structure that encompasses all the graph's vertices while minimizing the combined weight of its edges. MSTs find applications in various fields, including network design, circuit arrangement, and data clustering.

**Prim's Algorithm:**

Prim's algorithm stands as a greedy approach employed to identify the minimum spanning tree in a graph. It initializes from an arbitrary vertex and consistently selects the edge with the smallest weight, linking a vertex within the growing tree to a vertex outside it. The process persists until all vertices become part of the MST. Prim's algorithm can be implemented efficiently using either a priority queue or a Fibonacci heap to select the edge with the minimum weight.

**Kruskal's Algorithm:**

Kruskal's algorithm serves as another greedy approach used to find the minimum spanning tree of a graph. Unlike Prim's algorithm, which expands the MST from a single vertex, Kruskal's algorithm commences with an empty set of edges and incrementally adds edges with the smallest weight, ensuring that no cycles are created within the spanning tree. To achieve this, it employs a disjoint-set data structure (also referred to as a union-find data structure) to efficiently verify the absence of cycles.

**Data structures used:**

**PriorityQueue:**

PriorityQueue is used in Prim's algorithm to efficiently select edges with the minimum weight. The priority queue ensures that the edge with the smallest weight is always selected first during the MST construction.

**HashSet:**

HashSet is used to keep track of visited vertices in Prim's algorithm. It ensures that each vertex is added to the minimum spanning tree only once.

**ArrayList:**

Array Lists are used in various places to store collections of objects. For example, neighbors in the Vertex class stores a list of neighboring edges, and adjacencyList in the Graph class stores adjacency information.

**Additional User defined Structures:**

**1. Edge Class:**

The Edge class represents an edge in a graph. It contains information about its source vertex, destination vertex, and weight. Edges are used to connect vertices in the graph, and they are an essential part of both Prim's and Kruskal's algorithms for finding minimum spanning trees.

**2. Vertex Class:**

The Vertex class represents a vertex in a graph. It contains information about its unique identifier and a list of neighboring edges. Vertices are used to build the graph structure and store information about the vertices and their connections.

**3. DisjointSet Class:**

The DisjointSet class represents a disjoint-set data structure, also known as a union-find data structure. It is used in Kruskal's algorithm to keep track of sets of vertices and efficiently determine whether adding an edge creates a cycle in the minimum spanning tree.

**4. Graph Class:**

The Graph class represents a graph data structure using an adjacency list representation. It contains a list of vertices and an adjacency list that stores information about the edges between vertices. This class provides methods for adding edges to the graph and computing minimum spanning trees using both Prim's and Kruskal's algorithms.

**Time and space complexity analysis:**

**Time Complexity analysis:**

**The time complexity of Prim's algorithm for finding the Minimum Spanning Tree (MST) in a graph is as follows:**

1. While Loop: The main part of the algorithm is the while loop, which iterates until all vertices are visited. In the worst case, the loop runs for all vertices, so it has a time complexity of O(V).

2. Priority Queue Operations: Within the while loop, you perform operations on the priority queue (min-heap) to extract the minimum-weight edge. Each insertion and extraction operation on the priority queue takes O(log V) time. In the worst case, you perform these operations for all edges (E) in the graph, so the total time spent on priority queue operations is O(E \* log V).

3. Checking If a Vertex is Visited: In each iteration of the while loop, you check whether a vertex is already visited, which takes O(1) time.

4. Neighbor Iteration: You iterate through the neighbors of the current vertex. In the worst case, this involves iterating through all edges in the graph (E).

Combining these factors, the overall time complexity of Prim's algorithm is:

O(V) + O(V) + O(E \* log V) + O(1) + O(E) = O(E \* log V)

So, the time complexity of primsMinimumSpanningTree() using a binary min heap is O(E \* log V), where E is the number of edges, and V is the number of vertices in the graph.

**The time complexity of krushkal's algorithm for finding the Minimum Spanning Tree (MST) in a graph is as follows:**

1. Constructing the Priority Queue (minHeap):

- In the first loop that iterates over all vertices and their neighbors, we add all edges to the priority queue. This loop runs for all vertices, and for each vertex, it adds its neighbors' edges to the priority queue.

- The number of edges in the priority queue is typically proportional to the number of edges in the graph (E), as each edge is added exactly once.

- Therefore, constructing the priority queue has a time complexity of O(E \* log(E)) due to the insertion of E edges into the priority queue.

2. While Loop:

- The while loop runs until edgesAccepted becomes equal to (vertices.size() - 1), where vertices.size() represents the number of vertices in the graph.

- In the worst case, where the loop runs until it constructs a minimum spanning tree with (vertices.size() - 1) edges, it iterates through the entire list of edges exactly once.

- Inside the loop, the find and union operations from the DisjointSet class take nearly constant time.

- Adding an edge to the mst list is a constant-time operation.

- Polling the minimum-weight edge from the priority queue takes O(log(E)) time in each iteration.

3. Overall Time Complexity:

- The dominant factor in the time complexity of this method is the construction of the priority queue, which is O(E \* log(E)).

- The while loop iterates through at most E edges, and the operations within the loop are constant time or logarithmic time.

- Therefore, the overall time complexity of the krushkalsMinimumSpanningTree () is O(E \* log(E)).

**2. Space Complexity:**

**Prim’s Space Complexity analysis:**

1**. Priority Queue:** The space required for the priority queue is O(E) because, in the worst case, all edges are added to the priority queue.

2**. Visited Set:** We maintain a set of visited vertices, which takes O(V) space.

3. **MST Edges:** The space required to store the MST edges depends on the size of the MST, which can have at most V-1 edges.

Therefore, the space complexity of Prim's algorithm is O(V + E), where V is the number of vertices and E is the number of edges in the graph. In most cases, the space complexity simplifies to O(E) because E can be much larger than V in dense graphs.

**Krushkals Space complexity analysis:**

1. **DisjointSet (ds):**

- The DisjointSet object ds requires space to store the parent array, which has a size equal to the number of vertices in the graph (vertices.size()).

- Therefore, the space complexity of ds is O(V), where V is the number of vertices in the graph.

2. **Priority Queue (minHeap):**

- The minHeap priority queue stores edges from the graph. In the worst case, it can contain all edges of the graph, which is E edges.

- Therefore, the space complexity of minHeap is O(E).

3. **ArrayList (mst):**

- The mst ArrayList is used to store the edges of the minimum spanning tree. In the worst case, it can contain (vertices.size() - 1) edges, as this is the maximum number of edges in a minimum spanning tree.

- Therefore, the space complexity of mst is O(V).

Therefore, the overall space complexity is O(V + E), where V is the number of vertices, and E is the number of edges in the graph.

**Summary of important methods of Graph class:**

**1. Edge Class:**

**- Edge(Vertex source, Vertex destination, double weight):** This function constructs an edge by specifying its source, destination, and weight.

**- compareTo(Edge other):** It compares this edge to another edge based on their respective weights.

**2. Vertex Class:**

**- Vertex(int vertex):** This constructor creates a vertex with a unique identifier.

**3. DisjointSet Class:**

**- DisjointSet(int size):** This constructor creates a disjoint-set data structure of a specified size.

- find(int element): It identifies the representative (root) element of the set to which a specified element belongs.

**- union(int x, int y):** This function combines (unions) two sets by designating one as the parent of the other.

**4. Graph Class:**

**- Graph(int numVertices):** This constructor establishes a graph with a given number of vertices.

- addEdge(int srcVertex, int dstVertex, double weight): It adds an edge between two vertices with provided source, destination, and weight.

- **primsMinimumSpanningTree():** This method calculates the minimum spanning tree of the graph using Prim's algorithm and returns a list of Edges that represents the MST.

- **krushkalsMinimumSpanningTree():** It computes the minimum spanning tree of the graph using Kruskal's algorithm and returns a list of Edges representing the MST.

**5. minSPT Class:**

**- readFile(String fileName):** This function reads graph data from a file and provides an ArrayList of Graph objects.

**- main(String[] args):** The main method is responsible for reading a file, finding minimum spanning trees of the graphs using Kruskal's and Prim's algorithms, and displaying the results.