PROJECT 4 (MINIMUM SPANNING TREE)

Implementation And Comparison of Minimum Spanning Tree Algorithm

INTRODUCTION

A Minimum Spanning Tree is a subgraph of a connected, weighted graph that connects all the vertices together with the minimum possible total edge weight. The significance of Minimum Spanning Trees is their applications including network design, clustering, and resource allocation. Consequently, understanding and implementing efficient algorithms to find MSTs are essential skills for any computer scientist or engineer.

In this project, I implemented and compared two MST algorithms for finding Minimum Spanning Trees:

- Kruskal's algorithm
- Prim's algorithm.

Kruskal's algorithm operates by greedily selecting the smallest edge not yet included in the MST until all vertices are connected. On the other hand, **Prim's algorithm** starts from an random vertex and grows the MST incrementally by greedily selecting the smallest edge adjacent to the current MST.

The objective of this project is two-fold: firstly, to implement both Kruskal's and Prim's algorithms for finding Minimum Spanning Trees, and secondly, to compare their performance in terms of efficiency and effectiveness.

METHODOLOGY

To accomplish the objectives, we follow:

Algorithm Implementation: We begin by implementing both Kruskal's and Prim's algorithms JavaScript. The implementation will involve translating the pseudocode of the algorithms into executable code.

Testing and Debugging: Once the implementations are complete, we test the algorithms using a variety of input graphs with different sizes and characteristics. Testing involves verifying that the algorithms produce correct Minimum Spanning Trees for various input scenarios and density of the graph.

Performance Evaluation: After confirming the correctness of the implementations, we proceed to evaluate the performance of Kruskal's and Prim's algorithms. This evaluation includes measuring the execution time of each algorithm for graphs of varying sizes and densities.

Comparative Analysis: Finally, we compare the performance of Kruskal's and Prim's algorithms based on their execution times and the quality of the generated Minimum Spanning Trees. This comparative analysis aims to highlight the strengths and weaknesses of each algorithm and provide insights into their suitability for different applications.

DATA STRUCTURES

Graph Object

Graph Object: The core data structure utilized for representing the connected graph is the custom graph object. This graph object generates a random connected graph using Erdos Renyi model. This object enables us to store the vertices and edges of the graph efficiently.

- **Adjacency List:** To optimize certain graph operations, such as finding neighbors of a vertex and formatting the adjacency list, we use an adjacency list representation. The adjacency list is a mapping of each vertex to a list of its adjacent vertices along with the corresponding edge weights. This representation allows for fast access to neighbors and efficient storage of graph connectivity information.
- **Compatible Nodes and Edges:** Additionally, we transform the graph's nodes and edges into compatible formats for visualization purposes. Each node is represented as an object containing an identifier and a name, while each edge is represented as an object containing source, target, weight, and name attributes. This transformation facilitates seamless integration with React-based visualization libraries for displaying the graph in a user-friendly manner.

DisjointSet Class

The DisjointSet class provides functionalities for initializing disjoint sets, finding the root of a set containing a specified node, and performing union operations to merge sets. Below are the key components of the DisjointSet class:

- **Initialization:** Upon instantiation, the DisjointSet class initializes with a specified number of nodes. It creates a parent array to track the parent of each node initially. Each node is its own parent, indicating that it belongs to its own disjoint set.
- **Find Operation:** The find method takes a node as input and recursively finds the root of the set containing the node. Along with finding the root, it performs path compression to optimize future find operations by directly connecting each node to its root during traversal.
- **Union Operation:** The union method merges the sets containing two specified nodes. It first finds the roots of the sets containing the nodes and then merges the smaller set into the larger set. This merging strategy helps optimize the depth of the tree representing the disjoint sets, leading to efficient union operations.

ALGORITHM

Kruskal's Algorithm

Kruskal's algorithm is a greedy algorithm that aims to find a Minimum Spanning Tree (MST) for a given connected, weighted graph. The main components of Kruskal's algorithm can be outlined as follows:

1. **Sorting Edges by Weight:** The algorithm begins by sorting all the edges of the graph in non-decreasing order of their weights. This step ensures that we can efficiently select edges with the smallest weight during subsequent iterations.
2. **Initializing Forest:** Initially, each vertex of the graph is considered as a separate component in the MST. This forms a forest of trees, where each tree contains a single vertex.
3. **Iterative Edge Selection:** Starting from the edge with the smallest weight, the algorithm iterates through the sorted list of edges. For each edge, it checks whether including it in the MST would create a cycle. If adding the edge does not form a cycle, it is added to the MST, and the two components it connects are merged into a single component.
4. **Termination Condition:** The algorithm continues this process until all vertices are connected, i.e., until there are n-1 edges in the MST, where n is the number of vertices in the graph.
5. **Output:** The output of Kruskal's algorithm is the set of edges that form the Minimum Spanning Tree of the given graph.

Prim's Algorithm

Prim's algorithm is another greedy approach for finding a Minimum Spanning Tree (MST) in a connected, weighted graph. Unlike Kruskal's algorithm, Prim's algorithm grows the MST incrementally from an random starting vertex. The main components of Prim's algorithm is:

1. **Initialization:** The algorithm begins by selecting an arbitrary vertex as the starting point for the MST. This vertex becomes the first node in the MST.
2. **Maintaining a Priority Queue:** Prim's algorithm maintains a priority queue (usually implemented using a heap) of candidate edges. Initially, this priority queue contains all edges incident to the selected starting vertex.
3. **Iterative Edge Selection:** At each iteration, the algorithm selects the edge with the smallest weight from the priority queue. If adding this edge to the MST does not create a cycle, it is included in the MST, and the vertex it connects is added to the MST as well.
4. **Updating the Priority Queue:** After adding a new vertex to the MST, the algorithm updates the priority queue by adding all edges incident to the newly added vertex that connect to vertices not already in the MST.
5. **Termination Condition:** The algorithm continues this process until all vertices are included in the MST.
6. **Output:** The output of Prim's algorithm is the set of edges that form the Minimum Spanning Tree of the given graph.

DESIGN OF THE USER INTERFACE

For this project, the user interface (UI) is developed using React.js, a popular JavaScript library for building interactive user interfaces. The UI aims to provide a user-friendly experience for interacting with and visualizing the Minimum Spanning Tree (MST) algorithms.

Input Field for Number of Nodes:

The UI features an input field where the user can input the number of nodes for the graph. This input serves as the basis for generating a random graph for algorithm execution and visualization.

Table for Comparing Algorithm Runtimes:

A table is incorporated into the UI to display the runtime of each algorithm for various input sizes. The table provides a comparative analysis of the efficiency of Kruskal's and Prim's algorithms in finding Minimum Spanning Trees.

Graph Visualizer:

Utilizing the Graphology library, the UI includes a graph visualizer component for displaying the generated graph and the resulting Minimum Spanning Tree. The visualizer offers an intuitive representation of the graph structure, allowing users to observe the algorithm's execution and the resulting MST visually.

User Interaction:

Users can input the number of nodes, triggering the generation of a random graph. Upon executing Kruskal's and Prim's algorithms, the runtime results are displayed in the table, enabling users to compare the efficiency of both algorithms. Additionally, users can visualize the generated graph and its corresponding Minimum Spanning Tree using the graph visualizer component.

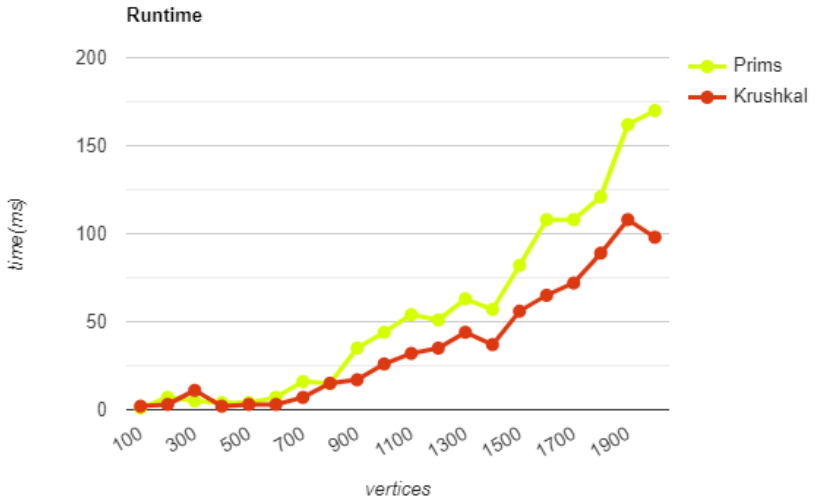
Responsive Design:

The UI is designed to be responsive, ensuring optimal viewing and interaction experiences across various devices and screen sizes. Whether accessed on desktop or mobile devices, users can easily interact with the application and visualize the algorithm results effectively.

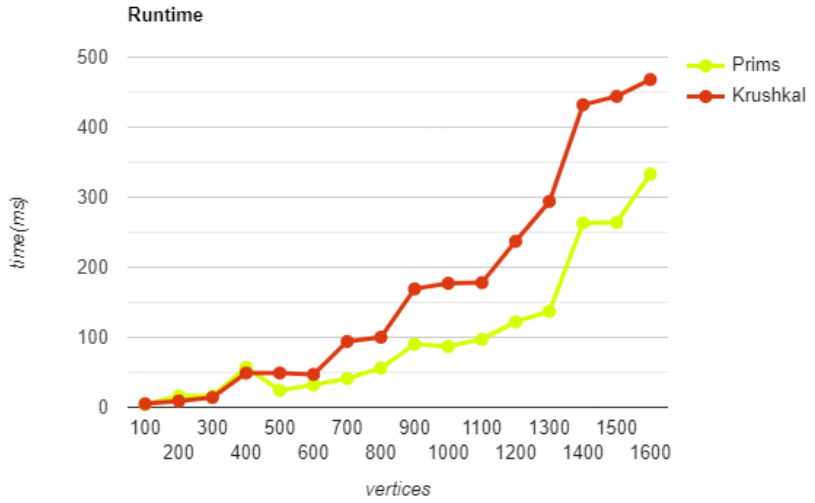
EXPERIMENTAL RESULTS

The experimental results present a comparative analysis of the runtime performance of Kruskal's and Prim's algorithms for finding Minimum Spanning Trees (MSTs) on Erdos-Renyi graphs. The runtime is measured in milliseconds for varying numbers of nodes in the graph. The results is below

With edge probability of 0.2:



With edge probability of 1.0:



With edge probability of 0.2:

Node Count	Kruskal run-time (ms)	Prims run-time (ms)
100	2	1
200	3	7
300	11	5
400	2	4
500	3	4
600	3	7
700	7	16
800	15	15
900	17	35
1000	26	44
1100	32	54
1200	35	51
1300	44	63
1400	37	57
1500	56	82
1600	65	108
1700	72	108
1800	89	121
1900	108	162
2000	98	170

With edges probability of 1(all complete graph):

Node Count	Kruskal run-time	Prims run-time
100	5	4
200	9	16
300	14	16
400	49	57
500	49	24
600	47	32
700	94	41
800	100	56
900	169	90
1000	177	87
1100	178	97
1200	237	122
1300	294	137
1400	432	263
1500	444	264
1600	468	333

For sparse graphs (probability = 0.2):

- **Kruskal's algorithm** generally exhibits lower runtime compared to Prim's algorithm for smaller graph sizes. However, as the graph size increases, the runtime of Kruskal's algorithm shows more variation and tends to increase rapidly.
- **Prim's algorithm** demonstrates a more consistent runtime across different graph sizes, with relatively slower growth compared to Kruskal's algorithm as the graph size increases.

For dense graphs (probability = 1):

- **Prim's algorithm** consistently outperforms Kruskal's algorithm in terms of runtime for all graph sizes. The difference in runtime between the two algorithms becomes more accurate as the graph size increases.
- **Kruskal's algorithm** exhibits higher runtime compared to Prim's algorithm, especially for larger graph sizes, indicating its inefficiency in dense graph structures.

CONCLUSION

When comparing Kruskal's and Prim's algorithms, their performance can vary depending on the characteristics of the graph they operate on.

In sparse graphs, where the number of edges is significantly less than the maximum possible edges, Kruskal's algorithm works well. Since sparse graphs have relatively fewer edges, the sorting step in Kruskal's algorithm is efficient, leading to lower runtimes for smaller graph sizes. However, as the graph size increases, Kruskal's runtime can become less predictable due to potential fluctuations in the number of edges.

Prim's algorithm works well on dense graphs, where the number of edges approaches the maximum possible. In these scenarios, Prim's algorithm outperforms Kruskal's due to its focus on vertices rather than edges.

In summary, while Kruskal's algorithm may have an advantage in the initial stages for sparse graphs, Prim's algorithm demonstrates more consistent and scalable performance across different graph densities.