Exploring Minimum Spanning Trees: Visualizing Prim's and Kruskal's Algorithms with Diameter Analysis

Bhupinder Gosal
University of the Fraser Valley
Abbotsford, BC
Bhupinder.Gosal@student.ufv.ca

Noor Sidhu
University of the Fraser Valley
Abbotsford, BC
Eknoor.Sidhu@student.ufv.ca

Justina Bradshaw
University of the Fraser Valley
Abbotsford, BC
Justina.Bradshaw@student.ufv.ca

***Abstract*— This study explores the application of Kruskal's and Prim's algorithms, two widely used greedy approaches for constructing MSTs. Graphs used in the analysis were generated using Nauty and Traces, providing a variety of structures for testing and visualization. A graphical user interface (GUI) built with Tkinter enables users to interactively visualize and analyze graphs and their MSTs. The program integrates NetworkX for graph processing and visualization, displaying the MST construction process and calculating key metrics such as graph and MST diameters. Randomly generated weighted graphs with varying degrees and vertex counts were used to test algorithm performance. Results show that MST diameters often increase due to the removal of redundant edges, with trends varying based on graph size and structure. These findings highlight the trade-offs between minimizing edge weight and maintaining optimal node connectivity in graph structures.**

Keywords—MST, Kruskal's Algorithm, Prim's Algorithm, Diameter

I. Introduction

The minimum-cost spanning tree (MST) problem involves a connected, undirected graph G, where each edge has a weight. The goal is to find a subset of edges that connects all the vertices in the graph while keeping the total edge weight as low as possible. An MST must meet two criteria: (1) the sum of the weights of the selected edges is minimized, and (2) the edges chosen to ensure that all the vertices remain connected. The MST also has no cycles [2]. Two main algorithms are used to find an MST, Kruskal's and Prim's. Both are greedy algorithms that make a locally optimal choice at each step. They both avoid creating cycles while adding edges to the MST.

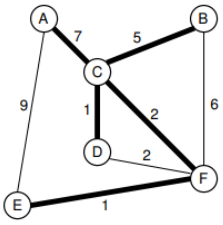

Figure 11.20 A graph and its MST. All edges appear in the original graph. Those edges drawn with heavy lines indicate the subset making up the MST. Note that edge (C, F) could be replaced with edge (D, F) to form a different MST with equal cost.

II. Prim's Algorithm

Prim's approach builds the MST from a single vertex and grows it by adding the smallest edge connecting a vertex inside the MST to one outside. It typically uses a priority queue data structure and an adjacency list.

A. Steps in Prims

1. Choose an arbitrary vertex and mark it as part of the MST.
2. Identify all edges connecting vertices in the MST to those outside it and select the edge with the minimum weight.
3. Add the chosen edge and the newly connected vertex to the MST.
4. Continue selecting and adding edges until all vertices are included in the MST.

B. Time Complexity:

Using an adjacency matrix, the time complexity is $\mathbf{O(V^2)}$. This occurs because, for each of the **V** vertices, the algorithm scans all **V** entries in the matrix to find the smallest edge, resulting in a total of $\mathbf{V \times V}$ or $\mathbf{O(V^2)}$. When using an adjacency list with a min-heap, the time complexity becomes $\mathbf{O((V+E)\ logV)}$. In this case, the algorithm processes **V** vertices and **E** edges, with the min-heap (priority queue) efficiently selecting the smallest edge in $\mathbf{O(log\ V)}$.time for each operation.

C. Space Complexity

The space complexity is $\mathbf{O(V+E)}$ for representing the graph using an adjacency list. Additionally, $\mathbf{O(V)}$ space is required for the priority queue and the data structure used to track the MST.

III. Kruskal's Algorithm

Kruskal's approach builds the MST by sorting the edges by weight and adding the smallest edge. It typically uses a disjoint-set data structure.

A. Steps in Kruskal

1. Sort all the edges in the graph by their weight in ascending order based on their weight.
2. Begin with the smallest edge and add it to the MST if it does not create a cycle with the edges already included.
3. Use a union-find data structure to efficiently check for and avoid cycles.

4. Continue selecting and adding edges until the MST contains V-1 edges, where V is the total number of vertices.

B. Time Complexity:

Kruskal's algorithm begins by sorting all the edges in the graph by weight. If there are EE edges, this sorting step takes **O(E logE)** time.

C. Space Complexity

The graph requires space to store its edges, typically in an adjacency list or edge list format. This representation requires **O(V+E)** space, where **V** is the number of vertices and **E** is the number of edges.

IV. Diameter

The diameter of a graph represents the longest shortest path between any two nodes in a graph. For unweighted graphs, the shortest path is the minimum number of edges required to travel between two nodes, and the diameter is calculated as the maximum of these shortest paths across all node pairs. In weighted graphs, where edges have assigned weights, the shortest path is determined by the minimum total weight of the edges along the path, and the diameter is calculated as the maximum of these weighted shortest paths across all node pairs.

V. Methodology

Our program includes a graphical user interface (GUI) built using Tkinter, designed to allow users to interact intuitively with the algorithms and graph configurations. Users can select between **Kruskal's** and **Prim's** algorithms for constructing the minimum spanning tree and explore various graph options. Graphs 1-10 feature either **5** or **10** vertices, while graphs **11** to **40** include varying numbers of nodes and degrees. Edge weights are generated randomly, ranging from **1** to **10**. Users can load and visualize any of the 40 pre-generated graphs, with nodes displayed in pink, edges in grey, and both node labels and edge weights visible on the graph.

To enhance the experience, the program uses NetworkX's built-in visualization tools to allow users to see the MST construction process. The graph is displayed with nodes and edges, with edges that are part of the MST highlighted using distinct visual styles. During the execution of Kruskal's and Prim's algorithms, an animation illustrates the step-by-step process, showing how edges are progressively added to the MST and how the tree's structure evolves. The GUI also calculates and displays the diameter of both the original graph and its MST, allowing users to examine any resulting changes. A "Show MST/Graph Only" button provides additional flexibility in viewing either the entire graph or just the MST. Graphs are rendered using Matplotlib's *plt.show()* for clear and interactive visualization.

A. Graph Generation

Graphs are represented as adjacency matrices, which are generated using Nauty. These matrices are then parsed into a format compatible with the program for visualization and algorithm execution. The adjacency matrices produced by Nauty define the graph structure, where each element *(i, j)* in the matrix is non-zero if an edge exists between node *i* and node *j*. The matrices are converted into NetworkX graph objects, which are used for visualization and the execution of Kruskal's and Prim's algorithms.

B. Diameter Testing Graph Types

We are testing graphs with 5 or 10 vertices, each having different minimum and maximum vertex degrees. Graphs numbered 11 to 40 are for visualization only and give users more options with different numbers of nodes and degrees.

TABLE I. Graph Types chosen for testing

Graph #	Minimum Degree	Maximum Degree
Graphs with 5 Vertices		
1	2	2
2	2	3
3	2	4
4	2	5
5	2	6
Graphs with 10 Vertices		
6	1	3
7	2	3
8	3	3
9	2	4
10	3	4

C. Python Code

- **Parse_txt(file_path):** This function reads the graph data from a file line-by-line. It detects when a new graph starts, creates an adjacency matrix from the data, converts the matrix into a NetworkX graph, assigns random edge weights to the graph, and stores the graphs in a list. This function is called when the line: graphs = parse_txt("graph_output.txt is executed (which stores the graphs as NetworkX graph objects in the graphs list).

- **visualize_all_graphs(graphs):** This function visualizes all the graphs in the graphs list using Matplotlib and NetworkX. It generates visual representations of each graph, displaying their nodes, edges, and edge weights. This function is called when the line: *visualize_all_graphs(graphs)* is executed, creating visual representations for each graph.

- **prim_mst(G):** finds the MST of the graph using Prim's algorithm, using a priority queue (*heapq*) data structure, to select the edge with the smallest weight that connects a visited node to an unvisited node, until all nodes are visited.

- **kruskal_mst(G):** finds the MST of the graph using Kruskal's algorithm, sorts all the edges by weight, and uses a union-find (*disjoint-set*) data structure to keep track of which nodes are connected, to prevent cycles.

- **update_graph(graph, steps, step_index, canvas, ax):** This function is responsible for updating the visualization of the graph at a given step of the MST construction. It clears the axis, recalculates the layout for the graph, assigns colors to nodes based on their visitation status, and highlights the edges that are part of the MST. The step_index indicates the current step in the algorithm. This function is called within the *update_graph()* function to calculate and display the diameters of both the original graph and the MST after the algorithm finishes.

- **calculate_diameter(graph, mst_edges=None):** This function computes the diameter of the graph. If mst_edges is provided, it calculates the diameter of the MST using the edges in mst_edges. If no mst_edges are provided, it computes the diameter of the original graph. This function is called within the update_graph() function to calculate and display the diameters of both the original graph and the MST.

- **plot_graph_gui(graph)**: This function creates a Tkinter GUI for interacting with the graph and visualizing the MST construction process. It includes buttons to navigate through the steps of the algorithm, a dropdown menu to select the graph to visualize, and labels to display the graph and MST diameters.

- The **load_graph(), next_step()**, and **prev_step()** functions within it are responsible for controlling the visualization of each step of the MST algorithm. This function is called when the line: plot_graph_gui(graphs[0]) is executed, which opens the GUI for the first graph in the graphs list and starts the visualization.

VI. Predictions

The change in diameter of a graph and its MST can differ depending on the structure of the graph and the weights of its edges. We will be testing on graphs number 1 through 10, which have either 5 or 10 vertices, varying minimum and maximum degrees, and random edge weights.

A. *Graphs with fewer edges (lower maximum degrees):*

We predict that the diameter will not change much, because the MST may closely resemble the original graph considering its sparse nature.

B. *Graphs with many edges (higher maximum degrees):*

We predict that the diameter will increase in the MST. The original graph likely has many short-circuiting paths that reduce diameter, however these paths are removed in the MST, forcing longer routes and increasing diameter.

Our program is designed to set random weights to all edges in every graph, which will differ each execution. Thus, our overall prediction is to see an increase in MST diameter for all graphs. This is because the MST prioritizes minimizing the total edge cost, rather than minimizing the distance between all nodes, which will often result in longer paths compared to the original graph.

VII. Observations

A. *Graphs with 5 vertices:*

For graphs with 5 vertices, the diameter of the MST generally remains the same or increases. This behavior aligns with our predictions, as smaller graphs are more likely to naturally resemble their MSTs. We also observed a slight trend of decreasing MST diameter as the maximum vertex degree increased, as shown in Figure 1 and Figure 2. This may be due to the higher number of potential edges in graphs with higher degrees, which, combined with randomly assigned edge weights, increases the likelihood of low-weight edges forming shorter paths in the MST.

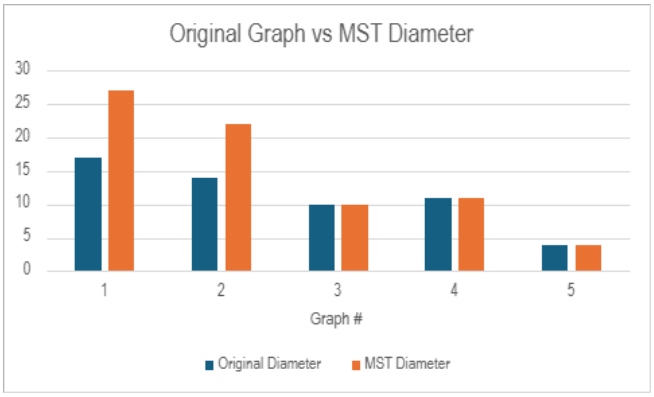

Figure 1: Bar Graph comparing the Diameter of the Original Graph to its Minimum Spanning Tree (MST) across graphs with 5 Vertices. (Bhupinder)

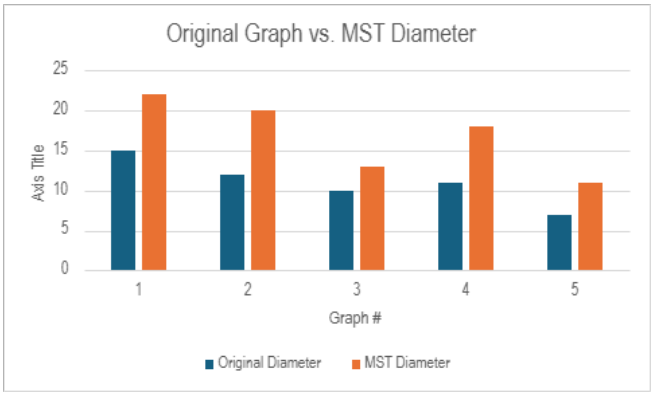

Figure 2: Bar Graph comparing the Diameter of the Original Graph to its Minimum Spanning Tree (MST) across graphs with 5 Vertices. (Justina)

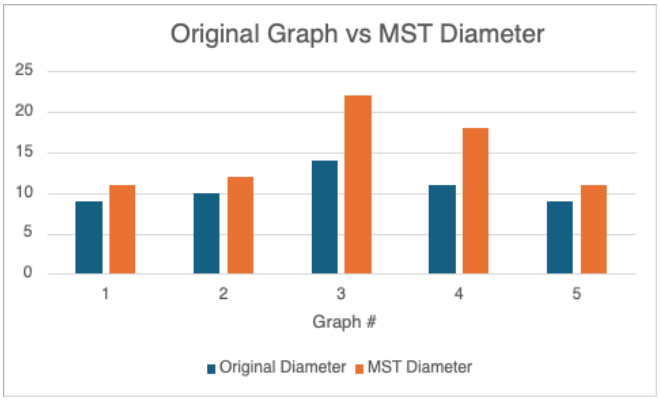

Figure 3: Bar Graph comparing the Diameter of the Original Graph to its Minimum Spanning Tree (MST) across graphs with 5 Vertices. (Noor)

B. Graphs with 10 vertices:

When considering graphs with 10 vertices, several cases show that the MST diameter significantly increases, especially in graphs where the original diameter is low. This suggests that the removal of certain edges in the MST forces longer paths between nodes. Unlike graphs with fewer vertices, there is no clear trend of decreasing MST diameter as the vertex degree increases. However, the presence of more vertices likely contributes to a greater variety of edge weights and possible paths.

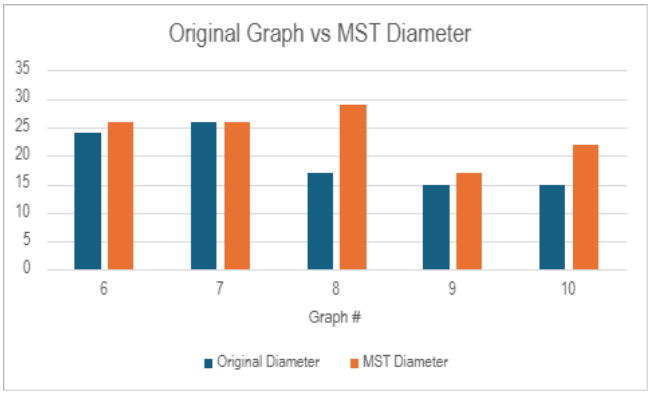

Figure 4: Bar Graph comparing the Diameter of the Original Graph to its Minimum Spanning Tree (MST) across graphs with 10 Vertices. (Bhupinder)

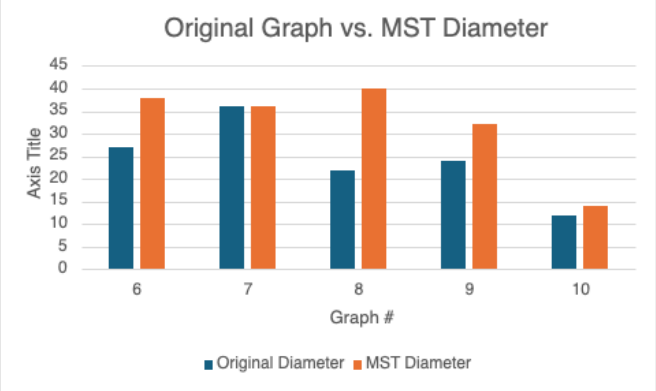

Figure 5: Bar Graph comparing the Diameter of the Original Graph to its Minimum Spanning Tree (MST) across graphs with 10 Vertices. (Justina)

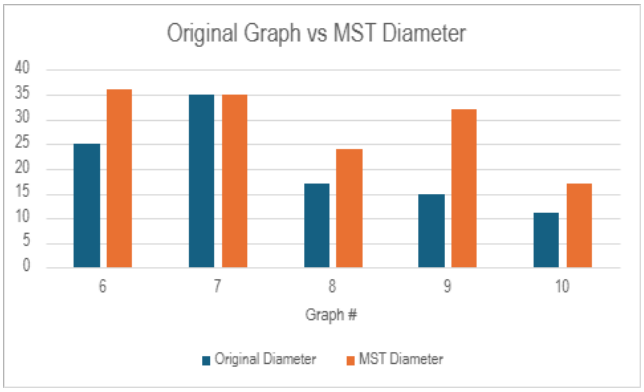

Figure 6: Bar Graph comparing the Diameter of the Original Graph to its Minimum Spanning Tree (MST) across graphs with 10 Vertices. (Noor)

VIII. Conclusion

In conclusion, our results indicate that the diameter of an MST will either remain the same or increase compared to the diameter of the original graph. This outcome occurs because the MST removes any redundant edges while preserving connectivity, often forcing longer paths between nodes, especially when the original graph contains multiple shortcuts or alternate routes. As a result, the MST's focus on minimizing total edge weight can lead to a greater overall diameter.

A higher vertex degree generally leads to a slight decrease in MST diameter, particularly in smaller graphs. However, larger graphs did not exhibit this trend. This can likely be attributed to the increased number of potential edges in graphs with higher degrees, which, along with randomly assigned edge weights, raises the probability of low-weight edges creating shorter paths in the MST.

References

[1] Bhatia, V. (2024) *Graph measurements: Length, distance, diameter, eccentricity, radius, center*, *GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/graph-measurements-length-distance-diameter-eccentricity-radius-center/ (Accessed: 29 November 2024).

[2] C. A. Shaffer, *Data Structures and Algorithm Analysis in Java*, 3rd ed. Dover Publications, Inc., 2013, pp. 393-398. [Online]. Available: https://people.cs.vt.edu/~shaffer/Book/JAVA3elatest.pdf

[3] R. Sedgewick, *Algorithms in C++*, 3rd ed. Addison-Wesley, 1998, p. 75. [Online]. Available: https://www8.cs.umu.se/kurser/TDBAfl/VT06/algorithms/BOOK/BOOK2/NODE75.

[4] R. Sedgewick, *Algorithms in C++*, 3rd ed. Addison-Wesley, 1998, p. 74. [Online]. Available: https://www8.cs.umu.se/kurser/TDBAfl/VT06/algorithms/BOOK/BOOK2/NODE74.HTM#SECTION02471000000000000000

[5] R. Sedgewick, *Algorithms in C++*, 3rd ed. Addison-Wesley, 1998, p. 73. [Online]. Available: https://www8.cs.umu.se/kurser/TDBAfl/VT06/algorithms