**2. Implementation**

**2.1. Data Structure:** Python Dictionary

**2.2. Libraries**

**NumPy**: Used for numerical operations, particularly for handling arrays and matrices.

**Pandas:** Used for creating a DataFrame to display the adjacency matrix.

**NetworkX:** A library for creating, analyzing, and visualizing complex networks or graphs.

**Random:** Used for generating random numbers.

**Matplotlib:** Used for plotting and visualization.

**2.3. Code Explanation**

**Graph Generation:** The code has Nodes variable which defines the number of Nodes and a Boolean Dense indicating whether the graph is dense or sparse. We have specified the graph as undirected with random edges with weights between 1 and 999.

**Edge Sorting:** The edges are extracted from the Adjacency Matrix and then sorted by their weights in ascending order and stored in a dict Edges.

**Kruskal MST:** The MST is found by iterating through all the sorted edges and adding the edges to our MST Graph called the sol\_graph. The Algorithm checks if adding an edge creates a cycle, the edge is skipped to avoid cycle creation. The solution is then visualized using NetworkX. The values are basically sorted at the start here and we are only popping them.

**Prim’s MST:** The adjacency matrix is converted into a dictionary of dictionaries. Each node is a key, and the corresponding value is a dictionary containing neighboring nodes and their weights. The algorithm starts with an arbitrary node (node 0 in this case), and iteratively adds the edge with the minimum weight that connects a visited node to an unvisited one. The original graph and the solution graph are visualized using NetworkX. The values are sorted here once initially and the minimum values are then found from the respected nodes and then we find the minimum values further from those respected nodes hence the double dictionary.

**3. Experimental Setup**

**3.1. Language:** Python Language

**3.2. Platform:** Google Collab

**3.3. Disk Information:**

**3.4. CPU Specifications:**

**3.5. Memory:**

**4. Results**

**4.1. Table**

**4.2. Line Graphs**

**4.3. Visualization of Graphs and MSTs**

**5. Discussion**

After conducting experiments and recording the time taken by Prim's and Kruskal's algorithms for different sets of vertices ('N') and edges in both dense and sparse graphs, the following observations were made.

**Prim's Algorithm:**

Demonstrated consistent better performance compared to Kruskal's algorithm for dense graphs.The implementation in networkx, utilizing heaps, resulted in an efficient time complexity of O((V+E)logV).Particularly well-suited for large dense graphs due to its efficiency in handling fewer edge operations.Prim's algorithm is recommended for creating a minimum spanning tree in scenarios involving large dense graphs. Continued to perform better than Kruskal's algorithm on smaller sparse graphs as well.

**Kruskal's Algorithm:**

Although Kruskal's algorithm performed slightly less efficiently than Prim's for dense graphs, it remains a viable option.The time complexity of Kruskal's algorithm is O(ElogE).Kruskal's algorithm can still be considered for smaller dense graphs, and the choice may depend on specific requirements. Demonstrated reasonable performance on sparse graphs, despite being outperformed by Prim's algorithm.

**Conclusion**

* Large and dense graphs.
* Sparse graphs where the efficiency of Prim's algorithm is advantageous.