CS-474

Design & Analysis of Algorithms

Semester Project

Group Members:

- Hamna Iftikhar (2020146)
- Hassan Ibrar (2020306)
- Zartaj Asim (2020526)

Submitted To:

Sir Ahsan Shah

List of Contents

1. Introduction:

Two well-liked techniques for locating the minimal spanning tree (MST) in a connected, undirected graph are Prim's and Kruskal's. A subset of a graph's edges with the smallest feasible total edge weight that joins all the vertices without creating any cycles is known as a minimal spanning tree.

1.1.Greedy approach & Algorithms:

The least spanning tree issue may be solved efficiently using the greedy method, as shown by the algorithms developed by Prim and Kruskal. By giving priority to locally optimum decisions, they arrive at globally optimal solutions and offer effective techniques for building minimal spanning trees in a variety of graph types.

1.2.Algorithmic Data Structures:

- **PRIM'S PSEUDO-CODE:**

```
# Pseudocode PRIMS'S
Nodes = 50
Dense = False
Visited = array of size Nodes initialized to 0
Vertices = 2D array of size Nodes x Nodes initialized to 0
Solution = 2D array of size Nodes x Nodes initialized to 0

# Generate Random Graph
if Dense:
    n = 0.5 * Nodes * (Nodes - 1) + randint(0, Nodes - 1)
else:
    n = 0.5 * Nodes * (Nodes - 1) - randint(0, Nodes - 1)

while n > 0:
    i, j = randint(0, Nodes - 1), randint(0, Nodes - 1)
    if i == j:
        continue
    Vertices[i][j] = randint(1, 999)
    Vertices[j][i] = Vertices[i][j]
    n = n - 1

# Pseudocode Algorithm for Prim's Algorithm
Edges = empty dictionary

# Construct Edges dictionary from Vertices
for i in range(Nodes):
    Edges[i] = {}
    for j in range(i):
        if Vertices[i][j] != 0:
            Edges[i][j] = Vertices[i][j]
            Edges[j][i] = Vertices[i][j]

for i in range(length(Edges)):
    Edges[i] = dictionary sorted by values in ascending order

graph = empty graph
sol_graph = empty graph
Temp = empty list
```

```
# Add Nodes to the Original Graph
for i in range(Nodes):
    add_node(graph, i)

# Add Edges to the Original Graph
for i in Edges:
    for j, value in Edges[i]:
        add_edge(graph, i, j, weight=value)

# Add Nodes to the Solution Graph
for i in range(Nodes):
    add_node(sol_graph, i)

Visited[0] = 1
Temp.append(0)

# Remove the 0th element from the adjacency lists of all nodes
for i in Edges:
    remove_element(Edges[i], 0)

while sum(Visited) != Nodes:
    X = empty list

    # Create a list of nodes in Temp and their minimum edges
    for i in Temp:
        if Edges[i]:
            X.append([i, get_min_edge(Edges[i])])

    # Find the minimum edge among the candidates
    index = find_min(X, key=lambda x: x[1][1])

    i, X = X[index]
    j, value = X

    # Remove the selected node from the adjacency lists of all nodes
    for X in Edges:
        remove_element(Edges[X], j)

    Temp.append(j)
```

```
    Visited[j] = 1
    Solution[i][j], Solution[j][i] = value, value

    # Add the selected edge to the solution graph
    add_edge(sol_graph, i, j, weight=value)

# Visualize the Original Graph and the Solution Graph
draw_networkx(graph)
draw_networkx(sol_graph)
```

- **KRUSKAL'S PSEUDO-CODE:**

```
# Pseudocode KRUSKAL's
Nodes = 25
Dense = False
Visited = np.zeros(Nodes, dtype=int)
Vertices = np.zeros((Nodes, Nodes))
Solution = np.zeros((Nodes, Nodes))
rand.seed(42)

if Dense:
    n = 0.5 * Nodes * (Nodes - 1) + rand.randint(0, Nodes - 1)
else:
    n = 0.5 * Nodes * (Nodes - 1) - rand.randint(0, Nodes - 1)

while n:
    i, j = rand.randint(0, Nodes - 1), rand.randint(0, Nodes - 1)
    if i == j:
        continue
    Vertices[i][j] = rand.randint(1, 999)
    Vertices[j][i] = Vertices[i][j]
    n = n - 1

Edges = {}
for i in range(Nodes):
    Edges[i] = {}

for i in range(Nodes):
    for j in range(i):
        if Vertices[i][j] != 0:
            Edges[i][j] = Vertices[i][j]
            Edges[j][i] = Vertices[i][j]

for i in range(len(Edges)):
    Edges[i] = dict(sorted(Edges[i].items(), key=lambda x: x[1]))

# Pseudocode for Graph Visualization using NetworkX
graph = nx.Graph()

for i in range(Nodes):
    graph.add_node(i)

for i, X in Edges.items():
    for j, value in X.items():
        graph.add_edge(i, j, weight=value)

nx.draw_networkx(graph)

# Pseudocode for Solving the Minimum Spanning Tree Problem
sol_graph = nx.Graph()
Temp = []

for i in range(Nodes):
    sol_graph.add_node(i)

Visited[0] = 1
Temp.append(0)

for i in Edges.keys():
    Edges[i].pop(0, None)

while sum(Visited) != Nodes:
    X = [[i, next(iter(Edges[i].items()))] for i in Temp if Edges[i]]
    index = min(X, key=lambda x: x[1][1])

    i, X = index
    j, value = X

    for X in Edges.keys():
        Edges[X].pop(j, None)

    Temp.append(j)

    Visited[j] = 1
    Solution[i][j], Solution[j][i] = value, value

    sol_graph.add_edge(i, j, weight=value)

nx.draw_networkx(sol_graph)
```

1.3.Theoretical Performance:

- For Prim's algorithm, the time complexity to construct a minimum spanning tree is O((V + E) log V), where V represents vertices and E represents edges. This holds true for both dense and sparse graphs.

- Kruskal's algorithm has different time complexities depending on graph density. In the best-case scenario, for a dense graph, the time complexity is O(V^2), and for a sparse graph, it is O(V log V). However, in the worst-case scenario, the time complexity becomes O(E log E), where E represents the edges of the graph and V represents the vertices.

1.4.Process:

- **Prim's Algorithm:**
  Prim's algorithm begins with an initial vertex and incrementally grows the minimum spanning tree by selecting the shortest edge connecting a vertex inside the current tree to one outside. It maintains a priority queue to efficiently choose the next vertex, updating key values for adjacent vertices. The process continues until all vertices are included, guaranteeing a minimum spanning tree with optimal weights. Prim's is particularly effective in dense graphs, where the number of edges is close to the maximum possible.

- **Kruskal's Algorithm:**
  Kruskal's algorithm takes a different approach by sorting all edges based on weight and then iteratively adding the smallest non-cyclic edge to the growing minimum spanning tree. It employs a disjoint-set data structure to efficiently detect cycles and union sets of vertices. Kruskal's algorithm works well in sparse graphs, where the number of edges is significantly less than the maximum possible. The result is a minimum spanning tree with the smallest total edge weight, achieved through the systematic consideration of edges in ascending order of their weights.

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 Data Frame to display the adjacency matrix.
- **NetworkX:** A library for creating, analysing, 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.

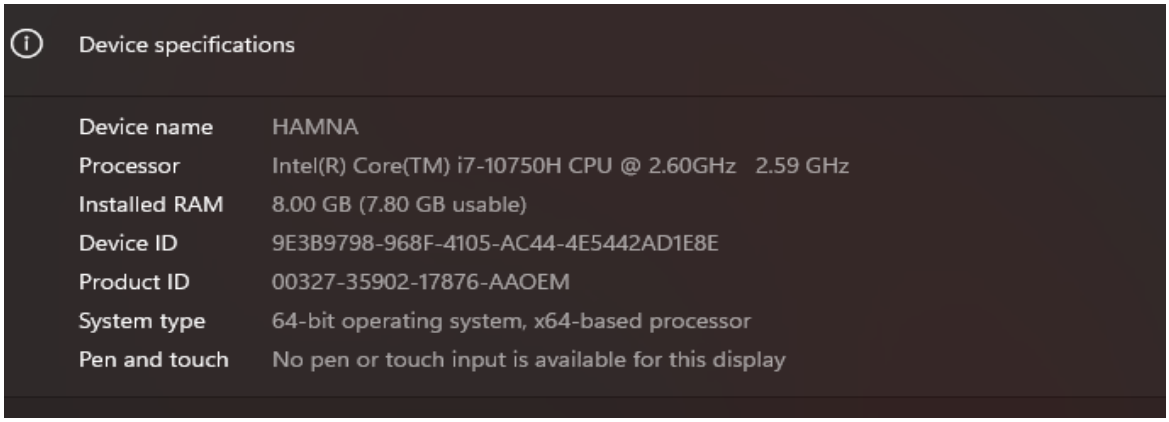
- **Edge Sorting:** The edges are extracted from the Adjacency Matrix and then sorted by their weights in ascending order and stored in a dict Edges.
- **Kruskal MST:** The MST is found by iterating through all the sorted edges and adding the edges to our MST Graph called the solution_graph. The Algorithm 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 Colab, Windows 11

3.3. Device Specifications:



4. Results:

4.1. Tables:

- **PRIMS's:**

Type: Prims	nodes	time (ms)	Wall time (ms)
Dense	100	5.23	5.74
Dense	500	357	360
Dense	1000	859	865
Dense	5000	31300	31400
Dense	10000	118000	118000
Sparse	100	5.19	5.2
Sparse	500	272	300
Sparse	1000	1060	1070
Sparse	5000	31700	31900
Sparse	10000	116000	116000

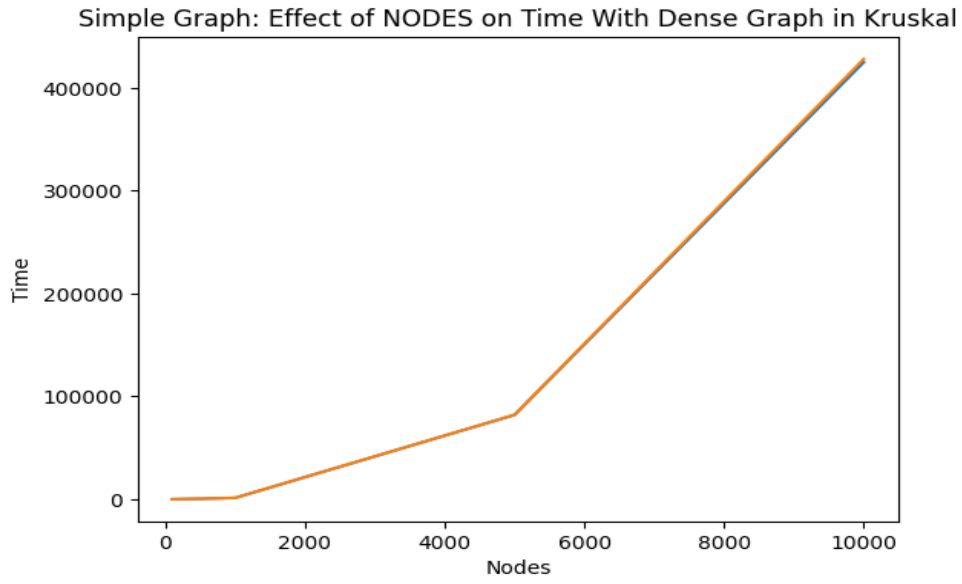
- **KRUSKAL's:**

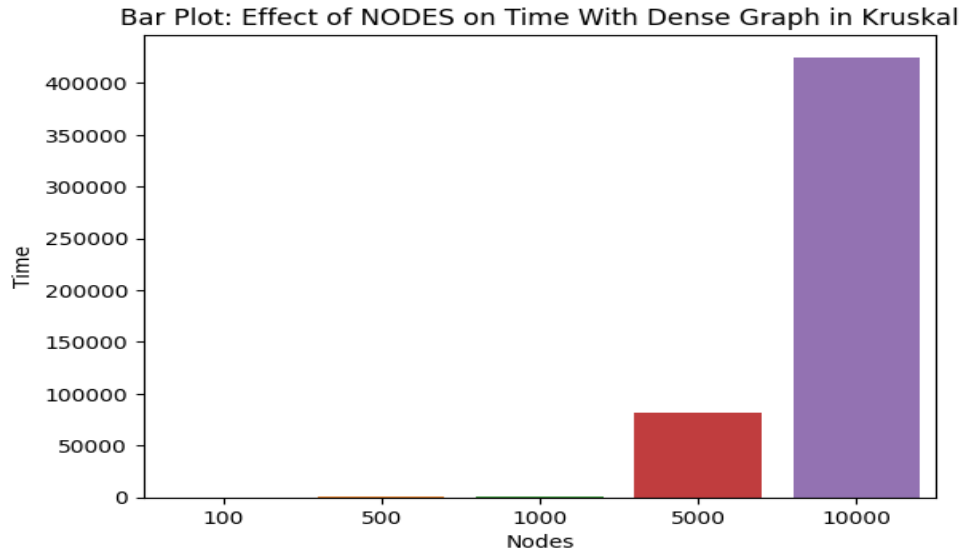
Type: Kruskal	Nodes	time(ms)	Wall time(ms)
Dense	100	10.6	10.4
Dense	500	474	479
Dense	1000	1320	1330
Dense	5000	82000	82000
Dense	10000	425000	428000
Sparse	100	9.59	12.1
Sparse	500	280	282
Sparse	1000	1580	1590
Sparse	5000	85000	85000
Sparse	10000	428000	431000

4.2. Line Graphs:

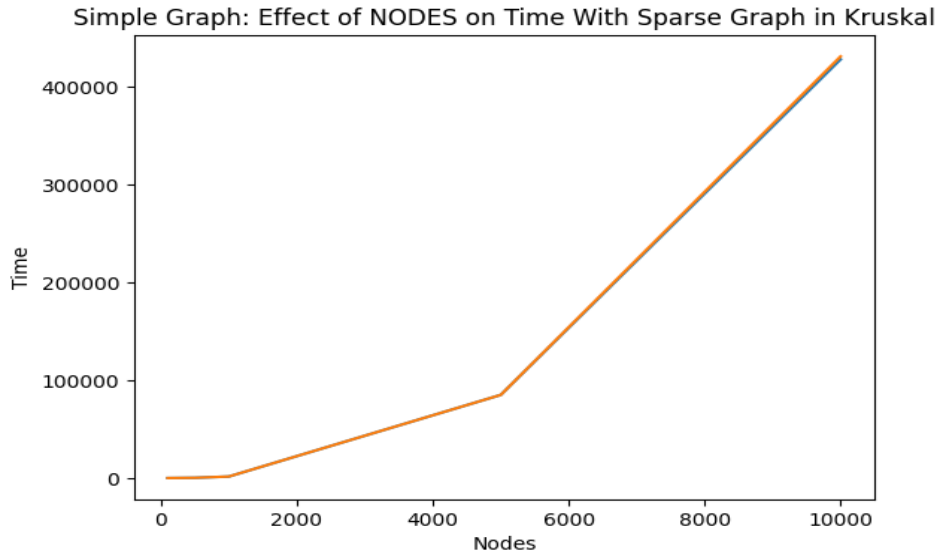
1. **Kruskal's Graphs:**

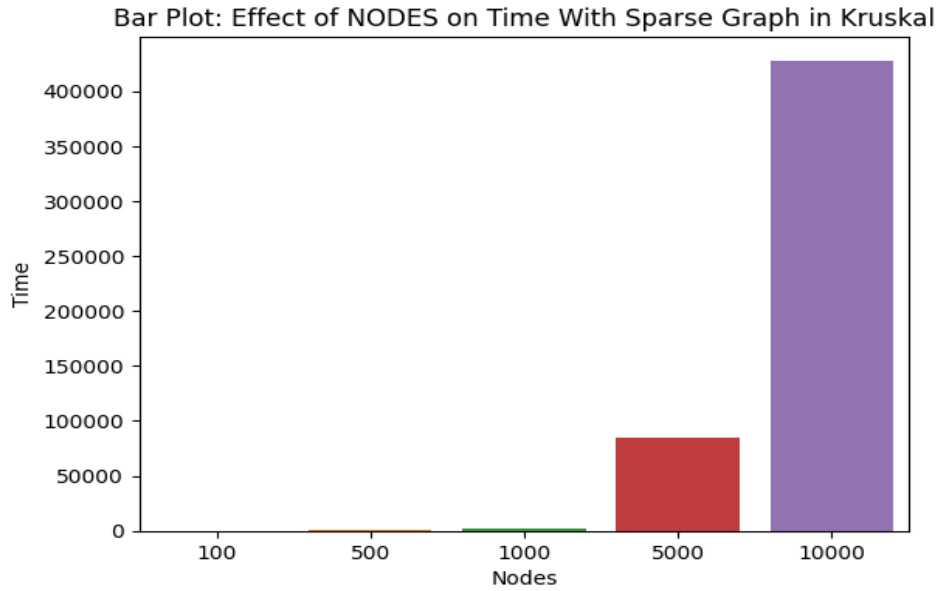
- **Dense Graph:**





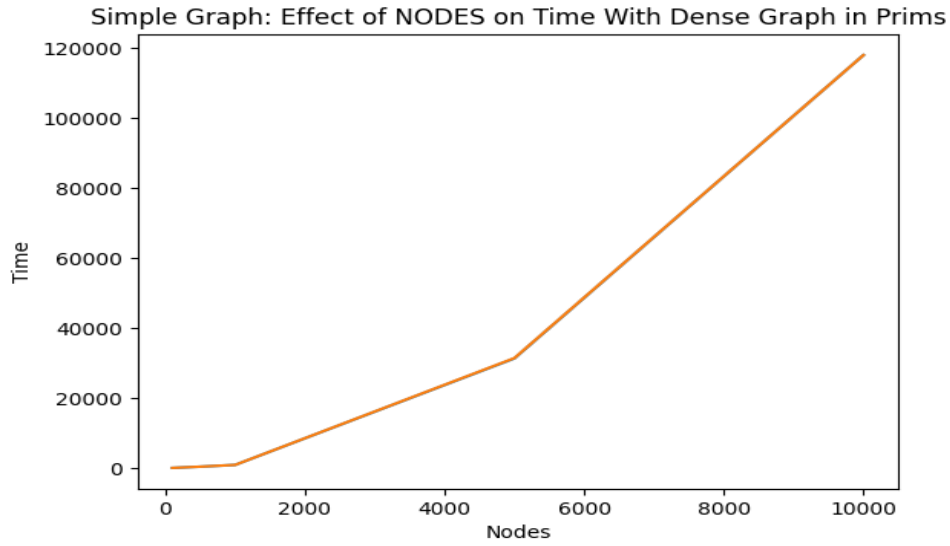
- **Sparse Graph:**

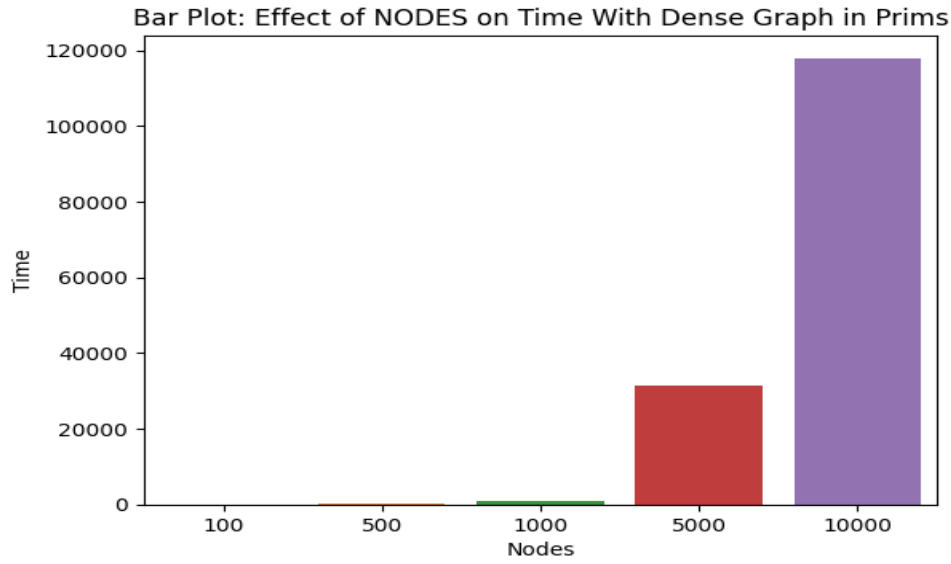




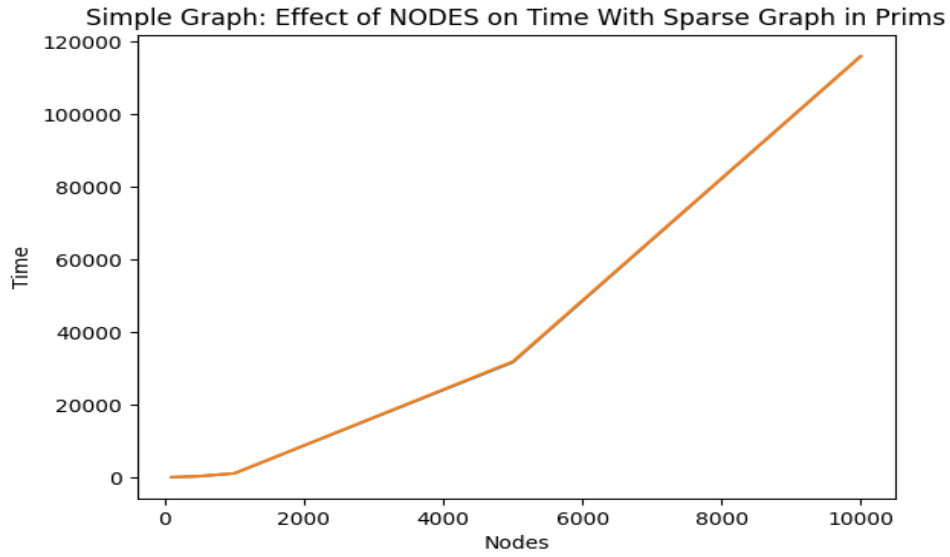
2. **Prim's Graph:**

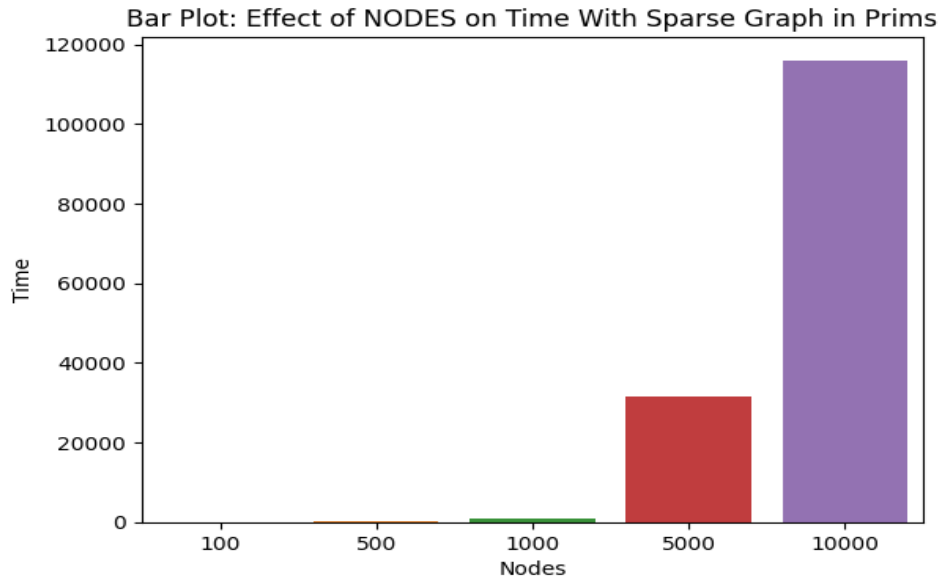
- **Dense Graph:**





- **Sparse Graph:**

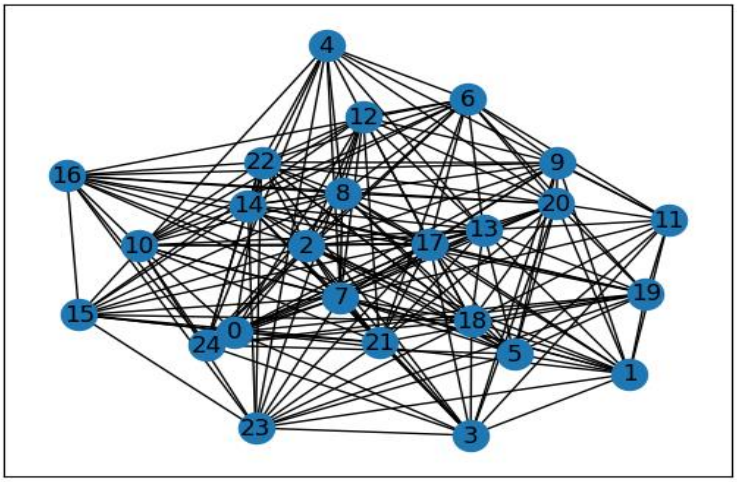


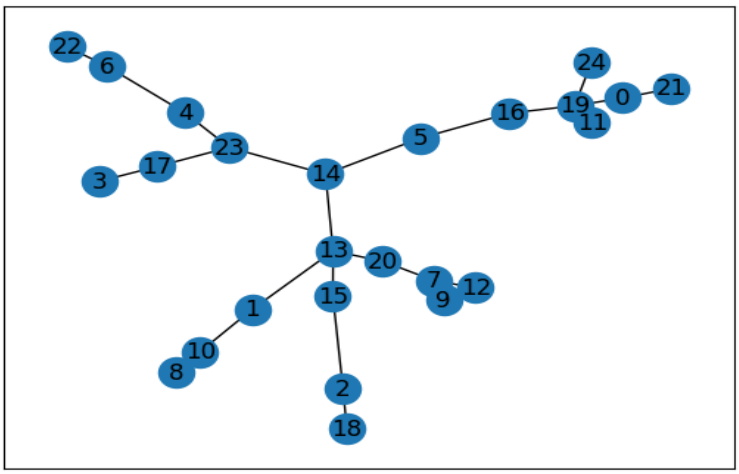


4.3. Visualization of Graphs & MSTs:

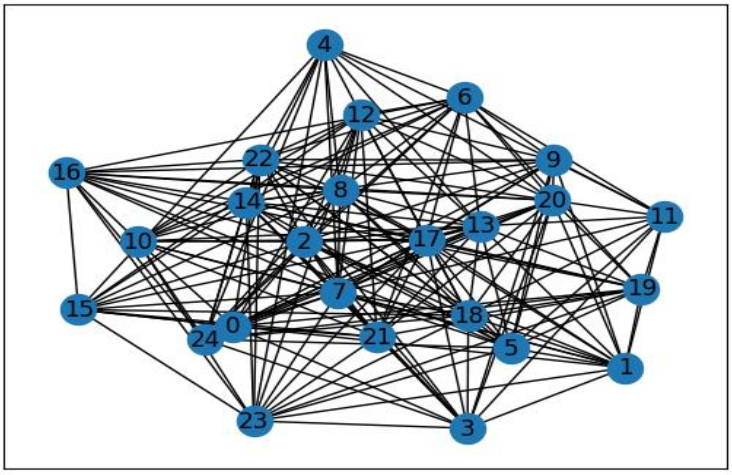
1. **PRIM's**

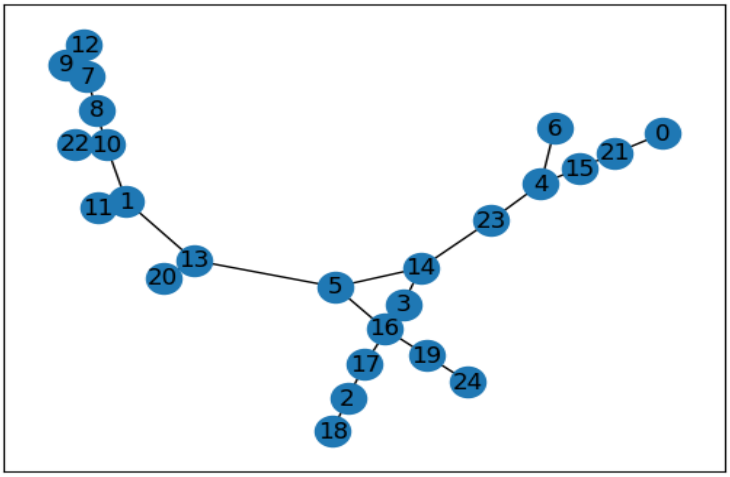
- **Dense:**





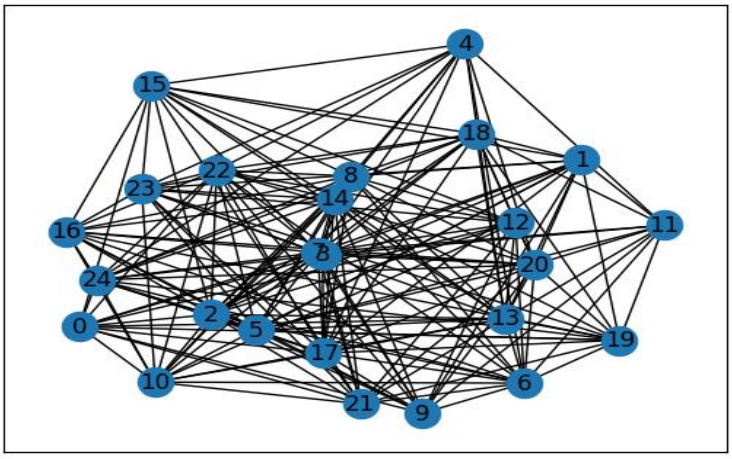
- **Sparse:**

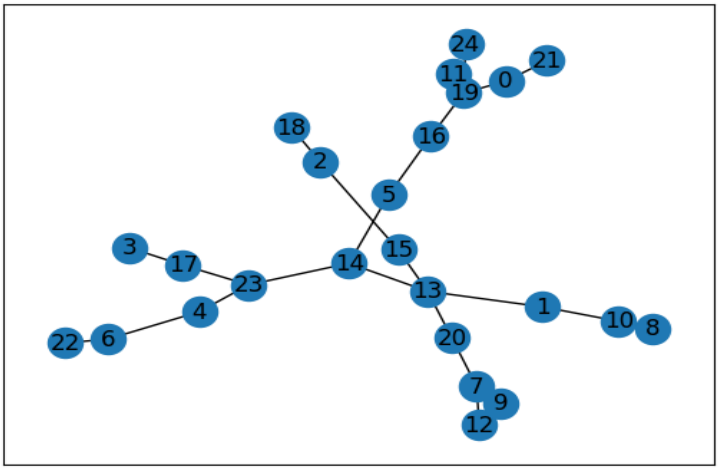




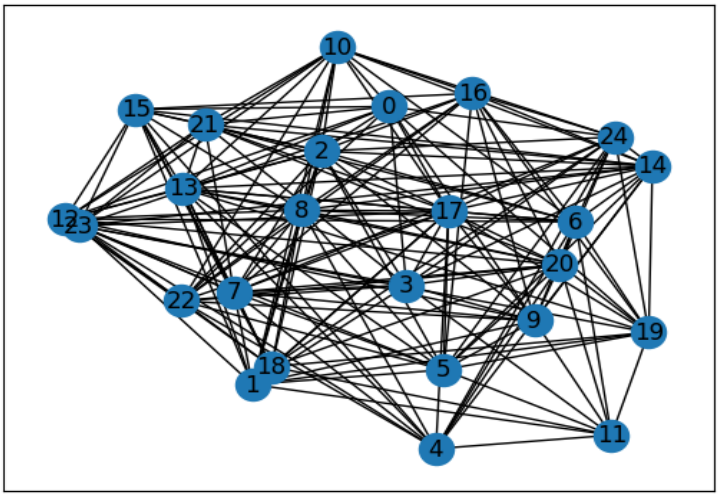
2. **KRUSKAL's**

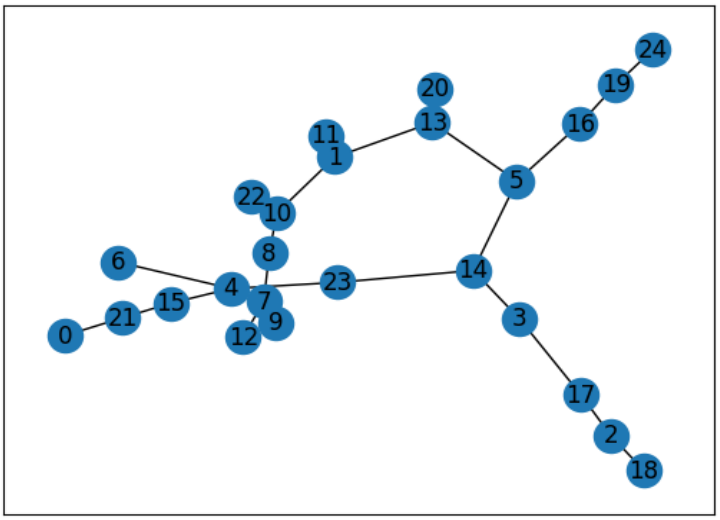
- **Dense:**




24
11
21
0
19
18
16
2
5
3
17
15
14
23
13
1
10
8
4
22
6
20
7
9
12

- **Sparse:**


10
16
15
0
21
24
2
14
13
8
17
12
23
6
20
7
3
22
9
19
18
5
1
11
4


24
20
19
13
16
11
1
5
22
10
8
6
14
4
23
7
15
3
21
9
0
12
17
2
18

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, 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:

Prim’s work better for:

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