🌳 Minimum Spanning Tree (MST) Algorithms

🌟 MST Properties

1. 🌲 MST for connected graph has exactly **V - 1** edges
2. 🌀 MST is **acyclic** (it's a tree)
3. 🔗 MST is **connected** (all vertices reachable)
4. 💰 Total cost of MST is **minimum**
5. 🌳 For disconnected graph we get a **forest** (multiple trees)

⚡ Prim's Algorithm

💡 **Main Idea:**
A greedy algorithm that **grows a tree from one starting vertex**
At each step, it adds the **minimum-weight edge** connecting the tree to a vertex outside the tree

🧭 **How it works:**

1. 🚀 Start from any vertex
2. 🌱 Add it to the tree
3. 🔍 Find the minimum edge connecting the tree to an outside vertex
4. 🔗 Add this edge and vertex to the tree
5. 🔁 Repeat until all vertices are in the tree

🧱 **Data Structures:**

- 🧮 **Priority Queue (Min-Heap)** — store edges efficiently
- 🪣 **Set** — track vertices already in the tree

⏱️ **Time Complexity:**

- **O((V + E) log V)**

💾 **Space Complexity:** O(V + E)

✅ **Best for:**

- 🌐 **Dense graphs** (many edges)
- 📊 Graphs represented as **adjacency matrices**

🧩 Kruskal's Algorithm

💡 **Main Idea:**
A greedy algorithm that **sorts all edges by weight** and adds them **one by one** if they don't create a cycle

🧭 **How it works:**

1. 📑 Sort all edges by weight (ascending)
2. 🧲 Take the minimum edge
3. ❌ If it creates a cycle → skip
4. ✅ If not → add to MST
5. 🔁 Repeat until MST has V−1 edges

🧱 **Data Structures:**

- 🔗 **Disjoint Set Union (DSU)** — detect cycles efficiently
- 🗒️ **Sorted list of edges**

⚙️ **DSU Optimizations:**

- 🪜 **Path compression** — flatten tree structure during find
- 🧮 **Union by rank** — merge smaller tree into larger

⏱️ **Time Complexity:**

- Sorting edges: **O(E log E)**
- DSU operations: **O(E α(V))** where α(V) ≈ constant
- Total: **O(E log E)**

💾 **Space Complexity:** O(V + E)

✅ **Best for:**

- 🌿 **Sparse graphs** (few edges)
- 📜 Graph as edge list
- ⚡ When edges are already sorted

⚔️ Comparison

Aspect	🟢 Prim	🔵 Kruskal
Complexity	**O((V+E) log V)**	**O(E log E)**
Best for	**Dense graphs**	**Sparse graphs**
Data Structure	**Heap + Adjacency List**	**Sorted Edges + DSU**
Implementation	**Medium difficulty**	**Simpler**

📈 **Dense graph (E ≈ V²):**

- Prim: O(V² log V)
- Kruskal: O(V² log V²) = O(V² log V)
  ➡️ **Almost equal, Prim slightly better**

🌾 **Sparse graph (E ≈ V):**

- Prim: O(V log V)
- Kruskal: O(V log V)
  ➡️ **Almost equal, Kruskal simpler**

Theory and In Practice

in Theory

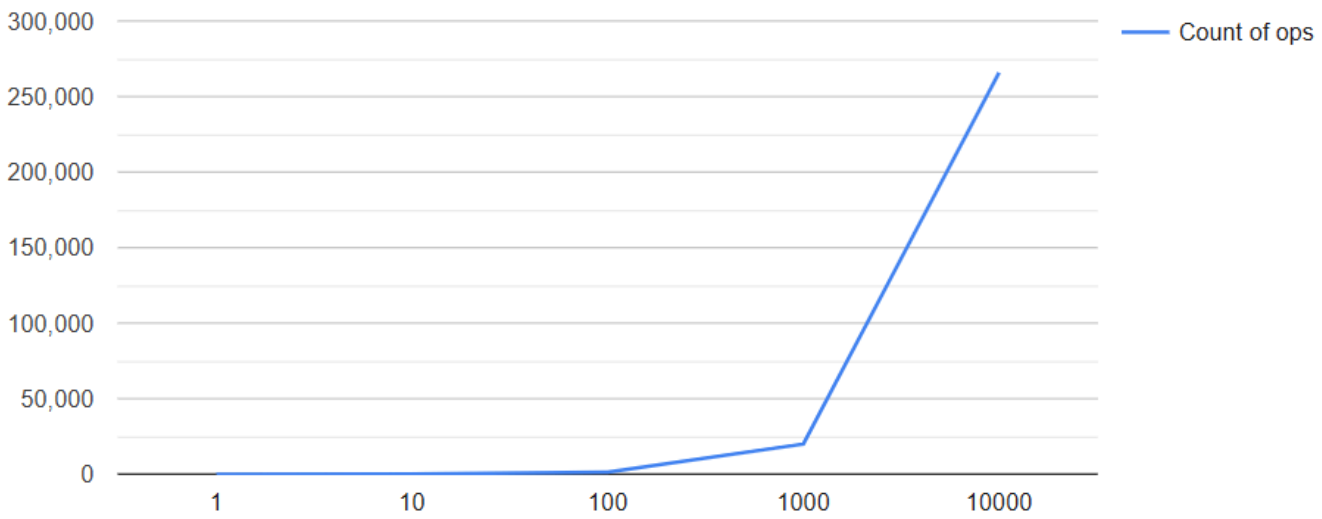
Prim algorithm

Sparse graph(E ≈ V):$T(V,E)=(V+E)\log_2(V) = 2V\log_2(V)$

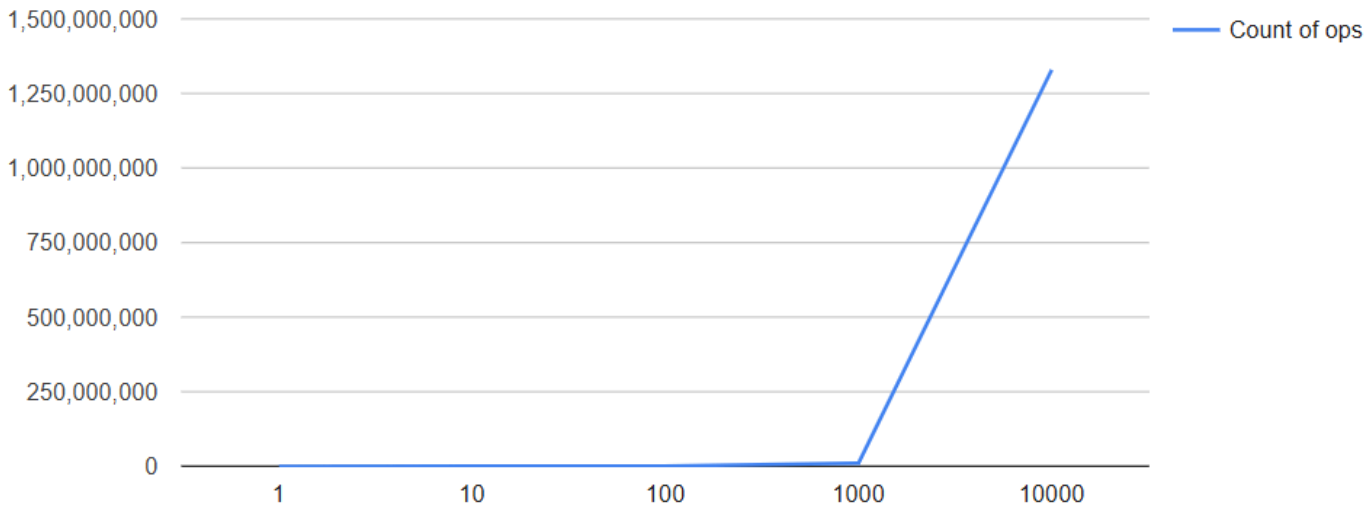
Dense graph(E ≈ V²):$T(V,E)=(V+V^2)\log_2(V)=V^2\log_2(V)$

V	$\log_2(V)$	$2 \cdot V \cdot \log_2(V)$ Sparse graph ($E \approx V$) count of operations	$V^2 \log_2(V)$ Dense graph ($E \approx V^2$) count of operations
1	0	0	0
10	3.32	66.43	332.19
100	6.64	1328.77	66,438.56
1000	9.96	19931.56	9,965,784.28
10000	13.28	265754	1,328,771,237.95

Sparse graphs:



Dense graphs:



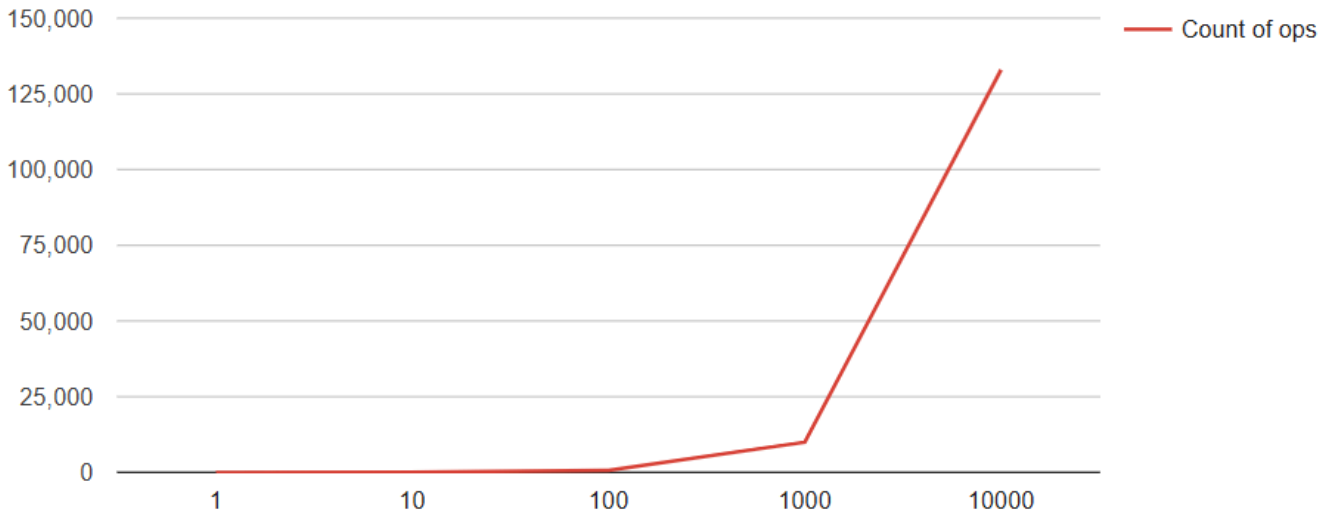
Kruskal algorithm

Sparse graph(E ≈ V):T(V,E) = (E$\log_2$E) = V$\log_2$V

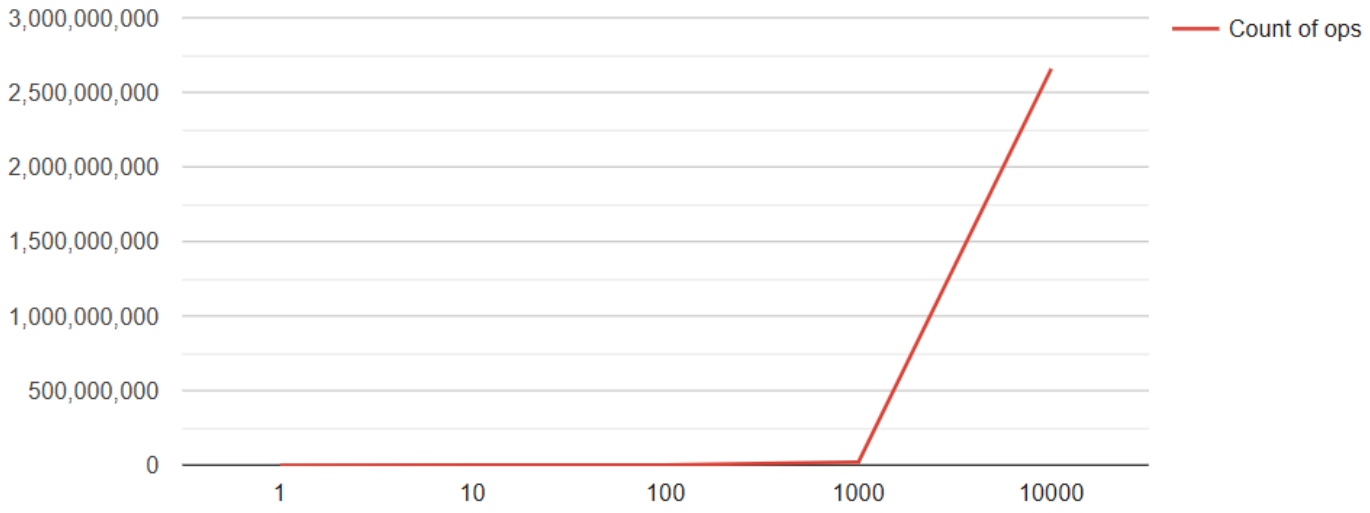
Dense graph(E ≈ V^2):T(V,E) = (E$\log_2$E) = $V^2\log_2V^2$

V	**$\log_2$V**	**E$\log_2$E(sparse)**	**$\log_2V^2$**	$V^2\log_2V^2$(dense)
1	0	0	0	0
10	3.32	33.2	6.64	664
100	6.64	664	13.29	132900
1000	9.97	9970	19.93	19930000
10000	13.29	132900	26.58	2658000000

Sparse graph:



Dense graph:

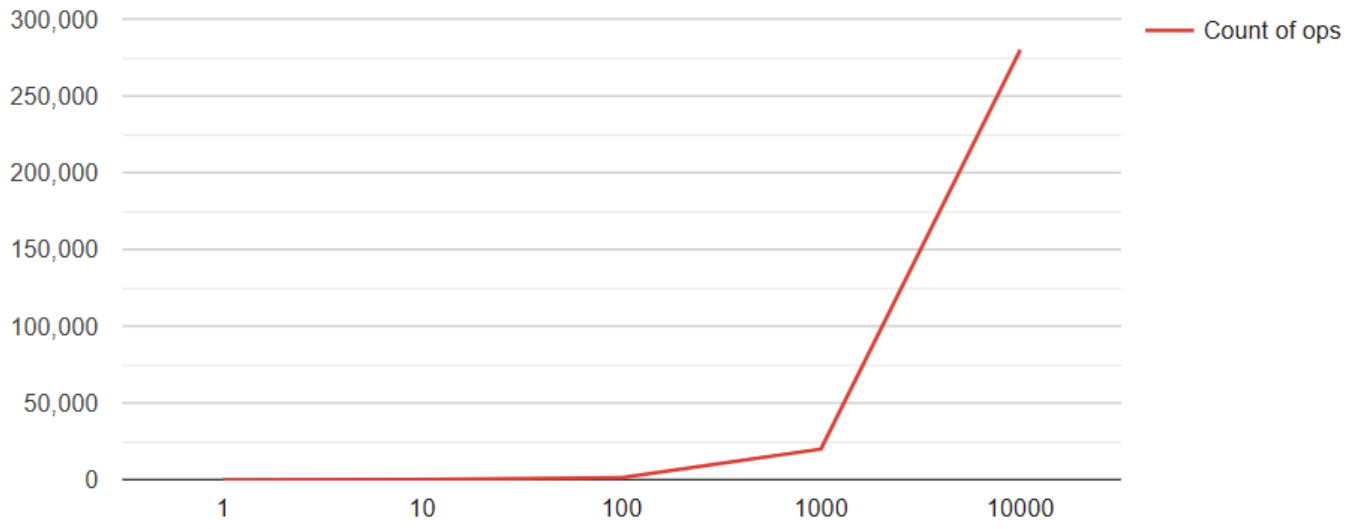


Practice

Prim algorithm

V	$2 \cdot V \cdot \log_2(V)$ Sparse graph ($E \approx V$) count of operations	$V^2 \log_2(V)$ Dense graph ($E \approx V^2$) count of operations
1	0	0
10	80	no data
100	1400	no data
1000	20000	no data
10000	280000	no data

Sparse graph in practice:



Dense graph in practice:

I cant do it cause it is difficult to create such a graph with a large number of vertices, and if possible, it would not be a correct comparison. So I simply abandoned this idea and decided to see the difference in a sparse graph

Kruskal algorithm

V	$E \log_2 E$(sparse)	$V^2 \log_2 V^2$
1	0	0
10	114	no data
100	1787	no data
1000	23907	no data
10000	318359	no data

Sparse graph in practice:

For the same reason as above

🌟 Conclusions

Both ⚙️ Prim's and 🧩 Kruskal's algorithms build a correct Minimum Spanning Tree (MST), but their performance depends on graph density, data structure, and implementation

🧠 1. Graph Density

- 🌿 Sparse graphs ($E \approx V$) → Kruskal is usually faster
  It sorts existing edges and efficiently uses the DSU structure
- 🌳 Dense graphs ($E \approx V^2$) → Prim performs better
  With a binary heap and adjacency list, it handles many edges efficiently

🧱 2. Edge Representation

- ⚡ Prim's algorithm works best with an adjacency list/matrix
- 🔗 Kruskal's algorithm suits an edge list, since it relies on sorting edges

💻 3. Implementation Complexity

- 🧮 Kruskal is simpler to implement and understand
- 🔧 Prim is more complex but can be optimized with a heap for better speed on dense graphs

⏱️ 4. Performance Comparison

- Theoretical complexity:

    - Prim (Binary Heap): O((V + E) log V)
    - Kruskal: O(E log E)
- Practical results:
    - ⚙️ Prim → better on large dense graphs
    - 🧩 Kruskal → better on small or sparse graphs

🌐 Final Conclusion

We can see that in practice, time complexity will be greater than in theory, because in theory we do not take constants into account. However, the data does not differ greatly from each other.

Metrics I relied on during the assignment

```
=== METRICS FOR CHARTS ===
ID  V       E           Prim_Cost          Kruskal_Cost    Prim_Ops    Kruskal_Ops Prim_Time_ms    Kruskal_Time_ms
1   1       0       0                       0               0           0           2,8160          13,9762
2   10      10      226                     226             80          44          2,2995          0,1629
3   100     100     2340                    2340            1400        487         0,5134          0,3936
4   1000    1000    24945                   24945           20000       4907        3,3036          1,8366
5   10000   10000   253988                  253988          280000      48359       17,9568         12,9900
```