**REPORT**

**1.**

The input data was read from a JSON file (input.json) describing a transportation network.  
Each dataset included a list of vertices (cities) and weighted edges (roads with distances or costs).

Both algorithms produced a Minimum Spanning Tree (MST) with identical total cost.

| **Algorithm** | **MST Edges (example)** | **Total Cost** | **Vertices** | **Edges** | **Operations** | **Execution Time (ms)** |
| --- | --- | --- | --- | --- | --- | --- |
| Prim | ( A–B, B–C, C–D) | 10 | 4 | 5 | 15 | 2.06 |
| Kruskal | ( A–B, B–C, C–D) | 10 | 4 | 5 | 5 | 1.32 |

Both algorithms produced the same total MST cost = 10.

Operation count includes key steps like comparisons, edge selections, and union/find operations.

The final results were stored in an output.json file

**2. Comparison: Prim’s and Kruskal’s Algorithm**

|  | **Prim’s Algorithm** | **Kruskal’s Algorithm** |
| --- | --- | --- |
| **Goal** | Grows a single tree by repeatedly adding the smallest edge connecting the tree to a new vertex. | Builds a forest and repeatedly connects the smallest edges that do not form a cycle. |
| **Best suited for** | Dense graphs (many edges). | Sparse graphs (fewer edges). |
| **Time complexity** | O(E log V) using a min-priority queue (heap). | O(E log E) ≈ O(E log V) due to sorting edges. |
| **Space complexity** | O(V + E). | O(V + E). |
| **Implementation complexity** | Slightly more complex due to heap updates. | Easier to implement because it mainly relies on sorting and Union-Find. |

**Practical Comparison (Based on Results)**

Execution Time:  
Kruskal’s algorithm executed faster (1.32 ms vs 2.06 ms) on the small graph.  
Prim’s algorithm required more time due to repeated key updates in the priority queue.

Operations:  
Kruskal’s performed only 5 operations, while Prim’s required 15, showing Kruskal’s better efficiency for smaller graphs.

Scalability:  
For larger graphs, the same pattern continues — Kruskal’s remains faster and lighter in operations for sparse networks,  
while Prim’s may perform better on dense graphs with optimized heap implementation.

**3. Conclusions**

Both algorithms correctly produced the same MST with identical total cost, confirming algorithmic consistency.  
However, their performance differs depending on graph structure:

Prim’s Algorithm is preferable when:

* + The graph is dense (many edges per vertex);
  + An adjacency matrix is used;
  + You need to grow the MST dynamically (e.g., in network expansion problems).

Kruskal’s Algorithm is preferable when:

* + The graph is sparse;
  + The input is naturally represented as an edge list;
  + You want simpler parallelization or easier implementation with Union-Find.

**In this project:**

* Kruskal’s algorithm demonstrated slightly better performance (lower runtime and fewer operations).
* Prim’s algorithm was still efficient and robust, but required more comparisons during heap updates.

Therefore, Kruskal’s algorithm is more efficient for this dataset, but the optimal choice depends on the graph’s density and data representation.