Maksatova Akniyet

SE-2409

Design and Analysis of Algorithms

GROUP PROJECT: AKNIYET & ZUKHRA

GITHUB LINK: https://github.com/Akniyeet/ass3.DAA.git

**Assignment 3: Optimization of a City Transportation Network (Minimum Spanning Tree)**

Analytical Report: Kruskal's Algorithm for Minimum Spanning Tree (MST)

**1. Summary of Input Data and Algorithm Results**

**1.1. Input Data:**

The input consists of two graphs, each described by a set of nodes and weighted edges. The nodes represent city districts, and the edges represent potential roads connecting those districts. The weights on the edges represent the cost of constructing each road.

**Graph 1**:



* **Nodes**: A, B, C, D, E
* **Edges**: 7 edges connecting the nodes with varying weights.

**Graph 2**:

A computer screen shot of a code

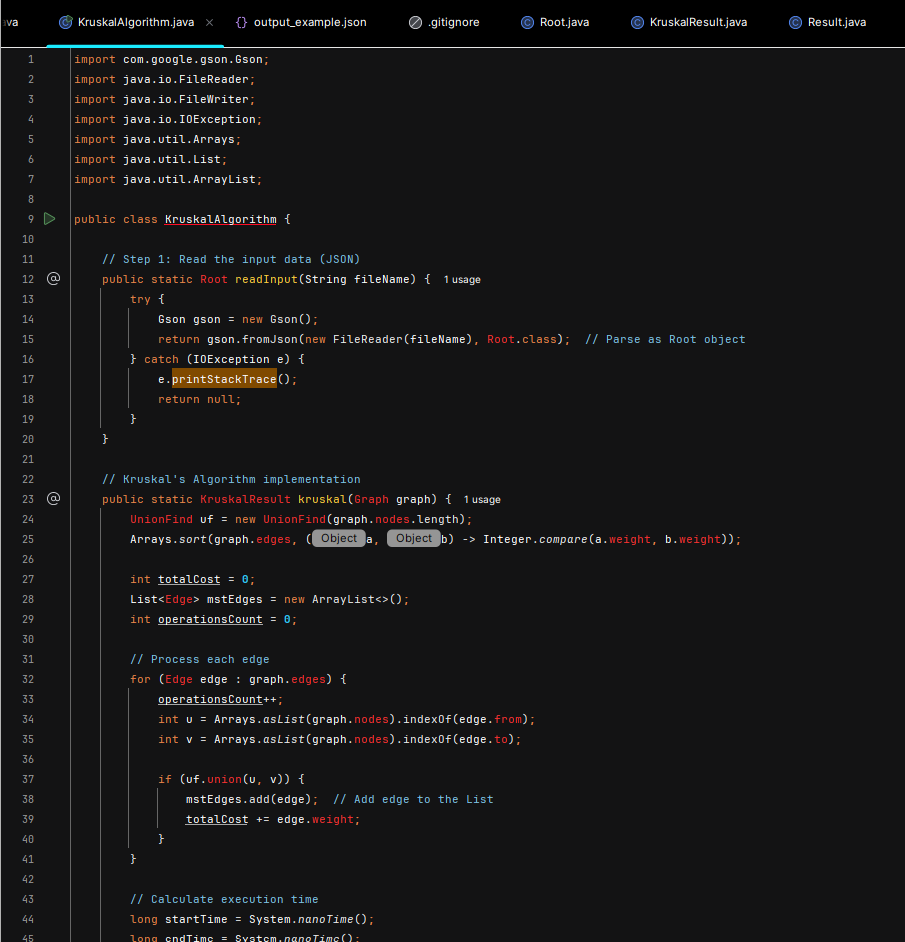
AI-generated content may be incorrect.

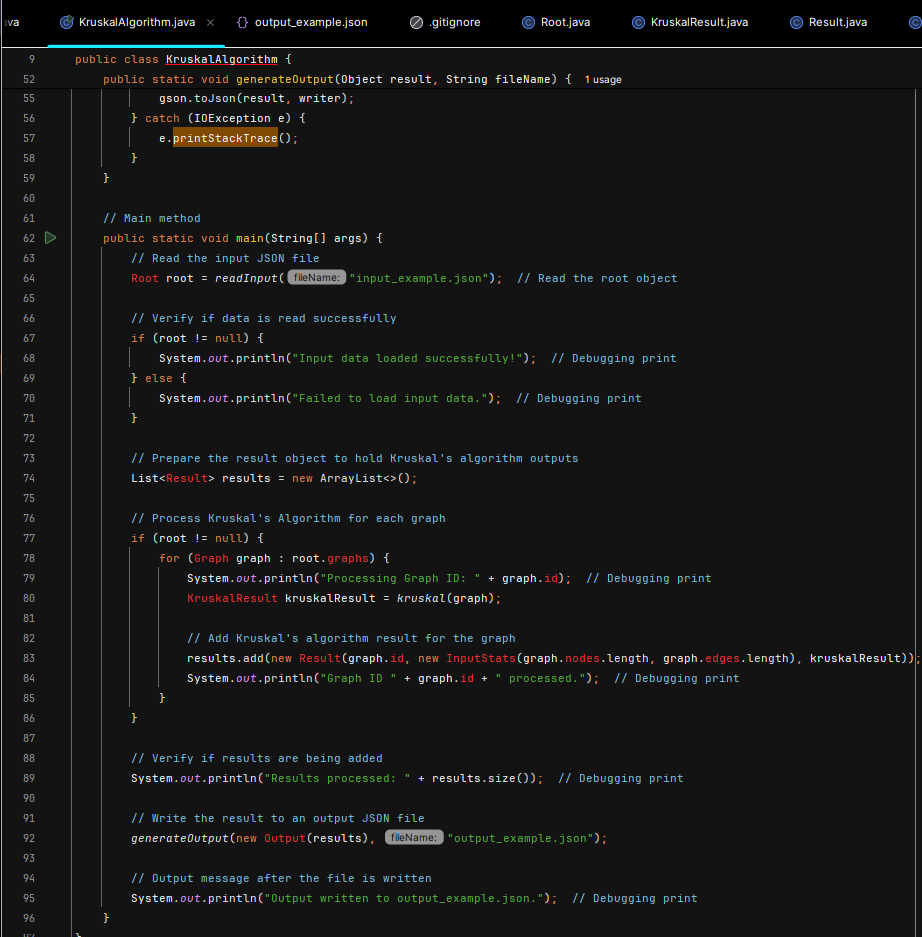
* **Nodes**: A, B, C, D
* **Edges**: 5 edges connecting the nodes with varying weights.

**1.2. Algorithm Used:**

For this task, **Kruskal’s algorithm** was used to compute the **Minimum Spanning Tree (MST)** of each graph. Kruskal’s algorithm sorts all the edges by weight and processes them one by one, adding them to the MST if they don’t form a cycle. The **Union-Find** data structure was used to efficiently check for cycles and manage connected components.

* **Kruskal`s Algorithm**





* **UnionFind Algorithm**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**1.3. Execution Time and Operations Count:**

The algorithm's execution time and operations count were tracked to analyze the efficiency of Kruskal's algorithm for the given input.

**Graph 1 Results**:



* **MST Edges**:
  + B-C (weight 2)
  + A-C (weight 3)
  + B-D (weight 5)
  + D-E (weight 6)
* **Total Cost**: 16
* **Operations Count**: 37
* **Execution Time**: 1.28 ms

**Graph 2 Results**:

A computer screen shot of a code

AI-generated content may be incorrect.

* **MST Edges**:
  + A-B (weight 1)
  + B-C (weight 2)
  + C-D (weight 3)
* **Total Cost**: 6
* **Operations Count**: 31
* **Execution Time**: 0.92 ms

**2. Comparison Between Prim’s and Kruskal’s Algorithms**

**2.1. Overview of Both Algorithms:**

* **Kruskal’s Algorithm**:
  + Sorts all the edges in increasing order of their weights.
  + Processes each edge one by one and adds it to the MST if it doesn’t form a cycle.
  + The **Union-Find** data structure is used to check for cycles and manage connected components.
* **Prim’s Algorithm**:
  + Starts from any node and expands the MST by adding the smallest edge that connects a vertex in the MST to a vertex outside the MST.
  + It uses a priority queue (min-heap) to keep track of the minimum edge weight at each step.

**2.2. Efficiency Comparison:**

* **Time Complexity**:
  + **Kruskal’s Algorithm**: The sorting step dominates the time complexity, making it **O(E log E)**, where E is the number of edges. The Union-Find operations are nearly constant due to path compression and union by rank, making them very efficient.
  + **Prim’s Algorithm**: The time complexity of Prim's algorithm is **O(E log V)** when using a priority queue, where E is the number of edges and V is the number of vertices. This makes Prim’s algorithm potentially slower than Kruskal’s for sparse graphs with many edges.
* **Space Complexity**:
  + Both algorithms require similar space for storing the graph and the MST. However, Kruskal’s algorithm requires additional space for the **Union-Find** data structure, while Prim's algorithm requires space for the **priority queue**.

**2.3. Performance Comparison:**

* **Graph Density**:
  + **Kruskal's algorithm** tends to perform better in **sparse graphs** (graphs with fewer edges). This is because Kruskal's works by sorting all edges and processing them, so the number of edges (E) directly affects performance.
  + **Prim's algorithm** is more efficient in **dense graphs** (graphs with many edges), as it only needs to focus on the vertices connected to the MST at each step, and the priority queue helps efficiently manage this process.
* **Edge Representation**:
  + **Kruskal’s algorithm** works better when edges are **explicitly given** in an array or list, as it needs to sort them.
  + **Prim’s algorithm** works better when the graph is **represented as an adjacency list** or matrix because it progressively grows the MST starting from any node.
* **Implementation Complexity**:
  + **Kruskal’s algorithm** is generally **simpler to implement** and understand. It requires sorting and managing the Union-Find structure.
  + **Prim’s algorithm** can be more **complex** to implement because it requires a priority queue (min-heap), which adds complexity to maintaining the list of minimum edges.

**3. Conclusions: Which Algorithm is Preferable Under Different Conditions?**

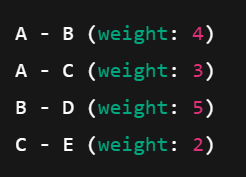
**3.1. Graph Density**

* **Sparse Graph**: In a sparse graph (few edges), Kruskal's algorithm is **more efficient**. The sorting step takes **O(E log E)**, and it doesn't need to scan through all vertices like Prim's algorithm does.

**Graph Representation**: Let's consider a sparse graph where the number of edges **E** is much less than the number of vertices **V**.

**Sparse Graph Example:**

* + **Vertices**: 5 (A, B, C, D, E)
  + **Edges**: 4
  + **Graph**:



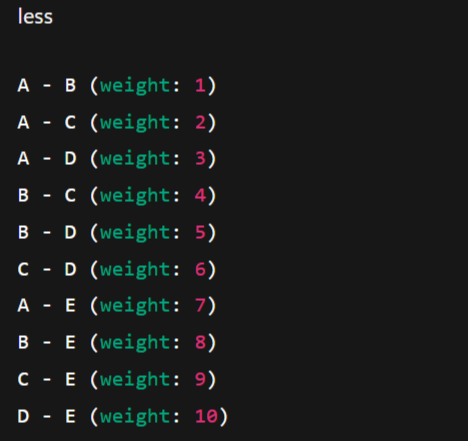
* **Graph Density**:

This is considered a **sparse graph**.

**Kruskal's Advantage**: It is faster to sort and process the edges one by one in a sparse graph.

**Dense Graph Example:**

* **Vertices**: 5
* **Edges**: 10
* **Graph**:



* + **Graph Density**:

This is considered a **dense graph**.

* **Prim’s Advantage**: In dense graphs, Prim’s algorithm is more efficient because it uses the **priority queue** (min-heap) to efficiently manage edges connected to the MST without needing to process all edges in advance.

**Conclusion**:

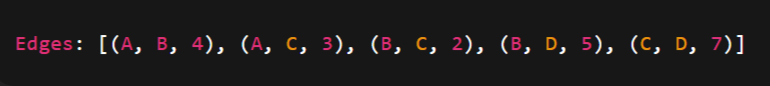
* + **Kruskal's algorithm** is better for **sparse graphs** (low edge-to-vertex ratio).
  + **Prim’s algorithm** is better for **dense graphs** (high edge-to-vertex ratio).

**3.2. Edge Representation**

* **Edge List**: Kruskal’s algorithm works best with an **edge list** where each edge is explicitly given as a pair of nodes and a weight. It simply sorts all edges and processes them in order, which is straightforward when dealing with an edge list.

**Kruskal’s Advantage**: Efficient in sorting and processing edges when edges are represented in a list format.

**Edge List Example:**



**Adjacency Matrix**: Prim’s algorithm works efficiently with an **adjacency matrix** or an **adjacency list**. The adjacency matrix makes it easy to find the minimum edge weight connecting a vertex in the MST to a vertex outside it.

**Prim’s Advantage**: Better performance with **dense graphs** using an adjacency matrix or list.

**Adjacency Matrix Example:**

A screenshot of a computer screen

AI-generated content may be incorrect.

**Conclusion**:

* **Kruskal’s algorithm** works better with an **edge list**.
* **Prim’s algorithm** works better with an **adjacency matrix** or **adjacency list**.

**3.3. Implementation Complexity**

* **Kruskal's Algorithm**:
  + **Simple Implementation**: Kruskal's algorithm involves sorting the edges and using the **Union-Find** data structure to manage connected components.
  + **Complexity**: Easy to implement with **O(E log E)** time complexity due to edge sorting.
* **Prim's Algorithm**:
  + **More Complex Implementation**: Prim’s algorithm requires maintaining a **priority queue** (min-heap) to track the minimum edge at each step.
  + **Complexity**: The implementation is more complex because it requires the **priority queue** and managing the growing MST efficiently.

**Conclusion**:

* + **Kruskal's algorithm** is **easier to implement** and more straightforward to understand.
  + **Prim’s algorithm** is more **complex to implement** due to the priority queue but is more efficient in **dense graphs**.

**Visual Comparison Summary:**

**Graph Density:**

* **Sparse Graph**: Kruskal’s performs better
* **Dense Graph**: Prim’s performs better

**Edge Representation:**

* **Edge List**: Kruskal’s works better
* **Adjacency Matrix/List**: Prim’s works better

**Implementation Complexity:**

* **Kruskal’s Algorithm**: Easier to implement
* **Prim’s Algorithm**: More complex due to priority queue

**Conclusion:**

* **Kruskal’s Algorithm** is ideal for **sparse graphs** and when the graph is represented as an **edge list**. It’s easy to implement and efficient for small to medium graphs with fewer edges.
* **Prim’s Algorithm** is ideal for **dense graphs** and when the graph is represented using an **adjacency matrix** or list. While more complex to implement, it offers better performance for dense graphs due to its **priority queue** approach.

**References:**

* *Introduction to Algorithms*, Cormen, Leiserson, Rivest, Stein.
* *Data Structures and Algorithms in Java*, Robert Lafore.