Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work 5:
Empirical analysis of algorithms: Prim and Kruskal (Greedy Algorithms)

Elaborated:

st. gr. FAF-221 Lupan Lucian

Verified:

asist. univ. Fiștic Cristofor

Chişinău – 2024

Contents

ALGORITHM ANALYSIS....3
Steps for Creating a Greedy Algorithm: 3
Applications of Greedy Algorithms:.... 3
Introduction: 6
Comparison Metric: 6
Input Format: 6
IMPLEMENTATION....7
Prim's algorithm: 7
Kruskal's algorithm: 9
CONCLUSION 12

ALGORITHM ANALYSIS

Study the concept of greedy algorithms, implement Prim and Kruskal algorithms in a programming language and analyze them empirically.

Tasks:

1. Study the greedy algorithm design technique.
2. To implement in a programming language algorithms Prim and Kruskal.
3. Empirical analyses of the Kruskal and Prim
4. Increase the number of nodes in graph and analyze how this influences the algorithms. Make a graphical presentation of the data obtained
5. To make a report.

Theoretical Notes:

Greedy algorithms are a class of algorithms that make locally optimal choices at each step with the hope of finding a global optimum solution. In these algorithms, decisions are made based on the information available at the current moment without considering the consequences of these decisions in the future. The key idea is to select the best possible choice at each step, leading to a solution that may not always be the most optimal but is often good enough for many problems.

Steps for Creating a Greedy Algorithm:

1. Define the problem: Clearly state the problem to be solved and the objective to be optimized.
2. Identify the greedy choice: Determine the locally optimal choice at each step based on the current state.
3. Make the greedy choice: Select the greedy choice and update the current state.
4. Repeat: Continue making greedy choices until a solution is reached.

Applications of Greedy Algorithms:

- Assigning tasks to resources to minimize waiting time or maximize efficiency.
- Selecting the most valuable items to fit into a knapsack with limited capacity.
- Dividing an image into regions with similar characteristics.
- Reducing the size of data by removing redundant information.

Greedy Algorithms:

1. **Prim's Algorithm for Minimum Spanning Tree (MST):**

Prim's algorithm is a Greedy algorithm. This algorithm always starts with a single node and moves through several adjacent nodes, in order to explore all of the connected edges along the way. The algorithm starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, and the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

Algorithm for Prim's Algorithm:

1. Determine an arbitrary vertex as the starting vertex of the MST.
2. Follow steps 3 to 5 till there are vertices that are not included in the MST (known as fringe vertex).
3. Find edges connecting any tree vertex with the fringe vertices.
4. Find the minimum among these edges.
5. Add the chosen edge to the MST if it does not form any cycle.
6. Return the MST and exit

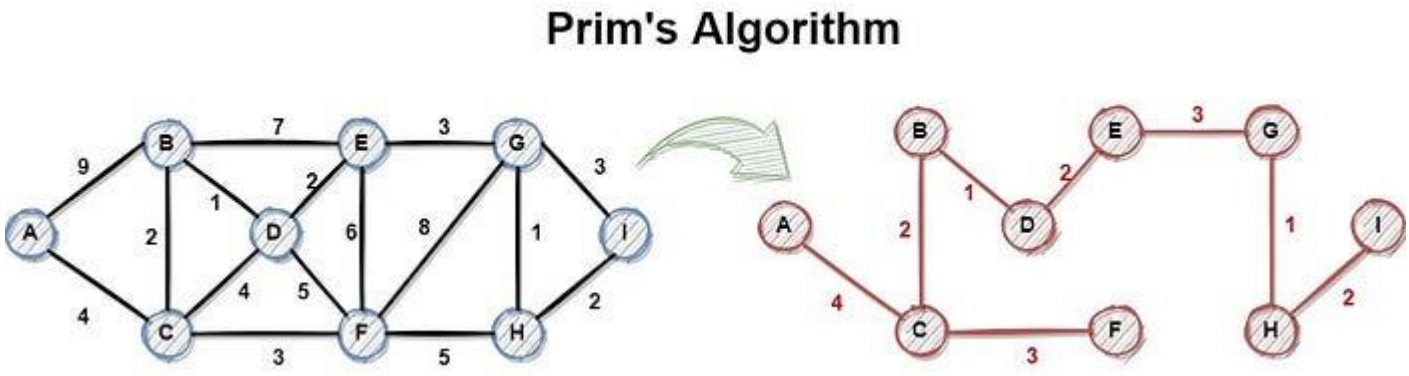

Figure 1. Prim algorithm representation.

2. Kruskal Algorithm:

In Kruskal's algorithm, sort all edges of the given graph in increasing order. Then it keeps on adding new edges and nodes in the MST if the newly added edge does not form a cycle. It picks the minimum weighted edge at first and the maximum weighted edge at last. Thus we can say that it makes a locally optimal choice in each step in order to find the optimal solution. Hence this is a Greedy Algorithm.

Algorithm for Kruskal's algorithm:

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If the cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.

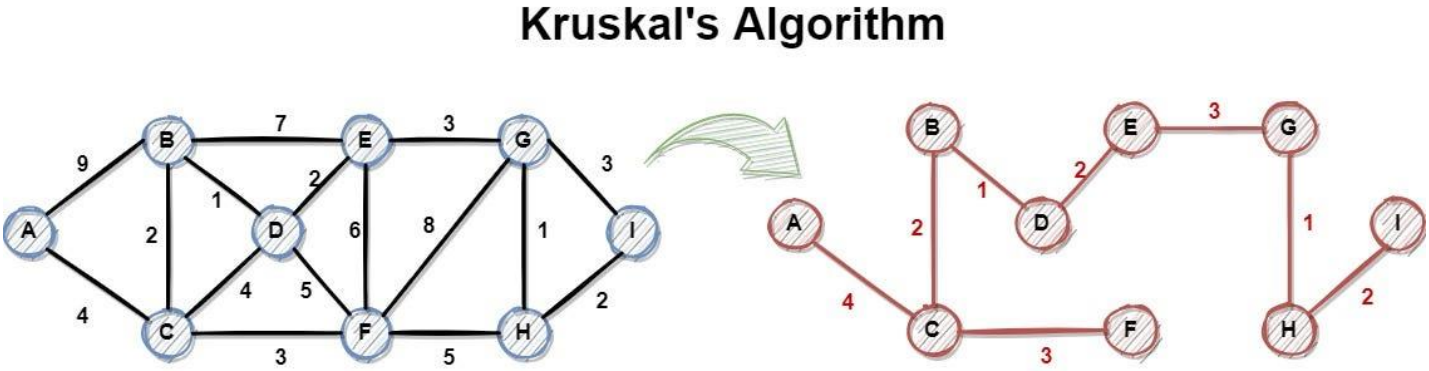

Figure 2. Kruskal algorithm representation.

Empirical Comparison:

Empirical comparison involves testing sorting algorithms on real-world datasets to observe their performance characteristics. Here's why it is crucial:

1. **Practical Efficiency:**
    - Theoretical analyses provide insights into the worst-case, average-case, and best-case scenarios. However, real-world data often exhibits unique patterns that may affect the actual performance of an algorithm.
2. **Adaptability:**
    - Different sorting algorithms may excel in specific scenarios. Empirical comparison helps identify which algorithm performs best under particular conditions, enabling the selection of the most suitable algorithm for a given task.
3. **Implementation Factors:**
    - The actual implementation of an algorithm can impact its performance. Empirical comparisons help identify subtle implementation details or optimizations that can significantly influence the practical efficiency of an algorithm.
4. **Hardware Dependencies:**
    - The efficiency of sorting algorithms can be influenced by the underlying hardware architecture. Empirical testing on different machines and environments provides a more realistic understanding of their performance.
5. **Scaling Behavior:**
    - The scalability of sorting algorithms is crucial, especially when dealing with large datasets. Empirical studies reveal how algorithms scale as the size of the input data increases.
6. **Benchmarking:**
    - Empirical comparison facilitates benchmarking, which is essential for assessing the relative merits of various algorithms in a specific context. Benchmark results guide the selection of the most efficient sorting algorithm for a particular application.

Introduction:

A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result. The algorithm never reverses the earlier decision even if the choice is wrong. It works in a top-down approach. This algorithm may not produce the best result for all the problems. It's because it always goes for the local best choice to produce the global best result. However, we can determine if the algorithm can be used with any problem if the problem has the following properties:

1. Greedy Choice Property

   If an optimal solution to the problem can be found by choosing the best choice at each step without reconsidering the previous steps once chosen, the problem can be solved using a greedy approach. This property is called greedy choice property.

2. Optimal Substructure

   If the optimal overall solution to the problem corresponds to the optimal solution to its subproblems, then the problem can be solved using a greedy approach. This property is called optimal substructure.

Comparison Metric:

The comparison metric for this laboratory work will be considered the time of execution of each algorithm T(n).

Input Format:

As input, the algorithms will take in a randomized graph data set, with an initial node_count of 1000, increment of 100 and test_cases of 10. The graph data type will be a connected graph, and it will have randomly defined edges, with a 5/10 chance of adding one between 2 vertices. The weights between the edge will be a random integer from 1 to the vertices count.

IMPLEMENTATION

Both of the algorithms are implemented in Python, making use of the random, time, collections and matplotlib.pyplot libraries in order to both ensure the functionality of the algorithms, analyze the algorithms empirically and visually representing the execution times.

Prim's algorithm:

Algorithm Description:

Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

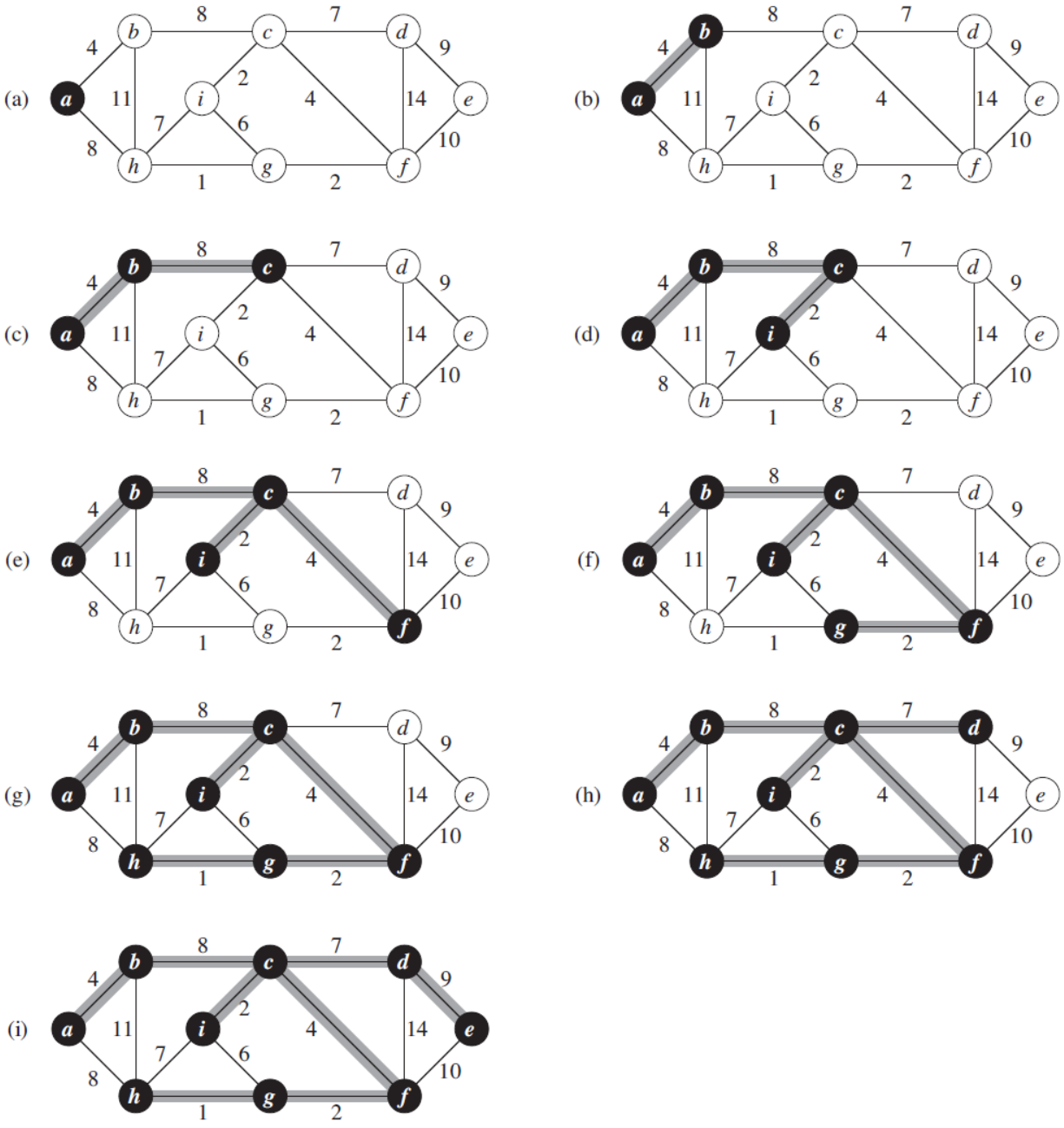

Figure 3. Prim algorithm explanation.

The complexity of Prim's algorithm is as follows:

- Time Complexity: O(V^2), If the input graph is represented using an adjacency list, then the time complexity of Prim's algorithm can be reduced to O(E * logV) with the help of a binary heap. In this implementation, we are always considering the spanning tree to start from the root of the graph
- Auxiliary Space: O(V)

Implementation:

```
def primMST(self):
    key = [float('inf')] * self.V
    parent = [-1] * self.V
    mstSet = [False] * self.V

    key[0] = 0
    parent[0] = -1

    for cout in range(self.V):
        u = self.minKey(key, mstSet)
        mstSet[u] = True

        for v, w in self.graph[u]:
            if not mstSet[v] and w < key[v]:
                key[v] = w
                parent[v] = u

    # print("Edges in the constructed MST")
    minimumCost = 0
    for i in range(1, self.V):
        # print(parent[i], "--", i, "==", key[i])
        minimumCost += key[i]
    # print("Minimum Spanning Tree:", minimumCost)

1 usage  new *
def minKey(self, key, mstSet):
    min = float('inf')
    min_index = -1

    for v in range(self.V):
        if key[v] < min and not mstSet[v]:
            min = key[v]
            min_index = v

    return min_index
```

Figure 4. Prim's algorithm code in Python.

Kruskal's algorithm:

Algorithm Description:

Kruskal's algorithm finds a minimum spanning forest of an undirected edge-weighted graph. If the graph is connected, it finds a minimum spanning tree. It is a greedy algorithm that in each step adds to the forest the lowest-weight edge that will not form a cycle.

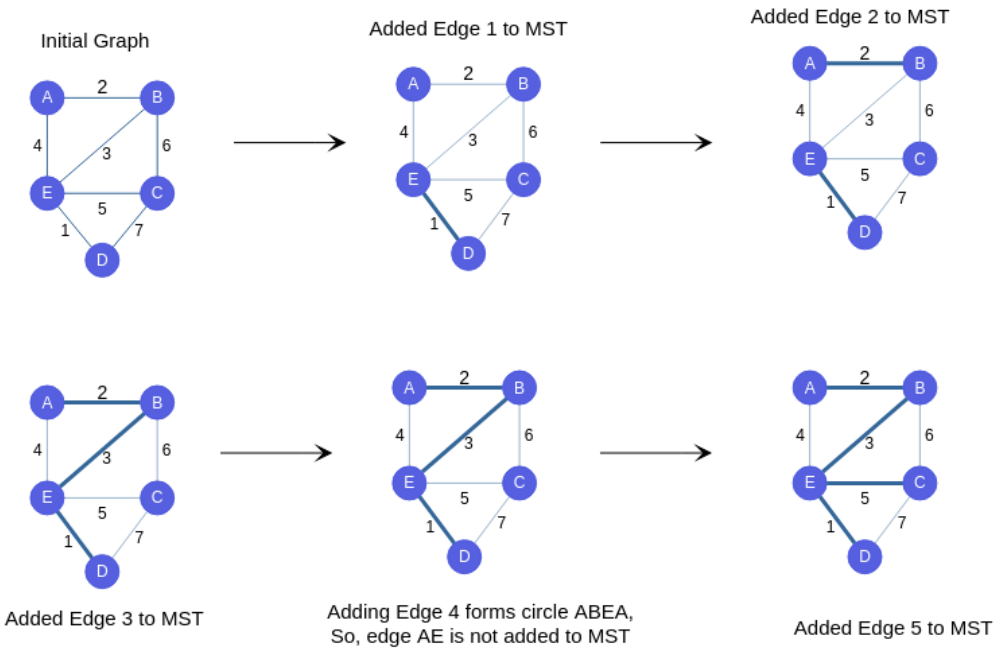

Figure 5. Kruskal algorithm explanation.

The complexity of Floyd-Warshal algorithm is as follows:

- **Time Complexity:** O(E * logE) or O(E * logV)
  - Sorting of edges takes O(E * logE) time.
  - After sorting, we iterate through all edges and apply the find-union algorithm. The find and union operations can take at most O(logV) time.
  - So overall complexity is O(E * logE + E * logV) time.
  - The value of E can be at most $O(V^2)$, so O(logV) and O(logE) are the same. Therefore, the overall time complexity is O(E * logE) or O(E*logV)
- **Auxiliary Space:** O(V + E), where V is the number of vertices and E is the number of edges in the graph.

Implementation:

```python
def KruskalMST(self):
    result = []
    i = 0
    e = 0
    edges = []

    for u in range(self.V):
        for v, w in self.graph[u]:
            edges.append((u, v, w))

    edges.sort(key=lambda item: item[2])

    parent = [i for i in range(self.V)]
    rank = [0] * self.V

    while e < self.V - 1:
        u, v, w = edges[i]
        i += 1
        x = self.find(parent, u)
        y = self.find(parent, v)
        if x != y:
            e += 1
            result.append([u, v, w])
            self.union(parent, rank, x, y)

    minimumCost = 0
    # print("Edges in the constructed MST")
    for u, v, weight in result:
        minimumCost += weight
        # print("%d -- %d == %d" % (u, v, weight))
    # print("Minimum Spanning Tree", minimumCost)
```

Figure 6. Kruskal algorithm code in Python.

Algorithm comparison results:

After performing the results on randomized datasets (according to the input data), saving the execution times, and plotting them on a graph, the results are as follows:

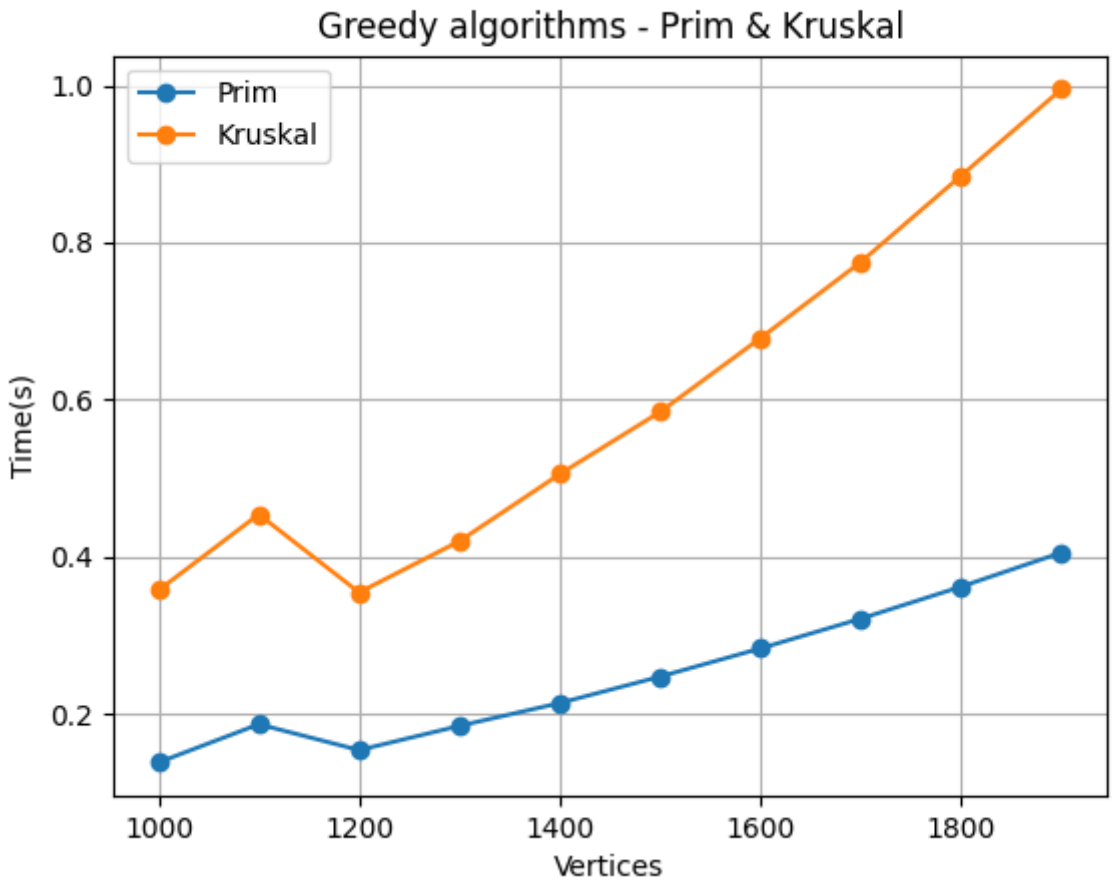

Figure 7. Algorithm comparison, first execution.

In figure 7, after the first execution, we can notice that both algorithms run very fast, finding the MST within 1 second, even for a graph with 2000 vertices. The time complexity between the 2 algorithms, however, varies, and as a result, the Kruskal algorithm runs slower than the Prim algorithm: O(E * logE + E * logV) vs O(E * logV). Because Kruskal's algorithm needs to find the union after sorting, it's significantly slower, and it

only gets slower on denser graphs. Thus, if we'd have increased the odds of generating an edge from 5/10 to 10/10, we'd experience and even bigger difference between the 2 algorithms.

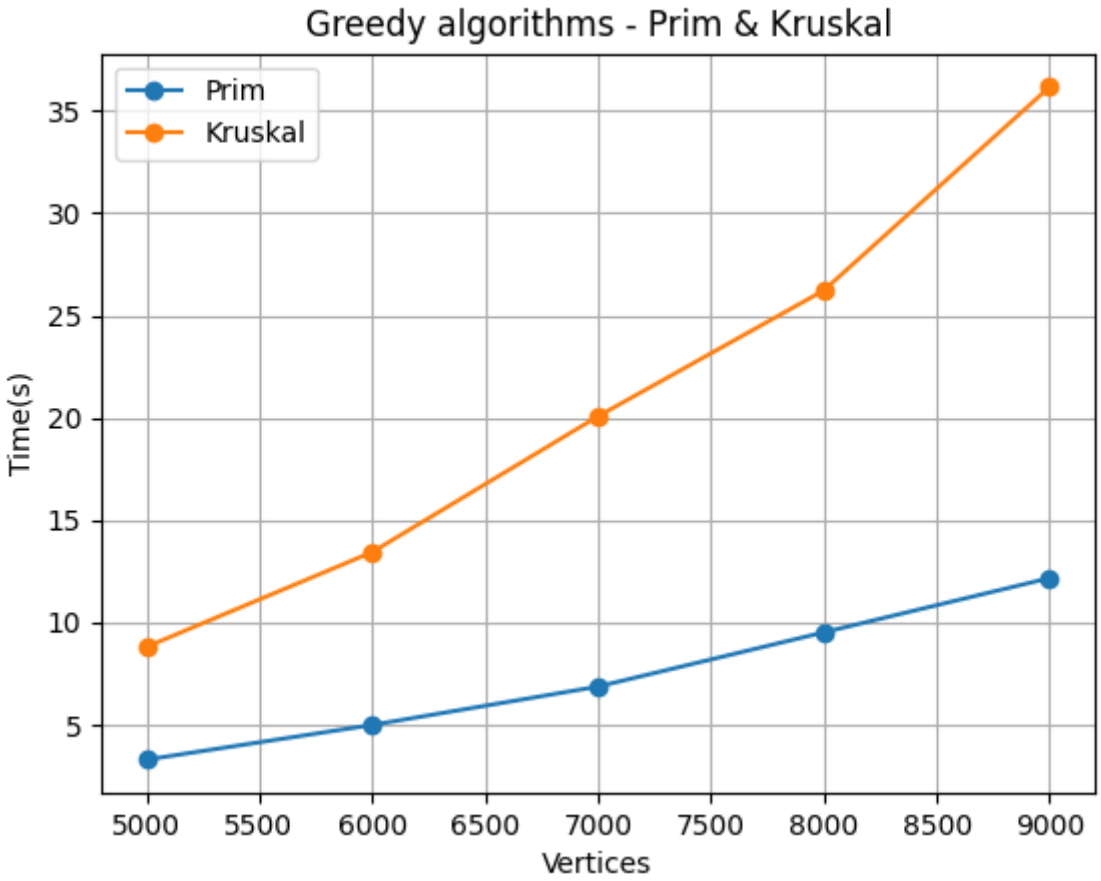

Figure 8. Algorithm comparison, second execution.

In figure 8, we work on graphs which range from 5k-9k vertices. As expected, due to the graph being somewhat dense, Prim's algorithm performs much better, beating Kruskal by approximately 2x, with a much better operational time at higher vertices count (~12 seconds vs ~37 seconds at 9k vertices).

CONCLUSION

After finishing this laboratory work, I learnt how greedy algorithms work in general, and for finding the MST/minimum spanning tree in a graph. By studying the Prim and Kruskal algorithms, I've remembered how they work on a pseudocode level, and practiced implementing them in Python, making them work with randomized datasets (vertices, edges and weights).

The algorithms in this laboratory work had a much better time complexity than in the previous laboratory work - O(E * logE + E * logV) for Kruskal vs O(E * logV) for Prim, which allowed me to work with higher vertices count, allowing for a more thorough analysis of the algorithms. As a result of the 2 tests, I've concluded that Prim's algorithm runs faster, with a significant increase in higher vertices count, outperforming Kruskal's algorithm by as much as 3x at 9k vertices. This is due to Kruskal's algorithm not operating well on dense graphs, due to having to sort the edges – which takes O(E * logE) or O(E * logV) time, and then having to perform an additional find-union algorithm, which takes O(log V) time for all the edges, thus ending up at a

O(E * logE + E * logV) time complexity, much slower than Prim's O(E * logV).

To conclude, greedy algorithms can be useful in programming, as they're easy to program and simple to understand, however, their solution might not always be the best one. When it comes to MST algorithms, Prim's algorithm outperforms Kruskal's for dense graphs, resulting in a much faster operational time, with the gap increasing only as we increase the vertices count.

GitHub repository link - https://github.com/Lucian-Lu/AA-Labs