Advanced Algorithms

Assignment 1: Minimum Spanning Trees

April 21, 2022

Budai Matteo	2057217
Burke Jamie	2044062
Vande Moore Carter	2062138

Contents

1 Introduction

For this assignment, we implemented and analyzed the running times of three MST algorithms. The algorithms implemented are:

1. Prim's Algorithm
2. Kruskal's Algorithm using a Naive Implementation
3. Kruskal's Algorithm using Union-Find

2 Prim's Algorithm

Naive version:

```
PRIM (G, s)
    X = {s} //set of vertexes included in the MST
   A = ∅ //set of edges included in the MST
   while there is an edge (u, v) with u ∈ X and v ∉ X do
      (u*, v*) = a minimum cost such edge //light edge
      add vertex v* to X
      add edge (u*, v*) to A
   return A
```

Min heap implementation:

```
PRIM (G, s):
    for each node u ∈ V do
        key[u] ← ∞
        π[u] ← null //parent of u in the minimum spanning tree
    key[s] ← 0
    Q ← V  //Q contains all nodes not in the minimum spanning tree
    while Q ≠ ∅
        u ← extractMin(Q)
        for each v adjacent to u do
            if v ∈ Q and weight(u,v) < key[v] then
                key[v] ← weight(u,v)
                π[v] ← u
```

2.1 Data Structure

2.1.1 Node

The Node class initializes six instance variables for each node of the graph

- tag: integer identifier of a node
- key: default key value of null
- parent: default parent value of null
- isParent: boolean value to determine if a node is in a heap, default of true
- index: index of node of the min heap array
- adjacencyList: adjacency list of the node, default is an empty list

2.1.2 Graph

The Graph class takes the graph .txt file as an input, and initializes variables that construct the graph and support the min heap data structure.

- **Initialize**: calls Python's defaultdict dictionary type

- **createNodes**: takes number of nodes as an input, and initializes each node in the node dictionary
- **buildGraph**: takes the graph .txt file as an input, and passes the number of nodes to the createNodes method. Futhermore, it passes each connecting node and their edge weight to the makeNodes method, which appends to the nodes adjacencyList.
- **numNodes**: takes the graph .txt file as an input and returns the number of nodes

2.1.3 MinHeap

The MinHeap class creates the min heap data structure with an array heap, and is initialized by passing the node dictionary values and the starting node integer tag. In addition to methods that return the standard array heap rules parent, leftchild, and rightchild, the following methods are defined:

- **minHeapify**: this method is passed an index and checks if any node swaps are required to maintain the min heap data structure
- **shiftUp**: this method is passed an index and properly positions the index element in the array in order to maintain the min heap data structure
- **extractMin**: the method extracts the min, or root value, of the array heap. After extracting, it calls the minHeapify method in order to maintain the min heap data structure

2.2 Implementation

The solution to the cost of the minimum spanning tree is performed in the following steps:

1. Create the Graph object through the Graph() class and call the buildGraph method
2. Define a starting node tag
3. Pass the Graph object and the starting node to the Prim function, which performs the following steps in order to return the cost of the minimum spanning tree of the graph:
    - Define the key for each node as infinity (∞)
    - Define the key for the starting node as zero (0)
    - Initialize the min heap data structure by calling the MinHeap class, passing the nodes from the graph and the starting node. If the starting node is not already the root node, the call to initialize the min heap data structure will re-set the root node as the passed starting node, and will update the index for all other nodes.
    - Now that the MinHeap object has been created, and initially contains all nodes that are not in the minimum spanning tree, we will perform the following iterative process until the heap size is zero (i.e., all nodes have been visited by the minimum spanning tree):

- Extract the minimum from the min heap data structure by calling extractMin(), which in turn returns the minimum and calls minHeapify() to make any updates required in order to preserve the min heap data structure.
- For each node, v, in the adjacency list of the node extracted by the extractMin call, check if it is both present in the array heap and the weight of connecting edge is less than the key value of node extracted by the extractMin call.
- If the check above is true, check if node v had previously been assigned a key other than infinity, and if so remove the original key value from the cost minimum spanning tree. Then,
  (a) update the parent of v be the node extracted by the extractMin call
  (b) update it's key to their connecting edge weight
  (c) add the edge weight to the cost of minimum spanning tree
  (d) call the shiftUp method in order to update the position of node v based on it's updated key value
  (e) add the key value to the cost of the minimum spanning tree.

2.3 Complexity

To calculate the total complexity, we must consider the following components where n in the number of nodes and m is the number of edges:

- Initialization of each node: O(n)
- The ExtractMin has complexity of O(log n) and is performed n times in the while loop, so total complexity of O(n log n)
- The for loop is called O(n) times, the check within the for loop is of complexity O(1), and the shiftUp method is of complexity O(log n). Therefore, the total cost of the for loop is O(m log n).

Adding these components simplifies to a total complexity of O(m log n).

3 Kruskal's Algorithm with "naive" implementation

```
KRUSKAL (G)
A = ∅
sort edges of G by cost
for each edge e, in non decreasing order of cost do
    if A ∪ {e} is acyclic then
        A = A ∪ {e}
return A
```

3.1 Input and Data Structures

The input for this algorithm will be a graph represented as a list of lists. The first element in this list is the number of nodes in the graph and the rest of the elements are nodes with the following format: [Cost, Node 1, Node 2]. This format was chosen because we can then use .sort() to sort the list of costs in non-decreasing order because the cost is the first item in the list.

Furthermore, we use one important data structure in our algorithm, which is the list "parent" that we use to find the parent node of a specific node. At the beginning, every node's parent is itself because none of the nodes are connected, but once two nodes become connected, the node will be assigned the value of the index of its parent node.

3.2 Implementation

To implement this algorithm, we used three different functions: search, combine, and Kruskal.

1. The **search** function is used to find the subtree that a certain node belongs to. The function takes in as parameters the node "a" and the list called "parent." The function will look at the parent of node "a", and then the parent of that node, and onward until the parent of the node is itself. In doing so, we can find out which set a node belongs to by seeing who the parent node of the whole set is.

2. The **combine** function is used to combine two subtrees into one. The function works by getting the parent of both subtrees, and then assigning one of them to be the parent of the other subtree. Thus, all elements in this subtree will ultimately have this node be the parent of their subtree.

3. The **Kruskal** function is the main function in our algorithm. The algorithm starts by initiating the total cost to be 0 and uses the number of nodes to create the list of parents that is numNodes long. We then remove the number of nodes element from the list, and sort all of the costs in non-decreasing order using sort(). We then traverse the list of edges and check to see if the two nodes in the edge are part of the same subtree by using our search function. If they are not, then we will combine these two subtrees to create one subtree using our combine function and increase our total cost by the weight of the edge. Once we have gone through all of the edges, we will return the total cost.

3.3 Complexity

The algorithm uses O(m * (log n)) time complexity to sort all of the edges in the list since it uses the sort() function in Python. The **for** loop will take O(m) time because it will iterate through every edge. Within this loop, we call **search** and this function grows linearly because the more elements that are in the subtree, the more parents the function will have to look through.

The algorithm therefore has a time complexity of O(m * (log n)) + O(m * n) since the sorting is not embedded into the **for** loop. This simplifies to just O(m * n) time complexity because this time complexity is the slower of the two, and therefore will affect the time complexity the most.

4 Kruskal's Algorithm with Union-Find

```
KRUSKAL(G):
    A = ∅
    U = initialize(V)
    sort edges of G by cost
    for each edge e = (u,v), in non decreasing order of cost do
        if FIND(U, u) ≠ FIND(U, v):
            A = A ∪ {(u, v)}
            UNION(u, v)
    return A
```

4.1 Data Structure

4.1.1 Graph

The Graph class contains three methods that take the graph .txt file as an input, and returns data in useable format for the algorithm.

- **buildGraph** - This method returns a list of lists in format [u, v, weight], where u and v are connected vertices and weight is the cost of their connecting edge. Each value in the list is an integer.
- **numEdges** - This method returns the number of edges as an integer.
- **numNodes** - This method returns the number of nodes as an integer.

4.1.2 UnionFind

The UnionFind class contains three methods, which support the implementation of the Kruskal Union Find solution.

- **Initialize** - Input the number of nodes to the Initialize method, and it creates a group and rank each node. Each node is initialized to be in a group of itself with a rank of zero. This function has a linear complexity of O(n), where n is the number of nodes.
- **Find** - Input a node to the Find method, and it returns the group that the node belongs to. If the nodes does not belong to it's own group, then traverse it's parent nodes until finding the node which is it's own parent. This function has a complexity that is proportional to the depth of nodes traversed to find the root parent node, and therefore a complexity of O(n) where n is the total number of nodes in the graph.
- **Union** - Input two nodes u and v to the Union method, and call the Find function to determine the group for each node. If the nodes already exist within the same group then do nothing and return false. Else, return true and merge the nodes into the group of the node with the highest rank, and update the rank of the group. On the first call of the Union function, each node is in a distinct group of equal rank, so we default to merging the group of node u into the group of node v. This function has a complexity of O(log n)

4.2 Implementation

The solution to the cost of the minimum spanning tree is performed in the following steps:

1. Create the Graph object through the Graph() class and call the buildGraph method to define a variable "points", and call the numNodes() method to define a variable "numNodes".
2. Pass the "points" and "numNodes" variables as inputs to the KruskalUF method, which performs the following steps in order to return the cost of the minimum spanning tree of the graph:
    - Calculate the length of the list "points" to determine the number of the edges in the graph.
    - Sort the list "points" in non-decreasing order of weight
    - Create an object to initialize the Union Find data structure through the Union-Find() class, and pass the "numNodes" as the input
    - Initialize the cost of the minimum spanning tree as zero, and the number of edges within the minimum spanning tree as zero (i.e., initialize an empty minimum spanning tree).
    - While the number of edges in the minimum spanning tree is less than the number of edges in the graph, traverse each edge (u,v), and call the Union method with the nodes u and v as input. If the Union method returns true, then add their edge weight to the cost of the minimum weight spanning tree, and increase the number of edges in the minimum spanning tree. Else, if it returns false, move to the next edge.

4.3 Complexity

To calculate the total complexity, we must consider:

- Initializing the nodes in Union Find data structure, where n is number of nodes: O(n)
- Sorting the edges, where m is the number of edges: O(m log n)
- Find called twice, a total of 2m times: O(m log n)
- Updating the minimum spanning tree: O(m)
- Union performed n-1 times: O(n log n)

Considering all componets, the complexity is O (m log n)

5 Results

5.1 Table with calculated MST

Input file	num_nodes	num_edges	MST
input_random_01_10.txt	10	9	29316
input_random_02_10.txt	10	11	16940
input_random_03_10.txt	10	13	-44448
input_random_04_10.txt	10	10	25217
input_random_05_20.txt	20	24	-32021
input_random_06_20.txt	20	24	25130
input_random_07_20.txt	20	28	-41693
input_random_08_20.txt	20	26	-37205
input_random_09_40.txt	40	56	-114203
input_random_10_40.txt	40	50	-31929
input_random_11_40.txt	40	50	-79570
input_random_12_40.txt	40	52	-79741
input_random_13_80.txt	80	108	-139926
input_random_14_80.txt	80	99	-198094
input_random_15_80.txt	80	104	-110571
input_random_16_80.txt	80	114	-233320
input_random_17_100.txt	100	136	-141960
input_random_18_100.txt	100	129	-271743
input_random_19_100.txt	100	137	-288906
input_random_20_100.txt	100	132	-229506
input_random_21_200.txt	200	267	-510185
input_random_22_200.txt	200	269	-515136
input_random_23_200.txt	200	269	-444357
input_random_24_200.txt	200	267	-393278
input_random_25_400.txt	400	540	-1119906

Input file	num_nodes	num_edges	MST
input_random_26_400.txt	400	518	-788168
input_random_27_400.txt	400	538	-895704
input_random_28_400.txt	400	526	-733645
input_random_29_800.txt	800	1063	-1541291
input_random_30_800.txt	800	1058	-1578294
input_random_31_800.txt	800	1076	-1664316
input_random_32_800.txt	800	1049	-1652119
input_random_33_1000.txt	1000	1300	-2089013
input_random_34_1000.txt	1000	1313	-1934208
input_random_35_1000.txt	1000	1328	-2229428
input_random_36_1000.txt	1000	1344	-2356163
input_random_37_2000.txt	2000	2699	-4811598
input_random_38_2000.txt	2000	2654	-4739387
input_random_39_2000.txt	2000	2652	-4717250
input_random_40_2000.txt	2000	2677	-4537267
input_random_41_4000.txt	4000	5360	-8722212
input_random_42_4000.txt	4000	5315	-9314968
input_random_43_4000.txt	4000	5340	-9845767
input_random_44_4000.txt	4000	5368	-8681447
input_random_45_8000.txt	8000	10705	-17844628
input_random_46_8000.txt	8000	10670	-18798446
input_random_47_8000.txt	8000	10662	-18741474
input_random_48_8000.txt	8000	10757	-18178610
input_random_49_10000.txt	10000	13301	-22079522
input_random_50_10000.txt	10000	13340	-22338561
input_random_51_10000.txt	10000	13287	-22581384
input_random_52_10000.txt	10000	13311	-22606313
input_random_53_20000.txt	20000	26667	-45962292

Input file	num_nodes	num_edges	MST
input_random_54_20000.txt	20000	26826	-45195405
input_random_55_20000.txt	20000	26673	-47854708
input_random_56_20000.txt	20000	26670	-46418161
input_random_57_40000.txt	40000	53415	-92003321
input_random_58_40000.txt	40000	53446	-94397064
input_random_59_40000.txt	40000	53242	-88771991
input_random_60_40000.txt	40000	53319	-93017025
input_random_61_80000.txt	80000	106914	-186834082
input_random_62_80000.txt	80000	106633	-185997521
input_random_63_80000.txt	80000	106586	-182065015
input_random_64_80000.txt	80000	106554	-180793224
input_random_65_100000.txt	100000	133395	-230698391
input_random_66_100000.txt	100000	133214	-230168572
input_random_67_100000.txt	100000	133524	-231393935
input_random_68_100000.txt	100000	133463	-231011693

5.2 Graph of the performance of Prim's Algorithm

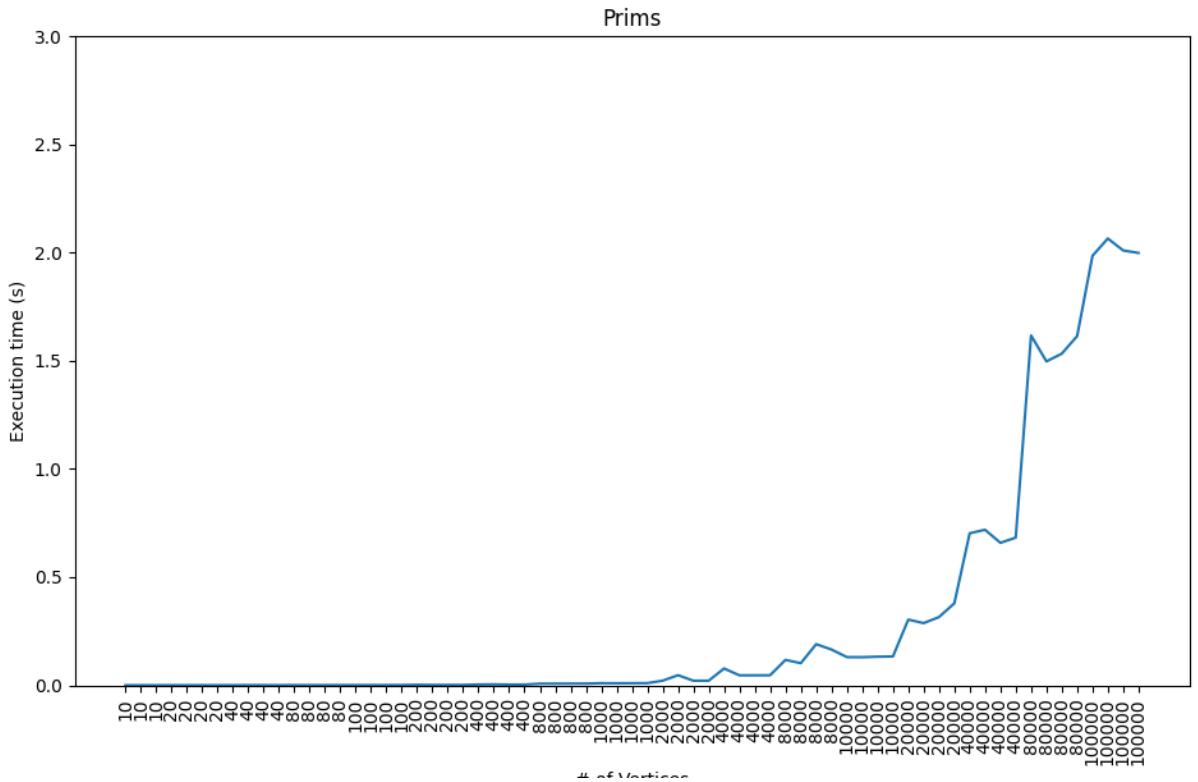

Figure 1: Performance of Prim's Algorithm

5.3 Graph of the performance of Kruskal's "naive" Algorithm

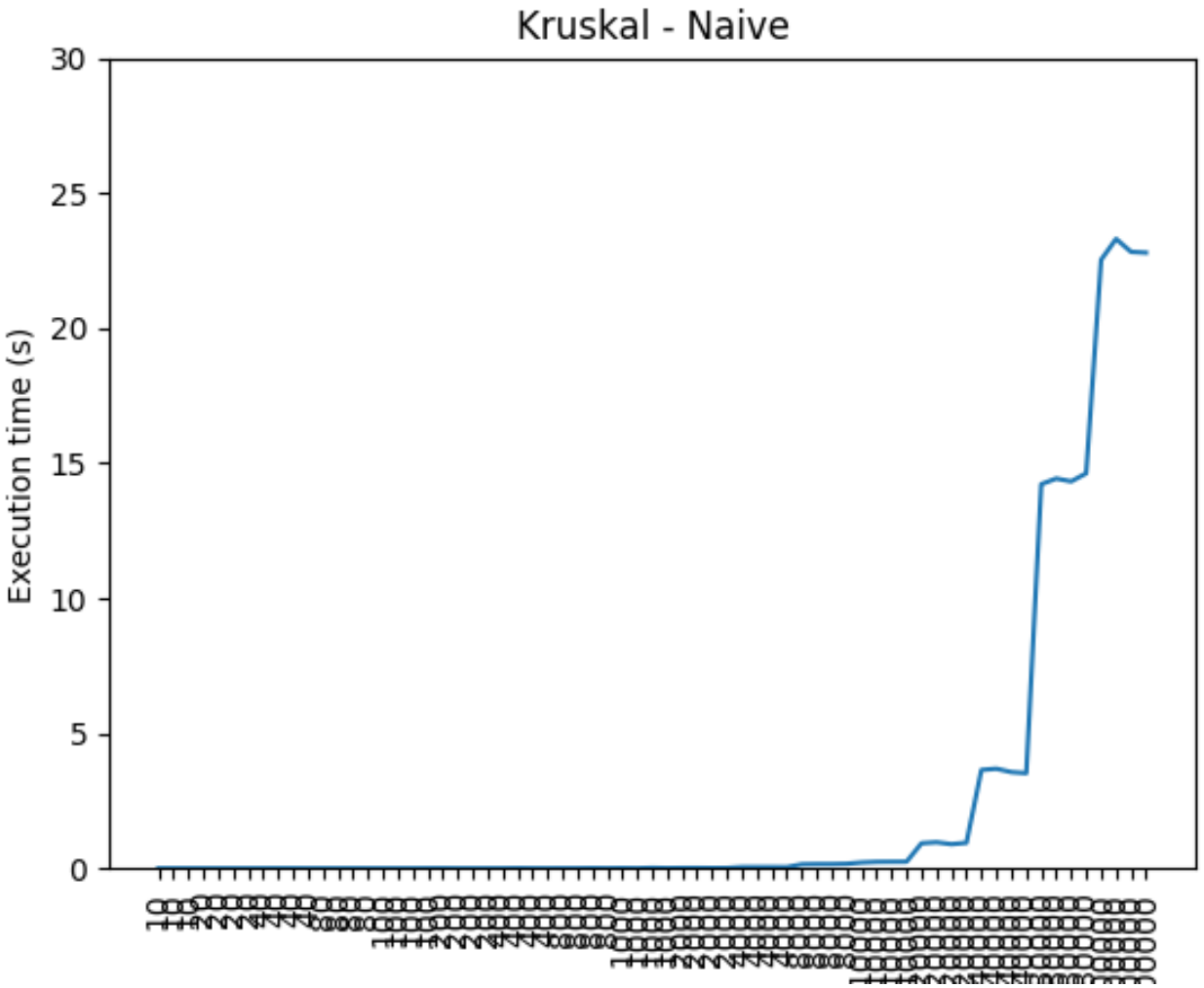

Figure 2: Performance of Kruskal's "naive" Algorithm

5.4 Graph of the performance of Kruskal's Union Find Algorithm

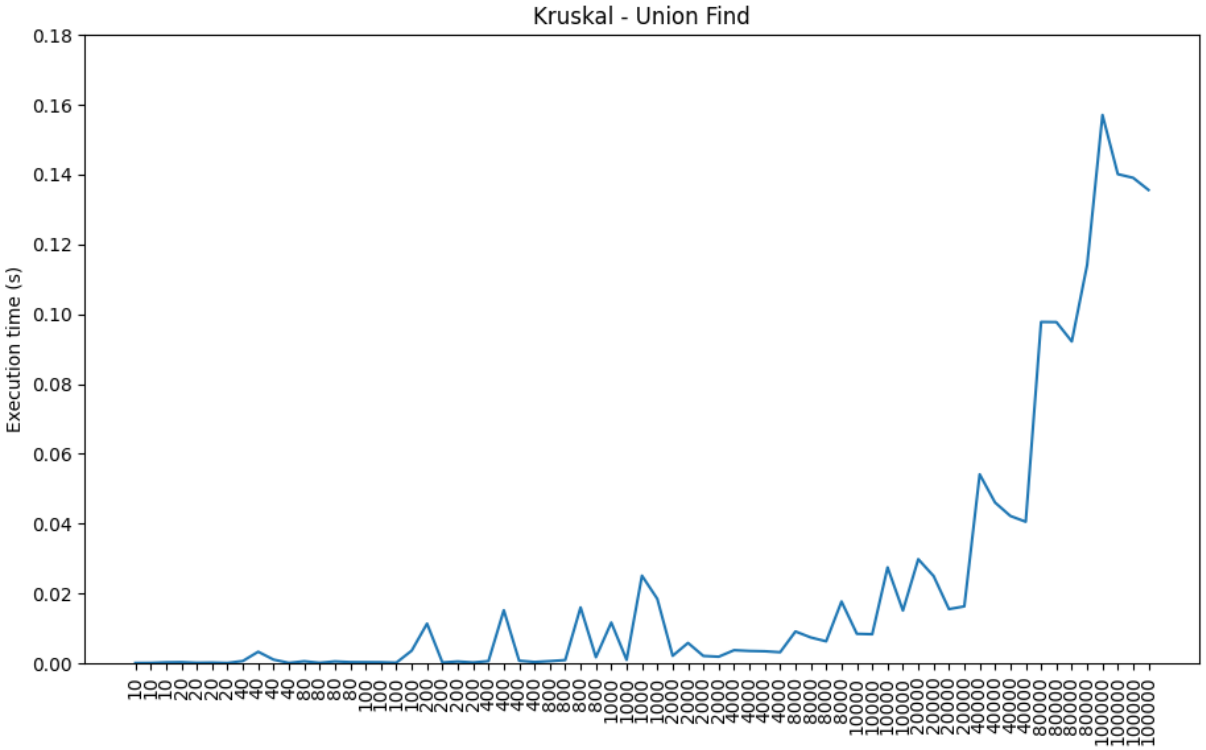

Figure 3: Performance of Kruskal's Union Find Algorithm

6 Conclusion

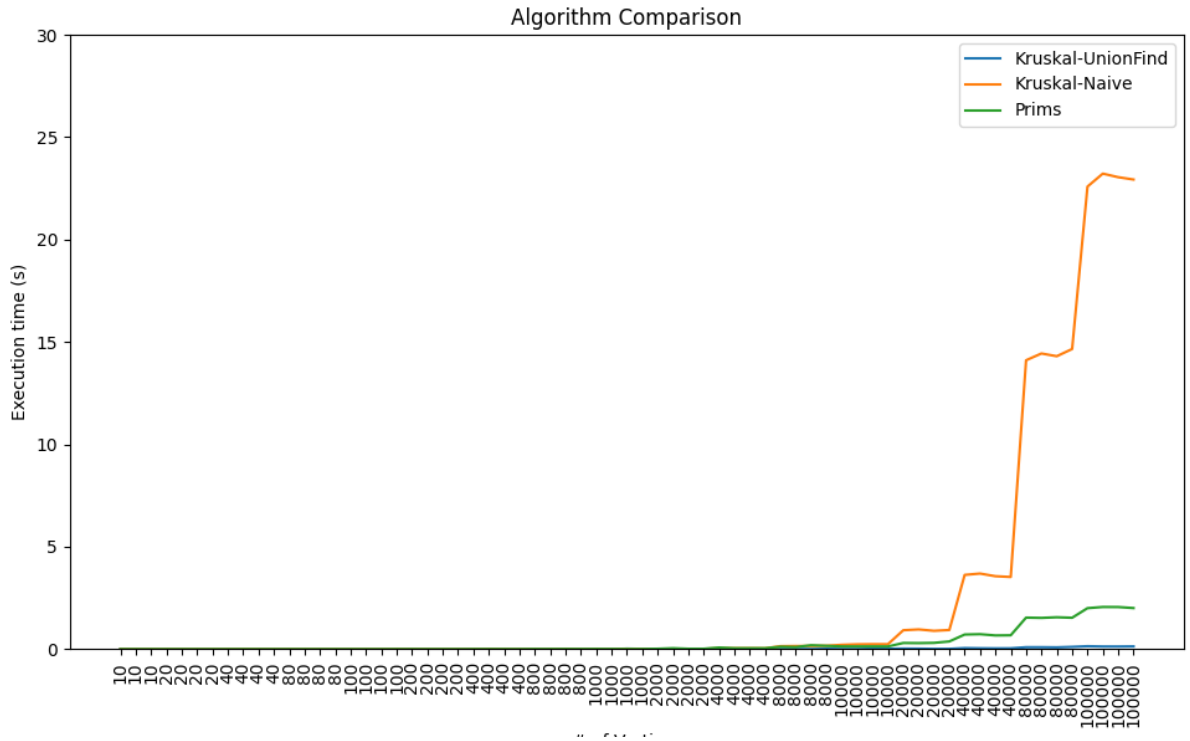

Figure 4: Comparison between the performances of the three algorithms

From Fig. 4 we can observe that for graphs with 1000 vertices or less the performance is more or less the same for the three different algorithms. As the number of vertices grows, the execution time of the Kruskal's "naive" algorithm grows exponentially and it can take nearly a minute to calculate the MST of the larger graphs within the dataset. The execution time of Prim's and Kruskal's Union Find are very similar. Prim's only takes a few seconds, and Kruskal's Union Find takes even less than a second, to calculate the MST of the larger graphs within the dataset. The results are consistent with the complexity of the algorithms. Kruskal's "naive" is the most complex algorithm, and while the other two have the same algorithmic complexity we still find that the Kruskal's Union Find has an execution time lower than Prim's.