UNIVERSITÀ DEGLI STUDI DI PADOVA
DEPARTMENT OF MATHEMATICS
COMPUTER SCIENCE

Advanced Algorithms
Minimum Spanning Trees

Homework 1 Report

Group: 5 Sara Buttau, Salvatore Gatto, Emad Efatinasab

April 25, 2022

Abstract

The problem of Minimum Spanning Trees has long been studied since 1920s, making it one of the fundamental problems in the theory of graphs and algorithms. Applications of Minimum Spanning Trees can vary from electronic circuits, in order to connect all its components by minimizing the cost of the wires, to the design of networks like computer networks, telecommunication networks and transportation networks. Several algorithms have been formulated to find an efficient solution to the Minimum Spanning Tree problem.
In this report we present our implementation of Prim's and Kruskal's (Naive and Efficient) algorithms. We first briefly introduce the problem in question and the three algorithms. Consequently, we explain our implementation choices and finally present our results, comparing the three algorithms' computational time also with respect to their high-order complexity (O(*mlog(n)*), O(*mn*)).

1 Minimum Spanning Trees: the theory

1.1 Introduction

Given an undirected, weighted and connected graph G with edges E and vertices V, a spanning tree is a subset of E that connects all of the vertices. A minimum spanning tree (hereon MST) is a subset T $\subseteq E$ of G such that

$$w(T) = \sum_{(u,v)\in T} w(u,v) \tag{1.1}$$

is minimized. In other words, it is a spanning tree whose sum of edge weights is as small as possible.

1.2 Prim's Algorithm

Prim's algorithm gradually builds the minimum spanning tree by adding edges one at a time. It starts from the starting vertex, and it incrementally constructs the MST with the minimum-weighted edge which connects the selected vertex with one unselected vertex. The process is repeated until the spanning tree contains all vertices.
A trivial implementation of Prim's algorithm searches the lightest edge by iterating over all possible edges, which takes O(m[1]). Extending this procedure to all vertices, we get an overall complexity of O(*mn*).
A more efficient implementation of Prim's uses the heap data structure, a priority queue that orders its elements according to their *key* or *priority*. A heap allows

[1] We indicate the number of edges in a graph with *m*, and the number of vertices with *n*

to efficiently extract the minimum-key element from its queue with a complexity of O(*log(n)*). Therefore, Prim's algorithm with the implementation of the heap becomes the following:

Algorithm 1 Prim's algorithm with heap

```
for each node u ∈ V do
    key[u] ← ∞
    mst ← null
end for
key[starting node] ← 0
Q ← V
while Q ≠ do
    u ← extractMin(Q)
    for each node v adjacent to u do
        if v ∈ Q and w(u, v) ≤ key[v] then
            key[v] ← w(u, v)
            mst[v] ← u
        end if
    end for
end while
```

The overall complexity of Prim's algorithm is therefore O(*mlog(n)*).

1.3 Naive Kruskal

Naive Kruskal is a greedy algorithm, since at each iteration it looks for the edge with the lowest weight in the graph. It gradually builds the minimum spanning tree, repeatedly checking that it is acyclic.
This implementation does not use an external data structure. Furthermore, in order to verify whether there exists a path between two nodes it utilizes a graph search algorithm, such as Depth First Search or Breadth First Search .
This implementation has a complexity of O(*mn*).

Algorithm 2 Kruskal's algorithm (Naive)

```
A ← ∅
sort edges of G by cost
for each edge e ∈ E, in ascending order of cost do
    if A ⋃ e is acyclic then
        A ← A ⋃ e
    end if
end for
```

1.4 Efficient Kruskal

Kruskal algorithm falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.We start from the edges with the lowest weight and keep adding edges until we reach our goal.

We will consider the data structure "Disjoint Set Union" for implementing Kruskal's algorithm, which will allow the algorithm to achieve the time complexity of O(*mlog(n)*).

Algorithm 3 Kruskal's algorithm with union find (Efficient kruskal)

$A \leftarrow \emptyset$
U=initialize(V)
for each edge $e = (V, W)$, in ascending order of weight **do**
  if $Find(U, V) \neq Find(U, W)$ **then**
    $A \leftarrow A \bigcup (U, V)$
    $Union(U, V, W)$
  end if
end for

2 Our implementation

Our algorithms are implemented in Python, and run on Google Colab. We implemented a class Graph in the following way:

- self.V: list of vertices of the graph.
- self.E: list of edges of the graph. An edge is a tuple of three elements, where the first element represents the source node, the second element the destination node and the third element is the weight of the edge.
- self.adjacency_list: default dictionary to represent the graph as an adjacency list.

The class Graph contains also the following methods:

- def add_edge(self, source_node, destination_node, weight): the method creates a tuple of (source_node, destination_node,weight) and adds it to the list "self.E". If source_node and destination_node are not inside the list of the vertices of the Graph, the method also updates the variable self.V. The graph is updated also appending the new edge to the adjacency list.
- def get_edges(self): the method returns the list of edges of the graph.
- def add_vertex(self,v): the method adds the vertex "v" to the list of vertices of the graph.
- def remove_vertex(self,v): the method removes the vertex "v" from the list of vertices of the graph.
- def get_vertices(self): the method returns the list of vertices of the graph.
- def get_value(self,key): the method returns the neighbours of the vertex corresponding to the key given as a parameter.
- def sort_edges(self): the method uses a lambda function to sort the list of edges "self.E" by the weight.
- def print_data(self): the method prints the adjancency_list of the graph.

As for what concerns the running time measurements, for each input file we read the graph and initialize its adjacency list, and counted the execution time of each algorithm using the *perf_counter_ns* function imported from the library *time*. We ran Prim's and Efficient Kruskal's algorithms 1000 times on each input graph, and computed the average running time, so to have more precise measurements. Due to significantly longer running times for Naive Kruskal's, we only ran it once per

input graphs. For this same reason, we ran Naive Kruskal's on a local environment. Furthermore, for each algorithm we also computed the weight of the returned MST. Lastly, we plotted the results using *matplotlib* library.

We have structured our work in the following way:

- our codes are divided into separate jupyter notebooks, one for Prim's algorithm, one for Kruskal's naive version, one for Kruskal's Union Find and one for comparing the results.
- a directory 'plots' contains the plots of the complexities we obtained, which are also present in this report
- a directory 'txt' contains the text files we used in our notebooks to reuse some outputs (such as input sizes or running times) without having to re-run the code multiple times.

All these can be found in the zip file we delivered.

2.1 Prim's implementation

To implement Prim's algorithm, we used Python's *heapq* data structure. The algorithm receives as input the graph's adjacency list and the starting vertex. It uses the following variables:

- mst: dictionary set in which the MST is constructed incrementally
- heap: heapq data structure which nodes are triples *(weight, vertex, next_vertex)*, which represent an edge. The heap's keys are the weights, sorted in ascending order.
- keys: an array which contains the vertices' priorities, i.e. their weights (initially set to infinity, except for the starting vertex, set to zero.

First, we initialize the keys of all nodes, setting the starting vertex's to zero, and the rest to infinity. To simplify comparisons among the weights, we chose to use a high value such as $1e6$ to represent infinity.

Instead of inserting all vertices into the heap at the beginning, we chose to insert few nodes at a time, so to reduce the computation time to the minimum.
At first, we insert only the edges related to the starting node, and heapify the heap structure using the *heapq.heapify()* method. We then iterate through the heap structure and extract the minimum using *heapq.heapop(heap)*, which preserves the heap invariant. To check that the extracted vertex has not been visited yet, we check that its key stored in the array *keys* is set to infinity.
This way, instead of going through a whole data structure to check if a value is present, we simply perform a check in constant time by having direct access to an array value at a given position. If the weight of the extracted edge is less than the key stored in the array, then the vertex is added to the *mst* and the new key is set to the weight value. Consequently, a new vertex is inserted into the heap, if it has not been visited yet.

2.2 Naive Kruskal's implementation

The naive implementation of Kruskal's algorithm starts executing the method *KruskalMST*. The method *KruskalMST* requires as parameter a Graph G, and it follows these steps:

1. It creates our MST "A".
2. The MST is initialised so to have the same number of vertices of the graph G.
3. The method creates the list of edges that it needs to check and sorts it by weight.

4. In this step, the algorithm iterates the list of sorted edges. For each edge, it checks that the addition of the edge to the minimum spanning tree does not imply the creation of a cycle.

5. To check the existence of a cycle, the algorithm calls the method *is_acyclic*. The method *is_acyclic* verifies whether there exists a path which links the source and the destination node.
   The first condition it checks is that source and destination nodes are different, in order to be sure not to have a self-loop. If the two nodes are different, it checks whether they are already present in the graph. If not, the nodes are added to the graph. Otherwise, the method calls the function *exists_path*, which checks whether such a path already exists through a Depth-First Search.
   If such a path does not exist, the edge will be added to the graph *A*. This step is repeated until the algorithm has checked all the edges of the graph.

6. The algorithm returns the MST *A* which the method has built.

2.3 Efficient Kruskal's implementation

Union Find is an algorithm to find the sets of connected graphs. This algorithm consists of two separate functions, the find and union functions. The find function needs two parameters, and the union function needs three parameters.

The find function takes a list that keeps track of the minimum spanning tree and our destination node. If the parent in the MST list is equivalent to the passed destination node, then we return the destination node. Otherwise, we recursively look for the destination node by calling the find function.

The union function's three parameters are the MST list, source node and destination node. First, we look for where the vertices(source node and destination node) are in terms of the MST. Then, we compare the values of the indices of the set tracking list and change the corresponding value in the MST list depending on the value comparison. If the value of roots of vertices are the same we return nothing. if the value of roots of vertices are different, then the index of the smaller value is set to the higher value and vice versa. If they are the same, then we create a new disjoint set.

The union find implementation of Kruskal's algorithm starts executing the method *KruskalUnionFind*. The method *KruskalUnionFind* requires as parameter a Graph G, and it follows these steps:

1. It creates list "mst".

2. then we will creat a DisjointSet object.

3. then we will sort the edges of G or our input graph.
4. In this step, the algorithm iterates the list of sorted edges. For each edge, it checks (by calling the find function in Disjointset) whether the source node found is different from the destination node found. In that case, it will append source node, destination node and weight to the mst.
5. Then we will call union function with source node and destination node as inputs.
6. The algorithm returns mst list which the method has built.

3 Results

3.1 Prim's Algorithm

As expected, the algorithm shows a complexity of O(*mlog(n)*), multiplied by some coefficient *c*. To estimate c, for each input we computed:

$$c\_estimates \leftarrow runtime/(m * log(n))$$

To plot our results, we took the average of all the coefficients stored in *c_estimates*.

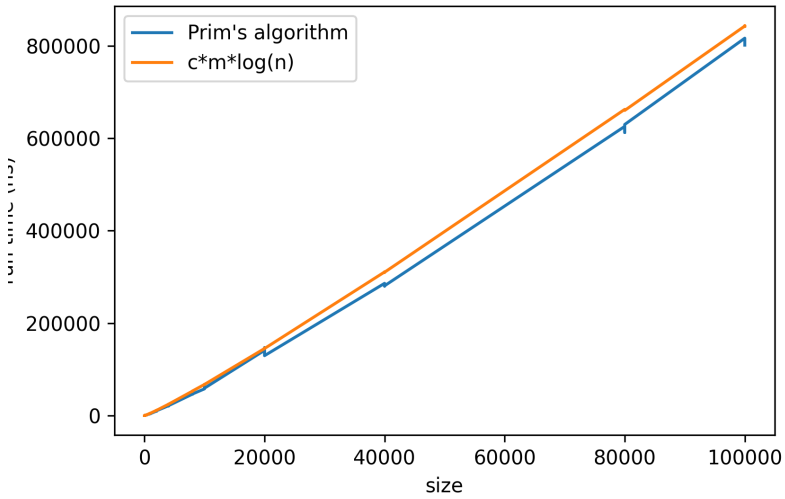

Figure 3.1: Algorithm complexity O(*mlog(n)*) in comparison with our implementation of Prim's

Figure 3.1 plots the algorithm's complexity in a linear scale. The orange line represents the O(*mlog(n)*) complexity, whilst the blue one is our implementation of Prim's, which asymptotically tends to the first one, as expected. We also plotted Prim's results in a logarithmic scale (see Figure 3.2).

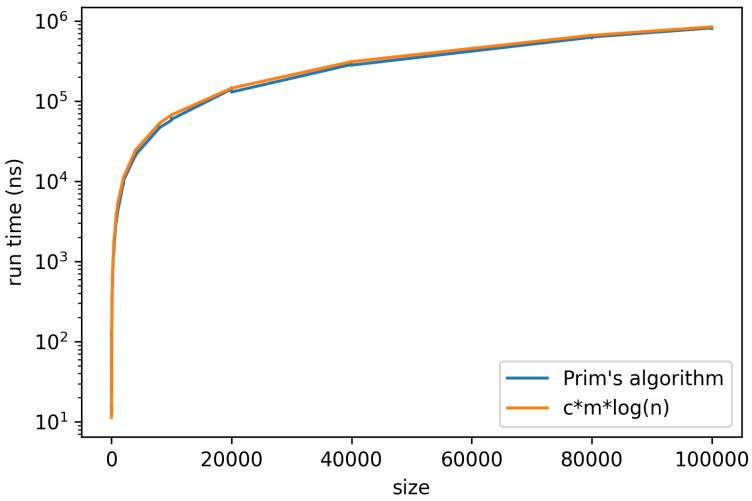

Figure 3.2: Algorithm complexity O(mlog(n)) in comparison with our implementation of Prim's in logarithmic scale

3.2 Naive Kruskal's Algorithm

Naive Kruskal's algorithm has a worse performance comparing to Prim's and Kruskal's algorithm with Union Find. The naive implementation of Kruskal's algorithm has a complexity of O(*mn*), where n is the number of vertices and m is the number of edges. The algorithm does not use an external data structure to accelerate the performance, but it just applies a graph search algorithm like Depth First Search or Breadth First Search. As we expected, the algorithm takes almost 9 hours to be executed on all input files. In this case, the estimate of c, for each input we computed is:

$$c \leftarrow runtime/(m * n)$$

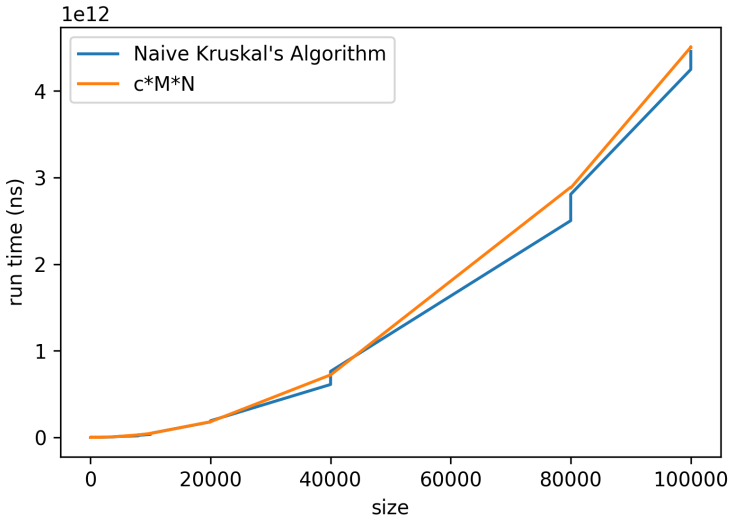

Figure 3.3: The orange line represents the complexity *O(mn)* in linear scale and is obtained by *c*m*n*.
The blue line represents the performance of our naive implementation of Kruskal's algorithm.

3.3 Efficient Kruskal

The algorithm shows a complexity of O(*mlog(n)*), as it was expected. Furthermore, Kruskal's algorithm with union find has a much better performance compared to naive Kruskal. Once again, to estimate the coefficient c, for each input we computed:

$$c\_estimates \leftarrow runtime/(m * log(n))$$

and then took the average of stored values in *c_estimates*.

As for the plots, figure 3.4 plots the algorithm's complexity in a linear scale, while figure 3.5 shows the logarithmic scale. The orange line represents the O(*mlog(n)*) complexity, whilst the blue one is our implementation of Kruskal union, which asymptotically tends to the first one, as expected.

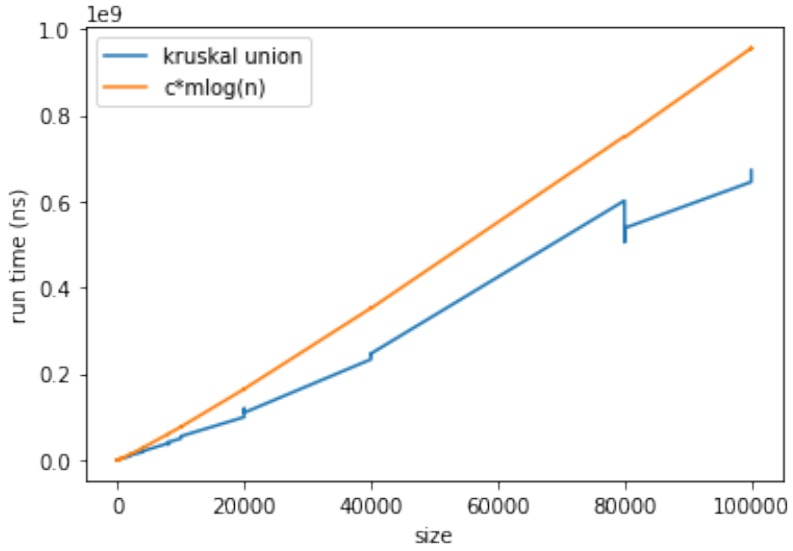

Figure 3.4: Algorithm complexity O(*mlog(n)*) in comparison with our implementation of Kruskal union

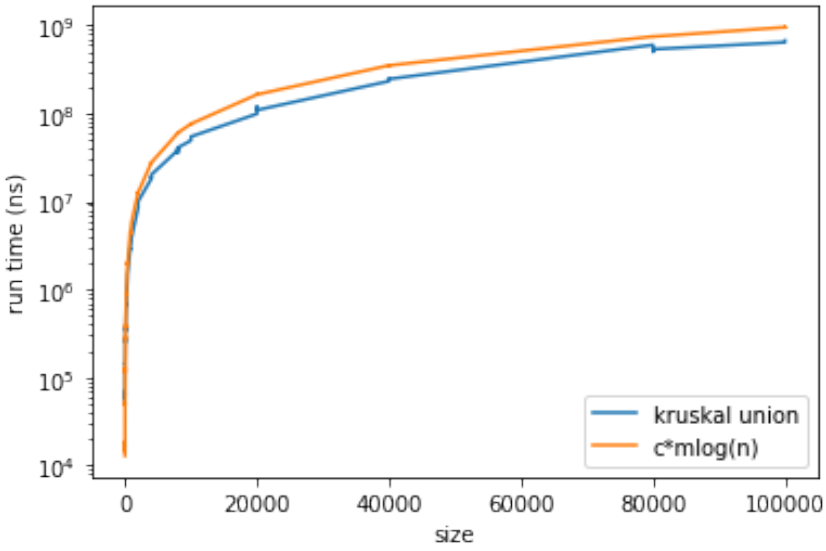

Figure 3.5: Algorithm complexity O(mlog(n)) in comparison with our implementation of Kruskal union in logarithmic scale

3.4 Comparisons

In the following section we compare the algorithms with one another. Theoretically, we would expect Prim and Efficient Kruskal to have more or less a similar complexity, which is O(*mlog(n)*) for both. As for Naive Kruskal, since its complexity is O(*mn*), we expect it to have a worse performance than the previous two.
Let us get into a more in-depth analysis by visualizing the plots.

3.4.1 Prim and Efficient Kruskal

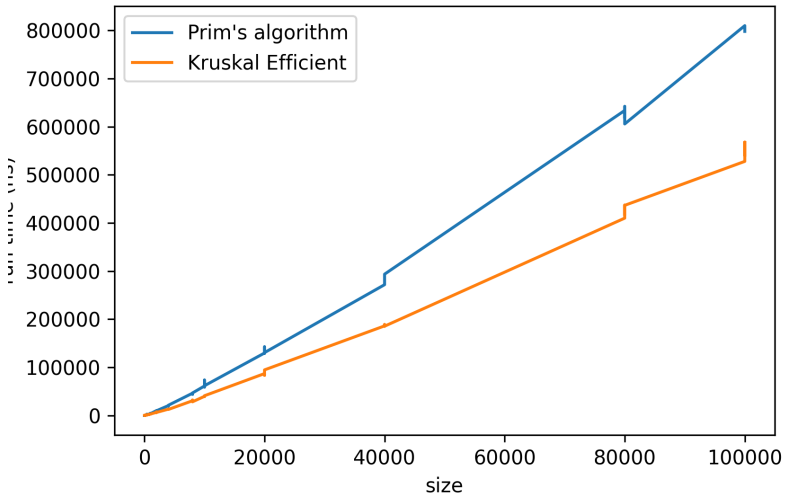

Figure 3.6: Comparing Prim's algorithm with Efficient Kruskal (with Union Find).

Figures 3.6 and 3.7 show Prim's complexity with respect to Efficient Kruskal's. In particular, Figure 3.7 is plotted in logarithmic scale. As the two plots show, the two algorithms perform quite well with the increase of the input size. As a matter of fact, along with the growth in the input dimension, the running times of the two algorithms grow logarithmically. Moreover, what is stressed by the graphs is that Kruskal's overall performance seems to be more efficient, over all sizes. However, while for larger input sizes (≥ 80000) Kruskal's performance worsens, Prim's running times decrese, albeit still remaining greater than Kruskal's. Finally, as for smaller inputs (≤ 10000) the two algorithms can be considered reasonably equivalent.

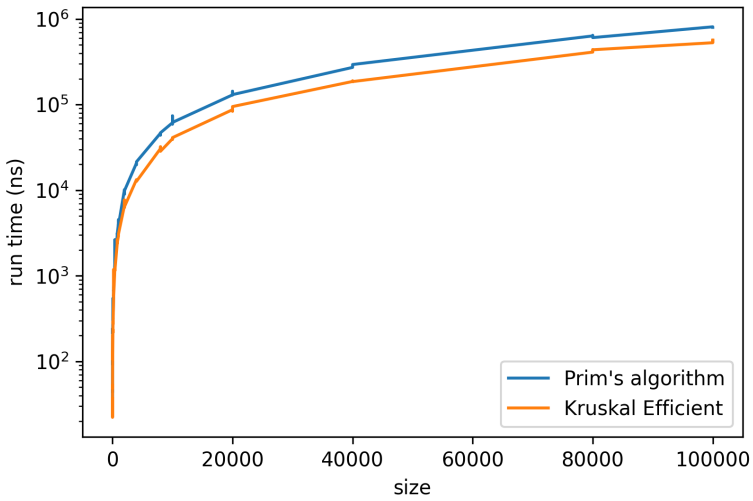

Figure 3.7: Comparing Prim's algorithm with Efficient Kruskal (with Union Find) - logarithmic scale.

3.4.2 Prim, Naive Kruskal and Efficient Kruskal

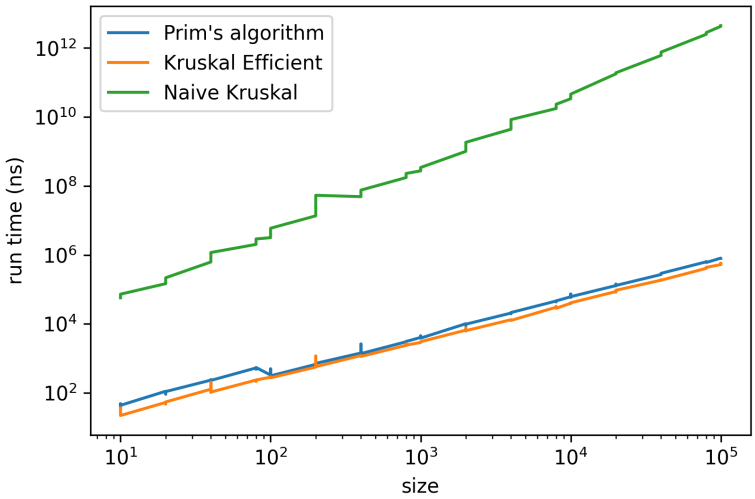

Figure 3.8: MST algorithms comparison: Prim, Naive and Efficient Kruskal

We compared the three algorithm's performances in Figure 3.8. We decided to

plot this in logarithmic scale as it allows to better appreciate the differences among the performances. Undoubtedly, Naive Kruskal's performance is the worst in terms of computational time, reaching values over $1e12$.
Instead, as previously shown, Prim and Efficient Kruskal have a similar performance, and require significantly less computational time. Nonetheless, such a behavior clearly meets our expectations, since Naive Kruskal's theoretical complexity is much higher than the remaining ones.

4 Conclusions

In conclusion, we have implemented three algorithms in Python for computing the minimum spanning tree given an input graph. In particular, we implemented Prim's algorithm with the heapq structure for the heap, Kruskal's naive algorithm and Kruskal's efficient algorithm with the Union Find data structure.
We ran the three algorithms on each input graph, measured their running time and computed the weights of the MST they returned (see Appendix). In section 2 we presented our implementation characteristics, and in section 3 we introduced our results. As expected, the overall worst-performing algorithm is Naive Kruskal's, having a complexity of O(mn). As for Prim's and Efficient Kruskal's, Kruskal's seems to be the most efficient one, especially for higher-dimension graphs. As for lower input sizes, we showed in Figure 3.6 that the two algorithms can be considered more or less equivalent.

Overall, computing the minimum spanning trees in graphs of increasing dimensions is a task that can be accomplished most efficiently by Kruskal's algorithm with Union Find, which provided good computation times on all graph sizes. The second-best algorithm in our implementation would be Prim's, followed by Naive Kruskal, which recorded the highest running times.

References

[1] Thomas H. Cormen (2009) *Introduction to Algorithms, Third Edition*, The MIT Press

[2] Lecture notes

5 Appendix

[1]

FILE	WEIGHT
input_random_01_10	29316
input_random_02_10	16940
input_random_03_10	-44448
input_random_04_10	25217
input_random_05_20	-32021
input_random_06_20	25130
input_random_07_20	-41693
input_random_08_20	-37205
input_random_09_40	-114203
input_random_10_40	-31929
input_random_11_40	-79570
input_random_12_40	-79741
input_random_13_80	-139926
input_random_14_80	-198094
input_random_15_80	-110571
input_random_16_80	-233320
input_random_17_100	-141960
input_random_18_100	-271743
input_random_19_100	-288906
input_random_20_100	-229506
input_random_21_200	-510185
input_random_22_200	-515136
input_random_23_200	-444357
input_random_24_200	-393278
input_random_25_400	-1119906
input_random_26_400	-788168
input_random_27_400	-895704
input_random_28_400	-733645
input_random_29_800	-1541291
input_random_30_800	-1578294
input_random_31_800	-1664316
input_random_32_800	-1652119

[1] See notebooks for more in-depth results on running times

input_random_33_1000	-2089013
input_random_34_1000	-1934208
input_random_35_1000	-2229428
input_random_36_1000	-2356163
input_random_37_2000	-4811598
input_random_38_2000	-4739387
input_random_39_2000	-4717250
input_random_40_2000	-4537267
input_random_41_4000	-8722212
input_random_42_4000	-9314968
input_random_43_4000	-9845767
input_random_44_4000	-8681447
input_random_45_8000	-17844628
input_random_46_8000	-18798446
input_random_47_8000	-18741474
input_random_48_8000	-18178610
input_random_49_10000	-22079522
input_random_50_10000	-22338561
input_random_51_10000	-22581384
input_random_52_10000	-22606313
input_random_53_20000	-45962292
input_random_54_20000	-45195405
input_random_55_20000	-47854708
input_random_56_20000	-46418161
input_random_57_40000	-92003321
input_random_58_40000	-94397064
input_random_59_40000	-88771991
input_random_60_40000	-93017025
input_random_61_80000	-186834082
input_random_62_80000	-185997521
input_random_63_80000	-182065015
input_random_64_80000	-180793224
input_random_65_100000	-230698391
input_random_66_100000	-230168572
input_random_67_100000	-231393935
input_random_68_100000	-231011693