|  |  |
| --- | --- |
| **Comparison and analysis of three algorithms for the Minimum Spanning Tree problem** | Logo  Description automatically generated with medium confidence  **University of Padova**  **Department of Mathematics**  **“Tullio Levi-Civita”** |
| **Syed Riaz Raza[[1]](#footnote-1)  Rana Mandal[[2]](#footnote-2) Muhammad Tabish[[3]](#footnote-3)**  \* **Homework 1 Report for “**[**Advanced Algorithms**](https://elearning.unipd.it/math/mod/assign/view.php?id=44431)**”** |

|  |  |
| --- | --- |
| **Keywords**  Advance Algorithm,  MST,  Minimum Spanning Tree,  Prims, Kruskal,  Union Find,  Heap, Graphs, Tree,  Asymptotic Complexity, Data Structure, Execution Time, Big O,  UNIPD,  Padova, Italy | **Abstract:**  **I**n this report we will do comparison and analysis between three algorithms for calculating the Minimum Spanning Tree problem given below:   1. **Prim's Algorithm**implemented with a Heap 2. **Naive Kruskal's Algorithm** having O(MN) complexity 3. **Efficient Kruskal's Algorithm** based on **Union-Find**   The report summarizes the result in following way:   * Visualize the result using matplotlib one-by-one * Comparing the result of one algorithm with other one and vice versa * Conclude the result |

Table of Contents

[1. Introduction: 4](#_Toc102237135)

[A. Definition of MST: 4](#_Toc102237136)

[a. Properties of the minimum spanning tree 4](#_Toc102237137)

[B. Generic Algorithm: 4](#_Toc102237138)

[a. Some notations for MST are given: 4](#_Toc102237139)

[b. To determine if a side is safe, the following theorem is used: 5](#_Toc102237140)

[2. Project Structure: 5](#_Toc102237141)

[A. Implementation: 5](#_Toc102237142)

[B. Dataset: 5](#_Toc102237143)

[C. Asymptotic Notation & Visualization: 6](#_Toc102237144)

[D. Flowchart: 7](#_Toc102237145)

[3. Question 1: 8](#_Toc102237146)

[A. Kruskal Naive: 8](#_Toc102237147)

[a. Working: 8](#_Toc102237148)

[b. Pseudocode: 8](#_Toc102237149)

[c. Code Structure: 9](#_Toc102237150)

[d. Complexity: 9](#_Toc102237151)

[e. Visualization 10](#_Toc102237152)

[B. Kruskal Union-Find Algorithm: 11](#_Toc102237153)

[b. Description 11](#_Toc102237154)

[c. Pseudo Code: 12](#_Toc102237155)

[d. Code Structure: 12](#_Toc102237156)

[e. Complexity 13](#_Toc102237157)

[f. Visualization: 13](#_Toc102237158)

[C. Prims Algorithm: 14](#_Toc102237159)

[b. Prims Algorithm implemented using Heap: 14](#_Toc102237160)

[c. Pseudo Code: 15](#_Toc102237161)

[d. Time complexities: 15](#_Toc102237162)

[e. Visualization: 16](#_Toc102237163)

[4. Question 2 17](#_Toc102237164)

[5. Conclusion 18](#_Toc102237165)

[6. Dataset Result 19](#_Toc102237166)

[A. Kruskal Naïve 19](#_Toc102237167)

[B. Kruskal Union Find: 21](#_Toc102237168)

[C. Prims Algorithm: 23](#_Toc102237169)

# Introduction:

## Definition of MST:

Given a weighted undirected graph. We want to find a subtree of this graph which connects all vertices (i.e., it is a spanning tree) and has the least weight (i.e., the sum of weights of all the edges is minimum) of all possible spanning trees. This spanning tree is called a minimum spanning tree.

### Properties of the minimum spanning tree

* A minimum spanning tree of a graph is unique, if the weight of all the edges are distinct. Otherwise, there may be multiple minimum spanning trees. (Specific algorithms typically output one of the possible minimum spanning trees).
* Minimum spanning tree is also the tree with minimum product of weights of edges. (It can be easily proved by replacing the weights of all edges with their logarithms)
* In a minimum spanning tree of a graph, the maximum weight of an edge is the minimum possible from all possible spanning trees of that graph. (This follows from the validity of Kruskal's algorithm).
* The maximum spanning tree (spanning tree with the sum of weights of edges being maximum) of a graph can be obtained similarly to that of the minimum spanning tree, by changing the signs of the weights of all the edges to their opposite and then applying any of the minimum spanning tree algorithm.

## Generic Algorithm:

A = empty\_set

while A doesn't form a spanning tree

find an edge (u, v) that is safe for A

A = A U {(u, v)}

return A

### Some notations for MST are given:

1. a cut (S; V \ S) of a graph G = (V; E) and a partition of V;
2. one side (u, v) ∈ E crosses the cut (S; V n S) if u ∈ S and v ∈ V \ S or vice versa.
3. a cut respects a set of sides A if no side of A crosses the cut.
4. given a cut, the side that passes through it with minimal weight is called the **light edge.**

### To determine if a side is safe, the following theorem is used:

**Theorem**: Let G = (V; E) be an undirected, connected, and weighted graph. Let A be a subset of E included in some MST of G, let (S; V \ S) be a cut respecting A, and let (u; v) be a light edge for (S; V \ S). Then (u; v) is safe for A.

# Project Structure:

## Implementation:

Other than the core data structures (for each algorithm), some functions are the same in all project files like:

**Core Program Units:**

* class Graph (diff for each algo)
* class MST: (diff for each algo)

**Additional functions (Same for all algo implementations):**

* funct load\_all\_dataset (Load All Dataset)
* funct populate\_each\_Graph\_from\_Dataset (Populate dataset to given Graph/MST class)
* funct execute\_each\_graph\_in\_dataset (Compute and execute each dataset with their complexity and export the result into csv)
* funct main (main calling function)

## Dataset**:**

The result created as output for each algorithm implementation is in .csv format, and all the project files implementing the algorithm are using the same format to export the data.

* n vertex (# of vertex in a graph)
* n edges (# of edges in a graph)
* nano seconds time (ns time took to compute MST for each graph)
* seconds time (s time took to compute MST for each graph)
* weight (weight of each MST)
* exe times (num\_time calculation done on each graph)

## Asymptotic Notation & Visualization:

To compute Asymptotic notation for each algorithm the following parameters were created for each graph in dataset (for each algorithm implementation):

**Note:** Our work here is to compute the complexity and visualize the behavior of MST algorithms when given a dataset of graphs with increasing nodes. In this case, our dataset is composed of graphs whose vertices is being repeated exactly 4 times.

* n\_vertices = Same number of Vertices/nodes in a dataset. (Exactly 4 e.g., 10 20)
* mean\_edges = Average of edges whose number of Vertices is same.
* time = Average of time whose number of Vertices is same.

Computing Ration and Constant:

**Note:** n here is the number of a graph in a dataset

* ratios= MST [estimated\_time][n+1] **/** MST [estimated\_time][n]
* constant= MST [estimated\_time] [n] / MST [mean\_edges][n]
* reference = MST [n]['mean\_edges']) \* MST[n]['n\_vertices']

We have created a separate file system which import the computational results of algorithm as mentioned in above paragraph and visualize the result of algorithms in following category

* Computational complexity of the algorithm
* Theoretical(C) computational complexity of the algorithm (reference variable)

**Note:** visualizing the comparison between Kruskal Union Find and Prims, as it has been assumed Kruskal UF and Prims is better than Kruskal Naïve.

## Flowchart:

The same structure is used to compute MST for all given three algorithms:

Diagram

Description automatically generated Diagram

Description automatically generated

**Figure 2:** Structure to compute and complexity and visualize data**.**

**Figure 1:** Coding structure to compute MST and saving the result in a .csv

# Question 1:

Run the three algorithms you have implemented (Prim, Kruskal naive and Kruskal efficient) on the graphs of the dataset. Measure the execution times of the three algorithms and create a graph showing the increase of execution times as the number of vertices in the graph increases. Compare the measured times with the asymptotic complexity of the algorithms. For each problem instance, report the weight of the minimum spanning tree obtained by your code.

## Kruskal Naive:

**Kruskal's algorithm** finds a [minimum spanning forest](https://en.wikipedia.org/wiki/Minimum_spanning_tree) of an undirected [edge-weighted graph](https://en.wikipedia.org/wiki/Weighted_graph). If the graph is [connected](https://en.wikipedia.org/wiki/Connectivity_(graph_theory)), it finds a [minimum spanning tree](https://en.wikipedia.org/wiki/Minimum_spanning_tree). (A minimum spanning tree of a connected graph is a subset of the [edges](https://en.wikipedia.org/wiki/Edge_(graph_theory)) that forms a tree that includes every [vertex](https://en.wikipedia.org/wiki/Vertex_(graph_theory)), where the sum of the [weights](https://en.wikipedia.org/wiki/Weighted_graph) of all the edges in the tree is minimized. For a disconnected graph, a minimum spanning forest is composed of a minimum spanning tree for each [connected component](https://en.wikipedia.org/wiki/Connected_component_(graph_theory)).)

It is a [greedy algorithm](https://en.wikipedia.org/wiki/Greedy_algorithm) in [graph theory](https://en.wikipedia.org/wiki/Graph_theory) as in each step it adds the next lowest-weight edge that will not form a [cycle](https://en.wikipedia.org/wiki/Cycle_(graph_theory)) to the minimum spanning forest.

### Working:

* Create a forest *F* (a set of trees), where each vertex in the graph is a separate [tree](https://en.wikipedia.org/wiki/Tree_(graph_theory))
* Create a set *S* containing all the edges in the graph
* While *S* is [nonempty](https://en.wikipedia.org/wiki/Nonempty) and *F* is not yet [spanning](https://en.wikipedia.org/wiki/Spanning_tree)
  + Remove an edge with minimum weight from *S*
  + If the removed edge connects two different trees, then add it to the forest *F*, combining two trees into a single tree

At the termination of the algorithm, the forest forms a minimum spanning forest of the graph. If the graph is connected, the forest has a single component and forms a minimum spanning tree.

### Pseudocode:

Kruskal-Naive(G)

         A = empty\_set

         sort edges if G by cost

         for each edge e in increasing order by weight

             if A U {e} is acyclic

                 A = A U {e}

         return A

### Code Structure:

    Class MergeSort:

        funct algorithm

        funct \_merge

    class KruskalNaive:

        funct kruskal\_naive

        funct \_is\_acyclic

        funct is\_there\_a\_path

        funct dfs

    class MST:

        funct kruskal\_naive

        funct get\_mst\_weight

    class Graph:

        funct \_\_init\_\_

        funct add\_vertex

        funct add edge

        funct remove\_vertex

        funct weightBetween

    funct load\_all\_dataset

    funct populate\_each\_Graph\_from\_Dataset

    funct execute\_each\_graph\_in\_dataset

    funct main

**Note:** lamda functions weren't used here because of bug found when a random link was missing after each iteration resulting into an error of listIndex out of Range

### Complexity:

To get the wight complexity we sorted the naive version of Kruskal's algorithm has a computational complexity equal to **O (mn)**. The idea behind this algorithm is to sort the sides in ascending order with respect to their weight w.

Once sorted, for each side it is checked whether adding it to the temporary graph creates a loop. If it creates a loop then that side will not be inserted, otherwise the side will be part of the MST.

Sorting the sides according to their weight and checking that the insertion of a side does not create a loop in the graph, we will obtain a tree whose sum of its sides will be minimal.

### Visualization

Chart, line chart

Description automatically generated

**Figure 3:** Kruskal Naive complexity with for each graph with equal number of nodes.

The graph just illustrated (fig. 3) shows the expected (in yellow) and effective (in blue) computational complexity for the Kruskal Naive algorithm with more executions of the algorithm. As can be seen from the image, the effective complexity curve remains slightly below the theoretical curve(C), and therefore we can say two complexities are comparable.

## Kruskal Union-Find Algorithm:

The naive version of Kruskal's algorithm repeatedly performs the acyclicity check operation of graph A. This continuous check is responsible for the computational complexity of the algo rhythm. To optimize performance, the control of the graph's cyclicality is captured through the implementation of a particular data structure: **disjoint sets.**

Consider the data structure ["**Disjoint Set Union**"](https://cp-algorithms.com/data_structures/disjoint_set_union.html) for implementing Kruskal's algorithm, which will allow the algorithm to achieve the time complexity of **O(M log N).**

#### Disjoint Set Union

This part discus about the data structure **Disjoint Set Union** or **DSU**. Often it is also called **Union Find** because of its two main operations.

This data structure provides the following capabilities. We are given several elements, each of which is a separate set. A DSU will have an operation to combine any two sets, and it will be able to tell in which set a specific element is. The classical version also introduces a third operation, it can create a set from a new element.

**Thus, the basic interface of this data structure consists of only three operations:**

* **make\_set(v)** - creates a new set consisting of the new element v
* **union\_sets(a, b**) - merges the two specified sets (the set in which the element a is located, and the set in which the element b is located)
* **find\_set(v)** - returns the representative (also called leader) of the set that contains the element v. This representative is an element of its corresponding set. It is selected in each set by the data structure itself (and can change over time, namely after union\_sets calls). This representative can be used to check if two elements are part of the same set or not. a and b are exactly in the same set, if **find\_set(a) == find\_set(b)**. Otherwise, they are in different sets.

As described in more detail later, the data structure allows you to do each of these operations in almost **O (1)** time on average.

Also, in one of the subsections an alternative structure of a DSU is explained, which achieves a slower average complexity of **O (log n)** but can be more powerful than the regular DSU structure.

### Description

Just as in the simple version of the Kruskal algorithm, we sort all the edges of the graph in non-decreasing order of weights. Then put each vertex in its own tree (i.e., its set) via calls to the **make\_set** function - it will take a total of **O(N)**. We iterate through all the edges (in sorted order) and for each edge determine whether the ends belong to different trees (with two **find\_set** calls in **O (1)** each). Finally, we need to perform the union of the two trees (sets), for which the DSU **union\_sets** function will be called - also in **O (1)**.

So, we get the total time complexity of **O (M \log N + N + M) = O (M \log N).**

### Pseudo Code:

Kruskal-Union-Find (G, w)

       A = empty\_set

       for each vertex belongs to G.V

         make-set(v)

       sort the edges of G.E into nondecreasing order by weight w

       for each edge (u, v) belongs to G.E, in nondecreasing weight order

         if find-set(u) != find-set(v)

           A = A U {(u, v)}

           Union(u, v)

       return A

### Code Structure:

    Class MergeSort:

        funct algorithm

        funct \_merge

    class KruskalUnionFind:

        funct kruskal\_union\_find

    class DisjointSet:

        funct \_\_init\_\_

        funct make\_set

        funct find\_set

        funct union\_by\_size

    class MST:

        funct kruskal\_union\_find

        funct get\_mst\_weight

    class Graph:

        funct \_\_init\_\_

        funct add\_vertex

        funct add edge

        funct remove\_vertex

        funct weightBetween

    funct load\_all\_dataset

    funct populate\_each\_Graph\_from\_Dataset

    funct execute\_each\_graph\_in\_dataset

    funct main

Note: lamda functions weren't used here because of bug found when a random link was missing after each iteration resulting into an error of listIndex out of Range

### Complexity

So, we get the total time complexity of:

### Visualization:

Chart, line chart

Description automatically generated

**Figure 4:** Kruskal with Union Find complexity with for each graph with equal number of nodes.

The graphs just illustrated (fig. 4,) show the expected (in yellow) and effective (in blue) computational complexity for the Kruskal Union Find algorithm. As it can be seen from the image, the curve of the effective complexity remains slightly above the theoretical curve as number of nodes increase.

## Prims Algorithm:

Prim’s Algorithm, an algorithm that uses the greedy approach to find the minimum spanning tree. It shares a similarity with the shortest path first algorithm. Spanning trees are the subset of Graph having all vertices covered with the minimum number of possible edges. **They are not cyclic and cannot be disconnected**. Spanning trees doesn’t have a cycle. A connected Graph can have more than one spanning tree.

* So, the major approach for the prim’s algorithm is finding the minimum spanning tree by the shortest path first algorithm.
* Basically, this algorithm treats the node as a single tree and keeps adding new nodes from the Graph.

#### Working of Prims Algorithm:

Prim's algorithm is a greedy algorithm that starts from one vertex and continue to add the edges with the smallest weight until the goal is reached. The steps to implement the prim's algorithm are given as follows -

* First, we have to initialize an MST with the randomly chosen vertex.
* Now, we have to find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree.
* Repeat step 2 until the minimum spanning tree is formed.

The applications of prim's algorithm are -

* Prim's algorithm can be used in network designing.
* It can be used to make network cycles.
* It can also be used to lay down electrical wiring cables.

### Prims Algorithm implemented using Heap:

1. Prim's algorithm selects the edge with the lowest weight between the group of vertexes already selected and the rest of vertexes so to implement Prim's algorithm, we need a minimum heap.
2. Each time we select an edge you add the new vertex to the group of vertexes We've already chosen, and all its adjacent edges go into the heap.
3. Then you choose the edge with the minimum value again from the heap.

### Pseudo Code:

Let X = nodes covered so far, V = all the nodes in the graph, E = all the edges in the graph

Pick an arbitrary initial node s and put that into X

for v ∈ V - X

key[v] = cheapest edge (u, v) with v ∈ X

while X ≠ V:

let v = extract-min(heap) (i.e., v is the node which has the minimal edge cost into X)

Add v to X

for each edge v, w ∈ E

if w ∈ V - X (i.e., w is a node which hasn’t yet been covered)

Delete w from heap

recompute key[w] = min(key[w], weight (v, w))

# key[w] would only change if weight of E (v,w) is **<** current weight of key).

reinsert w into the heap

### Time complexities:

* **Choosing minimum edge** = O (time of removing minimum) = **O(log(E)) = O(log(V))**
* **Inserting edges to heap** = O (time of inserting item to heap) = **1**
* **Minheap: Choosing minimum edge** = O (time of removing minimum from heap) = **O(log(E)) = O(log(V))**
* **Inserting edges to heap** = O (time of inserting item to heap) = **O(log(E)) = O(log(V))**

Remember that So, in total you have E inserts, and V minimum choosing (It's a tree in the end).

**So, in Min heap We'll get:**

### Visualization:

A picture containing line chart

Description automatically generated

**Figure 5:** Prim’s algorithm complexity with for each graph with equal number of nodes.

The graphs just illustrated (fig. 5) show the expected (in yellow) and effective (in blue) computational complexity for the Prim’s algorithm. As it can be seen from the image, the curve of the effective complexity remains below the theoretical curve. The difference is not great at the start but continues to increase as number of nodes increase. Maybe with even more large dataset, at the end it may reach a point where there the difference is greater than we analyzed in this report.

# Question 2

Comment on the results you have obtained: how do the algorithms behave with respect to the various instances? There is an algorithm that is always better than the others? Which of the three algorithms you have implemented is more efficient?

Chart, line chart

Description automatically generated

**Figure 6:** Comparison between Prim and Kruskal UF with execution of each graph.

The graphs just illustrated (fig. 6) show three comparisons between the Kruskal Union Find algorithm and Prim's algorithm on a linear scale

In this case it is possible to clearly see that Kruskal's UF algorithm turns out to be more efficient than Prim's algorithm. From this therefore we can conclude that the Kruskal UF algorithm always tends to do better than the others.

# Conclusion

At first, we had faced a lot of problems during the work of our assignment

* For Kruskal Naïve, it was getting the right complexity in our code, we faced problems with lambda function. We were trying to use lambda function for sorting, but after each sorting, one index was missing, we solved this problem by using merge sort.
* For Kruskal Union Find it was the same as Kruskal Naïve, but its implementation was easy than Kruskal Naïve as both their working is the same.
* For Prim’s algorithm we were facing problems related to heap library, which at the end we were able to solve.

To conclude, what we have turned out for the algorithms is in line with what we expected, in fact:

* Kruskal Naive is the slowest of the three algorithms and the least efficient in terms of average execution time for graphs with many nodes.
* Kruskal Union Find can be ranked first as it is the fastest and most efficient of the three algorithms, especially with large graphs.
* Prim can be ranked in second place with respect to Kruskal Union Find, having a sufficiently good execution time even for large graphs.

Concluding did good on our side, although our use of the Python language, although relatively simple, it was too slow as compared to working in Java. With java we could have made a quite good interactive application, and we wouldn’t have face typo error or library errors.

# Dataset Result

## Kruskal Naïve

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Dataset | Nodes | Edges | Time (ns) | Time (s) | Weight | Repetitions |
| 1 | 10 | 9 | 38309.58884 | 0.0000383 | 29316 | 13603 |
| 2 | 10 | 11 | 44351.39771 | 0.0000444 | 16940 | 16713 |
| 3 | 10 | 13 | 51044.94677 | 0.000051 | -44448 | 18487 |
| 4 | 10 | 10 | 43787.72818 | 0.0000438 | 25217 | 21286 |
| 5 | 20 | 24 | 98822.03781 | 0.0000988 | -32021 | 9787 |
| 6 | 20 | 24 | 105718.8574 | 0.0001057 | 25130 | 9276 |
| 7 | 20 | 28 | 134031.3407 | 0.000134 | -41693 | 7132 |
| 8 | 20 | 26 | 114722.4179 | 0.0001147 | -37205 | 8397 |
| 9 | 40 | 56 | 255071.3821 | 0.0002551 | -114203 | 3865 |
| 10 | 40 | 50 | 248330.5538 | 0.0002483 | -31929 | 3989 |
| 11 | 40 | 50 | 256882.2292 | 0.0002569 | -79570 | 3874 |
| 12 | 40 | 52 | 241759.0395 | 0.0002418 | -79741 | 3997 |
| 13 | 80 | 108 | 801811.0751 | 0.0008018 | -139926 | 1225 |
| 14 | 80 | 99 | 716404.7954 | 0.0007164 | -198094 | 1383 |
| 15 | 80 | 104 | 732120.7298 | 0.0007321 | -110571 | 1351 |
| 16 | 80 | 114 | 934089.6933 | 0.0009341 | -233320 | 1050 |
| 17 | 100 | 136 | 1141836.245 | 0.0011418 | -141960 | 874 |
| 18 | 100 | 129 | 924543.3148 | 0.0009245 | -271743 | 1080 |
| 19 | 100 | 137 | 922108.6626 | 0.0009221 | -288906 | 1067 |
| 20 | 100 | 132 | 926144.7015 | 0.0009261 | -229506 | 1072 |
| 21 | 200 | 267 | 2905119.571 | 0.0029051 | -510185 | 336 |
| 22 | 200 | 269 | 3079901.241 | 0.0030799 | -515136 | 328 |
| 23 | 200 | 269 | 3062370.963 | 0.0030624 | -444357 | 326 |
| 24 | 200 | 267 | 3449758.758 | 0.0034498 | -393278 | 289 |
| 25 | 400 | 540 | 11152807.22 | 0.0111528 | -1119906 | 89 |
| 26 | 400 | 518 | 9591474.857 | 0.0095915 | -788168 | 105 |
| 27 | 400 | 538 | 10962710.34 | 0.0109627 | -895704 | 90 |
| 28 | 400 | 526 | 10673518.1 | 0.0106735 | -733645 | 93 |
| 29 | 800 | 1063 | 37964609 | 0.0379646 | -1541291 | 26 |
| 30 | 800 | 1058 | 37701474.62 | 0.0377015 | -1578294 | 26 |
| 31 | 800 | 1076 | 36661267.07 | 0.0366613 | -1664316 | 27 |
| 32 | 800 | 1049 | 41528325.58 | 0.0415283 | -1652119 | 24 |
| 33 | 1000 | 1300 | 56174018.53 | 0.056174 | -2089013 | 17 |
| 34 | 1000 | 1313 | 63600892.13 | 0.0636009 | -1934208 | 15 |
| 35 | 1000 | 1328 | 59794412.06 | 0.0597944 | -2229428 | 16 |
| 36 | 1000 | 1344 | 61854901.81 | 0.0618549 | -2356163 | 16 |
| 37 | 2000 | 2699 | 236135664 | 0.2361357 | -4811598 | 4 |
| 38 | 2000 | 2654 | 236604744.5 | 0.2366047 | -4739387 | 4 |
| 39 | 2000 | 2652 | 234245281 | 0.2342453 | -4717250 | 4 |
| 40 | 2000 | 2677 | 246662528.5 | 0.2466625 | -4537267 | 4 |
| 41 | 4000 | 5360 | 912594104 | 0.9125941 | -8722212 | 1 |
| 42 | 4000 | 5315 | 968720773 | 0.9687208 | -9314968 | 1 |
| 43 | 4000 | 5340 | 899673136 | 0.8996731 | -9845767 | 1 |
| 44 | 4000 | 5368 | 1003618763 | 1.0036188 | -8681447 | 1 |
| 45 | 8000 | 10705 | 3698823422 | 3.6988234 | -17844628 | 1 |
| 46 | 8000 | 10670 | 3665222853 | 3.6652229 | -18798446 | 1 |
| 47 | 8000 | 10662 | 3684639303 | 3.6846393 | -18741474 | 1 |
| 48 | 8000 | 10757 | 3829282036 | 3.829282 | -18178610 | 1 |
| 49 | 10000 | 13301 | 5780728292 | 5.7807283 | -22079522 | 1 |
| 50 | 10000 | 13340 | 5819827165 | 5.8198272 | -22338561 | 1 |
| 51 | 10000 | 13287 | 5624067894 | 5.6240679 | -22581384 | 1 |
| 52 | 10000 | 13311 | 5843340473 | 5.8433405 | -22606313 | 1 |
| 53 | 20000 | 26667 | 23888895940 | 23.8888959 | -45962292 | 1 |
| 54 | 20000 | 26826 | 23449849119 | 23.4498491 | -45195405 | 1 |
| 55 | 20000 | 26673 | 23745057886 | 23.7450579 | -47854708 | 1 |
| 56 | 20000 | 26670 | 23544887011 | 23.544887 | -46418161 | 1 |
| 57 | 40000 | 53415 | 1.1889E+11 | 118.8903587 | -92003321 | 1 |
| 58 | 40000 | 53446 | 1.20359E+11 | 120.3589457 | -94397064 | 1 |
| 59 | 40000 | 53242 | 1.17939E+11 | 117.938669 | -88771991 | 1 |
| 60 | 40000 | 53319 | 1.18289E+11 | 118.2887952 | -93017025 | 1 |
| 61 | 80000 | 106914 | 6.4347E+11 | 643.4695731 | -186834082 | 1 |
| 62 | 80000 | 106633 | 6.58111E+11 | 658.1106275 | -185997521 | 1 |
| 63 | 80000 | 106586 | 6.6782E+11 | 667.8196926 | -182065015 | 1 |
| 64 | 80000 | 106554 | 6.56983E+11 | 656.9827757 | -180793224 | 1 |
| 65 | 100000 | 133395 | 1.11311E+12 | 1113.112826 | -230698391 | 1 |
| 66 | 100000 | 133214 | 1.11261E+12 | 1112.607324 | -230168572 | 1 |
| 67 | 100000 | 133524 | 1.13136E+12 | 1131.36465 | -231393935 | 1 |
| 68 | 100000 | 133463 | 1.1103E+12 | 1110.298442 | -231011693 | 1 |

## Kruskal Union Find:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Dataset | Nodes | Edges | Time (ns) | Time (s) | Weight | Repetitions |
| 1 | 10 | 9 | 54614.10765 | 0.0000546 | 29316 | 1765 |
| 2 | 10 | 11 | 58675.09989 | 0.0000587 | 16940 | 5506 |
| 3 | 10 | 13 | 71753.69458 | 0.0000718 | -44448 | 6090 |
| 4 | 10 | 10 | 50375.4487 | 0.0000504 | 25217 | 5906 |
| 5 | 20 | 24 | 153512.1652 | 0.0001535 | -32021 | 3099 |
| 6 | 20 | 24 | 177683.2212 | 0.0001777 | 25130 | 2080 |
| 7 | 20 | 28 | 144335.4093 | 0.0001443 | -41693 | 3115 |
| 8 | 20 | 26 | 134900.0453 | 0.0001349 | -37205 | 4418 |
| 9 | 40 | 56 | 326419.3285 | 0.0003264 | -114203 | 1102 |
| 10 | 40 | 50 | 297562.7688 | 0.0002976 | -31929 | 2976 |
| 11 | 40 | 50 | 272656.4812 | 0.0002727 | -79570 | 2893 |
| 12 | 40 | 52 | 323830.334 | 0.0003238 | -79741 | 2545 |
| 13 | 80 | 108 | 656220.8375 | 0.0006562 | -139926 | 1003 |
| 14 | 80 | 99 | 630728.5337 | 0.0006307 | -198094 | 1514 |
| 15 | 80 | 104 | 620904.9947 | 0.0006209 | -110571 | 941 |
| 16 | 80 | 114 | 826494.9064 | 0.0008265 | -233320 | 1335 |
| 17 | 100 | 136 | 949866.1314 | 0.0009499 | -141960 | 685 |
| 18 | 100 | 129 | 929337.5143 | 0.0009293 | -271743 | 877 |
| 19 | 100 | 137 | 873792.435 | 0.0008738 | -288906 | 846 |
| 20 | 100 | 132 | 935897.354 | 0.0009359 | -229506 | 1096 |
| 21 | 200 | 267 | 2034106.383 | 0.0020341 | -510185 | 517 |
| 22 | 200 | 269 | 1815860.123 | 0.0018159 | -515136 | 489 |
| 23 | 200 | 269 | 1745549.468 | 0.0017455 | -444357 | 188 |
| 24 | 200 | 267 | 2043655.382 | 0.0020437 | -393278 | 576 |
| 25 | 400 | 540 | 3539380.357 | 0.0035394 | -1119906 | 112 |
| 26 | 400 | 518 | 3355874.667 | 0.0033559 | -788168 | 225 |
| 27 | 400 | 538 | 3878332.127 | 0.0038783 | -895704 | 221 |
| 28 | 400 | 526 | 3395911.034 | 0.0033959 | -733645 | 145 |
| 29 | 800 | 1063 | 9213444.444 | 0.0092134 | -1541291 | 108 |
| 30 | 800 | 1058 | 10307181.55 | 0.0103072 | -1578294 | 103 |
| 31 | 800 | 1076 | 8155842.975 | 0.0081558 | -1664316 | 121 |
| 32 | 800 | 1049 | 8317628.44 | 0.0083176 | -1652119 | 109 |
| 33 | 1000 | 1300 | 9543010.87 | 0.009543 | -2089013 | 92 |
| 34 | 1000 | 1313 | 13849045.65 | 0.013849 | -1934208 | 46 |
| 35 | 1000 | 1328 | 11038460.87 | 0.0110385 | -2229428 | 69 |
| 36 | 1000 | 1344 | 9326254.639 | 0.0093263 | -2356163 | 97 |
| 37 | 2000 | 2699 | 20448892.86 | 0.0204489 | -4811598 | 42 |
| 38 | 2000 | 2654 | 22608710.26 | 0.0226087 | -4739387 | 39 |
| 39 | 2000 | 2652 | 21037453.33 | 0.0210375 | -4717250 | 45 |
| 40 | 2000 | 2677 | 20349945 | 0.0203499 | -4537267 | 40 |
| 41 | 4000 | 5360 | 46450720 | 0.0464507 | -8722212 | 20 |
| 42 | 4000 | 5315 | 44838038.46 | 0.044838 | -9314968 | 13 |
| 43 | 4000 | 5340 | 48587966.67 | 0.048588 | -9845767 | 18 |
| 44 | 4000 | 5368 | 44489063.16 | 0.0444891 | -8681447 | 19 |
| 45 | 8000 | 10705 | 116701075 | 0.1167011 | -17844628 | 8 |
| 46 | 8000 | 10670 | 114580162.5 | 0.1145802 | -18798446 | 8 |
| 47 | 8000 | 10662 | 103291377.8 | 0.1032914 | -18741474 | 9 |
| 48 | 8000 | 10757 | 109845587.5 | 0.1098456 | -18178610 | 8 |
| 49 | 10000 | 13301 | 130327450 | 0.1303275 | -22079522 | 6 |
| 50 | 10000 | 13340 | 133966585.7 | 0.1339666 | -22338561 | 7 |
| 51 | 10000 | 13287 | 147425150 | 0.1474252 | -22581384 | 6 |
| 52 | 10000 | 13311 | 140303457.1 | 0.1403035 | -22606313 | 7 |
| 53 | 20000 | 26667 | 274782600 | 0.2747826 | -45962292 | 3 |
| 54 | 20000 | 26826 | 299046433.3 | 0.2990464 | -45195405 | 3 |
| 55 | 20000 | 26673 | 282015300 | 0.2820153 | -47854708 | 3 |
| 56 | 20000 | 26670 | 278584666.7 | 0.2785847 | -46418161 | 3 |
| 57 | 40000 | 53415 | 839815200 | 0.8398152 | -92003321 | 1 |
| 58 | 40000 | 53446 | 708695700 | 0.7086957 | -94397064 | 1 |
| 59 | 40000 | 53242 | 679061400 | 0.6790614 | -88771991 | 1 |
| 60 | 40000 | 53319 | 720011500 | 0.7200115 | -93017025 | 1 |
| 61 | 80000 | 106914 | 1817621900 | 1.8176219 | -186834082 | 1 |
| 62 | 80000 | 106633 | 1619099800 | 1.6190998 | -185997521 | 1 |
| 63 | 80000 | 106586 | 1476226800 | 1.4762268 | -182065015 | 1 |
| 64 | 80000 | 106554 | 1592142800 | 1.5921428 | -180793224 | 1 |
| 65 | 100000 | 133395 | 2651459600 | 2.6514596 | -230698391 | 1 |
| 66 | 100000 | 133214 | 2160093100 | 2.1600931 | -230168572 | 1 |
| 67 | 100000 | 133524 | 2475796100 | 2.4757961 | -231393935 | 1 |
| 68 | 100000 | 133463 | 2364580300 | 2.3645803 | -231011693 | 1 |

## Prims Algorithm:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Dataset | Nodes | Edges | Time (ns) | Time (s) | Weight | Repetitions |
| 1 | 10 | 9 | 73384.47605 | 0.0000734 | 29316 | 8873 |
| 2 | 10 | 11 | 85553.08836 | 0.0000856 | 16940 | 10921 |
| 3 | 10 | 13 | 97357.07557 | 0.0000974 | -44448 | 9912 |
| 4 | 10 | 10 | 86931.52518 | 0.0000869 | 25217 | 10762 |
| 5 | 20 | 24 | 246858.1311 | 0.0002469 | -32021 | 3965 |
| 6 | 20 | 24 | 218168.3059 | 0.0002182 | 25130 | 4328 |
| 7 | 20 | 28 | 253380.6183 | 0.0002534 | -41693 | 3898 |
| 8 | 20 | 26 | 227881.9873 | 0.0002279 | -37205 | 4317 |
| 9 | 40 | 56 | 606313.1716 | 0.0006063 | -114203 | 1632 |
| 10 | 40 | 50 | 561691.547 | 0.0005617 | -31929 | 1757 |
| 11 | 40 | 50 | 566737.393 | 0.0005667 | -79570 | 1743 |
| 12 | 40 | 52 | 576771.5317 | 0.0005768 | -79741 | 1721 |
| 13 | 80 | 108 | 1440989.358 | 0.001441 | -139926 | 688 |
| 14 | 80 | 99 | 1414840.605 | 0.0014148 | -198094 | 730 |
| 15 | 80 | 104 | 1369863.437 | 0.0013699 | -110571 | 725 |
| 16 | 80 | 114 | 1521327.684 | 0.0015213 | -233320 | 649 |
| 17 | 100 | 136 | 1916219.344 | 0.0019162 | -141960 | 520 |
| 18 | 100 | 129 | 1859230.528 | 0.0018592 | -271743 | 536 |
| 19 | 100 | 137 | 1958658.725 | 0.0019587 | -288906 | 512 |
| 20 | 100 | 132 | 1903094.246 | 0.0019031 | -229506 | 524 |
| 21 | 200 | 267 | 4419799.484 | 0.0044198 | -510185 | 225 |
| 22 | 200 | 269 | 4485446.865 | 0.0044854 | -515136 | 223 |
| 23 | 200 | 269 | 4513123.795 | 0.0045131 | -444357 | 220 |
| 24 | 200 | 267 | 4533419.136 | 0.0045334 | -393278 | 221 |
| 25 | 400 | 540 | 10407676.17 | 0.0104077 | -1119906 | 95 |
| 26 | 400 | 518 | 10170872.37 | 0.0101709 | -788168 | 97 |
| 27 | 400 | 538 | 10533477.81 | 0.0105335 | -895704 | 94 |
| 28 | 400 | 526 | 10292546.25 | 0.0102925 | -733645 | 96 |
| 29 | 800 | 1063 | 23454574.36 | 0.0234546 | -1541291 | 42 |
| 30 | 800 | 1058 | 23645769.29 | 0.0236458 | -1578294 | 42 |
| 31 | 800 | 1076 | 23929648.95 | 0.0239296 | -1664316 | 41 |
| 32 | 800 | 1049 | 23516296.57 | 0.0235163 | -1652119 | 42 |
| 33 | 1000 | 1300 | 30235913.06 | 0.0302359 | -2089013 | 33 |
| 34 | 1000 | 1313 | 30572566.41 | 0.0305726 | -1934208 | 32 |
| 35 | 1000 | 1328 | 30896512.25 | 0.0308965 | -2229428 | 32 |
| 36 | 1000 | 1344 | 30906809.16 | 0.0309068 | -2356163 | 32 |
| 37 | 2000 | 2699 | 70873597.64 | 0.0708736 | -4811598 | 14 |
| 38 | 2000 | 2654 | 69789933.64 | 0.0697899 | -4739387 | 14 |
| 39 | 2000 | 2652 | 69632619.64 | 0.0696326 | -4717250 | 14 |
| 40 | 2000 | 2677 | 70234522.86 | 0.0702345 | -4537267 | 14 |
| 41 | 4000 | 5360 | 157046280.5 | 0.1570463 | -8722212 | 6 |
| 42 | 4000 | 5315 | 156028482.8 | 0.1560285 | -9314968 | 6 |
| 43 | 4000 | 5340 | 157181312 | 0.1571813 | -9845767 | 6 |
| 44 | 4000 | 5368 | 156269539.5 | 0.1562695 | -8681447 | 6 |
| 45 | 8000 | 10705 | 347440138.5 | 0.3474401 | -17844628 | 2 |
| 46 | 8000 | 10670 | 348194793 | 0.3481948 | -18798446 | 2 |
| 47 | 8000 | 10662 | 344711363.5 | 0.3447114 | -18741474 | 2 |
| 48 | 8000 | 10757 | 347015763 | 0.3470158 | -18178610 | 2 |
| 49 | 10000 | 13301 | 442112535.5 | 0.4421125 | -22079522 | 2 |
| 50 | 10000 | 13340 | 446760593.5 | 0.4467606 | -22338561 | 2 |
| 51 | 10000 | 13287 | 444743531.5 | 0.4447435 | -22581384 | 2 |
| 52 | 10000 | 13311 | 442997466 | 0.4429975 | -22606313 | 2 |
| 53 | 20000 | 26667 | 974610952 | 0.974611 | -45962292 | 1 |
| 54 | 20000 | 26826 | 973262147 | 0.9732621 | -45195405 | 1 |
| 55 | 20000 | 26673 | 974384105 | 0.9743841 | -47854708 | 1 |
| 56 | 20000 | 26670 | 973670869 | 0.9736709 | -46418161 | 1 |
| 57 | 40000 | 53415 | 2136591753 | 2.1365918 | -92003321 | 1 |
| 58 | 40000 | 53446 | 2133940840 | 2.1339408 | -94397064 | 1 |
| 59 | 40000 | 53242 | 2125702872 | 2.1257029 | -88771991 | 1 |
| 60 | 40000 | 53319 | 2132769220 | 2.1327692 | -93017025 | 1 |
| 61 | 80000 | 106914 | 4700060910 | 4.7000609 | -186834082 | 1 |
| 62 | 80000 | 106633 | 4695625792 | 4.6956258 | -185997521 | 1 |
| 63 | 80000 | 106586 | 4694558420 | 4.6945584 | -182065015 | 1 |
| 64 | 80000 | 106554 | 4705053462 | 4.7050535 | -180793224 | 1 |
| 65 | 100000 | 133395 | 6060222656 | 6.0602227 | -230698391 | 1 |
| 66 | 100000 | 133214 | 6042919511 | 6.0429195 | -230168572 | 1 |
| 67 | 100000 | 133524 | 6050503941 | 6.0505039 | -231393935 | 1 |
| 68 | 100000 | 133463 | 6087769864 | 6.0877699 | -231011693 | 1 |

End of the Report.

1. **Syed Riaz Raza; E-mail:** [syedriaz.raza@studenti.unipd.it](mailto:syedriaz.raza@studenti.unipd.it); **Portfolio**: [riazraza.me](https://riazraza.me/) [↑](#footnote-ref-1)
2. **Rana Mandal; E-mail:** [rana.mandal@studenti.unipd.it](mailto:rana.mandal@studenti.unipd.it); [↑](#footnote-ref-2)
3. **Muhammad Tabish; E-mail:** [muhammad.tabish@studenti.unipd.it](mailto:muhammad.tabish@studenti.unipd.it); [↑](#footnote-ref-3)