Assignment 1-Advanced Algorithm

Sofia Chiarello 2056126

Melissa Tanios 2041602

Introduction

To implement this assignment we used Python programming language and the unit second for time measurement for easier readability.
We divided our work starting from the implementation of the three algorithms and then studying their complexity and time.
In this report each section describes the implementation of the algorithm and how we have studied the complexity. In the conclusion section it is described the comparison between them.

Prim Algorithm

Implementation

To implement Prim's Algorithm, I have created one method Prim that takes as input a graph and a starting node and returns the total weight obtained from a certain graph.
In the main, the program is iterating over the files in the directory mst_dataset and reading each file alone to create a graph. The first line represents the number of nodes and edges which are saved in variables. The graph is formed in a way that for each node u , we have all the connected nodes v with the weight of u and v.
After reading all the file and forming the graph, the timer starts to measure the function `prim` based on the number of callings indicated.
The `prim` function:

- Created a set `mst` that stores the minimum spanning tree obtained, i.e, at the beginning it has the starting node (the second parameter of the function).
- Initialized all parents to NULL and keys to INFINITE.
- Push to the minheap the first node with a key updated to 0.
- While the minheap is not empty:
  - Pop the minimum value from the heap.
  - If the popped value is not in the mst, add it (once the node is in the mst, we do not update anymore its key and parent).
  - Explore all the vertices v connected to u through an edge (u,v) of weight w.
  - Update the parent, key, and minHeap if the edge is minimum and the node is not already a part of MST.
  - Add the node with a weight less than the key value to the minheap.

Other implementations choices are the creation of two supported functions:

- plotResult that takes as input two lists of tuples of number of vertices and time to plot the lines of the number of vertices in function of the execution time of the measured time of prim algorithm and m log n in log scale

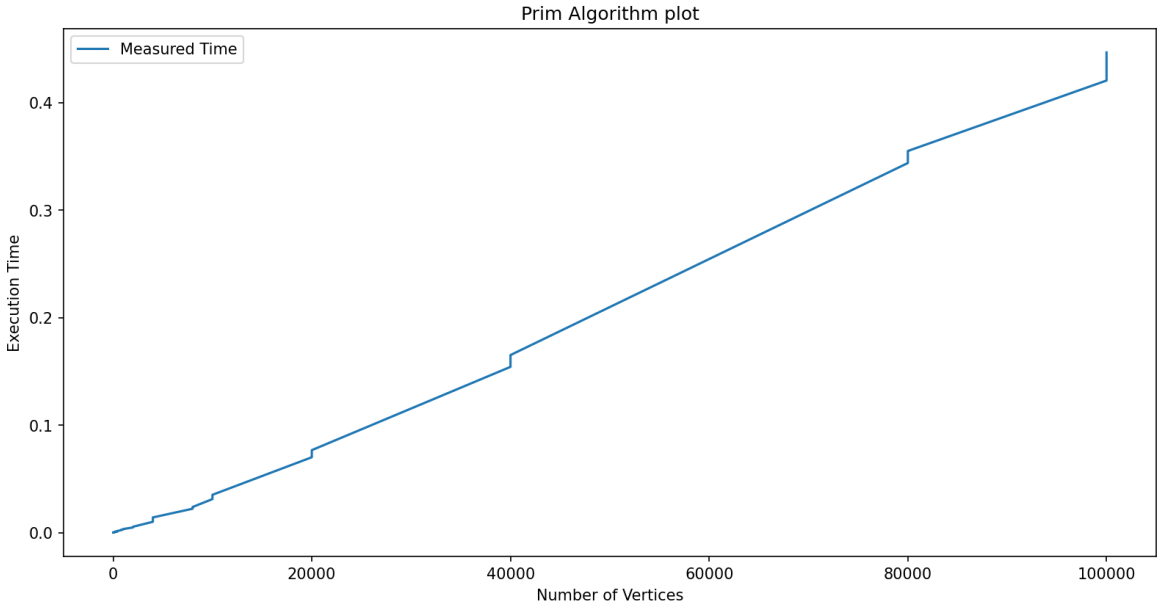
- plotMeasuredTime to plot only the measured time of the algorithm.
- printOutput that takes as input a list of tuples of values obtained to print the values of size, time, complexity, mlog n obtained from the algorithm.

Question 1:

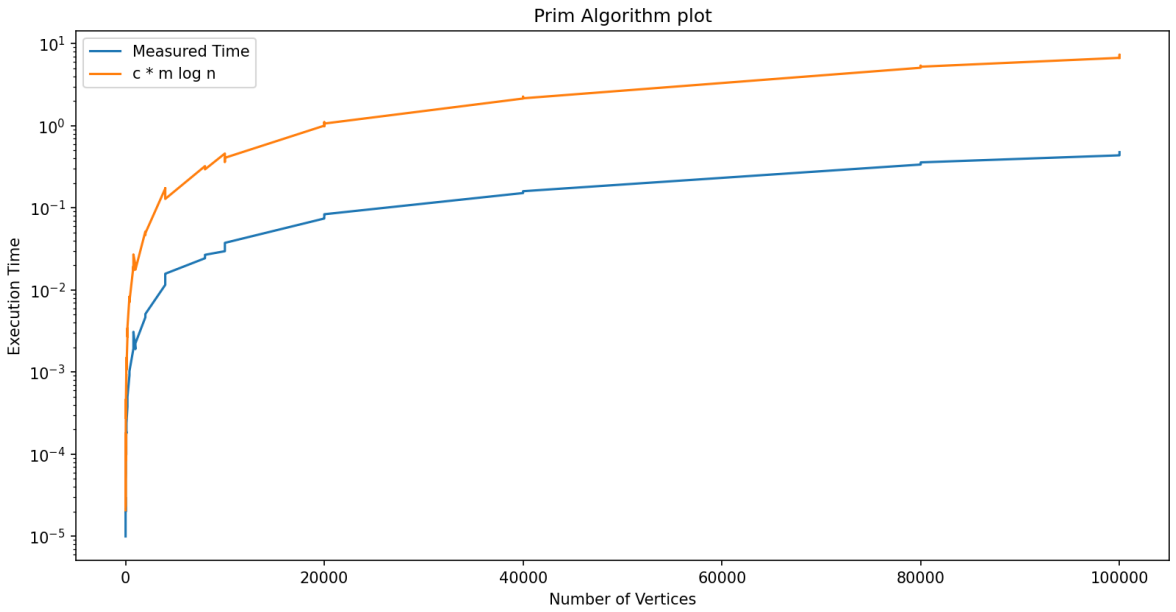
For Number of Calls = 1, the execution time of Prim's Algorithm over 68 files containing each random graph with vertices varying between 10 and 100000 is **4.603325366973877 s**. The time needed to execute graphs increases as the number of vertices increases. The average time needed is (s): **0.0676959612790276**

For number of calls = 100, as used in the algorithm, the total execution time is: **447.3743817806244 s** with an average of**: 6.579035026185653 s.**

This graph represents the execution time of Prim's algorithm in function of the number of vertices for a number of calls = 100**.**



From this graph, we can see that for both m log(n) and the measured time of Prim's algorithm, the execution time is increasing when the number of vertices is increasing. the c_estimate sum is equal to 0.00019416634141206733 and its average is 2.8553873737068723e-06



Size	Time of result
10	1.002788543701172e-05
10	1.996755599975586e-05
10	2.9914379119873047e-05
10	9.973049163818359e-06
20	4.986286163330078e-05
20	3.986358642578125e-05
20	1.9943714141845704e-05
20	2.9947757720947265e-05
40	9.008407592773437e-05
40	5.997657775878906e-05
40	7.007598876953125e-05
40	7.997989654541015e-05
80	0.00015958070755004884
80	9.970903396606445e-05
80	0.0001396465301513672
80	0.00018953084945678712
100	0.0002393627166748047
100	0.00017983913421630858
100	0.00017950057983398437
100	0.0001795172691345215
200	0.0003793525695800781
200	0.00047871828079223635
200	0.0003793144226074219
200	0.000498652458190918
400	0.0010272407531738282
400	0.0009177446365356446
400	0.001017153263092041
400	0.0009175038337707519
800	0.0022040963172912597
800	0.0021048235893249513
800	0.0019344735145568849
800	0.0030915665626525877

```
   1000              0.0022454142570495607
   1000              0.0020844197273254396
   1000              0.001991763114929199
   1000              0.0019032597541809083
   2000              0.005027780532836914
   2000              0.005095212459564209
   2000              0.004643974304199219
   2000              0.004832005500793457
   4000              0.015464627742767334
   4000              0.015766913890838622
   4000              0.013168997764587402
   4000              0.011539134979248047
   8000              0.026810703277587892
   8000              0.026185805797576903
   8000              0.025224740505218505
   8000              0.024347643852233886
  10000              0.037473807334899904
  10000              0.031796021461486815
  10000              0.029661247730255125
  10000              0.03320180892944336
  20000              0.07590766429901123
  20000              0.07419640064239502
  20000              0.08361054182052613
  20000              0.08053056478500366
  40000              0.15154582262039185
  40000              0.15955180406570435
  40000              0.1590648126602173
  40000              0.1528717064857483
  80000              0.3369351506233215
  80000              0.3514390230178833
  80000              0.35799886226654054
  80000              0.34812422513961794
 100000              0.43697901964187624
 100000              0.4565686631202698


 100000              0.47695388078689577
 100000              0.4361734485626221
-----------------------------------------
```

Below you can find the weight and time taken for each file with the number of vertices of the graph in each of these files.

Weight:	Time:	Vertices:	File:
29316	1.002788543701172e-05	10	input_random_01_10

16940	1.996755599975586e-05	10	input_random_02_10
-44448	2.9914379119873047e-05	10	input_random_03_10
25217	9.973049163818359e-06	10	input_random_04_10
-32021	4.986286163330078e-05	20	input_random_05_20
25130	3.986358642578125e-05	20	input_random_06_20
-41693	1.9943714141845704e-05	20	input_random_07_20
-37205	2.9947757720947265e-05	20	input_random_08_20
-114203	9.008407592773437e-05	40	input_random_09_40
-31929	5.997657775878906e-05	40	input_random_10_40
-79570	7.007598876953125e-05	40	input_random_11_40
-79741	7.997989654541015e-05	40	input_random_12_40
-139926	0.00015958070755004884	80	input_random_13_80
-198094	9.970903396606445e-05	80	input_random_14_80
-110571	0.0001396465301513672	80	input_random_15_80
-233320	0.00018953084945678712	80	input_random_16_80
-141960	0.0002393627166748047	100	input_random_17_100
-271743	0.00017983913421630858	100	input_random_18_100
-288906	0.00017950057983398437	100	input_random_19_100
-229506	0.0001795172691345215	100	input_random_20_100
-510185	0.0003793525695800781	200	input_random_21_200
-515136	0.00047871828079223635	200	input_random_22_200
-444357	0.0003793144226074219	200	input_random_23_200
-393278	0.000498652458190918	200	input_random_24_200
-1119906	0.0010272407531738282	400	input_random_25_400
-788168	0.0009177446365356446	400	input_random_26_400
-895704	0.001017153263092041	400	input_random_27_400

-733645	0.0009175038337707519	400	input_random_28_400
-1541291	0.0022040963172912597	800	input_random_29_800
-1578294	0.0021048235893249513	800	input_random_30_800
-1664316	0.0019344735145568849	800	input_random_31_800
-1652119	0.0030915665626525877	800	input_random_32_800
-2089013	0.0022454142570495607	1000	input_random_33_1000
-1934208	0.0020844197273254396	1000	input_random_34_1000
-2229428	0.001991763114929199	1000	input_random_35_1000
-2356163	0.0019032597541809083	1000	input_random_36_1000
-4811598	0.005027780532836914	2000	input_random_37_2000
-4739387	0.005095212459564209	2000	input_random_38_2000
-4717250	0.004643974304199219	2000	input_random_39_2000
-4537267	0.004832005500793457	2000	input_random_40_2000
-8722212	0.015464627742767334	4000	input_random_41_4000
-9314968	0.015766913890838622	4000	input_random_42_4000
-9845767	0.013168997764587402	4000	input_random_43_4000
-8681447	0.011539134979248047	4000	input_random_44_4000
-17844628	0.026810703277587892	8000	input_random_45_8000
-18798446	0.026185805797576903	8000	input_random_46_8000
-18741474	0.025224740505218505	8000	input_random_47_8000
-18178610	0.024347643852233886	8000	input_random_48_8000
-22079522	0.037473807334899904	10000	input_random_49_10000
-22338561	0.031796021461486815	10000	input_random_50_10000
-22581384	0.029661247730255125	10000	input_random_51_10000
-22606313	0.03320180892944336	10000	input_random_52_10000
-45962292	0.07590766429901123	20000	input_random_53_20000

-45195405	0.07419640064239502	20000	input_random_54_20000
-47854708	0.08361054182052613	20000	input_random_55_20000
-46418161	0.08053056478500366	20000	input_random_56_20000
-92003321	0.15154582262039185	40000	input_random_57_40000
-94397064	0.15955180406570435	40000	input_random_58_40000
-88771991	0.1590648126602173	40000	input_random_59_40000
-93017025	0.1528717064857483	40000	input_random_60_40000
-186834082	0.3369351506233215	80000	input_random_61_80000
-185997521	0.3514390230178833	80000	input_random_62_80000
-182065015	0.35799886226654054	80000	input_random_63_80000
-180793224	0.34812422513961794	80000	input_random_64_80000
-230698391	0.43697901964187624	100000	input_random_65_100000
-230168572	0.4565686631202698	100000	input_random_66_100000
-231393935	0.47695388078689577	100000	input_random_67_100000
-231011693	0.4361734485626221	100000	input_random_68_100000

The total weights obtained by the minimum spanning tree of Prim’s Algorithm are:

Weights: [29316, 16940, -44448, 25217, -32021, 25130, -41693, -37205, -114203, -31929, -79570, -79741, -139926, -198094, -110571, -233320, -141960, -271743, -288906, -229506, -510185, -515136, -444357, -393278, -1119906, -788168, -895704, -733645, -1541291, -1578294, -1664316, -1652119, -2089013, -1934208, -2229428, -2356163, -4811598, -4739387, -4717250, -4537267, -8722212, -9314968, -9845767, -8681447, -17844628, -18798446, -18741474, -18178610, -22079522, -22338561, -22581384, -22606313, -45962292, -45195405, -47854708, -46418161, -92003321, -94397064, -88771991, -93017025, -186834082, -185997521, -182065015, -180793224, -230698391, -230168572, -231393935, -231011693]

Using a heap data structure, the complexity of Prim's algorithm is in time O(|m| log |n|) where |m| is the number of edges and |n| is the number of vertices. In the graph above, we can see the plot of the asymptotic graph. In the table below, we have for each graph size, the corresponding asymptotic complexity of the algorithm.

The c_estimate value = Time/ Size for each graph is approximately equal to 0.00000285 and the of all c estimate is 0.00019148786748647686.

The c * m log n values are increasing with increasing time and size of the graph.

Size	Time of result	Constant(Time/Size)	Asymptotic Complexity	Ratio
10	1.002788543701172e-05	0.000001	0.000021	0.000000
10	1.996755599975586e-05	0.000002	0.000051	1.991203
10	2.9914379119873047e-05	0.000003	0.000090	1.498149
10	9.973049163818359e-06	0.000001	0.000023	0.333386
20	4.986286163330078e-05	0.000002	0.000179	4.999761
20	3.986358642578125e-05	0.000002	0.000143	0.799464
20	1.9943714141845704e-05	0.000001	0.000084	0.500299
20	2.9947757720947265e-05	0.000001	0.000117	1.501614
40	9.008407592773437e-05	0.000002	0.000465	3.008041
40	5.997657775878906e-05	0.000001	0.000277	0.665784
40	7.007598876953125e-05	0.000002	0.000323	1.168389
40	7.997989654541015e-05	0.000002	0.000384	1.141331
80	0.00015958070755004884	0.000002	0.000944	1.995260
80	9.970903396606445e-05	0.000001	0.000541	0.624819
80	0.00013964653015136 72	0.000002	0.000796	1.400540
80	0.00018953084945678712	0.000002	0.001184	1.357218
100	0.0002393627166748047	0.000002	0.001499	1.262922
100	0.00017983913421630858	0.000002	0.001068	0.751325
100	0.00017950057983398437	0.000002	0.001132	0.998117
100	0.00017951726913452 15	0.000002	0.001091	1.000093
200	0.0003793525695800781	0.000002	0.002683	2.113181
200	0.00047871828079223635	0.000002	0.003411	1.261935
200	0.0003793144226074219	0.000002	0.002703	0.792354
200	0.000498652458190918	0.000002	0.003527	1.314615
400	0.0010272407531738282	0.000003	0.008309	2.060033
400	0.0009177446365356446	0.000002	0.007121	0.893408
400	0.00101715326309204 1	0.000003	0.008197	1.108318
400	0.0009175038337707519	0.000002	0.007229	0.902031
800	0.0022040963172912597	0.000003	0.019577	2.402275
800	0.0021048235893249513	0.000003	0.018607	0.954960
800	0.0019344735145568849	0.000002	0.017392	0.919067
800	0.0030915665626525877	0.000004	0.027098	1.598144
800	0.0030915665626525877	0.000004	0.027098	1.598144
1000	0.0022454142570495607	0.000002	0.020164	0.726303
1000	0.0020844197273254396	0.000002	0.018905	0.928301
1000	0.001991763114929199	0.000002	0.018271	0.955548
1000	0.0019032597541809083	0.000002	0.017670	0.955565
2000	0.005027780532836914	0.000003	0.051572	2.641668
2000	0.00509521245956420 9	0.000003	0.051392	1.013412
2000	0.004643974304199219	0.000002	0.046806	0.911439
2000	0.004832005500793457	0.000002	0.049160	1.040489
4000	0.015464627742767334	0.000004	0.171874	3.200457
4000	0.015766913890838622	0.000004	0.173763	1.019547
4000	0.013168997764587402	0.000003	0.145814	0.835230
4000	0.011539134979248047	0.000003	0.128438	0.876235
8000	0.026810703277587892	0.000003	0.322425	2.323459
8000	0.026185805797576903	0.000003	0.313881	0.976692
8000	0.025224740505218505	0.000003	0.302134	0.963298
8000	0.024347643852233886	0.000003	0.294227	0.965229
10000	0.037473807334899904	0.000004	0.459079	1.539114
10000	0.031796021461486815	0.000003	0.390665	0.848487
10000	0.029661247730255125	0.000003	0.362988	0.932860
10000	0.03320180892944336	0.000003	0.407050	1.119367
20000	0.07590766429901123	0.000004	1.002347	2.286251
20000	0.07419640064239502	0.000004	0.985591	0.977456
20000	0.08361054182052613	0.000004	1.104310	1.126881
20000	0.08053056478500366	0.000004	1.063511	0.963163
40000	0.15154582262039185	0.000004	2.144446	1.881842
40000	0.15955180406570435	0.000004	2.259045	1.052829
40000	0.1590648126602173	0.000004	2.243554	0.996948
40000	0.1528717064857483	0.000004	2.159320	0.961066
80000	0.3369351506233215	0.000004	5.083660	2.204039
80000	0.3514390230178833	0.000004	5.288557	1.043046
80000	0.35799886226654054	0.000004	5.384897	1.018666
80000	0.34812422513961794	0.000004	5.234794	0.972417
100000	0.43697901964187624	0.000004	6.710978	1.255239
100000	0.4565686631202698	0.000005	7.002315	1.044830
100000	0.4769538807868957 7	0.000005	7.331982	1.044649
100000	0.4361734485626221	0.000004	6.702021	0.914498

Kruskal Algorithm using Union Find

Implementation

To implement this algorithm I first create two classes that represent the objects I will use: `UnionFind` and `Graph`.
Class `UnionFind` is the data structure needed to make the algorithm faster than primitive Kruskal, it has two lists: `parent` is the list of parents for each vertex and `rank` is the depth of each tree. I have implemented the methods needed for the algorithm:

- `init(n)`: initialize the data structure as every vertices that is in the graph has itself as parent and its rank is equal zero;
- `find(k)`: return the name of the set that contains the searched node, so reaching the edge such that parent(k) = k;
- `union(a,b)`: return the union of two sets. First at all, it invokes `find` for the two edges to obtain the names i and j that are the sets that contain them. If x==y return, cause we are considering the same set.
    - if rank(x) is bigger than rank(y): parent[x] become y and it's summed the rank of y to x;
    - if rank(y) is bigger than rank(x): parent[y] become x and it's summed the rank of x to y;
    - otherwise parent[x] become y and it's enough to add 1 to rank(y);

The other class `Graph` is used to represent as an object the data given as file txt. It is memorized the number of vertices, the number of edges and the edges as lists in the form `[vertex1, vertex2, weight of edged]`. The only method is `add_vertex` that is used to create the graph.

Other implementations choices is the creation of two supported functions:

- `create_graph` is used to create the object graph given data
- `plotResult` is used to plot the graph comparing two values.

To implement the algorithm I have created a function called `kruskal_algo(g)`. It has just a parameter `g` that is the graph of type `Graph`. First at all, it initializes the UnionFind structure with the values of g, and sorted the edges of g by nondecreasing order. For each edge in the graph, the algorithm calls the function `find` for the two vertices of the edge and compare the results, if they are different it is added to the minimum spanning tree and it is called `union` to make the sets of the two vertices just one. Otherwise this edge does not have to be added to the MST. The algorithm returns a list of the edges that compose the MST(`result`), and `tot_weight` that is the sum of all the weights of the edges of the MST.

Measure of the execution time

To study the complexity of this algorithm we need to compute how much time it takes for each graph to compute the MST. For each file in the dataset, it first creates the graph and disables the garbage collector which could add more time in the execution. At this point, with the help of the function `perf_counter_ns()` (in the library time) it computes the starting

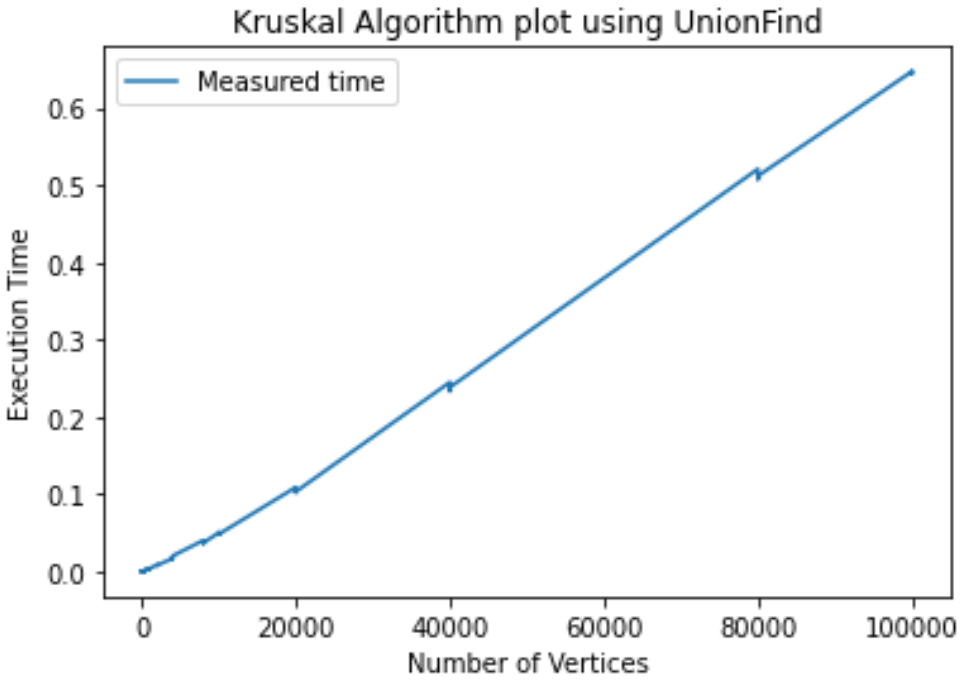
time and the end time after the execution of the Kruskal algorithm for each graph in the dataset. It calls the algorithm for a certain number of times to prevent errors.
For each graph it computes the complexity that in the worst case the algorithm can reach, that is equal to O(m logn). At this point, we have all the information to study the complexity of this algorithm.

Question 1

With a number of calls equal to 1, the Efficient Kruskal algorithm takes **7.994028917000001s** and by average **0.11755924877941179s**.

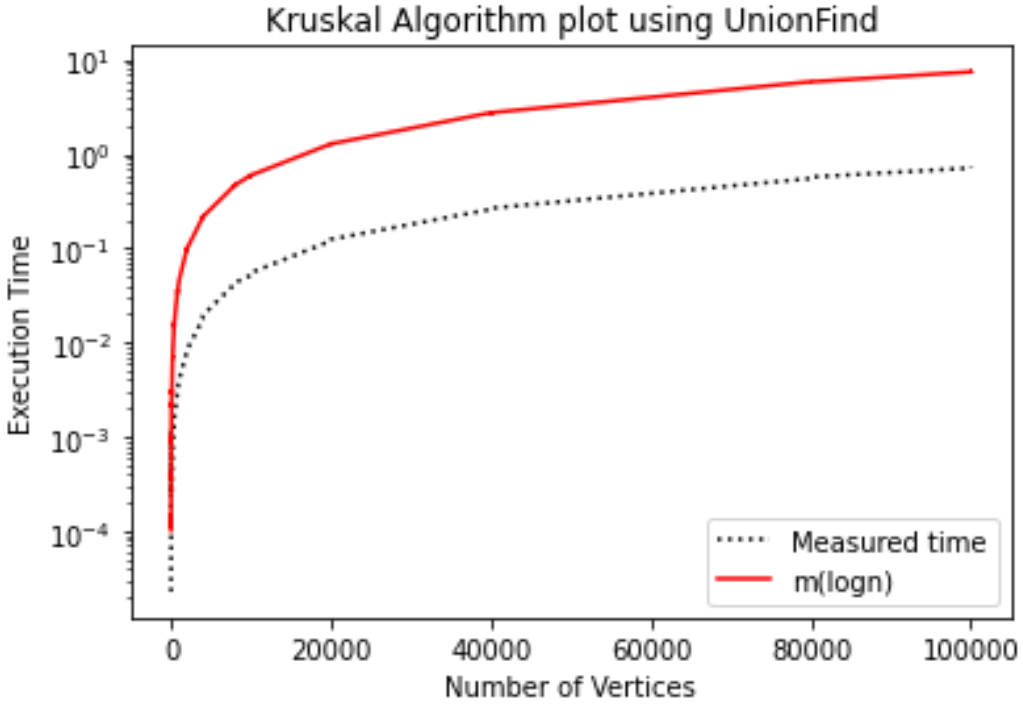
The following data is computed with num_calls=100, so it computes 100 times the algorithm for each graph and calculates the average to get a more accurate result.
The total time in seconds to compute all the graphs in the data set is **6.499942153400001s** and the average time is **0.09558738460882354s**.

This graph represents the execution time of Efficient Kruskal's algorithm in function against the number of vertices in graphs.



It has calculated the runs times and the value mlogn for each graph. To compare the measured times with the asymptotic complexity of the algorithms, it plotted in a graph comparing the two different lines. As we can see, the measured time is growing as the size of the graph increases. Furthermore it stays always below the line of the asymptotic complexity, as we expected.
It has multiplied the asymptotic complexity with the constant c, that represents the error committed by the machine we are running our algorithm to make it more accurate. It

computes the average of all c estimations of each graph, that is equal to **5.573621957070221e-06**.



In the following table we have the values represented in the graph:

Size(vertex)	Run Time(s)	Asyn Complexity	Constant	Ratio
10	2.33437e-05	20.7233	2.33437e-06	0
10	2.49358e-05	25.3284	2.49358e-06	1.068
10	2.81856e-05	29.9336	2.81856e-06	1.13
10	2.49561e-05	23.0259	2.49561e-06	0.885
20	5.48069e-05	71.8976	2.74035e-06	2.196
20	5.51311e-05	71.8976	2.75655e-06	1.006
20	5.99276e-05	83.8805	2.99638e-06	1.087
20	5.88587e-05	77.889	2.94294e-06	0.982
40	0.000135665	206.577	3.39162e-06	2.305
40	0.000110582	184.444	2.76456e-06	0.815
40	0.000164888	184.444	4.1222e-06	1.491
40	0.000128324	191.822	3.20809e-06	0.778
80	0.000307605	473.259	3.84506e-06	2.397
80	0.000229381	433.821	2.86726e-06	0.746
80	0.000234616	455.731	2.9327e-06	1.023
80	0.000245782	499.551	3.07228e-06	1.048
100	0.000308822	626.303	3.08822e-06	1.256
100	0.000302913	594.067	3.02913e-06	0.981
100	0.000321134	630.908	3.21134e-06	1.06
100	0.000298985	607.882	2.98985e-06	0.931
200	0.000613843	1414.65	3.06921e-06	2.053
200	0.000652432	1425.25	3.26216e-06	1.063
200	0.000654676	1425.25	3.27338e-06	1.003
200	0.000619244	1414.65	3.09622e-06	0.946
400	0.00143187	3235.39	3.57968e-06	2.312
400	0.00145056	3103.58	3.6264e-06	1.013
400	0.00137484	3223.41	3.43711e-06	0.948
400	0.00146729	3151.51	3.66823e-06	1.067
800	0.00306637	7105.74	3.83297e-06	2.09
800	0.00297917	7072.32	3.72397e-06	0.972
800	0.0042109	7192.64	5.26363e-06	1.413
800	0.00303938	7012.16	3.79922e-06	0.722
1000	0.00364879	8980.08	3.64879e-06	1.201
1000	0.0038739	9069.88	3.8739e-06	1.062
1000	0.00381527	9173.5	3.81527e-06	0.985
1000	0.00368955	9284.02	3.68955e-06	0.967
2000	0.00796806	20514.8	3.98403e-06	2.16
2000	0.007779	20172.8	3.8895e-06	0.976
2000	0.0150311	20157.6	7.51557e-06	1.932
2000	0.0185139	20347.6	9.25695e-06	1.232
4000	0.0463252	44456.1	1.15813e-05	2.502
4000	0.0233401	44082.9	5.83503e-06	0.504
4000	0.0166703	44290.2	4.16758e-06	0.714
4000	0.0166038	44522.5	4.15096e-06	0.996
8000	0.0362329	96207.9	4.52912e-06	2.182
8000	0.0367515	95893.4	4.59394e-06	1.014
8000	0.0379863	95821.5	4.74829e-06	1.034
8000	0.037719	96675.3	4.71487e-06	0.993
10000	0.0487271	122507	4.87271e-06	1.292
10000	0.048526	122866	4.8526e-06	0.996
10000	0.0491925	122378	4.91925e-06	1.014
10000	0.0491517	122599	4.91517e-06	0.999

20000	0.106651	264096	5.33254e-06	2.17
20000	0.106452	265671	5.32261e-06	0.998
20000	0.107561	264156	5.37803e-06	1.01
20000	0.116034	264126	5.80172e-06	1.079
40000	0.239496	566019	5.98739e-06	2.064
40000	0.239849	566348	5.99623e-06	1.001
40000	0.240609	564186	6.01521e-06	1.003
40000	0.239296	565002	5.98241e-06	0.995
80000	0.500245	1.20704e+06	6.25306e-06	2.09
80000	0.516951	1.20386e+06	6.46188e-06	1.033
80000	0.518488	1.20333e+06	6.4811e-06	1.003
80000	0.504947	1.20297e+06	6.31184e-06	0.974
100000	0.639033	1.53577e+06	6.39033e-06	1.266
100000	0.64001	1.53368e+06	6.4001e-06	1.002
100000	0.648138	1.53725e+06	6.48138e-06	1.013
100000	0.64755	1.53655e+06	6.4755e-06	0.999

It provides the weights of the MST found, they are printed in the following table.

Name of the file	Weight of MST
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
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

Naive Kruskal Algorithm:

Implementation

To implement the naive Kruskal algorithm, we first created a class `Graph` in order to read the files and construct the graphs. After creating the graph, the main function in our algorithm is `naive_kruskal_alg(g)` that takes a graph as input, runs the Kruskal algorithm on it and returns the weight and the edges and vertices in the minimum spanning tree. Naive Kruskal Algorithm is based on sorting all edges in the graph at the beginning based on weight, and then iterating over all the edges in non decreasing order of weights. If the edge does not create a cycle, we add it to the minimum spanning tree, else we skip it. In order to detect cycles, we used helper methods to create a DFS tree and do each time for a new edge a DFS traversal. We have implemented the methods needed for the DFS traversal of the algorithm:

- **`createDFSGraph(result, vertice)`**: take as parameter the edges to be appended to create a MST and the total number of vertices needed to construct a MST. It initializes a list equal to the number of vertices and appends to it the edges in result.
- **`addDFSEdge(adjacencyList, source, destination):`** in order to have an efficient DFS tree, instead of creating a new DFS tree each time we iterate on new edge, addDFSEdge takes one single edge and add it to the adjacency list of the DFS tree.
- **`deleteDFSEdge(adjacencyList, source, destination):`** this function deletes a DFS edge given both nodes, the source and the destination.
- **`DFS_Traversal(adjacencyList, v, visited, parent_node=-1):`** this helper function is a recursive method that return a boolean value indicating whether the new DFS tree, after appending one edge from the original graph to the new minimum spanning tree, creates a cycle or no.

Now, to dig deeper into the main function of naive kruskal algorithm:
First, it initializes the needed variables and sorts the edges of the graph in non decreasing order. It iterates over all the edges, and appends an edge to the MST. If not created, it creates a DFS tree using the MST or appends an edge to it, and then does a DFS traversal,

if no cycle is detected, it adds the weight of the edge to the total weight; otherwise, it removes the edge from the result. The algorithm returns a list of the edges that compose the MST(result), and tot_weight that is the sum of all the weights of the edges of the MST.

Other implementations choices is the creation of two supported functions:

- `create_graph` is used to create the object graph given data
- `plotResult` is used to plot the graph comparing two values. We used two functions to plot the result, one for the measured time only and the other for both the measured time and c*m*n

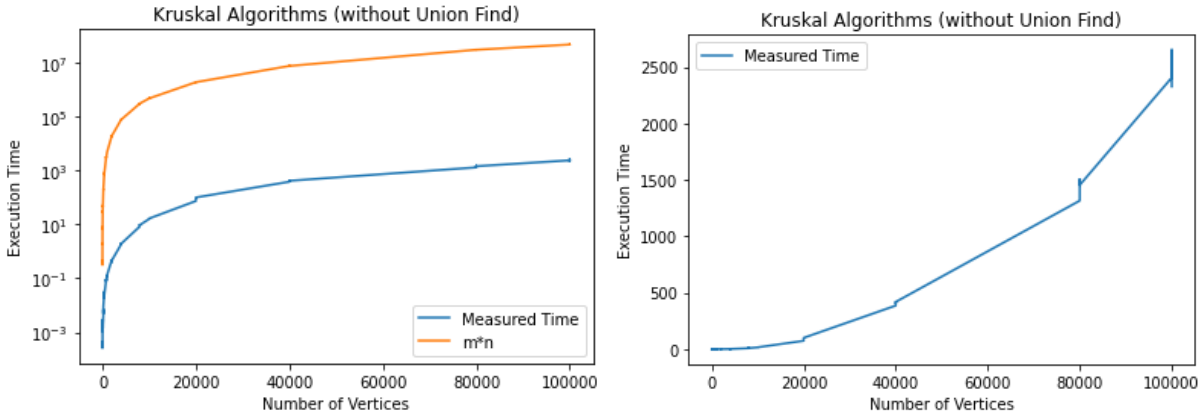
Naive Kruskal algorithm has a runtime of m*n complexity.

Question 1:

With a number of calls equal to 1, the Naive algorithm takes a total time of **17702.667455 s** and by average **260.33334492647055 s**.

The following data is computed with num_calls = 1, while the other two algorithms were computed with num_calls = 100, since this algorithm is not efficient and is taking approximately 5 hours to finish; therefore running it with 100 calls for accurate results requires days, so it computes 100 times the algorithm for each graph and calculates the average to get a more accurate result.

This graph represents the execution time of Naive Kruskal's algorithm in function against the number of vertices in graphs. We notice that the big graphs are taking so much time, as a graph with 10000 edges can last more than 30 minutes . and the measured time with respect to the number of vertices is increasing with a noticeable difference having a complexity of m * n. The average value of c_estimate is **0.0036050514616470584.**



In the following table we have the values represented in the graph:

Size(vertex)	Run Time(s)	Asyn Complexity	Constant	Ratio
10	0.000298	90	2.98e-05	0
10	0.0003164	110	3.164e-05	1.062
10	0.0003139	130	3.139e-05	0.992
10	0.0002559	100	2.559e-05	0.815
20	0.0003762	480	1.881e-05	1.47
20	0.0004368	480	2.184e-05	1.161
20	0.000427	560	2.135e-05	0.978
20	0.000394	520	1.97e-05	0.923
40	0.0005329	2240	1.33225e-05	1.353
40	0.0007831	2000	1.95775e-05	1.47
40	0.001205	2000	3.0125e-05	1.539
40	0.0013725	2080	3.43125e-05	1.139
80	0.001765	8640	2.20625e-05	1.286
80	0.0021124	7920	2.6405e-05	1.197
80	0.0010901	8320	1.36263e-05	0.516
80	0.0021982	9120	2.74775e-05	2.017
100	0.002974	13600	2.974e-05	1.353
100	0.0020709	12900	2.0709e-05	0.696
100	0.0014648	13700	1.4648e-05	0.707
100	0.0018278	13200	1.8278e-05	1.248
200	0.0042428	53400	2.1214e-05	2.321
200	0.0069469	53800	3.47345e-05	1.637
200	0.0049505	53800	2.47525e-05	0.713
200	0.008935	53400	4.4675e-05	1.805
400	0.0282049	216000	7.05123e-05	3.157
400	0.0196311	207200	4.90777e-05	0.696
400	0.0313372	215200	7.8343e-05	1.596
400	0.0236253	210400	5.90632e-05	0.754
800	0.0850228	850400	0.000106278	3.599
800	0.0797528	846400	9.9691e-05	0.938
800	0.0922996	860800	0.000115374	1.157

800	0.08383	839200	0.000104788	0.908
1000	0.108676	1300000	0.000108676	1.296
1000	0.118244	1313000	0.000118244	1.088
1000	0.114592	1328000	0.000114592	0.969
1000	0.131562	1344000	0.000131562	1.148
2000	0.477197	5398000	0.000238599	3.627
2000	0.410225	5308000	0.000205112	0.86
2000	0.451867	5304000	0.000225934	1.102
2000	0.456888	5354000	0.000228444	1.011
4000	1.81823	21440000	0.000454557	3.98
4000	1.92027	21260000	0.000480068	1.056
4000	1.8574	21360000	0.000464351	0.967
4000	1.91133	21472000	0.000477832	1.029
8000	8.41737	85640000	0.00105217	4.404
8000	9.56861	85360000	0.00119608	1.137
8000	9.48551	85296000	0.00118569	0.991
8000	9.5092	86056000	0.00118865	1.002
10000	15.9919	133010000	0.00159919	1.682
10000	16.1617	133400000	0.00161617	1.011
10000	16.5219	132870000	0.00165219	1.022
10000	16.6478	133110000	0.00166478	1.008
20000	74.3198	533340000	0.00371599	4.464
20000	98.1034	536520000	0.00490517	1.32
20000	100.092	533460000	0.00500461	1.02
20000	100.712	533400000	0.00503562	1.006
40000	386.178	2136600000	0.00965445	3.834
40000	416.817	2137840000	0.0104204	1.079
40000	411.088	2129680000	0.0102772	0.986
40000	418.852	2132760000	0.0104713	1.019
80000	1315.69	8553120000	0.0164461	3.141
80000	1380.39	8530640000	0.0172549	1.049
80000	1504.52	8526880000	0.0188065	1.09
80000	1454.44	8524320000	0.0181806	0.967
100000	2405.9	13339500000	0.024059	1.654
100000	2652.26	13321400000	0.0265226	1.102
100000	2543.32	13352400000	0.0254332	0.959
100000	2327.4	13346300000	0.023274	0.915

It provides the weights of the MST found, they are printed in the following table.

Name of the file	Weight of MST
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
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

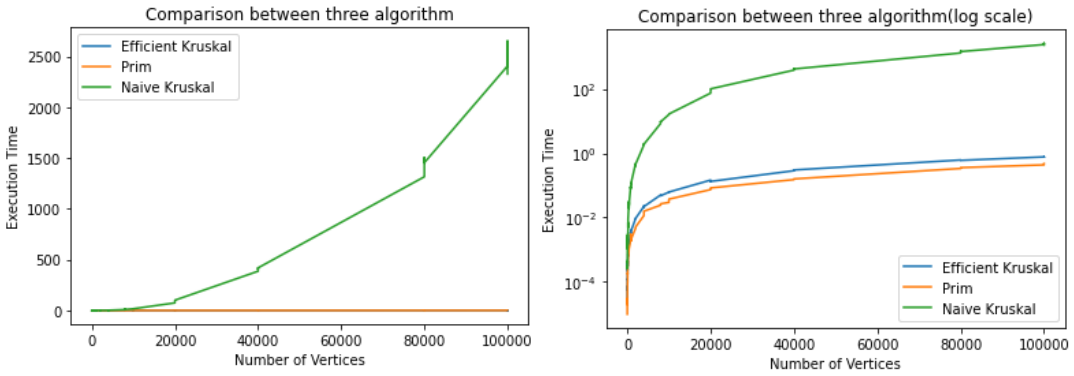
Comparison:

Time Taken for each of the algorithms:

Prim's Algorithm implemented with a Heap: **4.473743817806244 s**

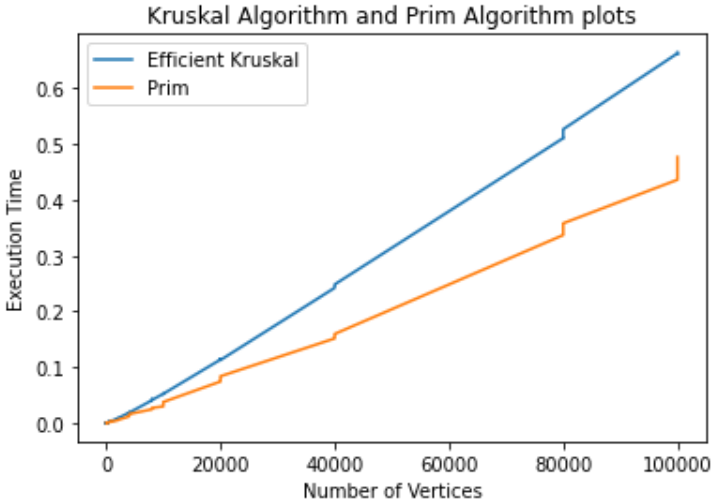
Naive Kruskal's Algorithm with O(mn) complexity: **17702.667455 s**

Efficient Kruskal's Algorithm based on Union-Find: **6.499942153400001s**



The execution time of the three algorithms is increasing in respect to the number of vertices. Both prim's algorithm and efficient Kruskal algorithms have a complexity of m log(n) while naive Kruskal has a complexity of m *n. However, the Prim Algorithm is faster than the other algorithms; hence, more efficient.
As we can see in the two graphs plotting the measured time for each algorithm, Naive Kruskal has a complexity that is exponentially in comparison to Prim and Efficient Kruskal, which complexity has more a linear behavior against it. In fact we can not distinguish the two functions from each other in the first plot.
Using instead a logarithmic scale (on the right graph), we can see again the big difference for Naive Kruskal, but see that Prim algorithm is more efficient than Efficient Kruskal.
When plotting the three graphs on the top left, we can notice that prim and efficient kruskal graphs are above each other due to the significant difference between their results and that of naive Kruskal that makes the difference between prim and efficient Kruskal insignificant and irrelevant.

To study which of Prim and Efficient Kruskal is more efficient, we plotted just their measured time. We can see that Prim's algorithm is more efficient and faster than Efficient Kruskal while both of the algorithms have mlog(n) complexity.



This is because Prim's algorithm is faster when using graphs with a lot of edges, on the other hand the difference between the two algorithms is not so significant, mostly if we consider the first graphs.

Algorithm	Complexity	Total Time	Average Time	Average C
Prim	O(m*n)	**4.4737s**	**0.0676**	2.8553e-06
Naive Kruskal	O(mlogn)	**17702.6674**	**260.3333**	**0.0036**
Efficient Kruskal	O(mlogn)	**6.4999**	**0.1175**	**5.573e-06**

In conclusion, even studying the data in the previous table, we can say that Prim's algorithm is faster and more efficient than the other two, but even Efficient Kruskal is a very efficient algorithm as the difference with the first is not so relevant.