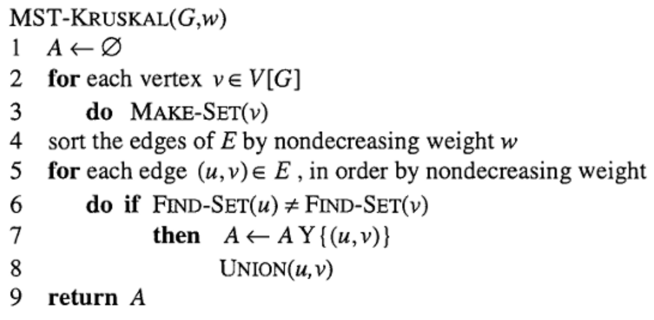
# CSE311 PROGRAMMING TERM PROJECT FINAL REPORT

In this project I'm comparing Kruskal and Prim's Algorithms.Both are Greedy algorithms to find the MST which is a tree that has the following characteristics:

1- acyclic
2- connected
3- undirected graph.

I used the following way the construct my Kruskal Algorithm:

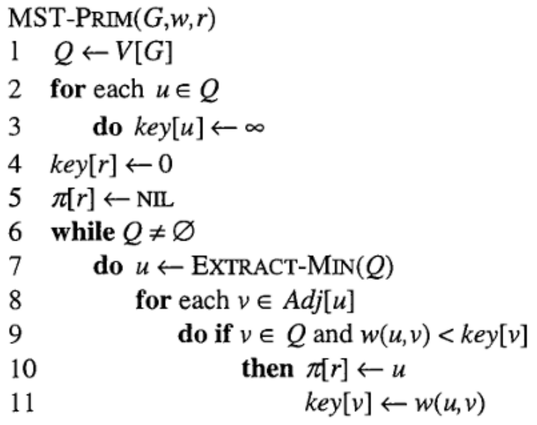


```
MST-KRUSKAL($G$,$w$)
1  $A \leftarrow \varnothing$
2  for each vertex $v \in V[G]$
3      do MAKE-SET($v$)
4  sort the edges of $E$ by nondecreasing weight $w$
5  for each edge $(u,v) \in E$ , in order by nondecreasing weight
6      do if FIND-SET($u$) $\neq$ FIND-SET($v$)
7          then $A \leftarrow A$ Y $\{(u,v)\}$
8               UNION($u,v$)
9  return $A$
```

**Algorithm 2 – Implementation of Kruskal's Algorithm**

I wrote the code for Kruskal Algo in C language. I used qsort to sort the edges of E by nondecreasing weight w. Kruskal has Θ(|V|log|V|) time complexity for sparse graphs and Θ(|V|^2log|V|) time complexity for dense graphs.

I used the following way the construct my Kruskal Algorithm:



```
MST-PRIM($G$,$w$,$r$)
1  $Q \leftarrow V[G]$
2  for each $u \in Q$
3      do $key[u] \leftarrow \infty$
4  $key[r] \leftarrow 0$
5  $\pi[r] \leftarrow$ NIL
6  while $Q \neq \varnothing$
7      do $u \leftarrow$ EXTRACT-MIN($Q$)
8          for each $v \in Adj[u]$
9              do if $v \in Q$ and $w(u,v) < key[v]$
10                 then $\pi[r] \leftarrow u$
11                      $key[v] \leftarrow w(u,v)$
```

**Algorithm 3 – Implementation of Prim's Algorithm**

I wrote the code for Prim Algo in C language. Prim algorithm has Θ(|V|^2) time complexity.

## INPUT STYLE

To compare these two algoritms I created adjacency matrices firstly.

I use the following input format for both of the algorithms:

For example 5 at the beginning represents that the graph is a V×V matrix

Here is an example :

5

0 2 0 6 0

2 0 3 8 5

0 3 0 0 7

6 8 0 0 9

0 5 7 9 0

## MATRIX GENERATOR

I'm using matrix generator written by Oğuzhan İçelliler. Code will be executed by typing

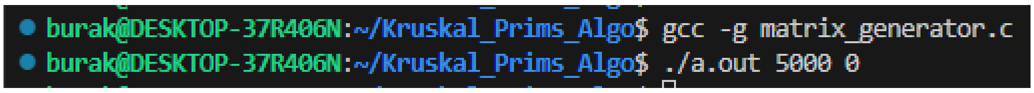
"./a.out 1000 0"   1000 means generate 1000x1000 matrix 0 means generate MST.

"./a.out 1000 1" if we give 1 that means increasing the density comparing giving 0.

The graph that we created will be atleast MST.The logic for this code is basically like

the following:

1) Type the matrix size (V x V) for example typing  500 will create 500x500 adjacent matrix.
2) Randomly select a number between 1 and 100 (inclusive).
3) If the chosen number is less than or equal to the user's chance value, select a random number between 1 and 9 (inclusive) and assign it to the current edge's value.



```
burak@DESKTOP-37R406N:~/Kruskal_Prims_Algo$ gcc -g matrix_generator.c
burak@DESKTOP-37R406N:~/Kruskal_Prims_Algo$ ./a.out 5000 0
```

## TESTING ENVIRONMENT

CPU: AMD Ryzen 7 5800X (8 cores (physical), 16 threads (logical))

OS: Windows 10 Pro

Frequency: With a base clock speed of 3.8 GHz and a max boost clock speed of 4.7 GHz

Windows Version: 22H2

RAM: 16 GB (2×8 dual channel) 3200 MHZ

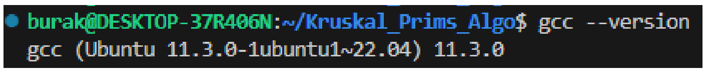
Cache L1: 64K (per core)

Cache L2: 512K (per core)

Cache L3: 32MB

EMPIRICAL ANALYSIS

I will be running each for 5 times but first one will be discarded due to caching issues.

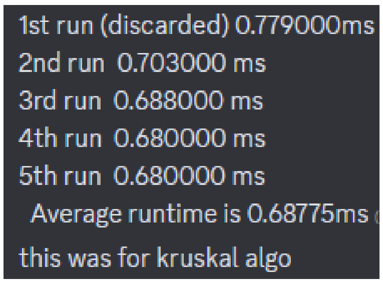


```
burak@DESKTOP-37R406N:~/Kruskal_Prims_Algo$ gcc --version
gcc (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0
```

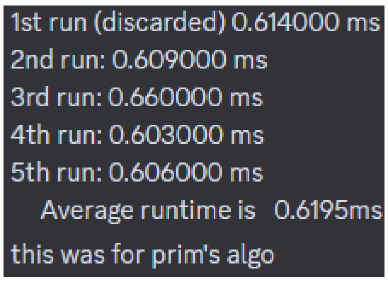
All programs will be compiled with GCC version 11.3.0 and with default level of optimization.

My first experiment will be like the following. Constant density (2) with increasing matrix size

1) 500x500 adjacency matrix to compare my Kruskal and Prim's Algorithms.

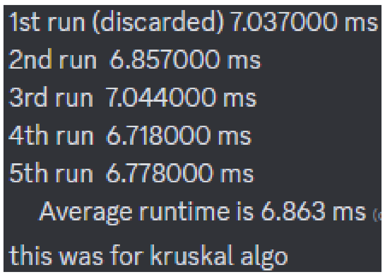


```
1st run (discarded) 0.779000ms
2nd run  0.703000 ms
3rd run  0.688000 ms
4th run  0.680000 ms
5th run  0.680000 ms
  Average runtime is 0.68775ms
this was for kruskal algo
```

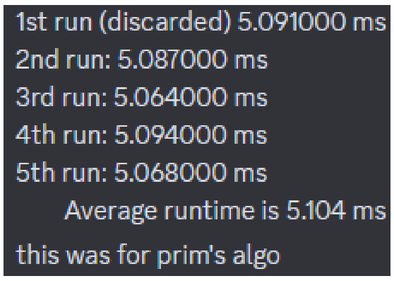


```
1st run (discarded) 0.614000 ms
2nd run: 0.609000 ms
3rd run: 0.660000 ms
4th run: 0.603000 ms
5th run: 0.606000 ms
    Average runtime is   0.6195ms
this was for prim's algo
```

2) 1500x1500 adjacency matrix to compare my Kruskal and Prim's Algorithms. "density_2_adj_matrix1500.txt"

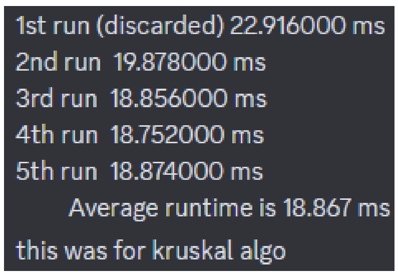


```
1st run (discarded) 7.037000 ms
2nd run  6.857000 ms
3rd run  7.044000 ms
4th run  6.718000 ms
5th run  6.778000 ms
    Average runtime is 6.863 ms
this was for kruskal algo
```

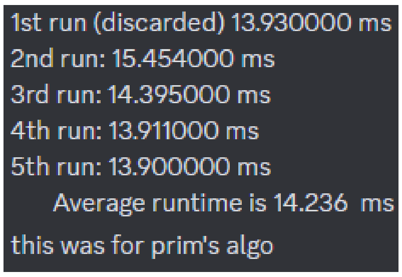


```
1st run (discarded) 5.091000 ms
2nd run: 5.087000 ms
3rd run: 5.064000 ms
4th run: 5.094000 ms
5th run: 5.068000 ms
      Average runtime is 5.104 ms
this was for prim's algo
```

3) 2500x2500 adjacency matrix to compare my Kruskal and Prim's Algorithms. "density_2_adj_matrix2500.txt"

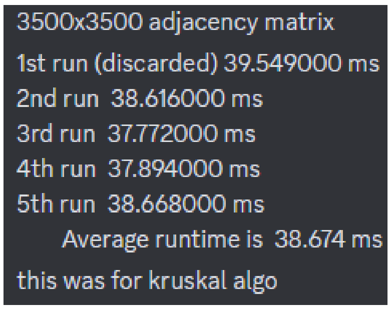


```
1st run (discarded) 22.916000 ms
2nd run  19.878000 ms
3rd run  18.856000 ms
4th run  18.752000 ms
5th run  18.874000 ms
      Average runtime is 18.867 ms
this was for kruskal algo
```

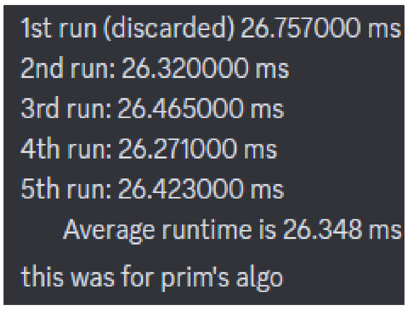


```
1st run (discarded) 13.930000 ms
2nd run: 15.454000 ms
3rd run: 14.395000 ms
4th run: 13.911000 ms
5th run: 13.900000 ms
      Average runtime is 14.236  ms
this was for prim's algo
```

4) 3500x3500 adjacency matrix to compare my Kruskal and Prim’s Algorithms.

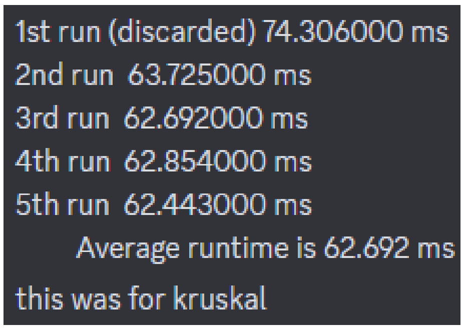


```
3500x3500 adjacency matrix
1st run (discarded) 39.549000 ms
2nd run  38.616000 ms
3rd run  37.772000 ms
4th run  37.894000 ms
5th run  38.668000 ms
      Average runtime is  38.674 ms
this was for kruskal algo
```

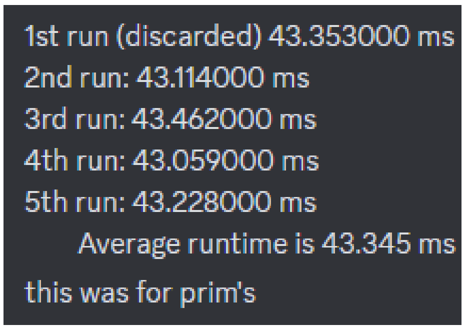


```
1st run (discarded) 26.757000 ms
2nd run: 26.320000 ms
3rd run: 26.465000 ms
4th run: 26.271000 ms
5th run: 26.423000 ms
      Average runtime is 26.348 ms
this was for prim's algo
```

5) 4500x4500 adjacency matrix to compare my Kruskal and Prim’s Algorithms. “density_2_adj_matrix4500.txt”

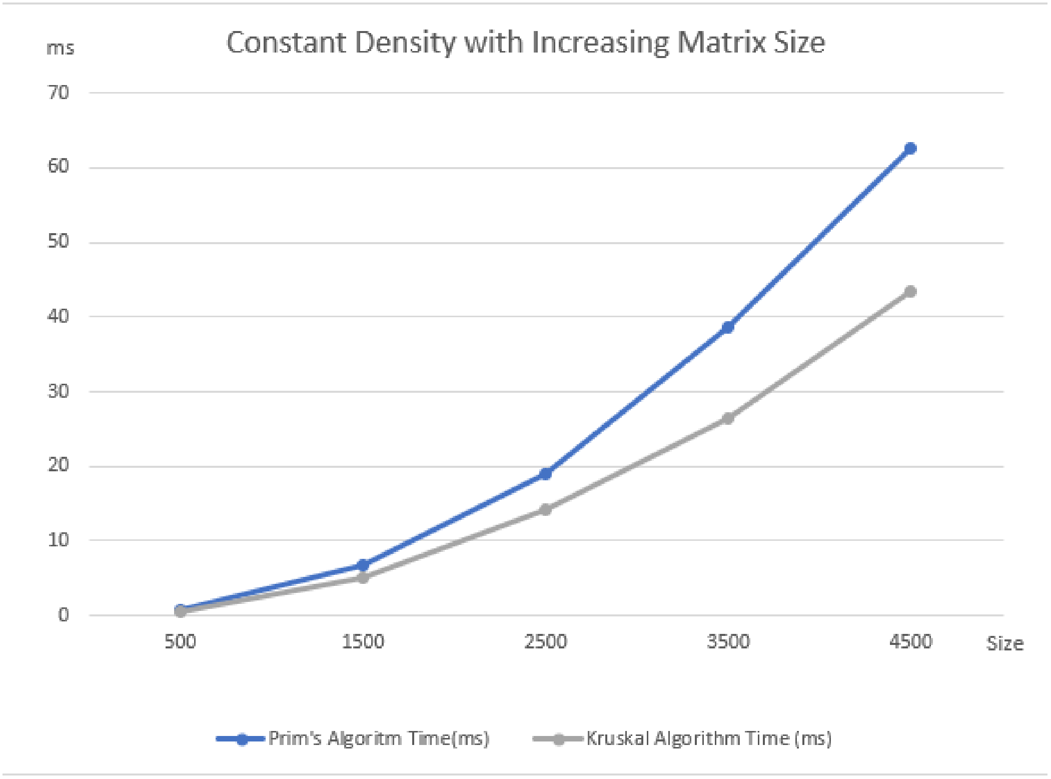


```
1st run (discarded) 74.306000 ms
2nd run  63.725000 ms
3rd run  62.692000 ms
4th run  62.854000 ms
5th run  62.443000 ms
      Average runtime is 62.692 ms
this was for kruskal
```



```
1st run (discarded) 43.353000 ms
2nd run: 43.114000 ms
3rd run: 43.462000 ms
4th run: 43.059000 ms
5th run: 43.228000 ms
      Average runtime is 43.345 ms
this was for prim's
```

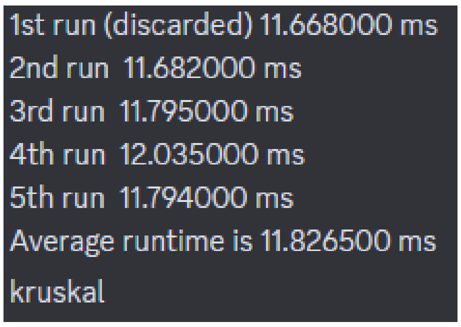
FIRST EXPERIMENT CHART



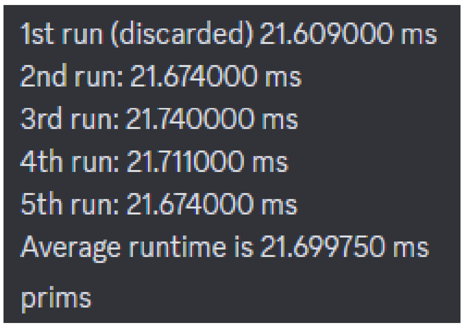
“Average values are used for the graph (Kruskal & Prim’s)”

My Second experiment will be like the following. Varying density with constant matrix size.

1) Density (0) “which means MST” with 3500x3500 matrix
2) “density_0_adj_matrix3500.txt”

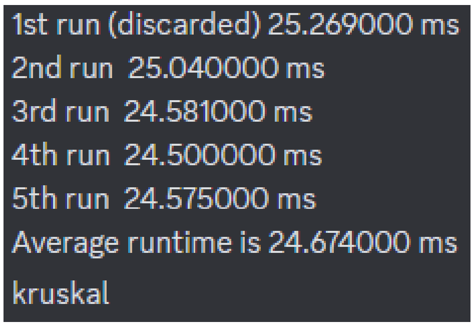


```
1st run (discarded) 11.668000 ms
2nd run  11.682000 ms
3rd run  11.795000 ms
4th run  12.035000 ms
5th run  11.794000 ms
Average runtime is 11.826500 ms
kruskal
```

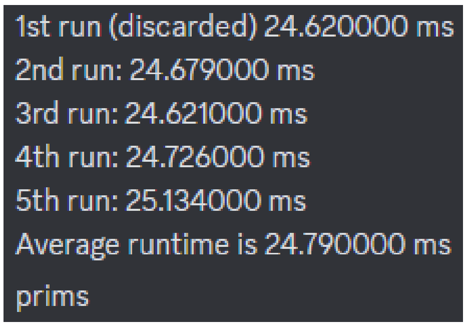


```
1st run (discarded) 21.609000 ms
2nd run: 21.674000 ms
3rd run: 21.740000 ms
4th run: 21.711000 ms
5th run: 21.674000 ms
Average runtime is 21.699750 ms
prims
```

3) Density (1) with 3500x3500 matrix

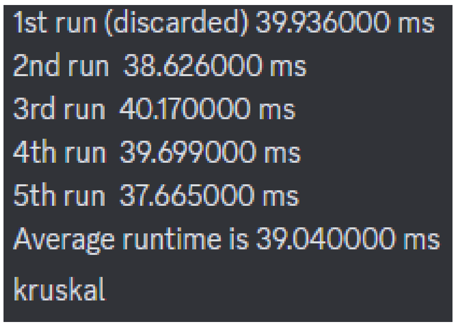


```
1st run (discarded) 25.269000 ms
2nd run  25.040000 ms
3rd run  24.581000 ms
4th run  24.500000 ms
5th run  24.575000 ms
Average runtime is 24.674000 ms
kruskal
```

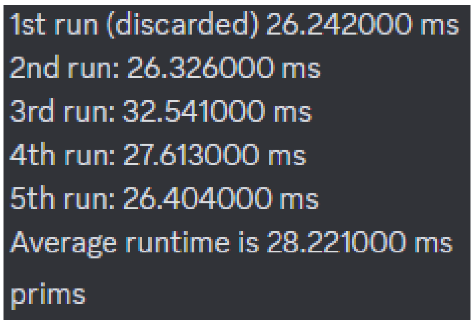


```
1st run (discarded) 24.620000 ms
2nd run: 24.679000 ms
3rd run: 24.621000 ms
4th run: 24.726000 ms
5th run: 25.134000 ms
Average runtime is 24.790000 ms
prims
```

4) Density (2) with 3500x3500 matrix

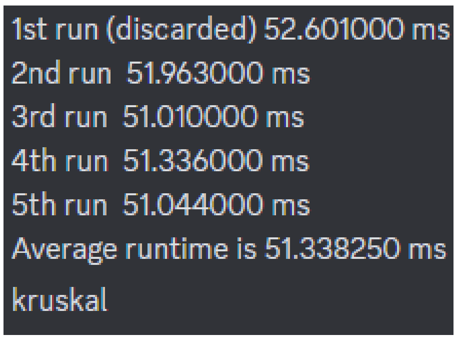


```
1st run (discarded) 39.936000 ms
2nd run  38.626000 ms
3rd run  40.170000 ms
4th run  39.699000 ms
5th run  37.665000 ms
Average runtime is 39.040000 ms
kruskal
```

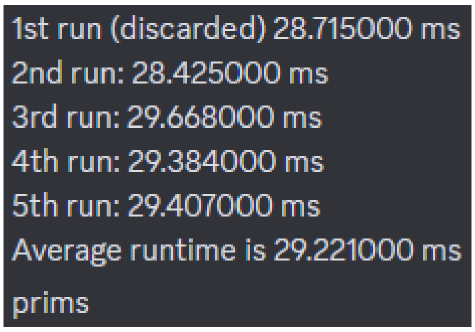


```
1st run (discarded) 26.242000 ms
2nd run: 26.326000 ms
3rd run: 32.541000 ms
4th run: 27.613000 ms
5th run: 26.404000 ms
Average runtime is 28.221000 ms
prims
```

5) Density (3) with 3500x3500 matrix

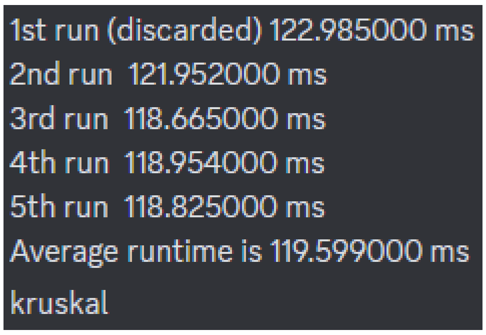


```
1st run (discarded) 52.601000 ms
2nd run  51.963000 ms
3rd run  51.010000 ms
4th run  51.336000 ms
5th run  51.044000 ms
Average runtime is 51.338250 ms
kruskal
```

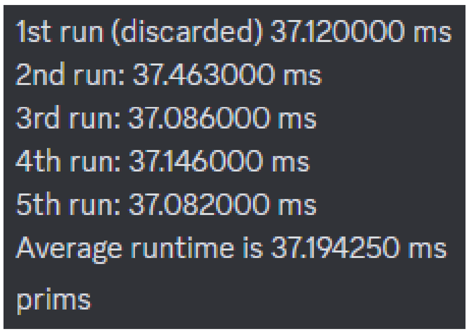


```
1st run (discarded) 28.715000 ms
2nd run: 28.425000 ms
3rd run: 29.668000 ms
4th run: 29.384000 ms
5th run: 29.407000 ms
Average runtime is 29.221000 ms
prims
```

6) Density (8) with 3500x3500 matrix

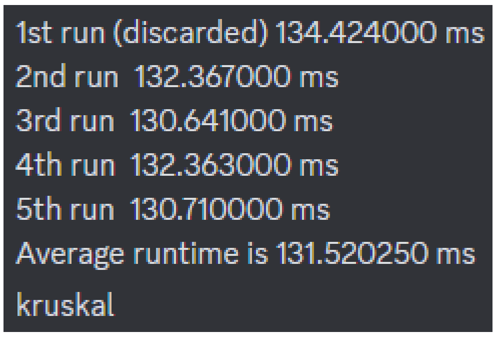


```
1st run (discarded) 122.985000 ms
2nd run  121.952000 ms
3rd run  118.665000 ms
4th run  118.954000 ms
5th run  118.825000 ms
Average runtime is 119.599000 ms
kruskal
```

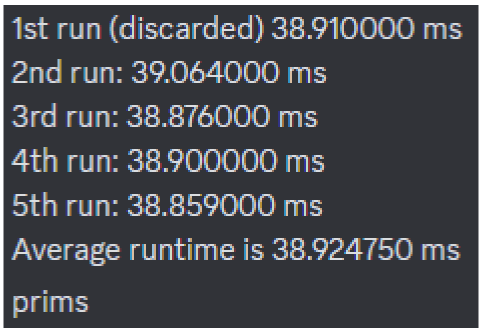


```
1st run (discarded) 37.120000 ms
2nd run: 37.463000 ms
3rd run: 37.086000 ms
4th run: 37.146000 ms
5th run: 37.082000 ms
Average runtime is 37.194250 ms
prims
```

7) Density (9) with 3500x3500 matrix

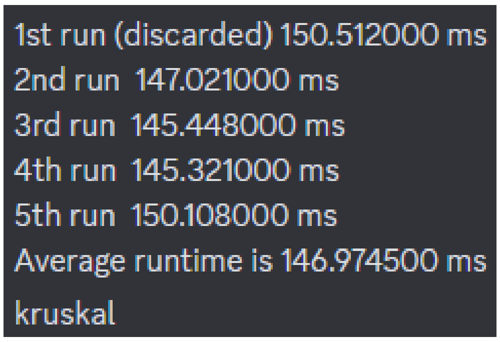


```
1st run (discarded) 134.424000 ms
2nd run  132.367000 ms
3rd run  130.641000 ms
4th run  132.363000 ms
5th run  130.710000 ms
Average runtime is 131.520250 ms
kruskal
```

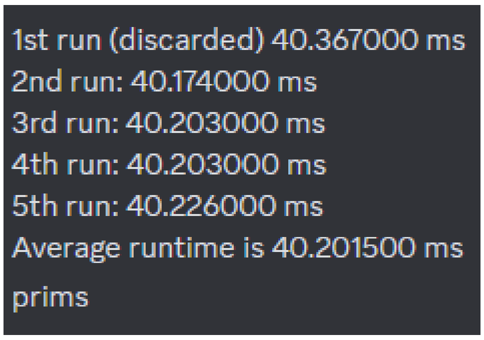


```
1st run (discarded) 38.910000 ms
2nd run: 39.064000 ms
3rd run: 38.876000 ms
4th run: 38.900000 ms
5th run: 38.859000 ms
Average runtime is 38.924750 ms
prims
```

8) Density (10) with 3500x3500 matrix

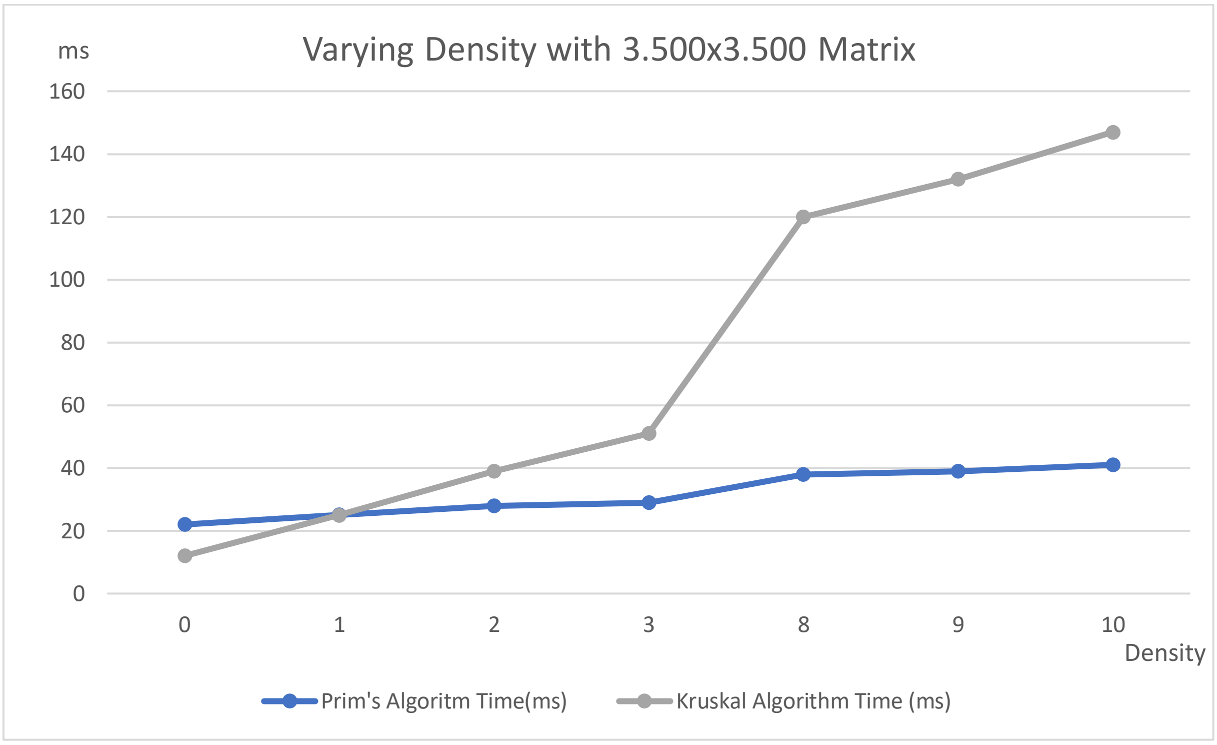


```
1st run (discarded) 150.512000 ms
2nd run  147.021000 ms
3rd run  145.448000 ms
4th run  145.321000 ms
5th run  150.108000 ms
Average runtime is 146.974500 ms
kruskal
```



```
1st run (discarded) 40.367000 ms
2nd run: 40.174000 ms
3rd run: 40.203000 ms
4th run: 40.203000 ms
5th run: 40.226000 ms
Average runtime is 40.201500 ms
prims
```

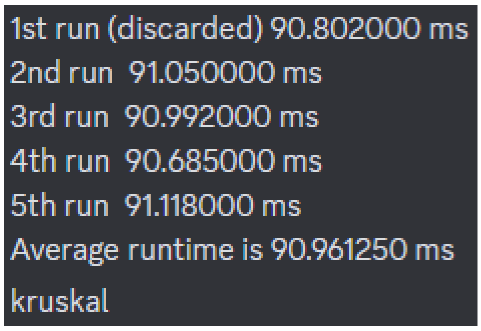
SECOND EXPERIMENT CHART



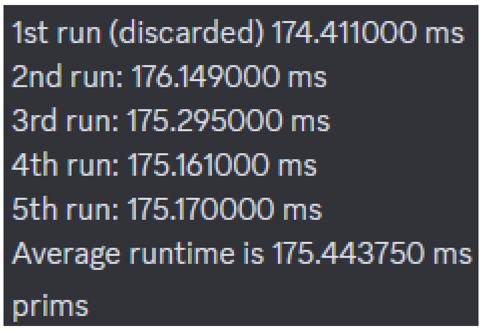
"Average values are used for the graph (Kruskal & Prim's)"

My third experiment will have increased matrix size compared to second experiment.

1) Density (0) "which means MST" with 10.000x10.000 matrix "density_0_adj_matrix10000.txt"

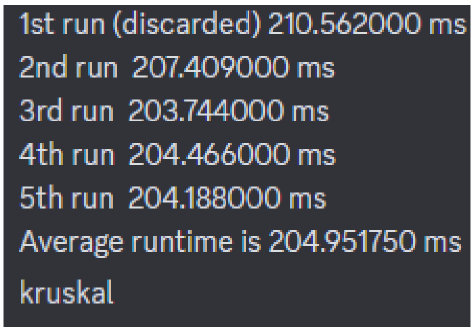


```
1st run (discarded) 90.802000 ms
2nd run  91.050000 ms
3rd run  90.992000 ms
4th run  90.685000 ms
5th run  91.118000 ms
Average runtime is 90.961250 ms
kruskal
```

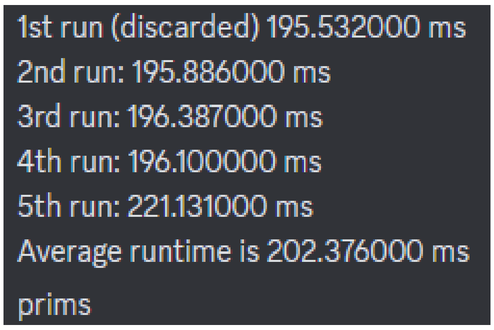


```
1st run (discarded) 174.411000 ms
2nd run: 176.149000 ms
3rd run: 175.295000 ms
4th run: 175.161000 ms
5th run: 175.170000 ms
Average runtime is 175.443750 ms
prims
```

2) Density (1) with 10.000x10.000 matrix

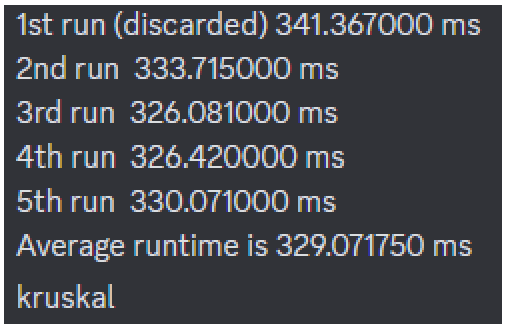


```
1st run (discarded) 210.562000 ms
2nd run  207.409000 ms
3rd run  203.744000 ms
4th run  204.466000 ms
5th run  204.188000 ms
Average runtime is 204.951750 ms
kruskal
```

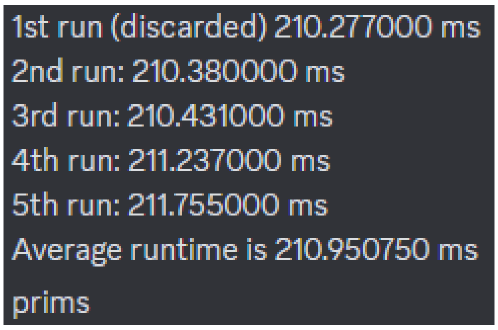


```
1st run (discarded) 195.532000 ms
2nd run: 195.886000 ms
3rd run: 196.387000 ms
4th run: 196.100000 ms
5th run: 221.131000 ms
Average runtime is 202.376000 ms
prims
```

3) Density (2) with 10.000x10.000 matrix

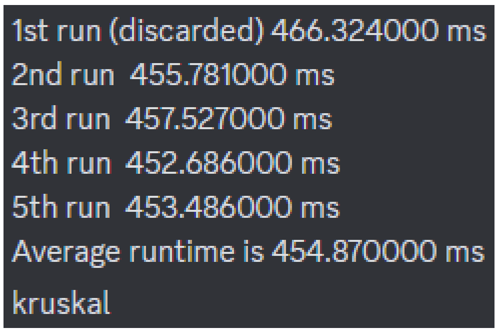


```
1st run (discarded) 341.367000 ms
2nd run  333.715000 ms
3rd run  326.081000 ms
4th run  326.420000 ms
5th run  330.071000 ms
Average runtime is 329.071750 ms
kruskal
```

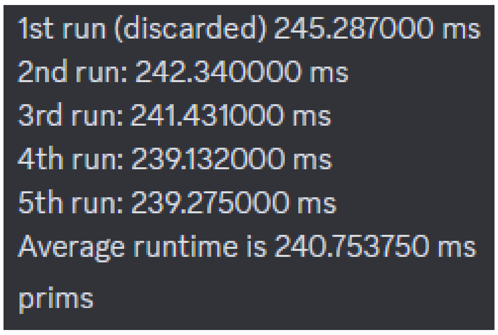


```
1st run (discarded) 210.277000 ms
2nd run: 210.380000 ms
3rd run: 210.431000 ms
4th run: 211.237000 ms
5th run: 211.755000 ms
Average runtime is 210.950750 ms
prims
```

4) Density (3) with 10.000x10.000 matrix

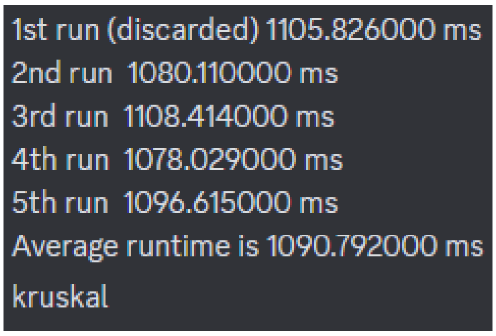


```
1st run (discarded) 466.324000 ms
2nd run  455.781000 ms
3rd run  457.527000 ms
4th run  452.686000 ms
5th run  453.486000 ms
Average runtime is 454.870000 ms
kruskal
```

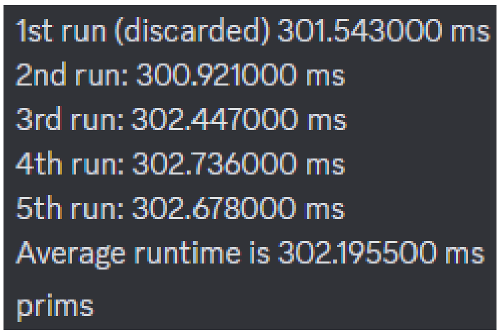


```
1st run (discarded) 245.287000 ms
2nd run: 242.340000 ms
3rd run: 241.431000 ms
4th run: 239.132000 ms
5th run: 239.275000 ms
Average runtime is 240.753750 ms
prims
```

5) Density (8) with 10.000x10.000 matrix

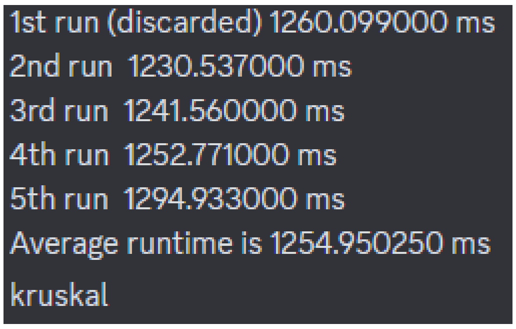


```
1st run (discarded) 1105.826000 ms
2nd run  1080.110000 ms
3rd run  1108.414000 ms
4th run  1078.029000 ms
5th run  1096.615000 ms
Average runtime is 1090.792000 ms
kruskal
```

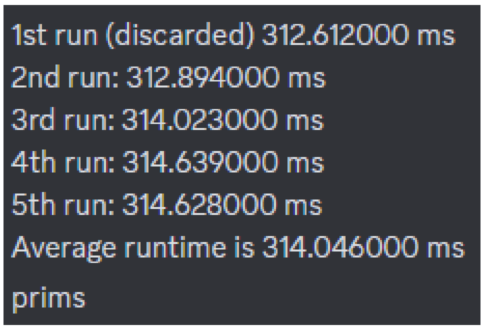


```
1st run (discarded) 301.543000 ms
2nd run: 300.921000 ms
3rd run: 302.447000 ms
4th run: 302.736000 ms
5th run: 302.678000 ms
Average runtime is 302.195500 ms
prims
```

6) Density (9) with 10.000x10.000 matrix

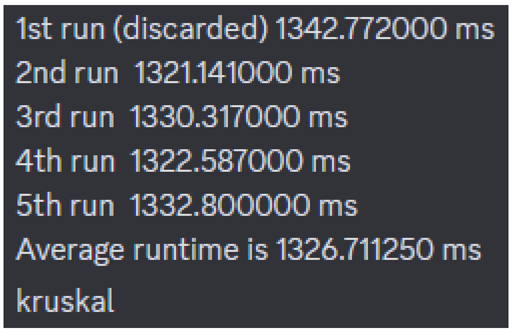


```
1st run (discarded) 1260.099000 ms
2nd run  1230.537000 ms
3rd run  1241.560000 ms
4th run  1252.771000 ms
5th run  1294.933000 ms
Average runtime is 1254.950250 ms
kruskal
```

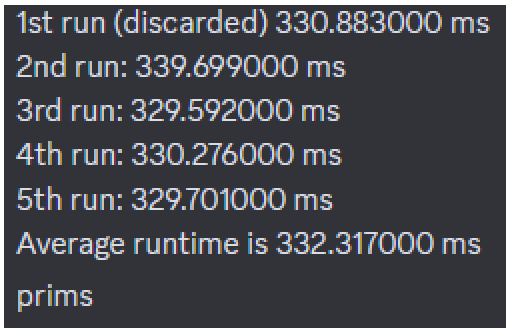


```
1st run (discarded) 312.612000 ms
2nd run: 312.894000 ms
3rd run: 314.023000 ms
4th run: 314.639000 ms
5th run: 314.628000 ms
Average runtime is 314.046000 ms
prims
```

7) Density (10) with 10.000x10.000 matrix
“density_10_adj_matrix10000.txt”

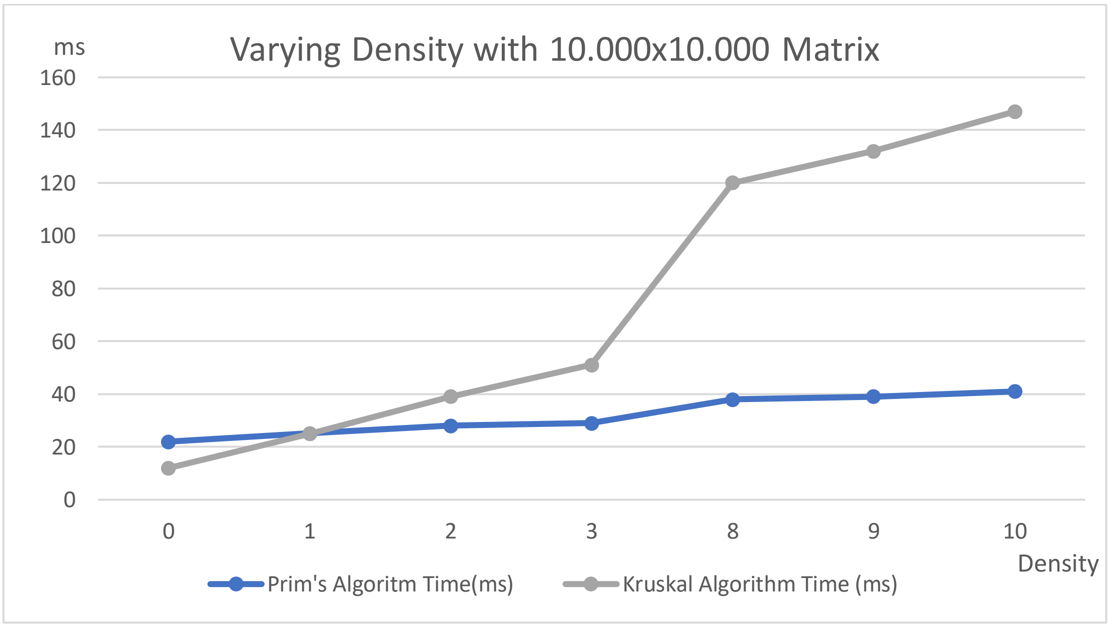


```
1st run (discarded) 1342.772000 ms
2nd run  1321.141000 ms
3rd run  1330.317000 ms
4th run  1322.587000 ms
5th run  1332.800000 ms
Average runtime is 1326.711250 ms
kruskal
```



```
1st run (discarded) 330.883000 ms
2nd run: 339.699000 ms
3rd run: 329.592000 ms
4th run: 330.276000 ms
5th run: 329.701000 ms
Average runtime is 332.317000 ms
prims
```

THIRD EXPERIMENT CHART



“Average values are used for the graph (Kruskal & Prim’s)”

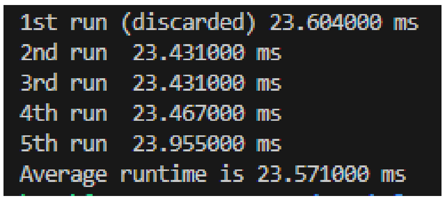
My thoughts about Empirical Analysis:

1) Kruskal algorithm performs worse than I expected considering especially my 2nd and 3rd experiments.Kruskal algorithm only performs better than Prim's algorithm with MST. Theoretically in sparse graphs Kruskal should perform better considering it's Time complexity is Θ(|V|log|V|) and Prim's algo's time complexity is Θ(|V|^2).

2) The code that I got from "geeksforgeeks" was using qsort to do the sorting required for Kruskal algorithm. qsort() is a sorting algorithm from the C standard library (stdlib.h) and it's not the same thing with quick sort algorithm it can even be not-in-place sorting algorithm.That may causing the efficiency problem for Kruskal algorithm.

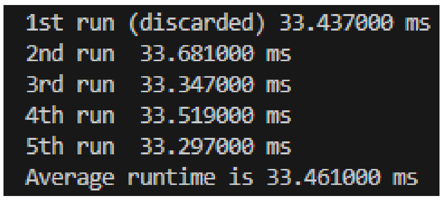
3) I tried shuffling for my Kruskal algorithm to see if it is performing better or not. It didn't solve my problem with the Kruskal algorithm's performance. It performed worse.

Without shuffing (Kruskal)



```
1st run (discarded) 23.604000 ms
2nd run  23.431000 ms
3rd run  23.431000 ms
4th run  23.467000 ms
5th run  23.955000 ms
Average runtime is 23.571000 ms
```

with shuffling (Kruskal)



```
1st run (discarded) 33.437000 ms
2nd run  33.681000 ms
3rd run  33.347000 ms
4th run  33.519000 ms
5th run  33.297000 ms
Average runtime is 33.461000 ms
```

I used "density_0_adj_matrix5000.txt"

4) I checked if I did any I/O operations in Kruskal algorithm but that wasn't the problem also.

5) I also tried to optimize KruskalAlgo function by giving edge set to it as a parameter. It increased the performance but still it wasn't enough. I kept this updated version in my code.

6) I checked if there is a  memory leak but I fixed it with using malloc.

7) After all the check operations I couldn't find the main reason for Kruskal's bad performance.

References:

For Kruskal Algorithm:

https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/

For Prim's Algorithm:

https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/

Prim's Algorithm's and Kruskal Algorithm's Pseudocodes:

Levitin, A. (2014). Introduction to the Design and Analysis of Algorithms: International Edition

Pearson Higher Ed.

Name-Surname: Burak Eymen Çevik                                   Student ID:20200702123