1)

Music Composer

I use AIVA (Artificial Intelligence Virtual Artist) as a preference. AIVA is an electronic composer based on AI.

Performance Measures:

Music has a unit to be measured, musicians measure music using the concept of *pitch*, *measure*, *notes*, *scale*, *mode*,*rhythm*, *melody*, *chords*,etc. Our **AI agent has to fulfill those measures**.

The music generated with the agent has to be audible to human, means that the generated music must be in the **range of frequencies within human audible zone**

Most of the time what the artist do is that, they compose the music by computer and they play in the event in front of the a large audience, so in order to the human to be able to play a music that our AI agent composed , **the composed music have to be easy to be played by the human.**

Environment:

The environment is **virtual**, when we look at AIVA it train itself by reading over 30, 000 classical music scores from the world's greatest composers, this tell us that it is a **software agent** that operate in **a fully observable environment**

Actuator:

This can be pure softbot but if it has to be play what it composed then it may need a speaker or related music player instrument

Sensors:

As a software agent the AI composer uses **a code that reads the parameters** specified by the user.

Aircraft Autolander

Performance Measures:

The AI has to take the **right time** to land. It has not been too long or too short. It also answer the economical quasense of fuel, the AI have to be **economical**

The AI hase to land the correct airport o the correct airplane,(**correctness**),without any damage on the airplane as well as other airplane and the airport, and also to the cargo the plane cares(**safety).**,

Environment:

The AI mainly hase role in the **airport** and also **atmosphere** of the earth.

Actuator:

The AI may use the **Throttle**(to controls the amount of fuel provided to the engine), **Landing** gear to land on the ground surface,**Rudders**(to controls rotation about the vertical axis of an aircraft**)**

Sensors:

The AI see the environment throw the **Camera** know the speed of the plane by reading the **Speedometer** of the plane, it also use **Altimeter** to measures the height of an aircraft above the ground

2)

A lit bit about my file structure:

I have my graph implementation in a file named graph.py, in my utility.py file I put the code that reads the text file of the graph(romania.txt) and also that does the searches(the 4 searches). Those files with names starting with 'test' are what I use in order to benchmark my search algorithms.

Finding The Path Between Each Node

I do a test by running test2-path.py file. This file will call those search algorithms and print the returned value to the console.

I repeat this process by changing the file name to the BuildGraph class constructor, since this is common to all other test classes I will copy paste some code here.

```
search =Search()
graph = BuildGraph('romania4x.txt').giveMeGraph()
nodes = graph.nodes()
```

My four text file in the file structure is corponed to the the number of nodes in the graph as a multiple of the original romania graph which is in represented in 'romania.txt' file

Average Time Needed

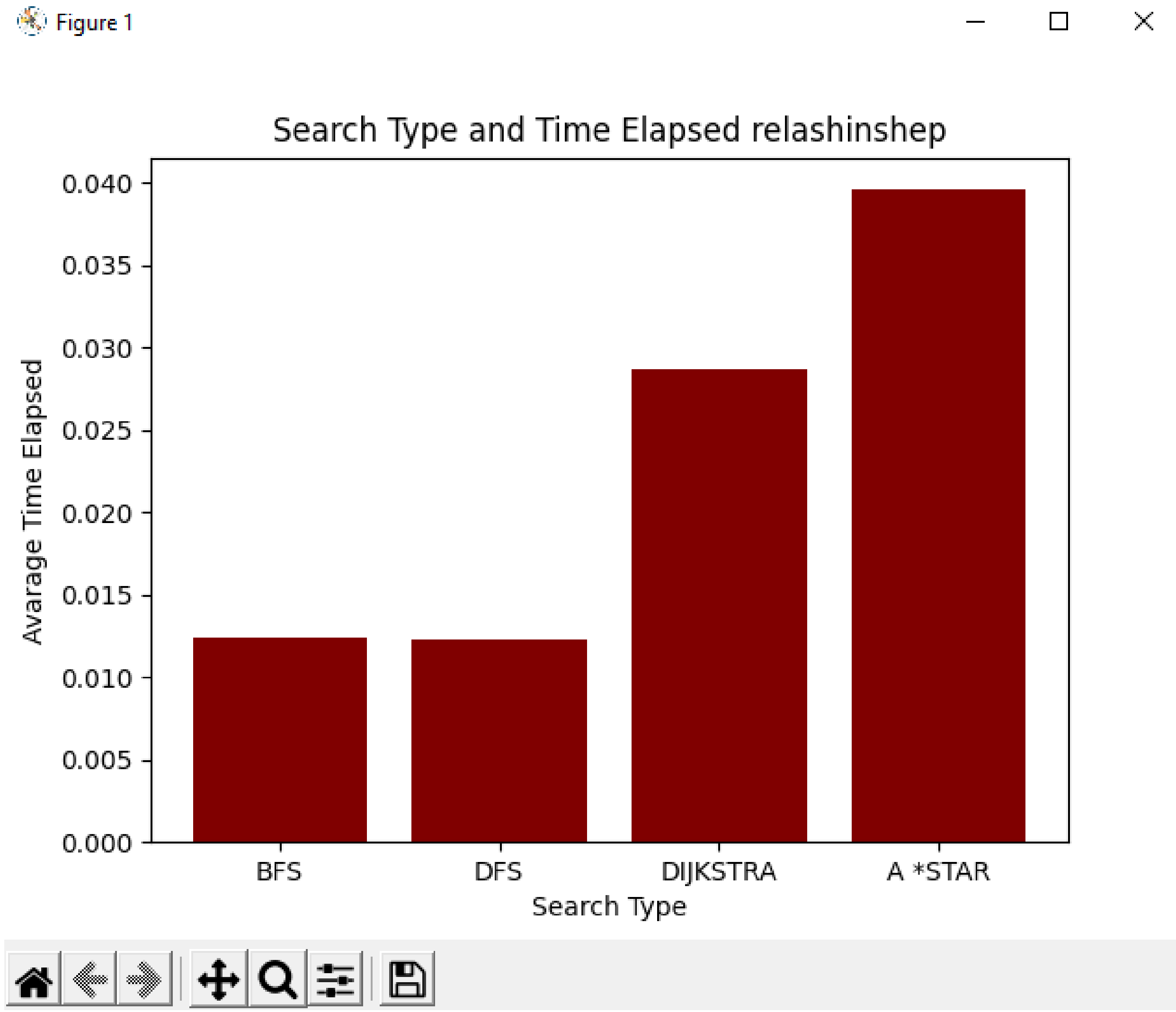
I use timeit and matplotlib mougle for the benchmark process, in order to benchmark the algorithms based on average time needed to find a solution

First I will present the data and I will explain about it at the end of this part.

When I test the search within the original **romania** graph, by running those algorithm 100 time fevry node combination except the some two node the timeit module generate the following elapsed average time time by the algorithm (by running 'test1-time.py' file I change the name of the text file to change the number of nodes of the graph).

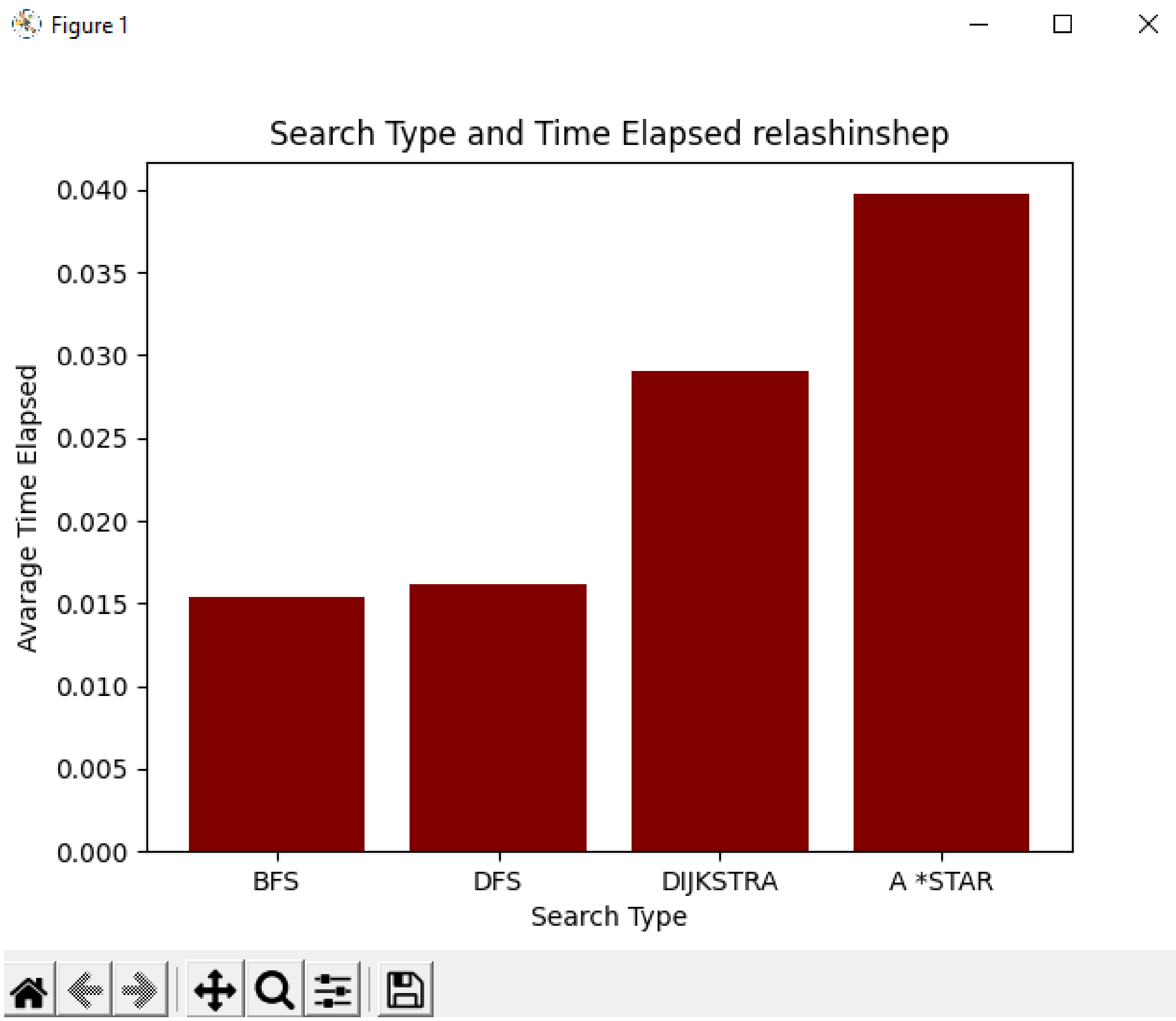
Breadth Elapsed Time: 0.012359211000000002
Depth Elapsed Time: 0.012299807999999999
Dijkstra Elapsed Time: 0.028729002
A Star Elapsed Time: 0.039556504000000006

And the plot generated by the matplotlib with the above data is this:



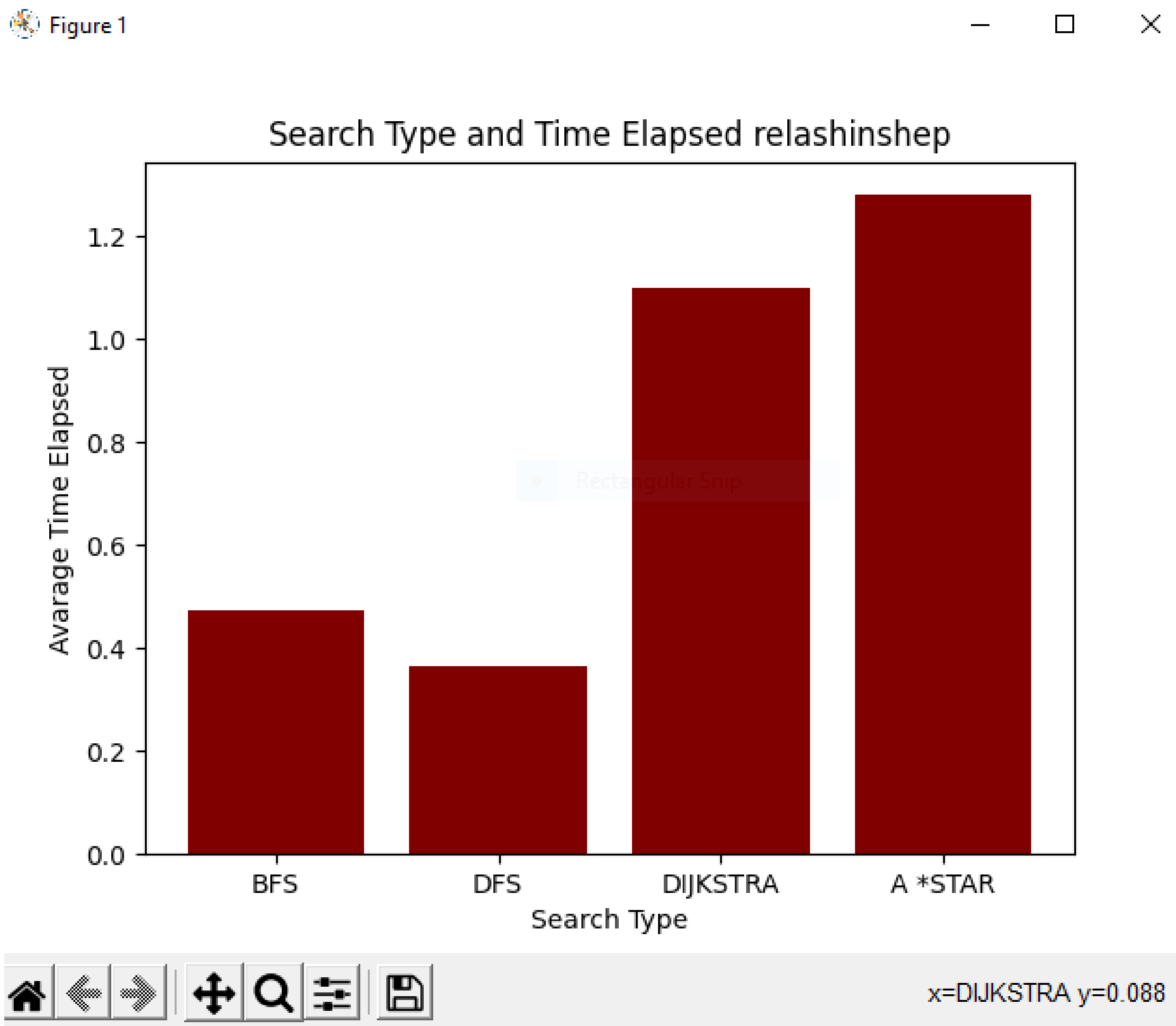
When I increase the node number of the graph by double and plot the graph within the following time it generated data after running the search 100 times and taking the average is like this

Breadth Elapsed Time: 0.115510463
Depth Elapsed Time: 0.12208503600000001
Dijkstra Elapsed Time: 0.27051646700000004
A Star Elapsed Time: 0.316912074



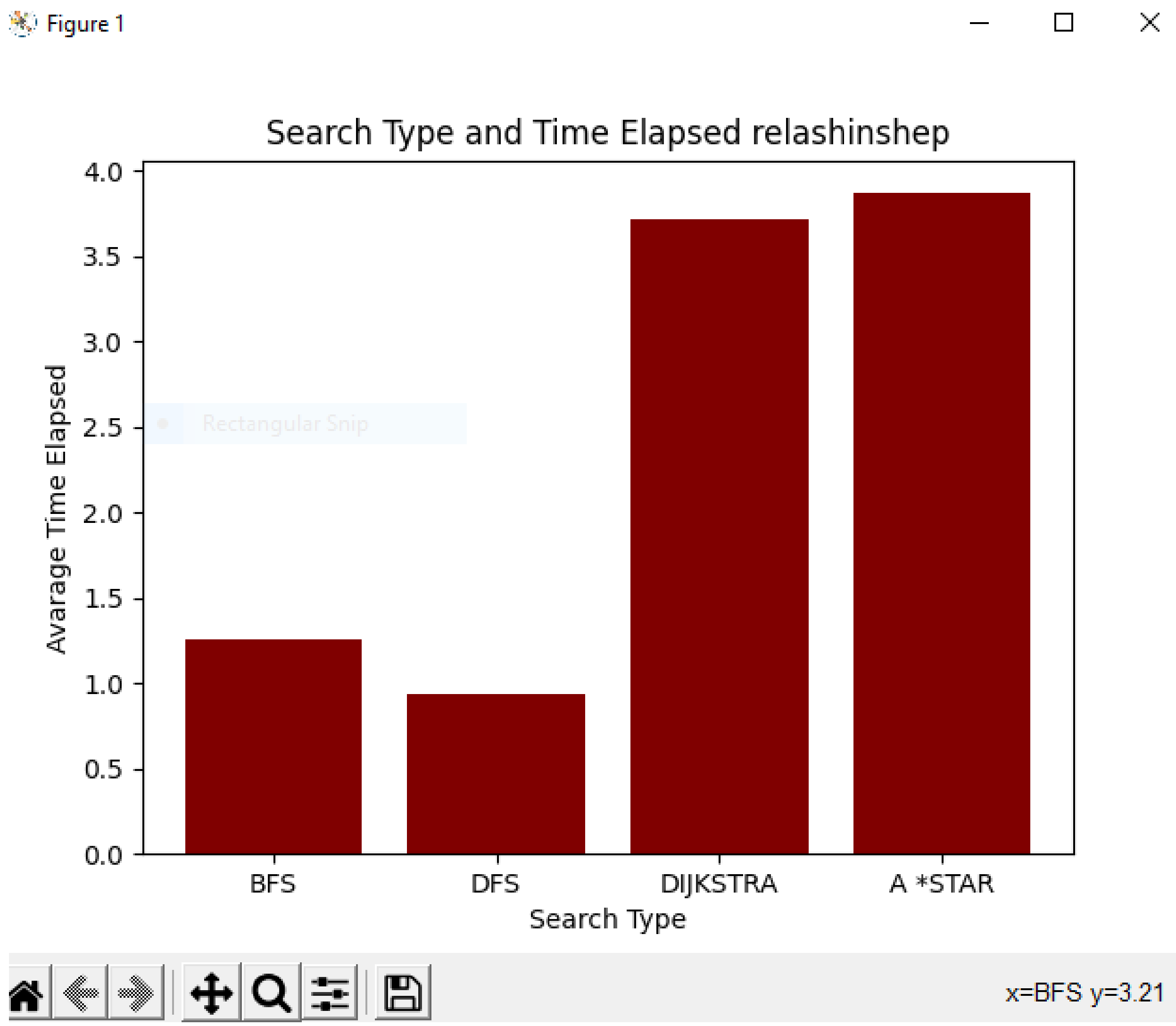
I increased the size of the graph agan in 3x size and run the text-1-time.py file with the apporprate tixt file specfird(romania3x.txt) . I found the following data and graph.

Breadth Elapsed Time: 0.472043023
Depth Elapsed Time: 0.36403834700000004
Dijkstra Elapsed Time: 1.099875477
A Star Elapsed Time: 1.280971609



Within the 4x size of the original graph i found

Breadth Elapsed Time: 1.259295541
Depth Elapsed Time: 0.9329613280000001
Dijkstra Elapsed Time: 3.7164976580000006
A Star Elapsed Time: 3.867440663



From the above data and graph we observe that our DFS function is performing better than BFS searches, this indicates to us that most of our graph nodes are placed far from each other. I said this because DFS performs better than BFS when the starting node and the target node have a long distance(in terms of the number of nodes) between each other.

We also can observe that with our graph node number increased the elapsed number of the algorithm also increased. This relationship can be easy obser by running the time-in-com.py file.

The result of the above file is this
Breadth search
original size time 0.012359211000000002
2x size time increased by fold of 9.3461033232623
3x size time increased by fold of 38.193621178568755
4x size time increased by fold of 101.89125673151787

Depth search
original size time 0.012299807999999999
2x size time increased by fold of 9.3912411478293
3x size time increased by fold of 29.597075580366788
4x size time increased by fold of 75.85169849805787

Dijkstra search
original size time 0.012359211000000002
2x size time increased by fold of 9.416145642650589
3x size time increased by fold of 38.284499997598246
4x size time increased by fold of 43.83359857053161

A Star search
original size time 0.012359211000000002
2x size time increased by fold of 8.01162999642233
3x size time increased by fold of 32.38333723829588
4x size time increased by fold of 97.77003202810843

As it is easily noticed the BFS search is taking longer as the node number increases than the previous one. In all cases DFS is doing well when compared with other algorithms.

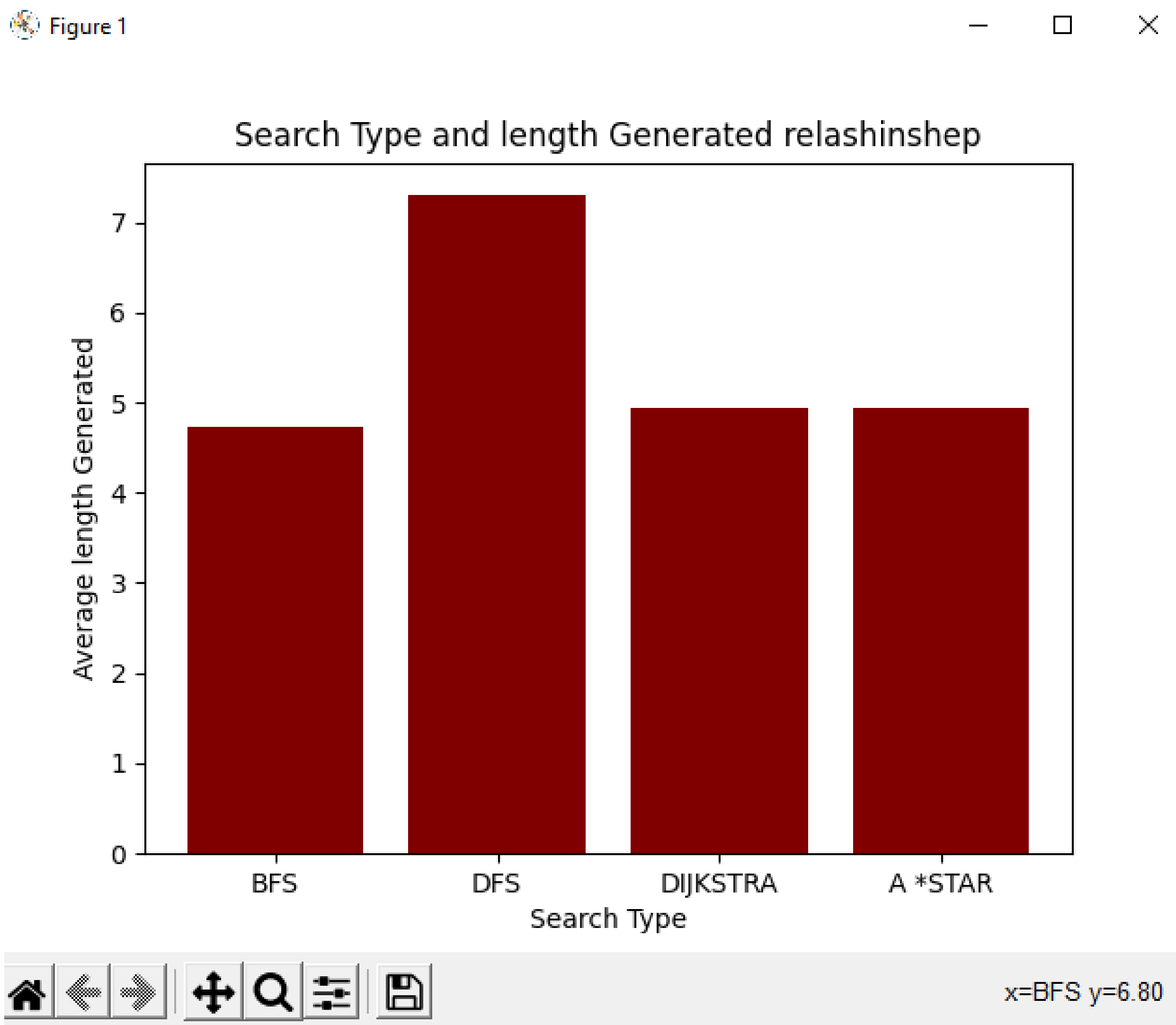
This data is dependent on the processor speed and even it is different as we repeat running the file.

Average solution length

My ‘test3-length.py’ file is dedicated to calculate the average solution length of each algorithm in different sizes of the graph.

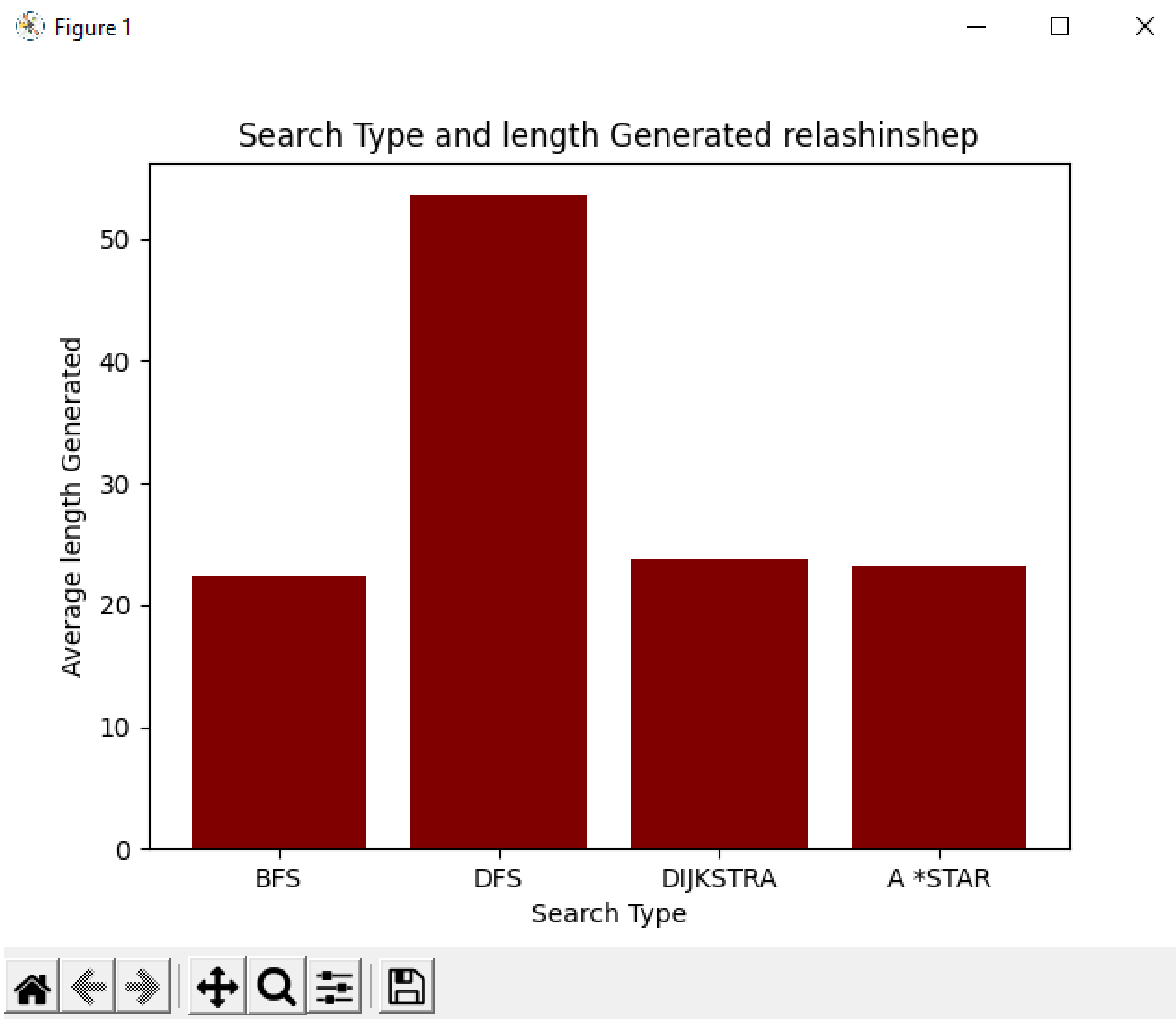
Within the original romania map when I do an average solution length test I found the following data and the plot next to it.

Breadth Length: 4.721052631578948
Depth Length: 7.2973684210526315
Dijkstra Length: 4.936842105263158
A Star Length: 4.936842105263158



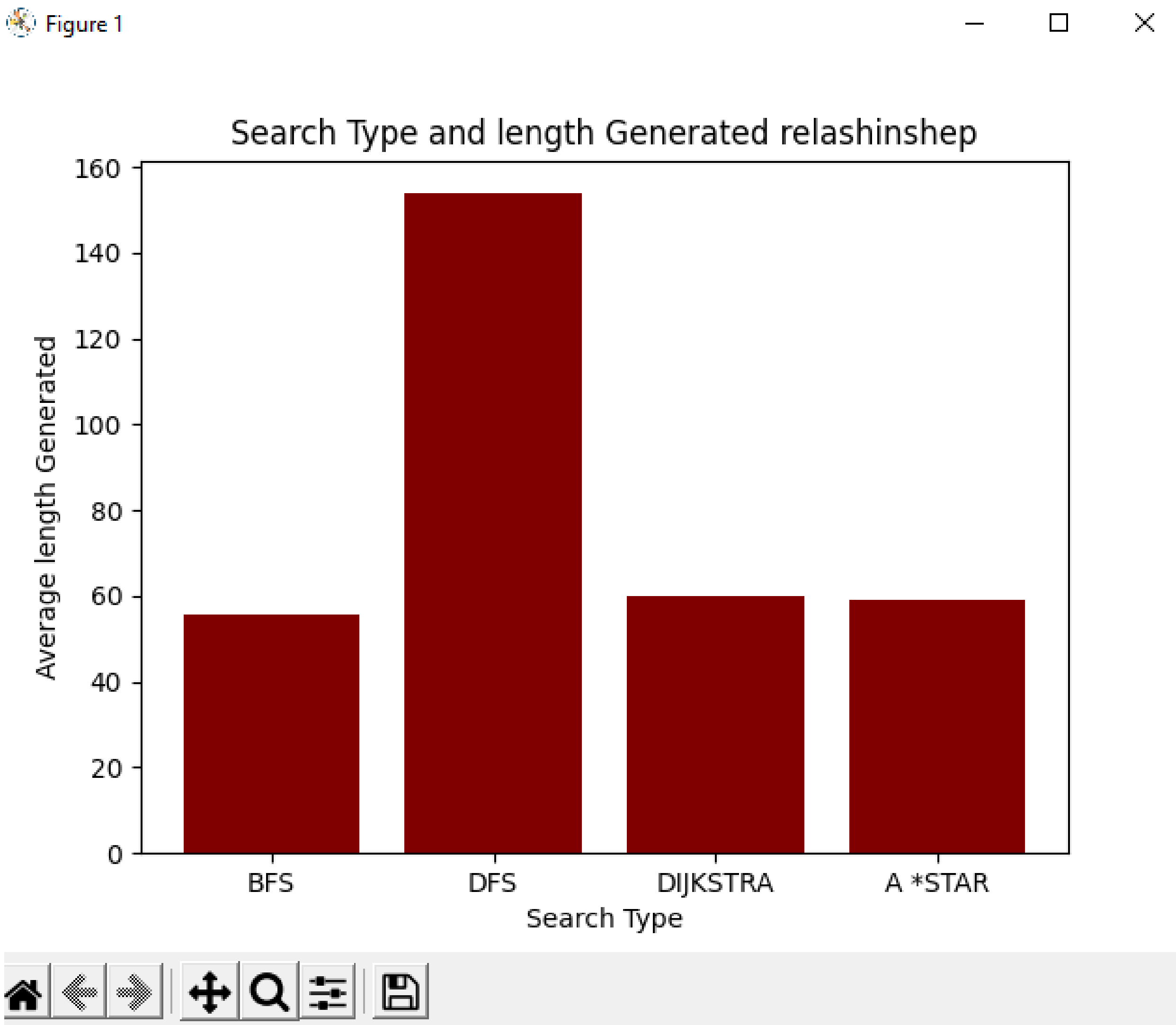
When we double the size of the graph the algorithm will have the following average solution length and I draw the plot nest.

Breadth Length: 22.347368421052632
Depth Length: 53.578947368421055
Dijkstra Length: 23.705263157894738
A Star Length: 23.236842105263158



With the size of the graph increased by a factor of 3 we will have the following information.

Breadth Length: 55.50526315789474
Depth Length: 153.81052631578947
Dijkstra Length: 59.94210526315789
A Star Length: 59.21842105263158



Again when we increase the by the factor of 4 we have:

Breadth Length: 107.36842105263158
Depth Length: 385.4973684210526
Dijkstra Length: 115.15263157894736
A Star Length: 113.98947368421052

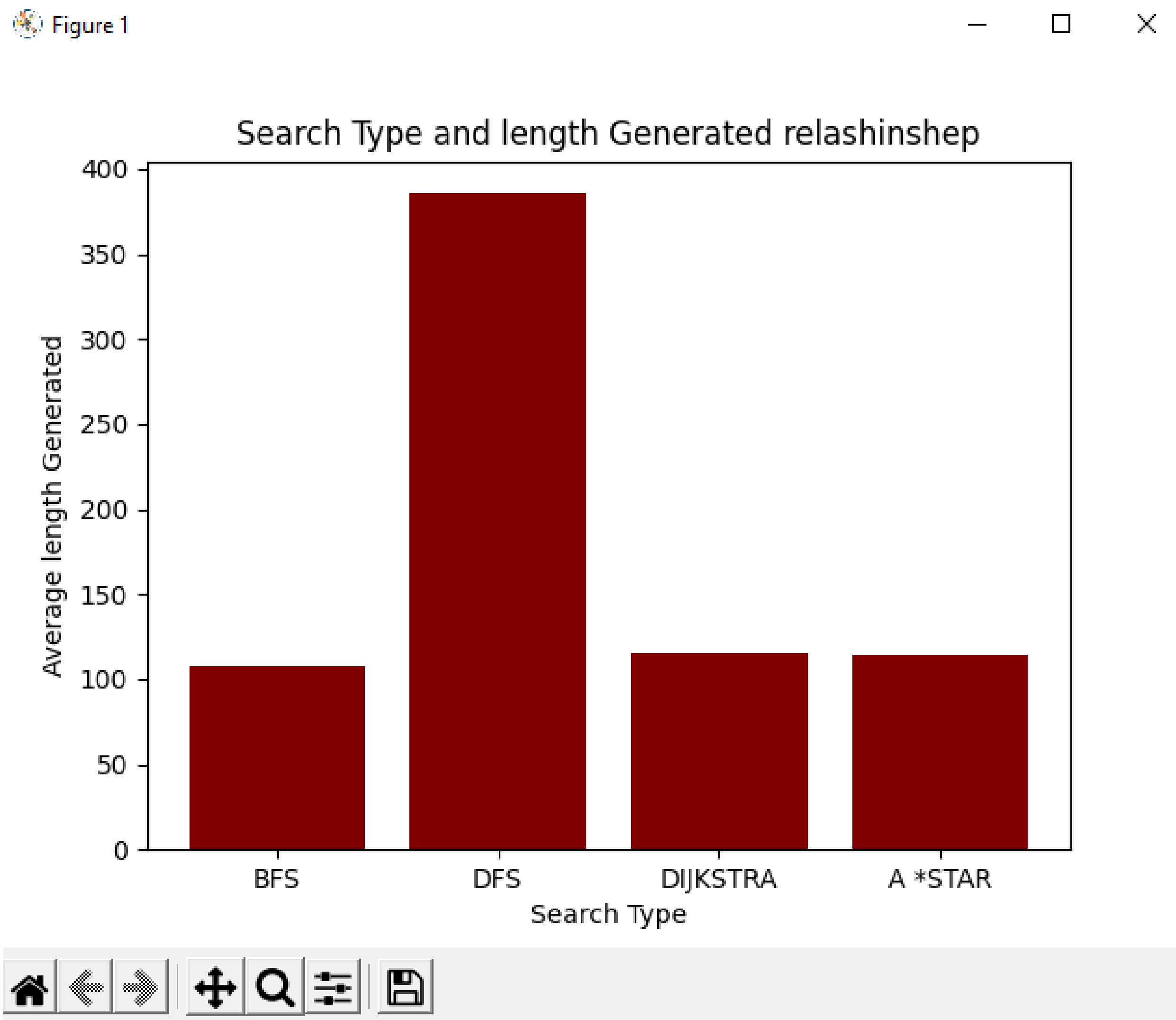


Figure 1
Search Type and length Generated relashinshep
Average length Generated
0
50
100
150
200
250
300
350
400
BFS
DFS
DIJKSTRA
A *STAR
Search Type

By observing the above image and data(actually they are the same) we can decide that a BFS search always finds a small path in terms of the number of nodes(if all edges have the same weight then this path becomes the shortest path otherwise it is difficult to decide). The DFS is always returning a path with a lot of nodes , since it returns the forest path it finds. It doesn't care about the distance)

Even If the path length of Dijkstra and a star search looks long this is because I only consider the number nodes that the returned pathe is contain. But the path length(computed from the weight of the edge) is shortest in those two searches.
As long as the number of nodes and edges are not changed the average distance of the path is constant with the DFS search as an exception.

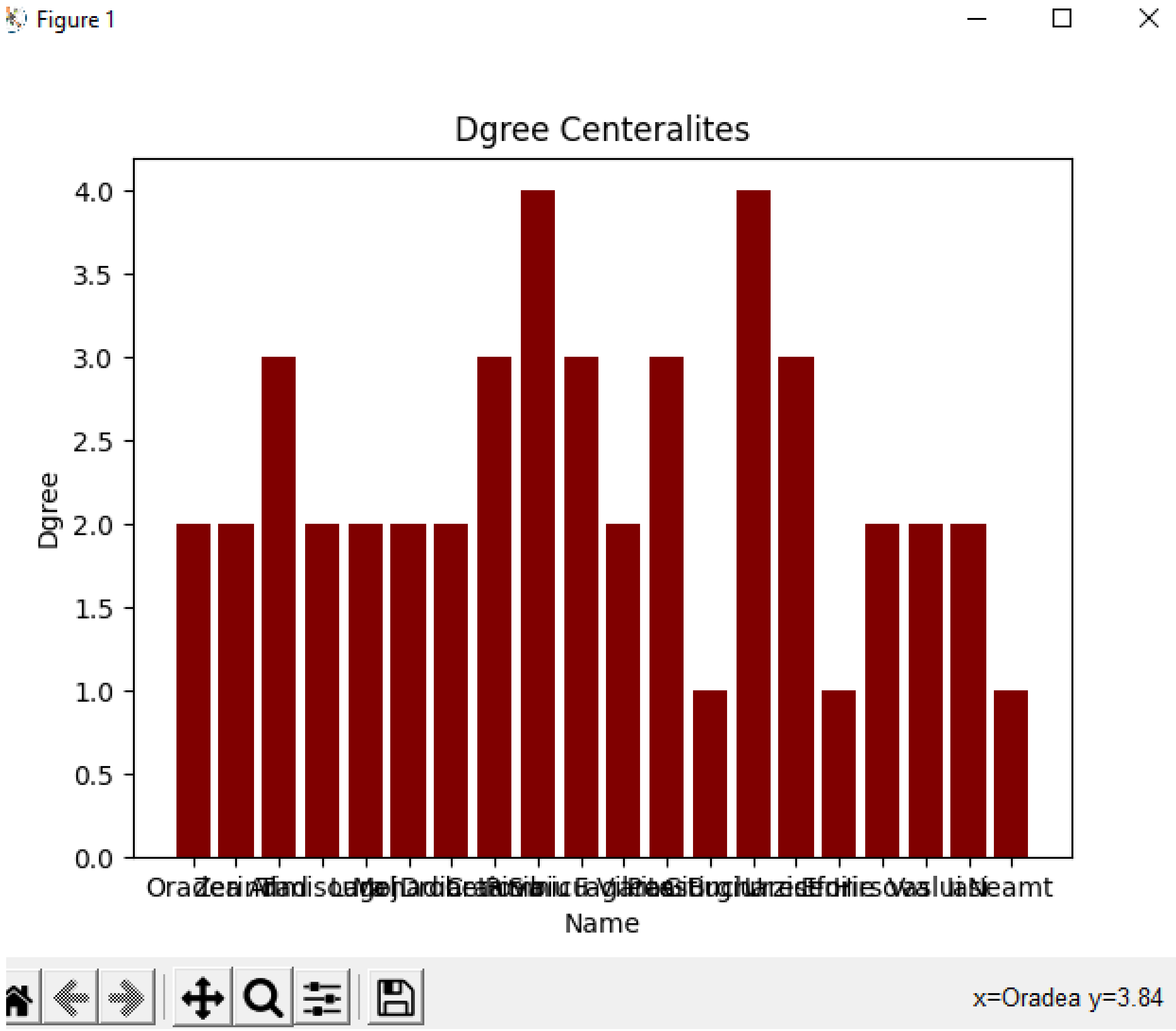
GROUP WORK

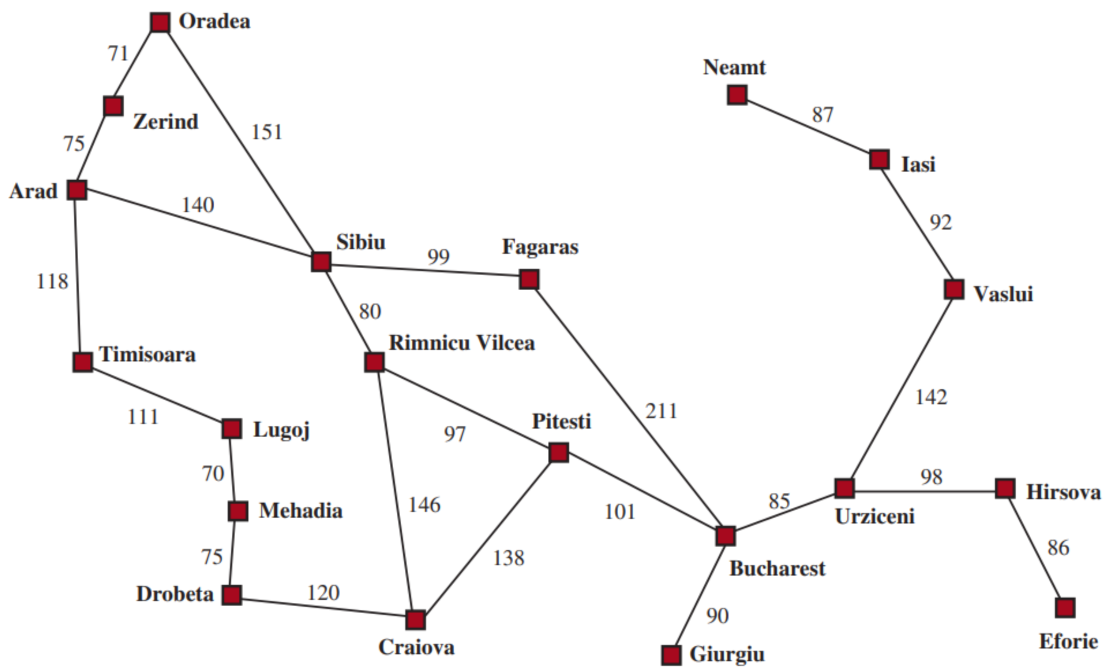
Degree centralities

By running test-5-dgree-centerality.py file we found the following node degree combination

Oradea   2
Zerind   2
Arad   3
Timisoara   2
Lugoj   2
Mehadia   2
Drobeta   2
Craiova   3
Sibiu   4
Rimnicu-Vilcea   3
Fagaras   2
Pitesti   3
Giurgiu   1
Bucharest   4
Urziceni   3
Eforie   1
Hirsova   2
Vaslui   2
Iasi   2
Neamt   1

When we plot this data we found:





When we analyzed the original graph and and the graph(or the data) we conclude:

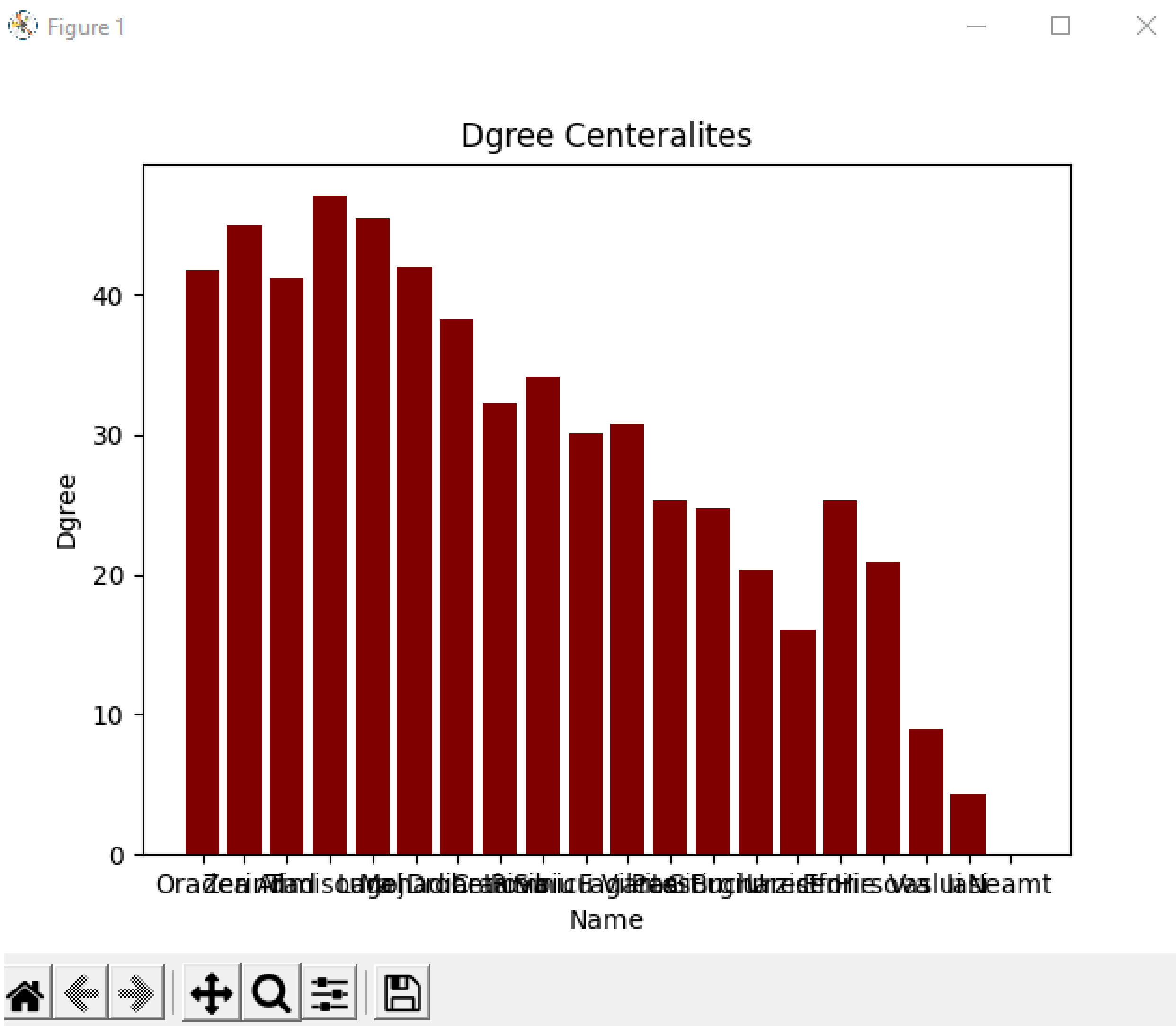
Sibiu and Bucharest have a degree of 4 .

Closeness centralities

For this test I make the return value of the dijkstra and A star search the length. By running test-6-cloness.py file I get the following data and graph.

Oradea   41.75
Zerind   44.95
Arad   41.2
Timisoara   47.1
Lugoj   45.5
Mehadia   42.0
Drobeta   38.25
Craiova   32.25
Sibiu   34.2
Rimnicu-Vilcea   30.2
Fagaras   30.85
Pitesti   25.35
Giurgiu   24.8
Bucharest   20.3
Urziceni   16.05

Eforie 25.25
Hirsova 20.95
Vaslui 8.95
Iasi 4.35



From the above data since I didn't take the reciprocal the gig data is the the closest that is Timisoara

Betweenness centralities

By running test4-betweenness-centralities.py file we found the folloind data :

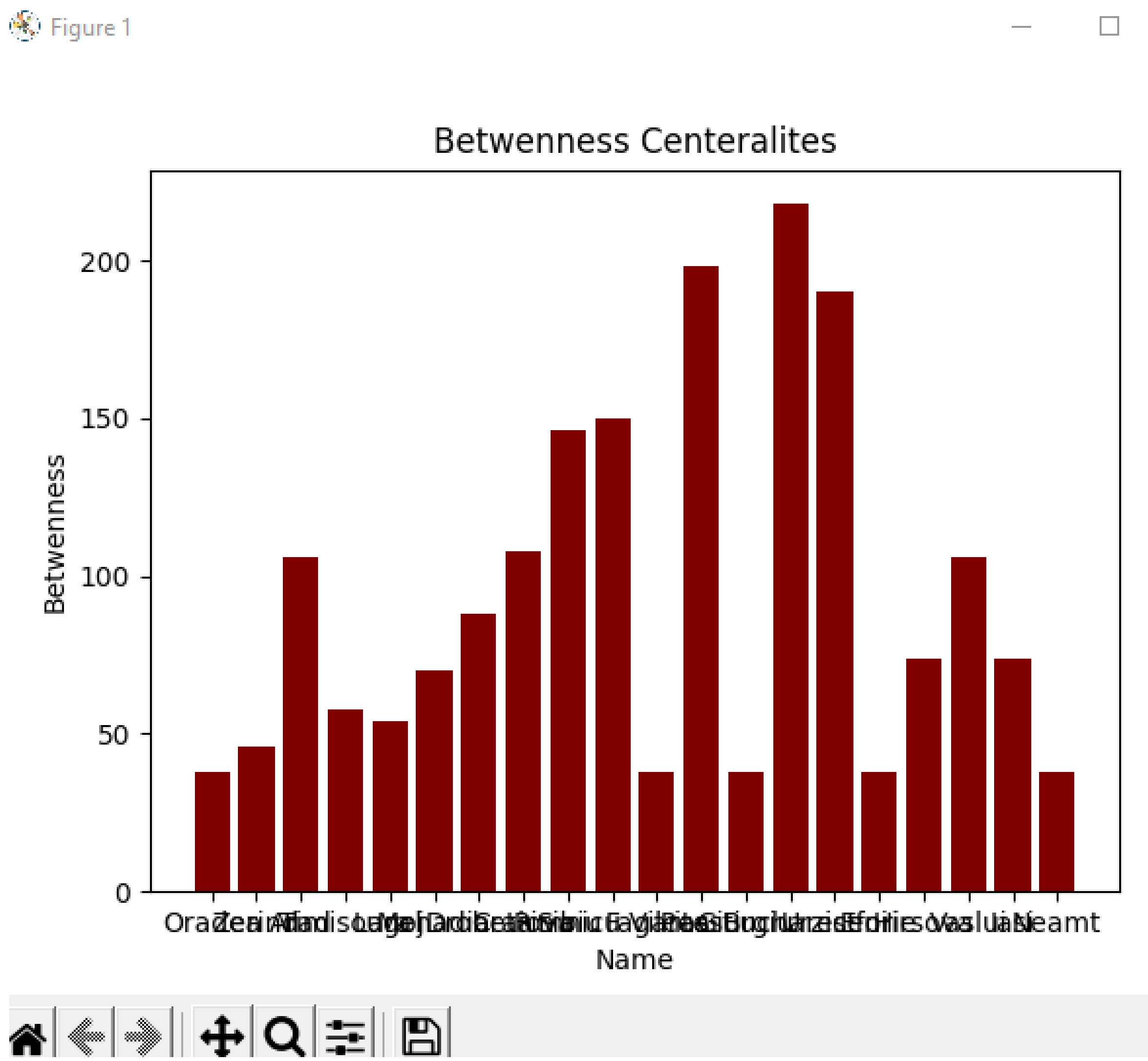
Betweenness centralities By Dijkstra Search
Oradea 0.1
Zerind 0.12105263157894737

Arad 0.2789473684210526
Timisoara 0.15263157894736842
Lugoj 0.14210526315789473
Mehadia 0.18421052631578946
Drobeta 0.23157894736842105
Craiova 0.28421052631578947
Sibiu 0.38421052631578945
Rimnicu-Vilcea 0.39473684210526316
Fagaras 0.1
Pitesti 0.5210526315789473
Giurgiu 0.1
Bucharest 0.5736842105263158
Urziceni 0.5
Eforie 0.1
Hirsova 0.19473684210526315
Vaslui 0.2789473684210526
Iasi 0.19473684210526315
Neamt 0.1

Betweenness centralities By Star Search

Oradea 0.1
Zerind 0.12105263157894737
Arad 0.2789473684210526
Timisoara 0.15263157894736842
Lugoj 0.14210526315789473
Mehadia 0.18421052631578946
Drobeta 0.23157894736842105
Craiova 0.28421052631578947
Sibiu 0.38421052631578945
Rimnicu-Vilcea 0.39473684210526316
Fagaras 0.1
Pitesti 0.5210526315789473
Giurgiu 0.1
Bucharest 0.5736842105263158
Urziceni 0.5
Eforie 0.1
Hirsova 0.19473684210526315
Vaslui 0.2789473684210526
Iasi 0.19473684210526315
Neamt 0.1

I show the data within the two searches even if they are the same. The plot of the above data is



As seen from the data Bucharest has a large betweenness value indicating it`s important in the graph . we can also see decide this by observing the graph as Bucharest is used as a junction point to connect the above part and the lower part of romania.