Table of Contents

Introduction to Artificial Intelligence Assignment I

1. Peas problem formulation

1. Identify two very different existing AI systems and characterize them based on the PEAS problem formulation. Give a detailed explanation of the applications based on these four fundamental concepts. (Read Part I, Chapter 1, and Chapter 2)

What is PEAS problem formulation?
PEAS stands for Performance, Environment, Actuator and Sensor.

- **Performance** is used to judge the success of the agent. It involves things we can evaluate an agent against to know how well it performs.
- **Environmen**t is where the agent needs to deliberate actions. what the agent can perceive from.
- **Actuators** are the tools, equipment or others in which an agent performs actions in the environment. This works as an output of the agent.
- **Sensors** are tools in which an agent captures the state of the environment. This works as input to the agent

Two different AI systems I chose to explain based on PEAS problem formulation are **recommendation systems and vacuum cleaner robots.**

AI recommendation systems

Recommendation systems are systems that are designed to recommend things to the user based on many different factors. These systems predict the most likely products the user will be interested in and are used in many applications like amazon,netflix and youtube.

The system achieves its work by dealing with a large amount of information to identify the most important information provided by the user and finding similarities between users and items used by them for recommendation purposes.

This system based on the PEAS problem formulation is explained below

1. Performance

The performance of recommendation systems can be measured as follows

Accuracy, is the fraction of correct recommendations out of total possible recommendations.

Coverage, it measures the fraction of objects in the search space the system is able to provide recommendations for.

2. Environment

Recommendation systems work dependably on other software data especially users information for this reason their environment is considered to be softwares in which the

system is used and a sea of information the software provides.

3. Actuators

4. Sensors

Vacuum cleaner robots

Vacuum cleaner Robot is a smart home appliance AI system which can clean the floor automatically. It uses different sensors to detect and measure the world around them and their own progress through it.This combination of sensors means that the robot knows a few things about the world around it such as how far it has gone, things it has bumped into and things it could fall off from.

This system based on the PEAS problem formulation is explained below

1. Performance

The performance of vacuum cleaner is measured by How Effectively and Efficiently it can clean a given environment Distance travelled to clean How long it can last(Battery life) Safety, amount of probable damage it may cause

2. Environment

The surrounding environment includes room, table, wooden floor, carpet, and different obstacles the robot faces.

3. Actuators

This system acts on the environment through Wheels, to travel the distance it should clean, Different brushes and vacuum extractor.

4. Sensors

This system perceives its environment through Dirt detection sensors Cliff sensor Bump sensor Infrared wall sensor Optical encoders

2. Creating a graph

2. Using your self-made graph library, try loading the graph data presented on page 83rd of the textbook.

I created a create graph function that accepts the edges information file and heuristic data file, reads them, and adds the nodes into the graph. I read the heuristic data and stored it in a globally created variable. The code looks like the following.

```
def create_graph(self, graph_file, heuristic_file):
        with open(graph_file) as file:
            for line in file:
                connection = line.split()
                if connection[0] not in self.g.verticies:
                    node1 = gi.Node(connection[0])
                    self.g.add_node(node1)
                if connection[1] not in self.g.verticies:
                    node2 = gi.Node(connection[1])
                    self.g.add_node(node2)
                self.g.add_edge(self.g.verticies[connection[0]],
self.g.verticies[connection[1]],connection[2])

        with open(heuristic_file) as file:
            for line in file:
                data = line.split()
                self.heuristic_data[data[0]] =
[radians(float(data[1])),radians(float(data[2]))]
```

3. Searching Algorithms

3. Implement BFS, DFS, Dijkstra's shortest path, and A* Search algorithm. Using the graph from Question 2, evaluate each of your algorithms and benchmark them. The benchmark should be finding the path between each node. The benchmark result should include the average time needed to find a solution and the average solution length of each algorithm.

My algorithm for the BFS, DFS, Dijkstra and A* search algorithms is as follows.

BFS

```
    def bfs(self, start, end):
        count = 0
        queue = deque([start])
        visited = set()
        while queue:
            count+=1
            for _ in range(len(queue)):
                temp = queue.popleft()
                visited.add(temp.name)
                for nodes in temp.edge_list:
                    if nodes.name not in visited:
                        queue.append(nodes)
                        if nodes.name == end.name:
                            return count
        return count
```

DFS

```
def dfs(self, start, end, dfs_visited):
        dfs_visited.add(start.name)
        if start.name == end.name:
            return
        for nodes in start.edge_list:
            if nodes.name not in dfs_visited:
                self.dfs(nodes, end, dfs_visited)
        return
```

Djikstra’s Shortest Path

```
def djikstra(self, start):
```

```
        dis_start = {}
        previous = {}
        for node_name in self.g.verticies:
            dis_start[node_name] = float("inf")
            previous[node_name] = start.name
        dis_start[start.name] = 0
        unvisited = [[0, start.name]]
        heapq.heapify(unvisited)
        visited = set()
        while unvisited:
            temp = heapq.heappop(unvisited)
            visited.add(temp[1])
            for node in self.g.verticies[temp[1]].edge_list:
                if node.name not in visited:
                    new_dis = dis_start[temp[1]] +
int(self.g.edges[(node.name, temp[1])].weight)
                    if new_dis < dis_start[node.name]:
                        dis_start[node.name] = new_dis
                        previous[node.name] = temp[1]
                        heapq.heappush(unvisited, [dis_start[node.name],
node.name])
        return [dis_start, previous]
```

A* search

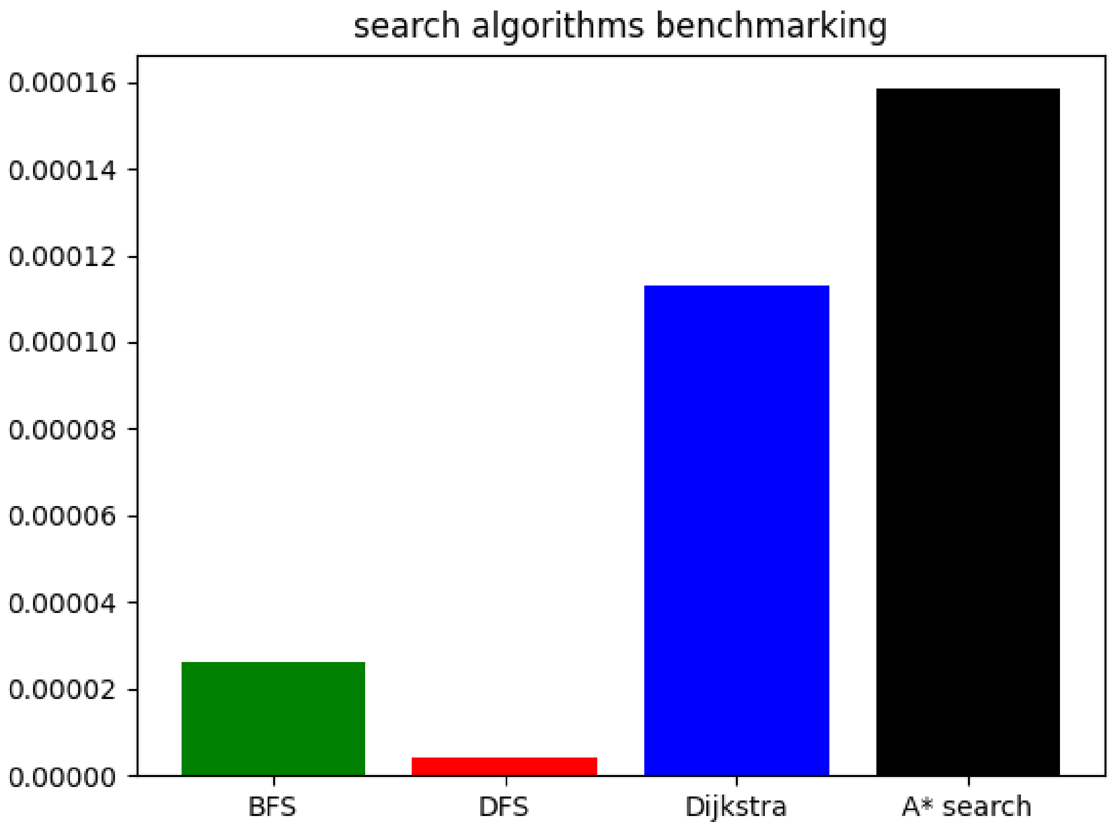
```
def Astarsearch(self, start, end):
        heuristic_dis = self.calc_heuristic_dis(end)
        f = {}
        dis_start = {}
        previous = {}
        for node_name in self.g.verticies:
            dis_start[node_name] = float("inf")
            previous[node_name] = start.name
            f[node_name] = float("inf")
        dis_start[start.name] = 0
        f[start.name] = heuristic_dis[start.name] + dis_start[start.name]
        unvisited = [[heuristic_dis[start.name], start]]
        heapq.heapify(unvisited)
        visited = set()
```

```
        flag = False
        while unvisited:
            temp = heapq.heappop(unvisited)
            visited.add(temp[1])
            for node in temp[1].edge_list:
                if node not in visited:
                    new_dis = dis_start[temp[1].name] +
int(self.g.edges[(node.name, temp[1].name)].weight)
                    temp_dis = heuristic_dis[node.name] + new_dis
                    if temp_dis < f[node.name]:
                        # if temp_dis < f[node.name]:
                        dis_start[node.name] = new_dis
                        previous[node.name] = temp[1].name
                        f[node.name] = temp_dis
                        heapq.heappush(unvisited, [f[node.name], node])
                    if node.name == end.name:
                        flag = True
                        break
            if flag:
                break
        # finding the path to reach end starting from start node
        path = []
        while end.name != start.name:
            path.append(end.name)
            end = self.g.verticies[previous[end.name]]
        path.append(start.name)
        shortest_path = path[::-1]
        return [dis_start,shortest_path]
```

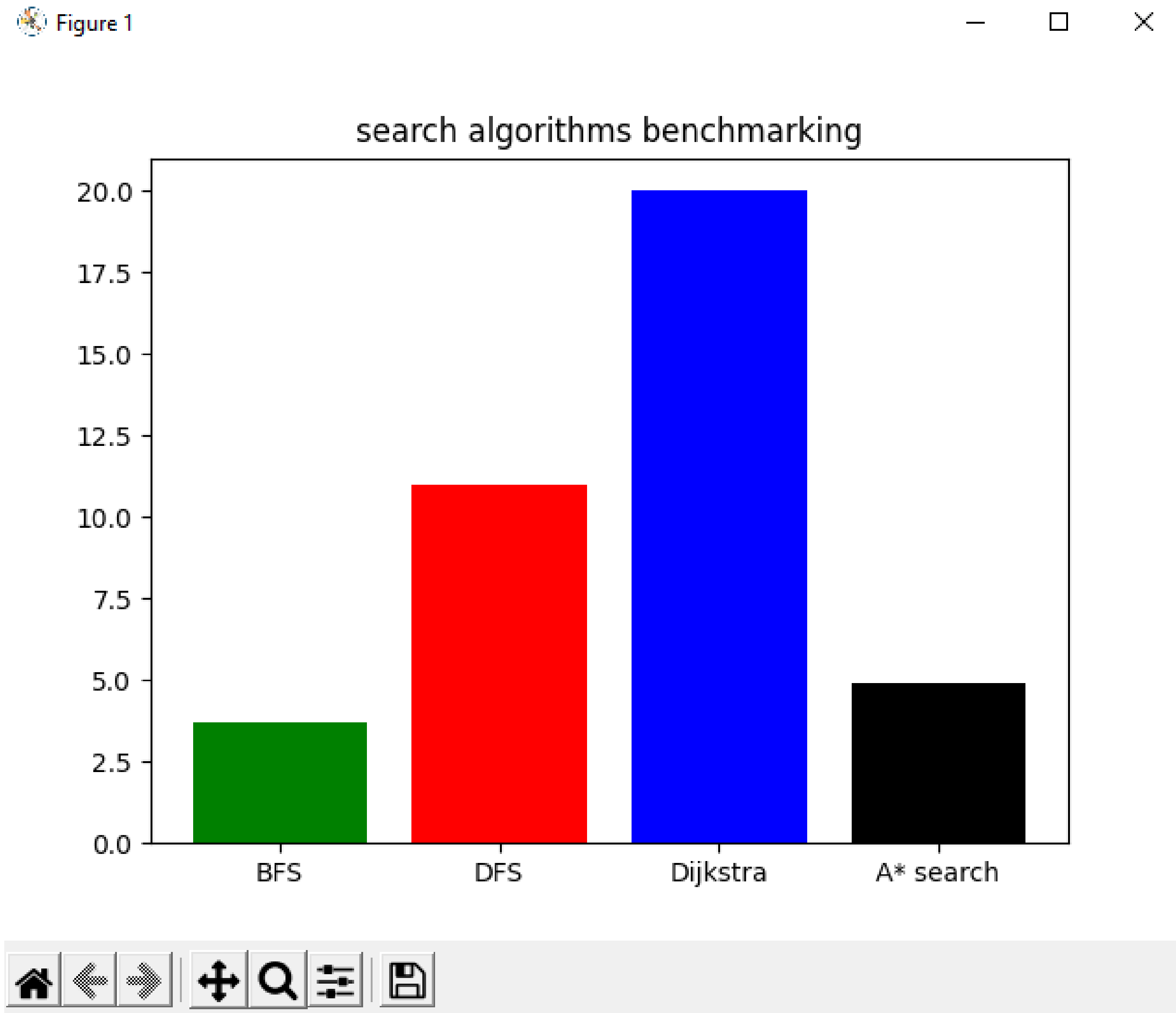
Benchmark

The following images show the average time and average solution length benchmark of the above four searching algorithms. These graphs show that DFS is the fastest algorithm to find the distance between two nodes, but not shortest. But we have to know that this graph is based on the 20 nodes provided in the book. Dijkstra looks faster than the A* search in the following graph, because of the node size. As the number of nodes increases, we will see the real benchmark. The Second factor why Dijkstra is faster in my implementation is because I added one more loop in my Astar function to get the real path.

Average time benchmark



Average solution length benchmark



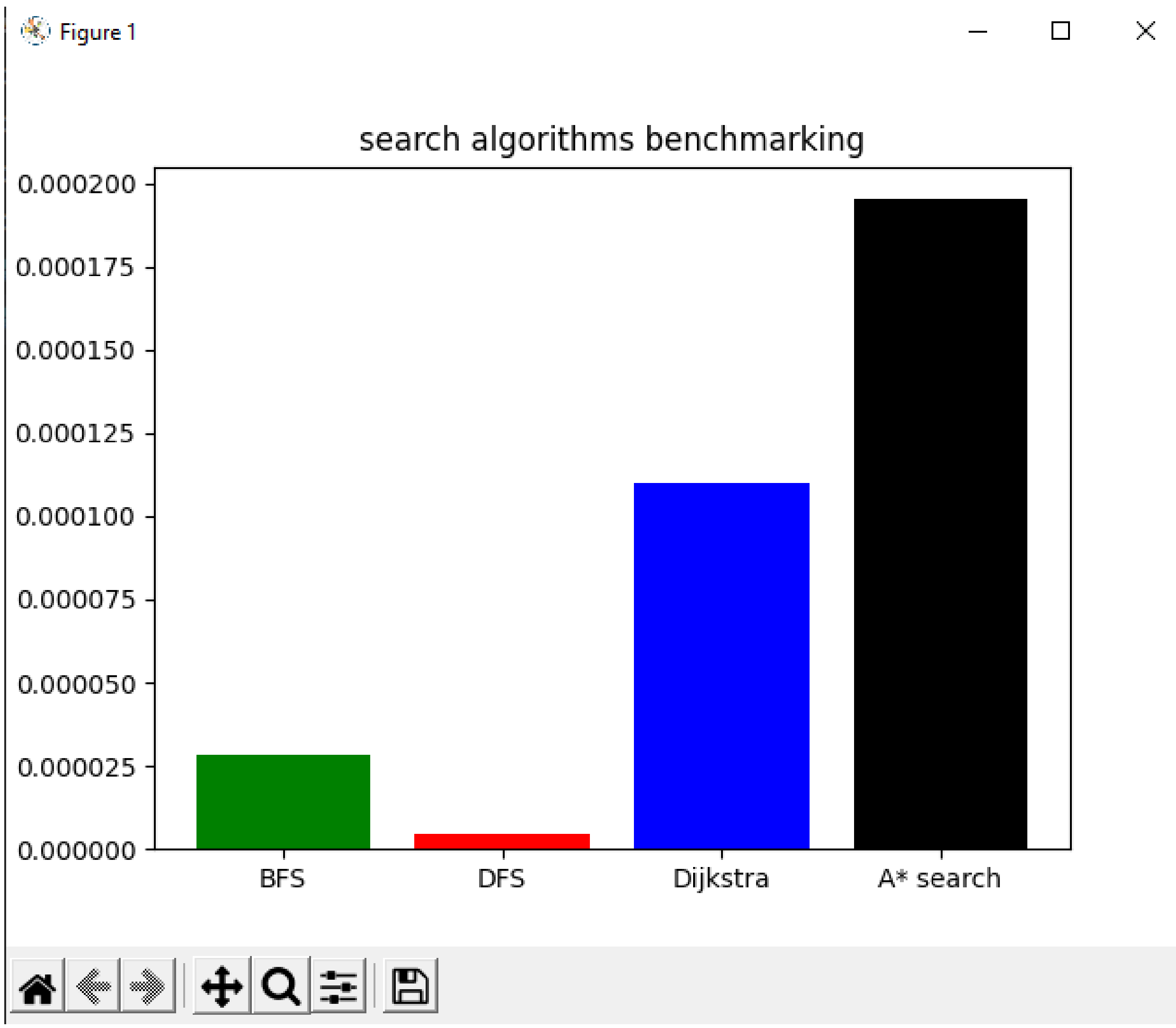
Bonus

a. Bonus - create random nodes of your own and randomly develop connections with the original graph. The number of your random nodes should be 1x, 2x, 3x, 4x.. of the original size. Evaluate each algorithm on these graph sizes and observe what happens to the benchmark. Use matplotlib.pyplot to plot their average time and solution length on each graph sizes

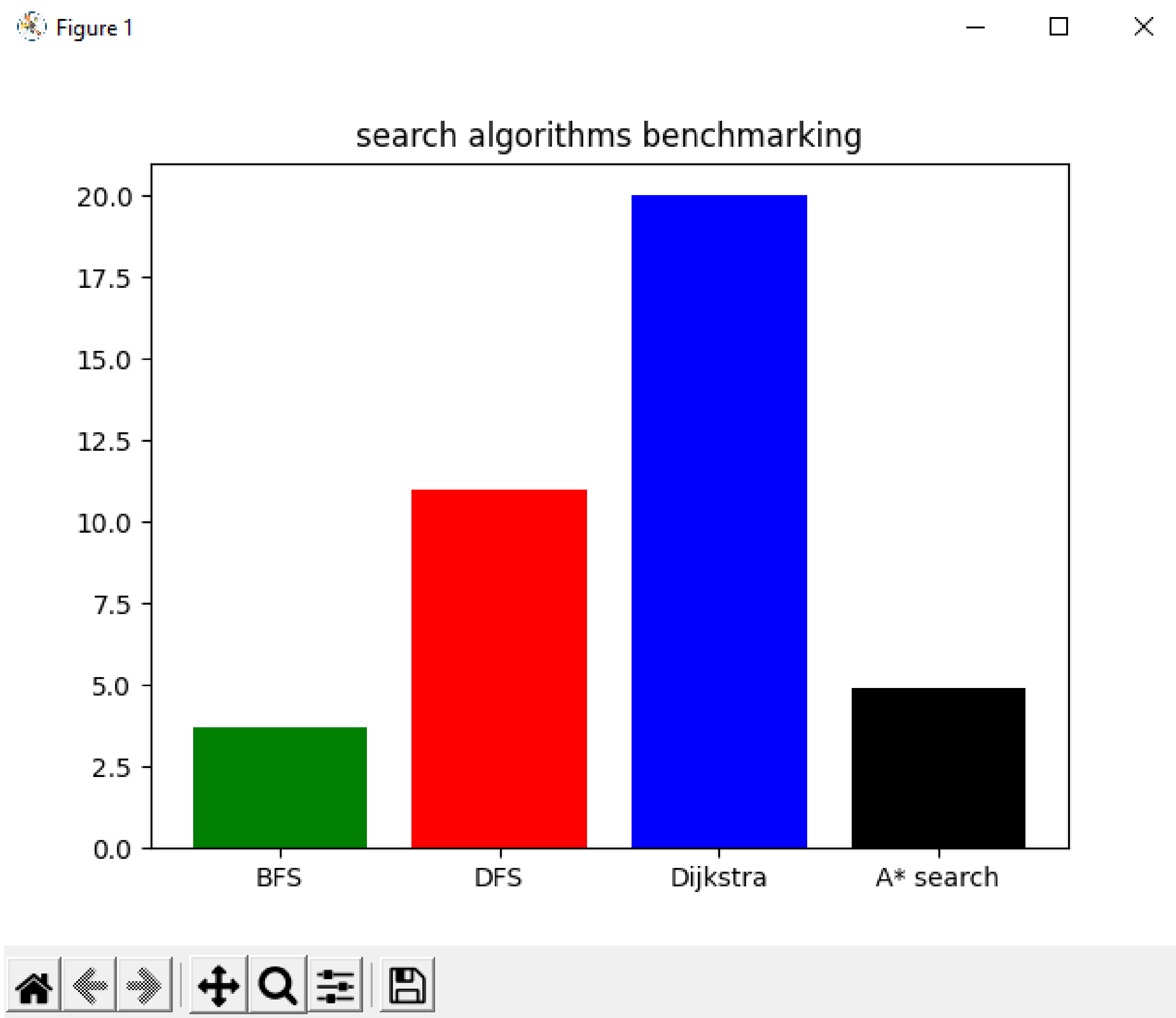
I generated 1x, 2x, 3x, and 4x random nodes and added them to the original file to create a more complex graph. I used those files to check the benchmarks of the four searching algorithms. The detailed benchmark for all of the five files is as follows.

The following graph is based on the 1x (20) original nodes given in the book.

Average time

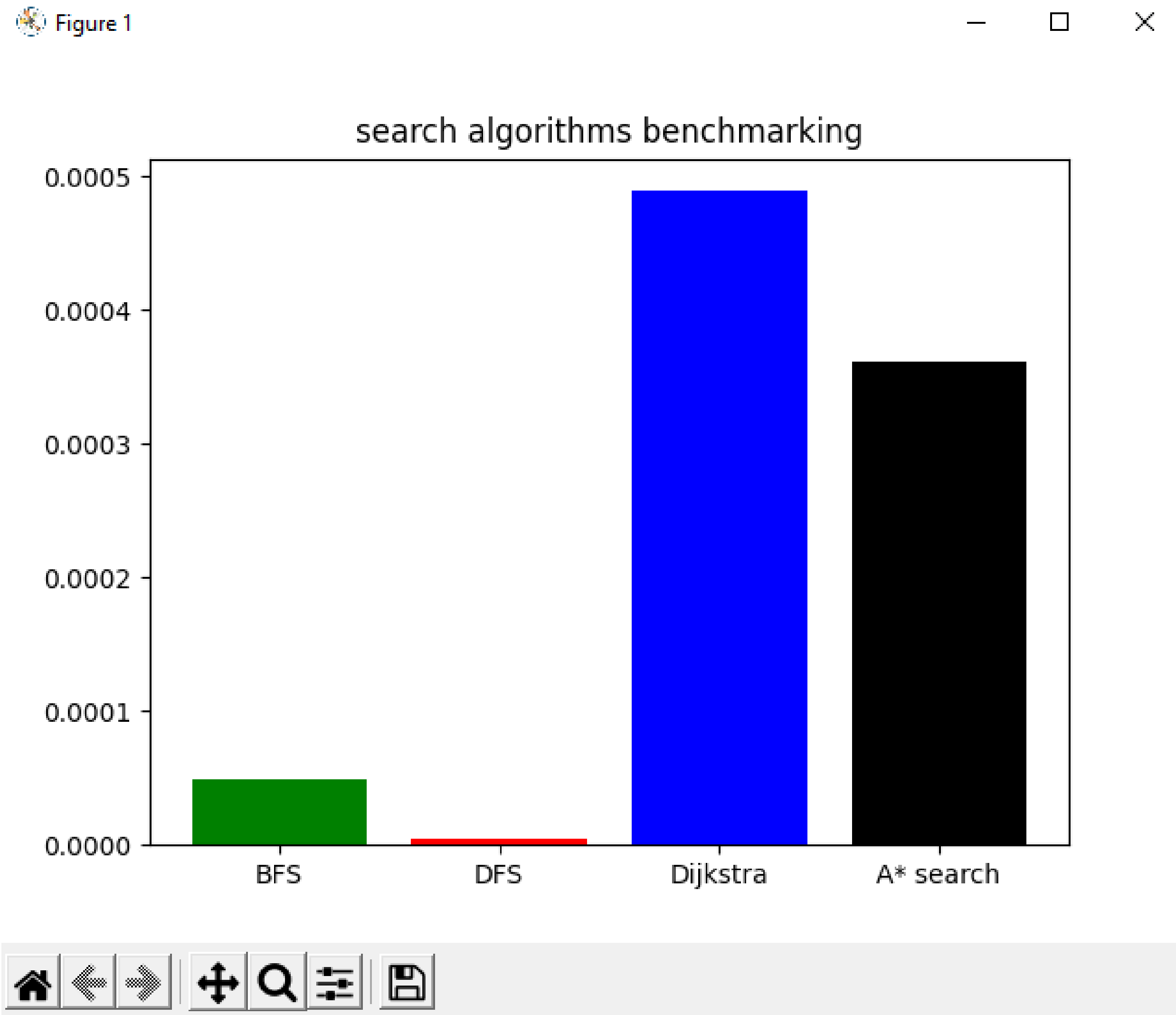


Average solution length

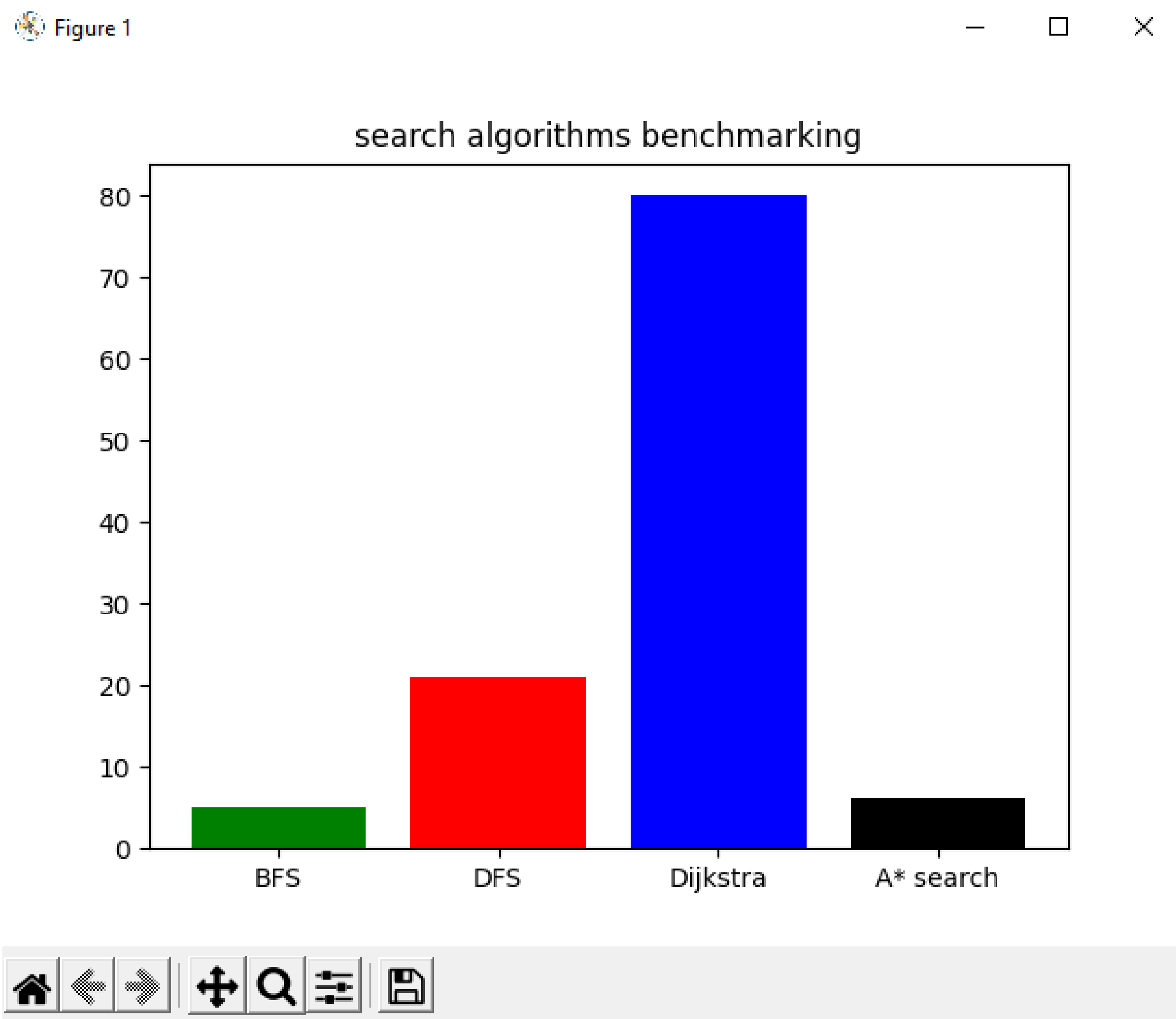


The following graph is a benchmark for 2x (40) nodes, where x is the number of nodes of the initially given nodes. This graph shows as number of nodes increases The A* search algorithm starts to be better than Dijkstra in finding the shortest path.

Average time

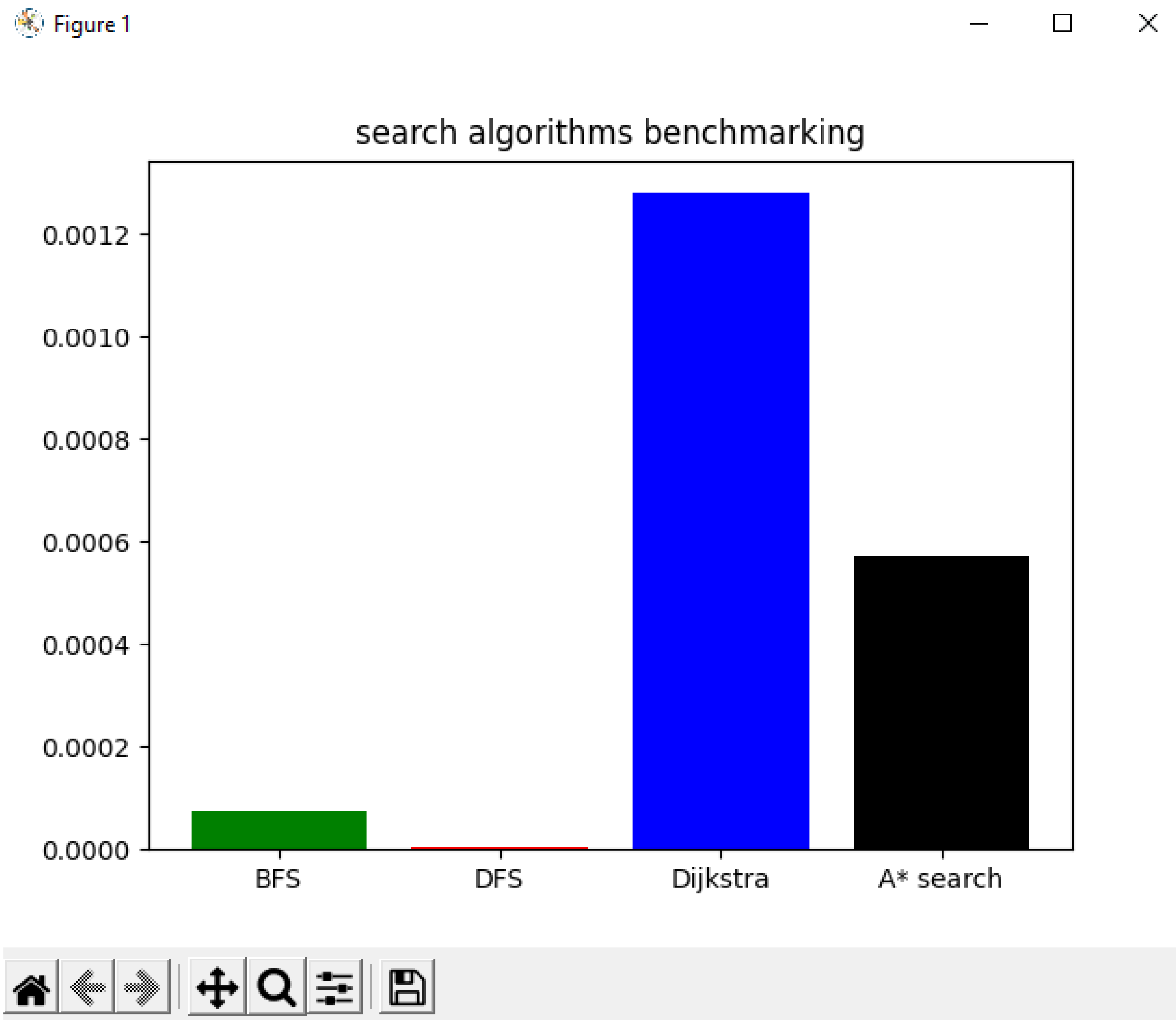


Average solution length

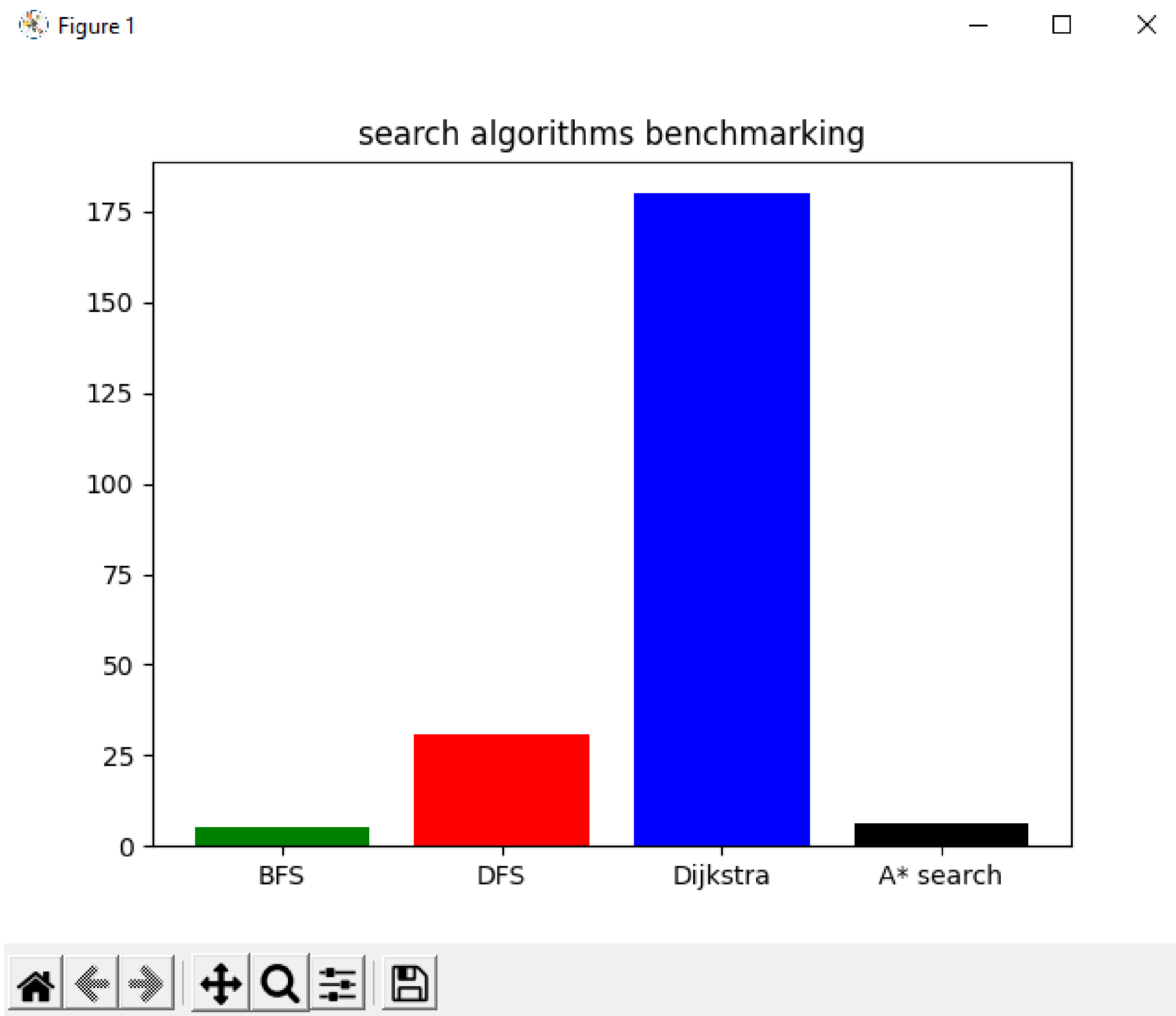


The following graph shows a benchmark for all of the four searching algorithms on 3x (60) nodes. As we can see from the following graph, The time graph heights of the A* search algorithm are becoming shorter than Dijkstra’s search algorithm. That is because, as the number of nodes increases, using the A* search is the best option to get the shortest path between two nodes even if it can not give us the exact answer. Since A* search works greedily, we might not get the optimal solution, but we will get the answer faster and without touching many nodes.

Average time

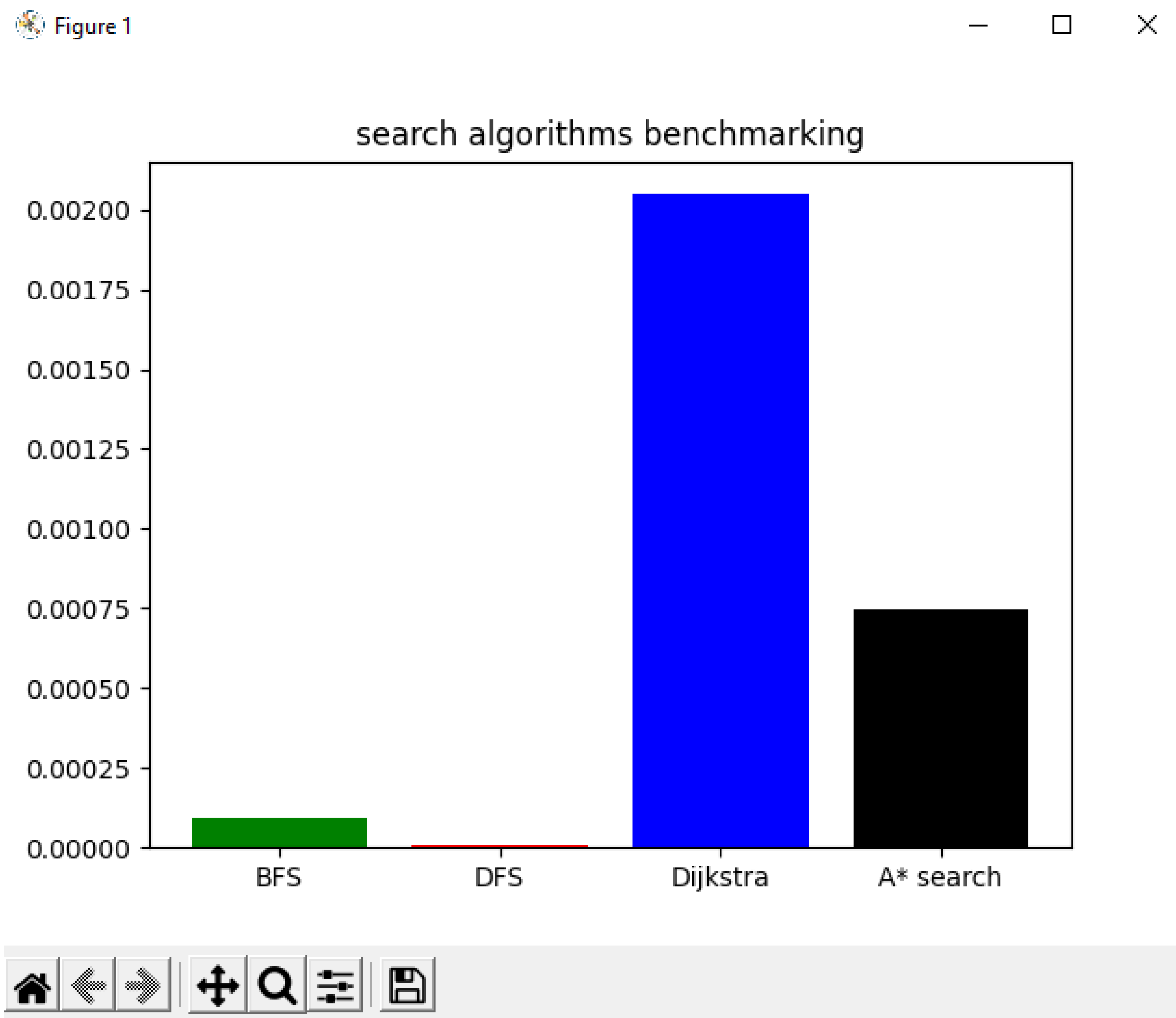


Average solution length

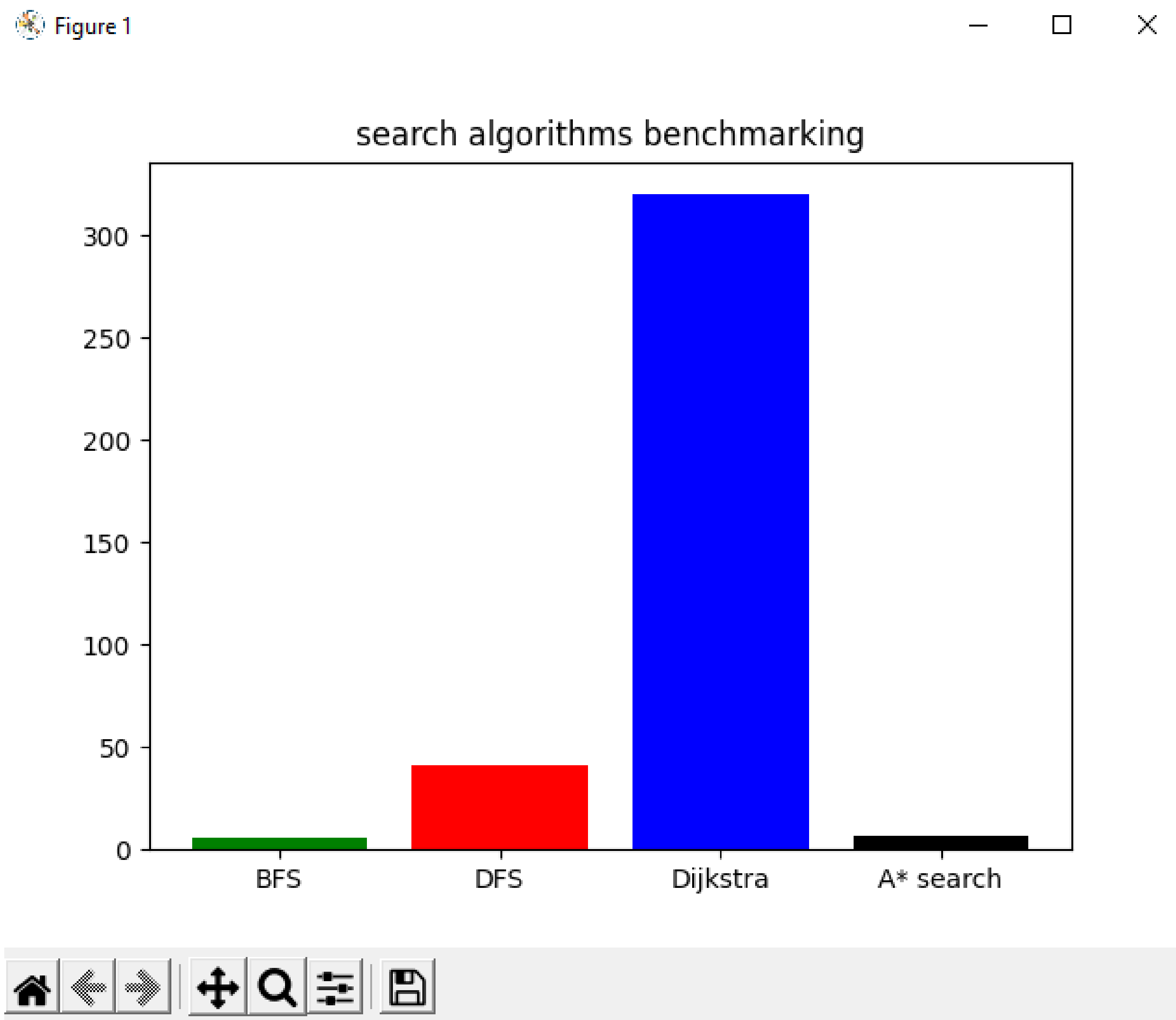


The following graph is a benchmark of the four searching algorithms on a 4x (80) number of nodes. As we can see from the following graph, it takes much more time to find the shortest path using Dijkstra than A* search. Therefore we can conclude that on a large number of nodes using the A* search algorithm is better for finding the shortest path faster.

Average time

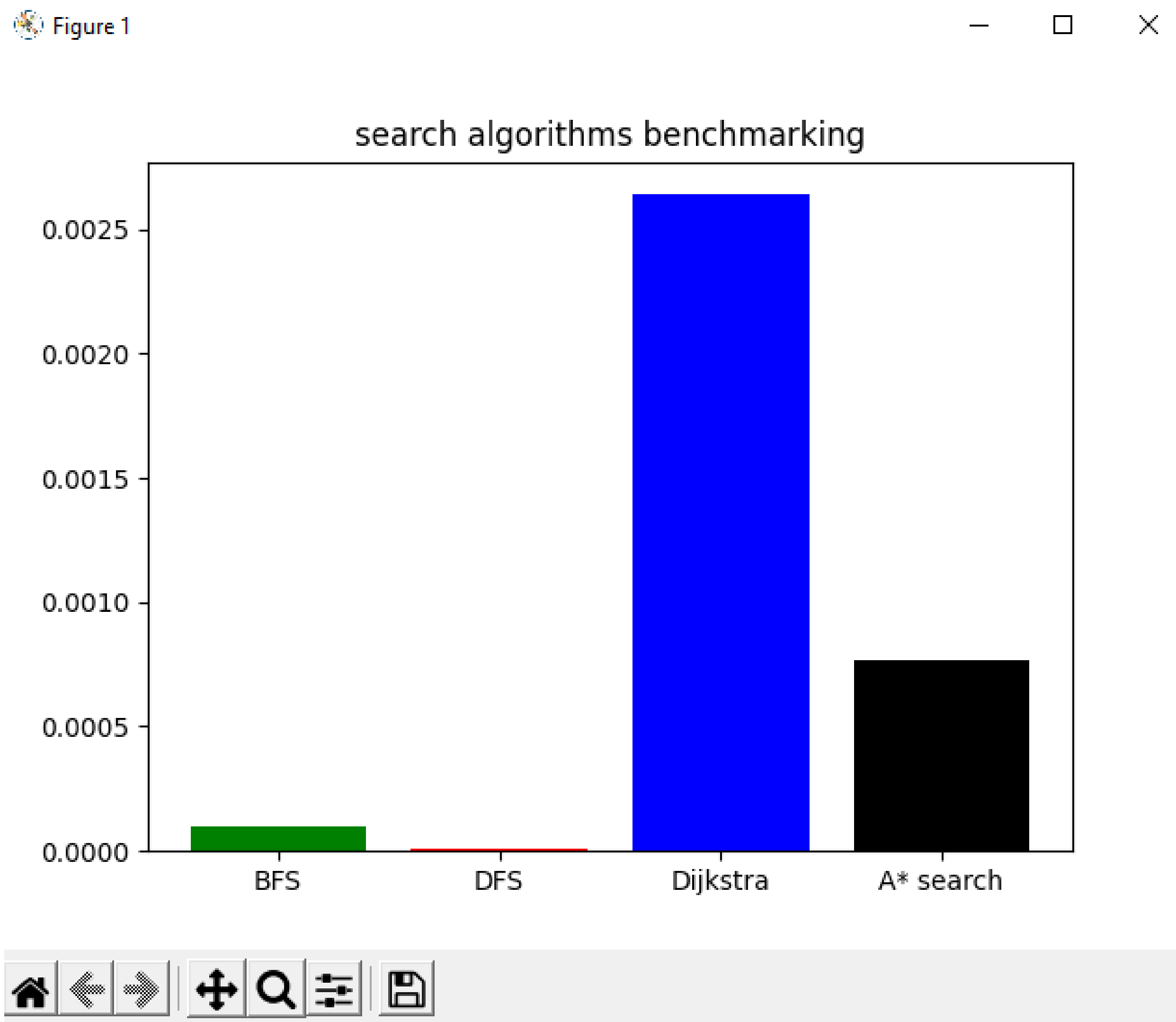


Average solution length

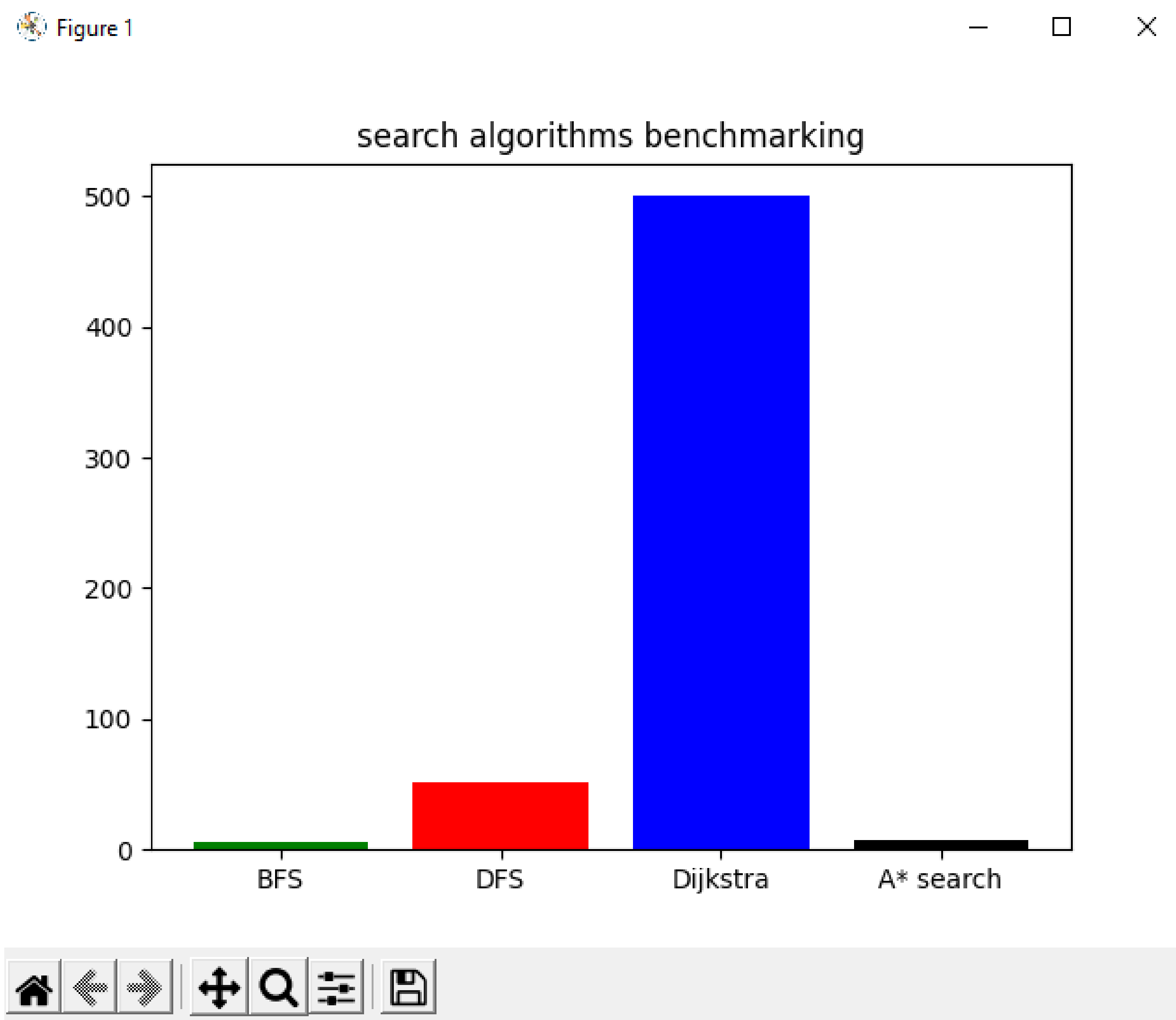


The following graph shows a benchmark of BFS, DFS, Dijkstra's shortest path, and A* searching algorithms on 5x (100) number of nodes. This graph confirms that as number of nodes increases A* search is better in finding the shortest path faster. Of Course with the limitations of it's being greediness.

Average time



Average solution length



4. Group Work

4. Use your A* search & Dijkstra's algorithms with your graph library to calculate Degree, Closeness, and Betweenness centralities on the graph from Question 2. Compare your A* and Dijkstra's Algorithm results. By looking at the graph drawing in the textbook, explain the results.

Degree Centrality

- **The degree** centrality of a node refers to the summation of the weight of edges attached to the node. In order to know the standardized score, we need to divide each score by the total sum of the edge weights. The following code shows how we calculate Degree centrality.

```
def degree_centrality(self):
        total_weight = self.total_weight_calc()
        degree_cent = {}
        for node in self.g.verticies:
            single_wight = 0
            for edge in self.g.verticies[node].edge_list:
                single_wight += int(self.g.edges[(node,edge.name)].weight)
            CD = single_wight/(total_weight)
            degree_cent[node] = CD
        return degree_cent
```

Closeness Centrality

- To calculate the **Closeness** centrality we need to calculate the inverted score after we calculated the total shortest distance of every node to a node. In order to know the standardized score, we need to multiply it with the total summation of all the edge weights. We used a helper function to calculate the summation of all the edge weights of our graph. The following shows the code.

```
def total_weight_calc(self):
        total_w = 0
```

```
        for edge in self.g.edges:
            total_w += int(self.g.edges[edge].weight)
        return total_w
```

We used both Dijkstra and A* search to calculate the closeness centrality. The following are the codes to calculate closeness centrality using Dijkstra and A* search.

Closeness centrality using Dijkstra's shortest path

```
def closeness_centrality_Dj(self):
        total_weight = self.total_weight_calc()
        clos_cent = {}
        for node1 in self.g.verticies:
            shortest_paths = self.djikstra(self.g.verticies[node1])[0]
            temp = 0
            for weights in shortest_paths:
                temp+=shortest_paths[weights]
            CC = (total_weight)/temp
            clos_cent[node1] = CC
        return clos_cent
```

Closeness centrality using A* search Algorithm

```
def closeness_centrality_As(self):
        total_weight = self.total_weight_calc()
        clos_cent = {}
        for node1 in self.g.verticies:
            temp = 0
            for node2 in self.g.verticies:
                if node1 != node2:
                    shortest_paths =
self.Astarsearch(self.g.verticies[node1], self.g.verticies[node2])[0]
                    temp += shortest_paths[node2]
            CC = (total_weight)/temp
            clos_cent[node1] = CC
        return clos_cent
```

Betweenness Centrality

- To calculate **betweenness** centrality, we take every pair of the network and count how many times a node can interrupt the shortest paths between the two nodes of the pair. For standardization, We can divide it by the total number of connections. The following are python codes to calculate betweenness centrality.

Betweenness centrality using Dijkstra's Shortest path

```
def betweenness_centrality_Dj(self):
        total_connections = 380
        bet_cent = {}
        for start in self.g.verticies:
            temp = 0
            for node1 in self.g.verticies:
                if node1 != start:
                    prev_nodes = self.djikstra(self.g.verticies[node1])[1]
                    for node2 in prev_nodes:
                        if node2 != start:
                            temp_n = node2
                            while temp_n != node1:
                                if prev_nodes[temp_n] == start or temp_n
== start:
                                    temp+=1
                                    break
                                temp_n = prev_nodes[temp_n]
            bet_cent[start] = temp/total_connections
        return bet_cent
```

Betweenness centrality using A* search algorithm

```
def betweenness_centrality_As(self):
        total_connections = 380
        bet_cent = {}
        for start in self.g.verticies:
            temp = 0
            for node1 in self.g.verticies:
                for node2 in self.g.verticies:
                    if node1 != node2 and node1!=start and node2!=start:
                        shortest =
self.Astarsearch(self.g.verticies[node1], self.g.verticies[node2])[1]
                        if start in shortest:
                            temp+=1
            bet_cent[start] = temp/total_connections
        return bet_cent
```

centrality comparisons

The following image shows the different centralities and we will compare them by looking at the graph given in the book.

Dijkstra VS A* search for Closeness centrality

The following table is a comparison between results of closeness centrality using Djikstra and A* search.

Figure 1

City names	Closeness using Dj	Closeness using A*	difference
Neamt	0.4323900740095777	0.43118867760701574	0.0012013964025619495
Iasi	0.5006553079947575	0.4990453220781831	0.0016099859165744634
Vaslui	0.5879010299514621	0.5856822738530487	0.002218756098413377
Urziceni	0.7688496671311349	0.7650593128947774	0.0037903542363575404
Hirsova	0.6186620156970225	0.6162054845514332	0.0024565311455893024
Eforie	0.5186422976501306	0.516914749661705	0.0017275479884255596
Bucharest	0.8593182211455269	0.854586129753915	0.004732091391611903
Giurgiu	0.6711717799702662	0.6682815233481362	0.0028902566221300496
Fagaras	0.7369045852500371	0.7369045852500371	0.0
Sibiu	0.7901352426412093	0.7861326579072344	0.004002584733974857
Pitesti	0.8962281176682909	0.8962281176682909	0.0
Craiova	0.7680173213733374	0.7531088868668486	0.014908434506488821
Rimnicu_Vilcea	0.8611062944338478	0.8554694229112834	0.005636871522564424
Oradea	0.6096243555119076	0.607238933724627	0.0023854217872806283
Arad	0.666935267257588	0.664081305161808	0.0028539620957800382
Zerind	0.5743696507055286	0.5685825509503092	0.005787099755219405
Timisoara	0.5672835275302719	0.5652173913043478	0.002066136225924109
Lugoj	0.5668302705170642	0.5668302705170642	0.0
Mehadia	0.5979530403371462	0.5979530403371462	0.0
Drobeta	0.6464462379588649	0.6445165476963011	0.0019296902625638435

Dijkstra VS A* search for Betweenness centrality

The following table shows the comparison between results of Betweenness centrality using Dijkstra and A* search.

Figure 1

City names	Betweenness using Dj	Betweenness using A*	difference
Neamt	0.0	0.0	0.0
Iasi	0.09473684210526316	0.09473684210526316	0.0
Vaslui	0.17894736842105263	0.17894736842105263	0.0
Urziceni	0.4	0.4	0.0
Hirsova	0.09473684210526316	0.09473684210526316	0.0
Eforie	0.0	0.0	0.0
Bucharest	0.47368421052631576	0.47368421052631576	0.0
Giurgiu	0.0	0.0	0.0
Fagaras	0.0	0.034210526315789476	0.034210526315789476
Sibiu	0.28421052631578947	0.28157894736842104	0.0026315789473684292
Pitesti	0.42105263157894735	0.3868421052631579	0.03421052631578947
Craiova	0.18421052631578946	0.17894736842105263	0.005263157894736831
Rimnicu_Vilcea	0.29473684210526313	0.25526315789473686	0.03947368421052627
Oradea	0.0	0.021052631578947368	0.021052631578947368
Arad	0.17894736842105263	0.1631578947368421	0.01578947368421052
Zerind	0.021052631578947368	0.02368421052631579	0.0026315789473684223
Timisoara	0.05263157894736842	0.060526315789473685	0.007894736842105267
Lugoj	0.042105263157894736	0.04736842105263158	0.005263157894736845
Mehadia	0.08421052631578947	0.0868421052631579	0.0026315789473684292
Drobeta	0.13157894736842105	0.13157894736842105	0.0

Centrality comparison summary

The following image shows a Summary of Centrality Comparisons.

Figure 1

City names	Degree Centrality	Closeness using Dj	Closeness using A*	Betweenness using Dj	Betweenness using A*
Neamt	0.01751913008457511	0.4323900740095777	0.43118867760701574	0.0	0.0
Iasi	0.036045106725735	0.5006553079947575	0.4990453220781831	0.09473684210526316	0.09473684210526316
Vaslui	0.04712041884816754	0.5879010299514621	0.5856822738530487	0.17894736842105263	0.17894736842105263
Urziceni	0.06544502617801047	0.7688496671311349	0.7650593128947774	0.4	0.4
Hirsova	0.03705195328231978	0.6186620156970225	0.6162054845514332	0.09473684210526316	0.09473684210526316
Eforie	0.017317760773258157	0.5186422976501306	0.516914749661705	0.0	0.0
Bucharest	0.09806685461135722	0.8593182211455269	0.854586129753915	0.47368421052631576	0.47368421052631576
Giurgiu	0.018123238018525976	0.6711717799702662	0.6682815233481362	0.0	0.0
Fagaras	0.06242448650825614	0.7369045852500371	0.7369045852500371	0.0	0.034210526315789476
Sibiu	0.09464357631896898	0.7901352426412093	0.7861326579072344	0.28421052631578947	0.28157894736842104
Pitesti	0.06766008860249698	0.8962281176682909	0.8962281176682909	0.42105263157894735	0.3868421052631579
Craiova	0.08135320177204994	0.7680173213733374	0.7531088868668486	0.18421052631578946	0.17894736842105263
Rimnicu_Vilcea	0.06504228755537655	0.8611062944338478	0.8554694229112834	0.29473684210526313	0.25526315789473686
Oradea	0.04470398711236408	0.6096243555119076	0.607238933724627	0.0	0.021052631578947368
Arad	0.06705598066854611	0.666935267257388	0.664081305161808	0.17894736842105263	0.1631578947368421
Zerind	0.029399919452275474	0.5743698507055286	0.5685825509503092	0.021052631578947368	0.02368421052631579
Timisoara	0.04611357229158276	0.5672835275302719	0.5652173913043478	0.05263157894736842	0.060526315789473685
Lugoj	0.03644784534836891	0.5668302705170642	0.5668302705170642	0.042105263157894736	0.04736842105263158
Mehadia	0.02919855014095852	0.5979530403371462	0.5979530403371462	0.08421052631578947	0.0868421052631579
Drobeta	0.03926701570680628	0.6464462379588649	0.6445165476963011	0.13157894736842105	0.13157894736842105