Assignment 1: SEARCH Report

Group Members & Work Done:

Ryan Sippy, Nathan Grilliot, Derek Corniello

All team members contributed an equal measure

Honor Statement:

In completing this assignment, all team members have followed the honor pledge specified by the instructor for this course

Bibliography:

None

Implementation Details:

For all of our algorithms, we chose to use Python. First, we will cover how we implemented the Romanian road map. We chose to implement this as a custom class "Map" in python. This class contains a dictionary named "roads" where each city is a key with value of another dictionary that contains all the cities that it connects to and their respective costs. We also created several member functions that will be used throughout the other algorithms. This was implemented in the module "CityMatrix.py" and is imported into each of the algorithm files.

Next, we will discuss our implementations of Depth-First Search and Breadth-First Search because both of these implementations are very similar. They take in an input of a start city and an end city and determine a possible path between them if one exists. In our implementation, Depth-First Search uses a LIFO stack and Breadth-First search uses a FIFO queue. For Depth-First Search, the connections of the current city that have not been visited already are added to the top of the stack, whereas, in Breadth-First Search, they are added to the back of the queue. If there are multiple cities that have not been explored, we choose to add them to the stack or queue in order of least cost, so that it would be easier for us to calculate the expected path for our testing. If the current city is not the goal destination, the implementation will take the next city in the stack or queue, set it to be the current city, mark that city as visited, and then repeat this process over again until it reaches the destination.

For our implementations of the Greedy and A* algorithms, we used two heuristics. For scenarios where the city or destination is Bucharest, we used the given straight-line distance to Bucharest as the heuristic. If the current city or destination is not Bucharest, we had to use a different heuristic. We chose to attempt to approximate the distance between the city and the goal destination by plugging in the straight-line distance to Bucharest for the city and goal into the

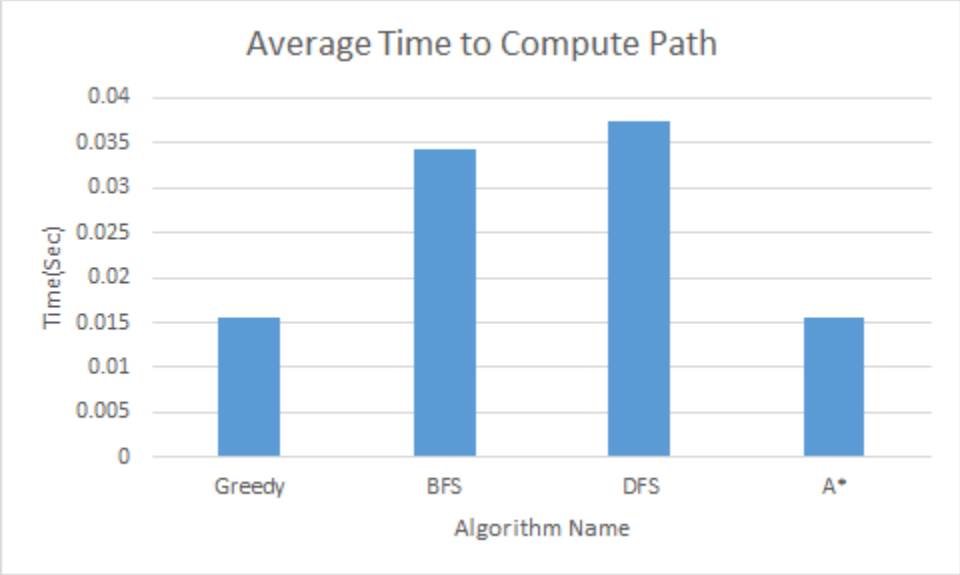
Euclidian distance formula and the Law of Cosines, assuming an angle of sixty degrees between the vectors. We must assume an angle of sixty degrees as we do not have an exact measurement of this, and it ensures our heuristic is admissible. Once we have these heuristics, we use a priority queue based on the heuristic value for each connection to the current city to get the next city to visit. For the Greedy algorithm, that is all we do to determine the next city. However, for the A* algorithm, we must factor in the previous cost to get to that city before we insert the cities into the priority queue. Using the priority queue gives us the desired path between the source city and the goal city.

Experimental Results:

We chose to evaluate our algorithm implementations on a variety of different criteria including performance speed, completeness, optimality, time complexity, and space complexity. A table outlining these criteria **for our implementations** can be found below (we will discuss the differences below), where b is the branching factor, d is the depth, ε is the relative error of the heuristic, and m is the maximum depth in the optimal solution:

	Avg. Speed	Complete?	Optimal?	Time Complexity	Space Complexity
BFS	0.034352s	Yes	No	O(b^d)	O(b^d)
DFS	0.037384s	Yes	No	O(b^m)	O(bm)
Greedy	0.015508s	Yes	No	O(b^m)	O(b^m)
A*	0.015468s	Yes	Yes	O(b^εd)	O(b^d)

First, we will discuss the performance speed of each implementation. To evaluate the performance speed, we created the “Compare.py” module that is used to return the time it takes for each algorithm to find a path. We used this module to compute the average time to return a path for each algorithm, given that we run a cartesian product of each city (every city mapped to every other city) one-hundred times. The Greedy Algorithm took an average of 0.015508 seconds. The BFS Algorithm took an average of 0.034352 seconds. The DFS Algorithm took an average of 0.037384 seconds. The A* Algorithm took an average of 0.015468 seconds. These are shown in the chart below:



As you can see, the BFS and DFS both took more than double the time to find a path than the Greedy and A* algorithms. This is interesting because those two algorithms are also the two Informed algorithms that make use of a heuristic. The use of this heuristic is directing these two algorithms towards the goal destination, whereas BFS and DFS have no direction and could be moving in the opposite direction of the goal destination during the process of finding a path.

Next, we will take a look at each algorithm individually and address its completeness, optimality, time complexity and space complexity. The first algorithm we will look at is BFS. BFS is complete if b is finite, but it is not optimal. BFS also has a time and space complexity of O(b^d) because it must expand each node until it reaches the goal and must keep each node in memory.

The second algorithm to discuss is DFS. Unlike BFS, DFS in general is not complete because it can get caught in loops. However, because in our implementation we cannot visit a city that is already visited, our implantation does not get stuck in loops our implementation is complete in finite space. However, DFS is like BFS in the fact that DFS also does not produce an optimal path. The time complexity of DFS is in O(b^m) and the space complexity is O(bm).

The third algorithm we implemented was the Greedy algorithm. Similar to DFS, greedy algorithms can get stuck in loops, however, because we implemented repeated-state checking, our implementation is complete. Because the greedy algorithm only looks at the cost of the heuristic for the next step, it is not optimal as there could be a cheaper overall path if you look at the previous costs. The greedy algorithm also has a time and space complexity in O(b^m), but the use of a good heuristic can reduce the time complexity. This is why we see the greedy algorithm perform better in the speed test above, even if it is technically in the same time complexity as BFS and DFS.

The final algorithm to look at is the A* algorithm. To start off, the A* algorithm is complete in finite space. The A* algorithm also provides an optimal solution because it will only expand nodes up to the cost of the optimal path at which point it will have found the optimal path. The time complexity is in O(b^εd) where ε is the relative error of the heuristic with respect to the actual cost. This means that using a good heuristic will reduce the value of ε and reduce the time complexity. This reduction is why we see the A* algorithm perform better than the BFS and DFS algorithms above. The A* algorithm also has a space complexity in O(b^d) because it, like BFS, must keep all the nodes in memory until it finds the path.

Execution Instructions:

To find the path using each of the four algorithms, run "Compare.py" and input your two cities to see the path and cost of each algorithm (an empty path and –1 value indicates there was no path, which should be handled in the output). To see the various execution times for each algorithm run 100 times on each possible path, run "Diagnostics.py". For specifics about each algorithm, run any of the files labeled with the algorithm name to check our premade test paths or input one of your own. This behavior can be controlled by changing the DEBUG variable in "CityMatrix.py". The DEBUG variable will have a verbose output and show the outputs of our test cases, while DEBUG as false will ask you for two city inputs and show you the path it finds with the cost.