**Project 2**

Jeremy Boyd

[Computer Science and Engineering]

Speed School of Engineering

University of Louisville, USA

[jrboyd03@louisville.edu](mailto:jrboyd03@louisville.edu)

1. **Introduction** (What did you do in this project and why?)

For this project I was responsible for solving the Travelling Salesperson Problem which is a problem programmers have been working on since forever. It involves a salesman being given a list of cities to travel to and their coordinates the problem is to find the most efficient route. But for this project not all cities were connected and the salesperson only needs to find the best path to get to a target city, not visit all cities.

The approach I took to solve this problem was the use of the Breadth First Search (BFS) and the Depth First Search (DFS) algorithms. These algorithms are not ideal to solve the TSP problem but they can both be manipulated in a way that gets you your desired solution in a rather efficient manner. For DFS, this approach is able to solve the problem because I created an algorithm that takes in the starting city and the destination city. And with a graph that has how all the cities are connected using DFS the algorithm generates all possible routes to get from the first city to the destination city. Then calculates which one has the shortest total distance and returns that value and the route. Because DFS isn’t specifically for finding the shortest path but it is helpful to find all the pathways within a graph. I was able to use that to generate what would be the shortest path using DFS.

For BFS, this approach is able to solve the problem using the Dijkstra version of the algorithm which respects edge cost. This algorithm finds the shortest distance to each point in the graph and so conceptually the proccess of exploring the graph is conceptually the same as BFS but with Dijkstra I’m able to use edge weights that are not all equal.

1. **Approach** (Describe algorithm you are using for this project)

The programming language I used for this project was Python. This project was difficult because BFS and DFS are not perfect for solving the TSP problem and it took some time to figure out the best way to use them to get the intended result. To start I’m going talk about how I created the DFS algorithm I used for this project:

I started by importing the math, collections, pandas, and numpy libraries. Then I brought in everything I’d be using from my previous project including my ‘distance’ function to calculate the distance between two points. Also the table function that would read in the data from the tsp file and store it in a table, a list that held all the x and y coordinates stored in the table, and a list called ‘distances’ that would recursively go through the list of coordinates and find the distance between each set of points. Then in conjunction with an asarray, all the values that are stored in the ‘distances’ list are imported into the array along with their indexes so that the asarray functions as a sort of look up table.

Next, I created a graph to hardcode in the connected edges to use in my function. Then I created my DFS function that would take in a graph, a starting point, and an ending point. I then created a stack data structure that would iteratively find each possible path from the starting city to the ending city. The function generates all the alternative paths and appends them into the stack using a for loop that looks at the next vertex in the graph subtracting the paths that have already been generated and if that vertex is the destination city it ends that path. This proccess continues until the end of the stack has been reached.

Then the paths generated from the function are stored in a list but the numbers in the paths are stored as strings instead of integers. So, I put the list of paths through a set of nested for loops that go through the numbers in each path and converts them from strings to integers. After this I incorporated a set of nested for loops that I used in my last project that will cycle through each path and go through the numbers in each path. By looking up the distance between each edge in the path using the asarray and adding the returned results in a variable. The total distance of each path is stored in a list and I created a variable that returned the smallest value in the list and another variable that uses the list of paths and the corresponding index in the list of distances that matches the smallest value in order to find the route that goes with the shortest distance. And printed those both on the screen.

Next I will talk about how I created the BFS algorithm I used for this project:

I imported all the same libraries but I also included the heapq library and brought in the same function, table, lists, and asarray from my previous project. Next, I created a graph to hardcode in the connected edges to use in my function and included next to each connected edge the distance between them. Then I created my BFS function which took in a graph the starting point and ending point. I then created a dictionary that took in all the vertexes and set all their distances to infinity and set the starting vertex distance to 0. I then created a heap that held the starting vertex and its initial distance of 0. Then in a while loop that would continue as long as there was something in the heap. I popped the distance and vertex into variables respectively and then used an if statement that would compare the current distance popped out of the heap to the distance of the vertex already in the dictionary. This is so vertex’s that have already been processed only go through if there’s a possibility of getting a shorter distance to the vertex. Then there’s a for loop that looks through the neighbors for each vertex and compares their distances and stores the shortest one in the dictionary. After this the shortest distance is printed to the screen.

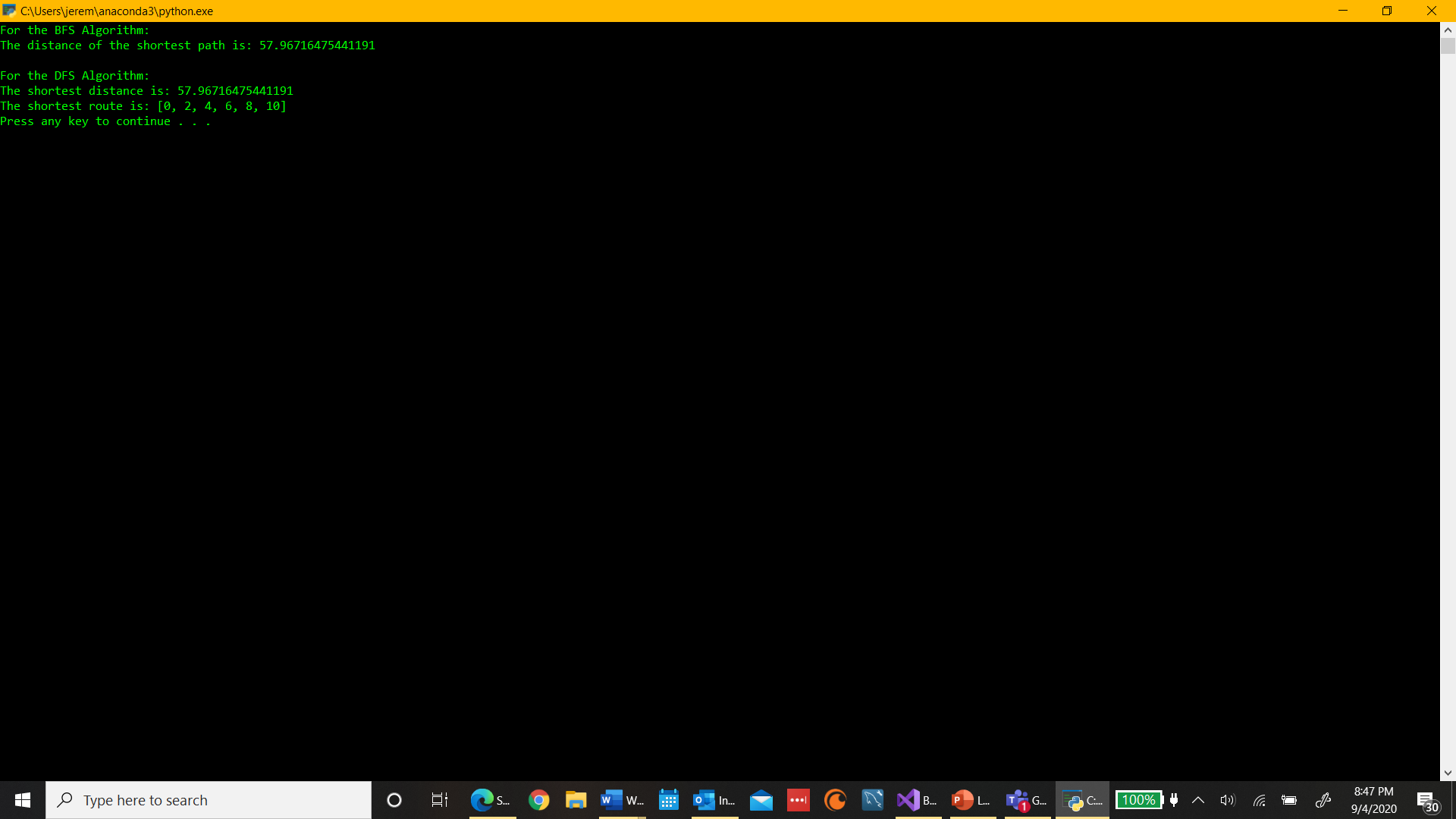
1. **Results** (How well did the algorithm perform?)

Both algorithms performed well for the dataset of 11 cities especially compared to the brute force approach. Which took around 13 minutes to find the shortest path with 11 cities. While both of these algorithms got me results in seconds.

* 1. **Data** (Describe the data you used.)

I just used the given data in the tsp file we were given, which included 11 numbered cities and their respective x and y coordinates.

* 1. **Results** (Numerical results and any figures or tables.)

- 

1. **Discussion** (Talk about the results you got and answer any specific questions mentioned in the assignment.)

I was able to get the shortest distance along with its corresponding route for the Depth First Search but for the Breadth First Search I was only able to get the shortest distance. I wasn’t able to come up with a way to find the path associated with it. But the distances were the same for both algorithms and they both performed at the same rate (tested them individually as well as in the same program) only taking around 4 seconds for both.

For my BFS function its time complexity is O(V+ ElogE), when you take into consideration the dictionary takes O(V) time. There are at most O(E) iterations of the while loop and the for loop iterates over outgoing edges, so among all iterations of the while loop, the body of the for loop executes at O(E) times. And lastly each priority queue operation is O(log E) the final runtime sums up to O(V+ ElogE).

For the DFS function I’m not as sure of its time complexity but I believe it’s O((V +E)!). Because stack operations are done in O(1) time but the function creates multiple permutations to generate each route. It works fast for smaller graphs but a graph with millions of pathways to get from the start to the destination wouldn’t work well with this function.

The computes I’m using is a LENOVO YOGA 720-12IKB with a Intel Core i7 CPU with a clock rate of 2.80 GHz, it has a 64-bit operating system, and I’m using Visual Studios as my IDE to run my code.

1. **References** (If you used any sources in addition to lectures please include them here.)