**Project 2: TSP – Search with BFS and DFS**

Stone Barrett

Computer Science Engineering

Speed School of Engineering

University of Louisville, USA

[scbarr04@louisville.edu](mailto:scbarr04@louisville.edu)

1. **Introduction**

In this project, the goal was to continue the idea established in the previous project. Previously, iterative permutations were used to brute force the solution to a Hamiltonian graph traveling salesperson problem. This time, the goal is to find the solution to a traveling salesperson problem that lies on a weighted, directed, graph using the search algorithms breadth-first search and depth-first search, or their derivatives. This time, a starting city is declared and an ending city is declared. The many cities in between the source and destination are chained together by one-way streets – the distances of which are found using a set of geographical coordinates located in a provided text file – and this data must be used to implement the aforementioned algorithms in such a way to find the shortest possible route.

1. **Approach**

This project required the use of two separate search algorithms that are used to search through a weighted, directed graph. The first algorithm is an implementation of breadth-first search, traditionally meaning that if ran on a search tree that grows downward, each layer of horizontal nodes would be searched first. In other words, the breadth of the tree would be searched before the depth.

Ex:

Figure 2.1

In this example, the nodes would be searched in numerically ascending order the way they are labeled. One derivative idea from breadth-first search algorithm is Dijkstra’s algorithm, which compares all lengths of paths between one layer of the tree and the next, and always takes the shortest one.

Ex:

1 3

Figure 2.2

5 2 4 6

In this example, the path chosen by Dijkstra’s algorithm would be 1, 2, 5 because 1 is the shortest path from layer 1 to layer 2 and 2 is the shortest path from layer 2 to layer 3. This is a version of breadth-first search that successfully returns the shortest path in a weighted, directed graph, like the one provided with the assignment. For this project, I implemented this algorithm in Python code using a class to create a graph instance and functions within said class to create edges within the graph. These edges require a source and destination parameter, as well as the weight, or length of the edge. After all distances on a given layer of nodes has been stacked and compared, the shortest distance is found and used to continue on to the next layer. This is done until the distances are summed at the end of the route to return the shortest possible distance from City 1 to City 11.

The second search algorithm required to be implemented in this project is depth-first search algorithm. This accomplishes the same goal as DFS and Dijkstra’s but through a perpendicular method. Depth-first search traverses each individual path possible from the starting nodes to the leaves on a tree.

Ex:

Figure 2.3

In this example, the nodes would be visited again in numerically ascending order, but they are ordered differently than before.

If the weights were the same as in Figure 2.2, then the shortest route would still be the same. I also used Python code to implement this search algorithm on the traveling salesperson problem by creating a visual representation of the graph using an adjacency matrix and Smartdraw.

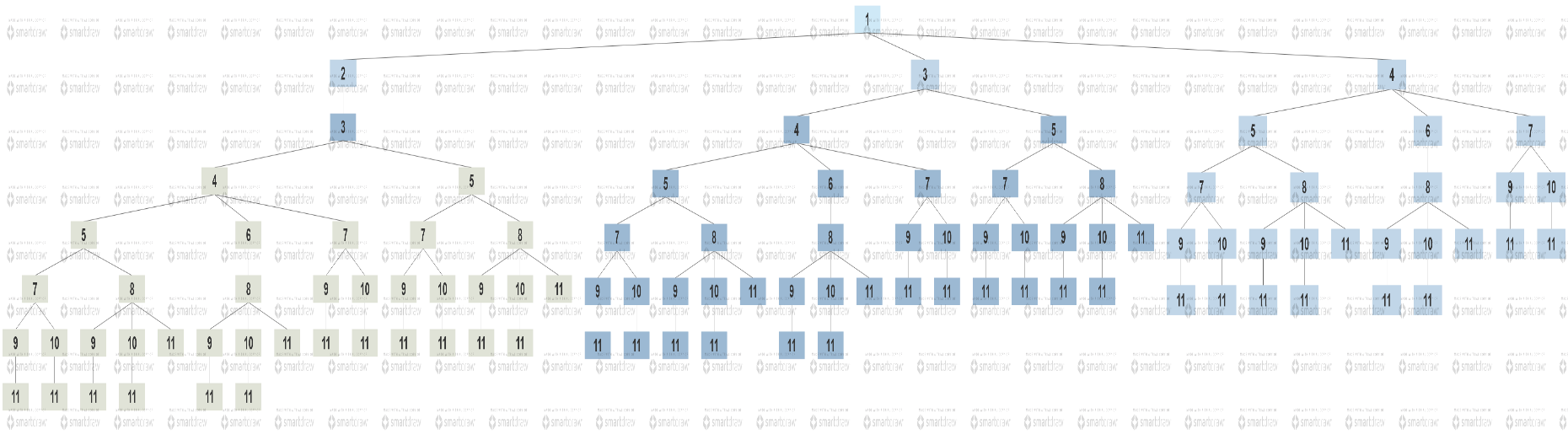
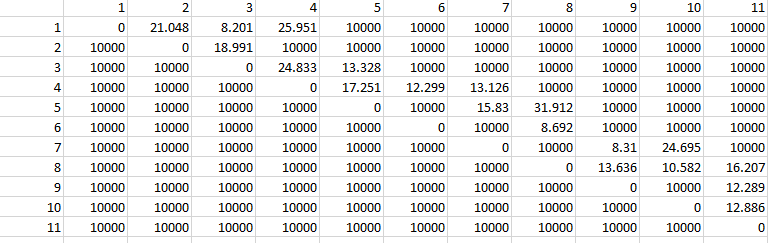
Figure 2.4

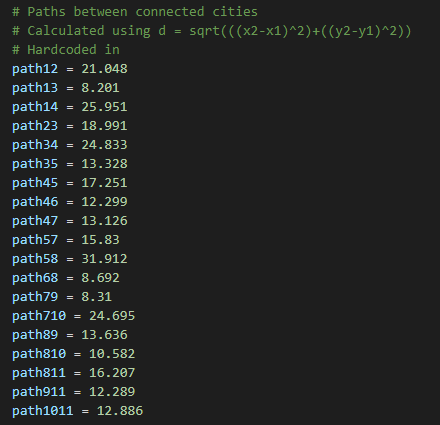
Figure 2.5

(Figure 2.4 Note: 10000 is used as a placeholder for “infinity” since comparing any of the actual distances would result in less than; there is no path from City X to City Y when 10000 is shown.)

The implementation in this problem would map out every single possible path from the head node to the leaves, storing all the distances in a list and finding the minimum sum of distances in the end.

1. **Results**
   1. **Data**

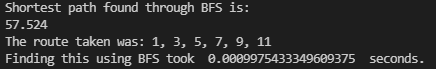
The data in question was the prompt assigned in class (which helped create Figure 2.4) and the .tsp text file containing a list of coordinate pairs that represent cities to be visited. These coordinates and list of possible paths from one city to the next were used to find all distances between connected cities. Using the distance formula, the connections between cities was calculated as such:

Figure 3.1

* 1. **Results**

The results of the breadth-first search implementation were fast and accurate. The following was returned when program was executed:

Figure 3.2



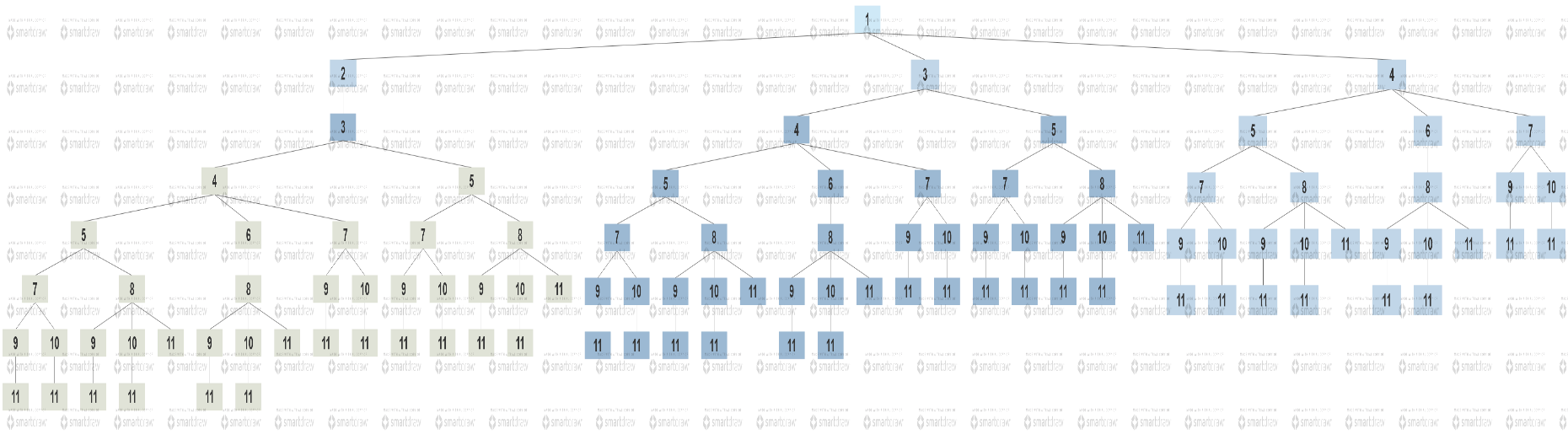
The results of the depth-first search algorithm implementation were just as fast but not quite as accurate. The following was returned when program was executed:

Figure 3.3



Obviously, assuming BFS is correct and found a path that was only about 58 in length, something almost double that length could not be correct. Due to time constraints, this issue was unable to be located and resolved before submission time.

Assuming BFS is correct, the shortest path from City 1 to City 11 is highlighted below.

Figure 3.4

1. **Discussion**

It seems that both algorithms work at roughly the same speed when it comes to data sets of the size of the one provided. I imagine larger data sets would quickly ramp up the computational times and allow me to discover which implementation would truly work faster. However, it is evident from this project that these search algorithms work much more quickly than a widespread brute force set of calculations. Implementations can be further optimized, but even what I’ve done here is significantly faster.

1. **References**

*Flowchart maker org Chart MAKER drawing Program floor Plan Creator*. SmartDraw. (n.d.). Retrieved September 13, 2021, from https://www.smartdraw.com/.