AI Project 2

Eric Dockery
Computer Engineering and Computer Science
Speed School of Engineering
University of Louisville, USA
eadock01@louisville.edu

Introduction:

For this assignment, we are asked to solve the traveling salesman problem using Depth First and Breadth First Search algorithms. This task asked us to generate all possible paths of a set of cities to travel, trace a Hamiltonian path through a directed graph, and to find the minimum cost solution to a traveling salesperson problem. This assignment is to show the implementation of data structures efficiency compared to the brute force permutation method from the last project.

Approach:

The data sets given to test our code had an extra 6 lines that needed to be removed and stripped. My solution prompted the user for the tsp file then striped the file and sorted the information that was needed for the solution by storing that data in an array (list in python.) Once the data was sorted the program prompts the user to select the either Breadth First Search or Depth First Search.

If Breadth First Search is selected the program runs the function BreadthFirstSearch (BFS) which takes in the hashmap, start, and end variables. Note that this is hard coded data that is given from the program details. The BFS program takes the first variable and makes a queue of the possible paths that it can travel. While there is still a queue to travel the program finds the first object that can be traversed to and for each value after that it looks for the end city. If it doesn't find the end city then it adds the next value to the queue always pulling from the first possible path. This generates a list of possible city traversals that is then stored in a python list (array.) The first value of this list is the shortest Vertex Path, but not the shortest traveled path otherwise known as the distance.

If Depth First Search is selected the program runs the function DepthFirstSearch (DFS) which takes in the hashmap, start, and end variables. The

DFS program takes the first variable and makes a queue of the possible paths that it can travel. While there is still a queue to travel the program finds the last or deepest object that can be traversed to and for each value after that it looks for the end city. If it doesn't find the end city then it adds the next value to the queue always pulling from the first possible path. This generates a list of possible city traversals that is then stored in a python list. This may generate the shortest distance but not the shortest path.

After the possible paths are stored in a list the program takes the possible city's and stores the coordinates for each city in the paths. This is accomplished with a nested for loop that is three deep comparing each possible value in the list and changing it to the correct value. For each of the paths the distance is calculated and if the value of the distance is lower than the hard coded 1 trillion distance value, I hard coded due to being the most efficient way to calculate, then the minimum cost and path is stored. At the end of the program the minimum cost and the minimum path is displayed and the program plots the final path as well as the minimum cost in a GUI.

3. Results:

3.1 Data:

The data that was used for this assignment was generated using Concorde. The format for the data was:

NAME: concorde11

TYPE: TSP

COMMENT: Generated by CCutil_writetsplib

COMMENT: Write called for by Concorde GUI

DIMENSION: 11

EDGE_WEIGHT_TYPE: EUC_2D

NODE_COORD_SECTION

1 5.681818 63.860370

2 11.850649 83.983573

3 13.798701 65.092402

4 16.883117 40.451745

5 23.782468 56.262834

6 25.000000 31.211499

7 29.951299 41.683778

8 31.331169 25.256674

9 37.175325 37.577002

10 39.935065 19.096509

11 46.834416 29.979466

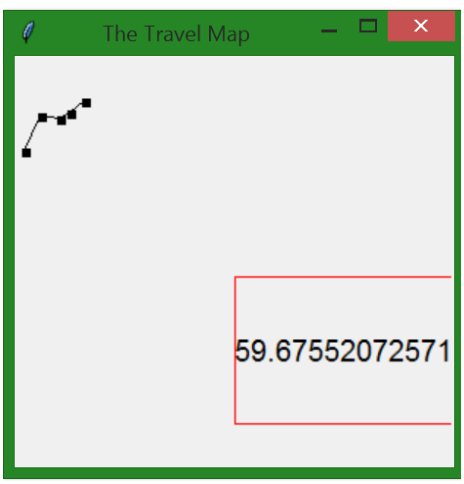
With hard coded vertexes:

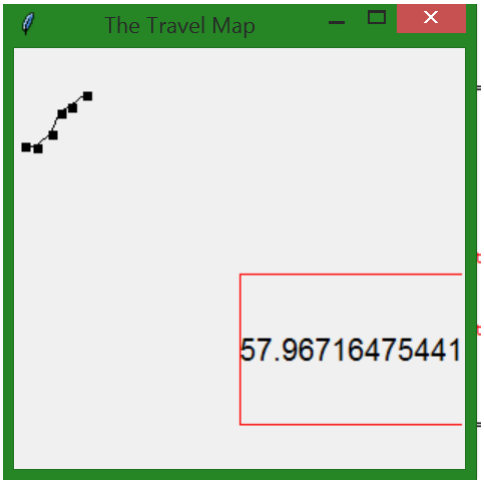
Table 1: Cities connected by a one way path of Euclidian distance (left = from, top = to).

pt	1	2	3	4	5	6	7	8	9	10	11
1		x	x	x							
2			x								
3				x	x						
4					x	x	x				
5							x	x			
6								x			
7									x	x	
8									x	x	x
9											x
10											x

3.2 Results:

```
>>>
Enter B for BreadthFirstSearch or D for DepthFirstSearch: B
Calculating the distance
The minimum cost is:
59.675520725710555
The minimum path is:
[1, 4, 7, 9, 11]
Duration: 0:00:01.156565
>>> ================================ RESTART ================================
>>>
Enter B for BreadthFirstSearch or D for DepthFirstSearch: D
Calculating the distance
The minimum cost is:
57.96716475441191
The minimum path is:
[1, 3, 5, 7, 9, 11]
Duration: 0:00:01.219432
>>>
```





4. Discussion:

These results ran very effectively in comparison to the first project of brute force programming the permutation of the paths. Breadth First Search Runs at Duration: .093691 seconds Depth First Search Runs at Duration: 0.078371 seconds. Making Depth First Search much more efficient than Breadth First Search. Note Duration on screenshot is adding the time it takes to put the B or D value in the program.

5. References:

https://en.wikipedia.org/wiki/Depth-first_search

https://en.wikipedia.org/wiki/Breadth-first_search