**Traveling Salesman Problem: Brute Force**

Jacob Taylor Cassady

Computer Engineering & Computer Science

Speed School of Engineering

University of Louisville, USA

[jtcass01@louisville.edu](mailto:NameID@louisville.edu)

1. **Introduction**

The Traveling Salesman Problem (TSP) is a well-known non-deterministic polynomial-time hard problem that has been studied within mathematics since the 1930s. The "salesman” is given a list of cities with their locations and is asked the shortest route to travel to each city once and then return to the starting point. A program was developed using Python 3.7 and accompanying 3rd party libraries: NumPy, Pandas, and matplotlib to determine the shortest path.

1. **Approach**

Two approaches were taken for solving the TSP. Both approaches were based on search algorithms and include breadth first search (BFS) and depth first search (DFS). Throughout this document cities will be referred to as “vertices” and the route between the vertices as “edges.”

* 1. **Breadth-First Search**

The breadth-first search algorithm focuses on visiting all edge-neighbors of a current vertex before visiting the edge-neighbors of its edge-neighbor. The implementation presented in this paper uses a dictionary of layer keys to properly iterate over the possibilities.

At the start, the starting vertex is represented as its enumeration id. This id is used to retrieve the current vertex from the graph. A route is initialized with this starting vertex id and a while loop begins looping over every vertex in a current layer, while generating the next layer, before incrementing the current layer and repeating this process. For instance, the source vertex (part of layer 0) uses its adjacent vertices to develop the list under the dictionary’s “layer 1” key.

This method requires every vertex to be visited from every possible path. This provides the opportunity for ensuring the minimum path distance to each vertex. Whenever a vertex is to be added, it is checked to see if it’s vertex representation already exists within the breadth-first search tree. If it does, then the route distances are compared between the two vertices and the vertex with the minimum route distance is kept.

At the end, the generated bfs\_tree is iterated over to find the vertex of interest. It’s minimum\_route is returned at this time. This is an object containing the order of vertices visited as well as the distance traveled. For more information, please reference the upper level function from source code shown in **Figure 2** of the Appendix.

* 1. **Depth-First Search**

The depth-first search algorithm focuses on traveling as deep as possible within a tree before visiting other adjacent vertices from a given starting vertex. The implementation presented in this paper uses a recursive function to continually dive deeper into a tree until there are no remaining adjacent vertices.

At the start, the starting vertex is represented as its enumeration id. This id is used to retrieve the source vertex from the graph. The source vertex is then plugged into the “search\_deeper” recursive function which takes a current vertex and a current route.

The search\_deeper recursive function starts by getting a list of unfinished adjacent vertices to the current vertex. It updates the route to include the current vertex and pushes it on top of a stack. If there are remaining adjacent vertices, it continues to search deeper from the first remaining adjacent vertex. If there aren’t, the vertex is popped off the stack and the vertex is finished. If there still vertices on the stack, it searches deeper from the vertex on top. If not, the program finishes.

The distance traveled to each vertex is kept within the route. At the end, all nodes include the distance traveled to them at the time of being marked finish. For more information, please reference the upper level function from source code shown in **Figure 3** of the Appendix.

1. **Results**

Both the breadth-first search and depth-first search algorithms were correctly implemented. They visited the cities in the expected order although, the breadth-first search was the only algorithm that was correctly augmented to produce the shortest path to the target city. It is also important to note breadth-first search did run much slower than depth-first search.

* 1. **Data**

The algorithms were tested using a single dataset. Within the datafile, cities are enumerated, and x and y coordinates are provided. The input data was formatted like the example shown in **Figure 1** below.

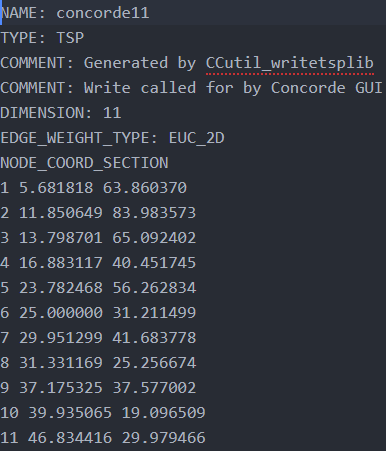


Figure : 11PointDFSBFS.tsp Input Data

Additionally, an adjacency matrix was provided to define which cities are connected and in which fashion. Please reference **Table 1** to see the adjacency matrix utilized.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **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 |

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

* 1. **Results**

The results from processing 11PointDFSBFS.tsp using the previously described algorithms can be found in **Table 2**.

|  |  |  |  |
| --- | --- | --- | --- |
| **algorithm** | **runtime** | **minimum\_route** | **minimum\_distance** |
| Breadth-first search | 0.2894378 | 1, 3, 5, 7, 9, 11 | 57.96716475 |
| Depth-first search | 0.0229423 | 1, 2, 3, 4, 5, 7, 9, 11 | 118.5519187 |

Table : Algorithm Performances

The table indicates breadth-first search performed over 10 times slower than depth-first search. But only breadth-first search produced the correct minimum route and minimum distance. Graphical representations of both the input and output figures for the algorithms can be found in the appendix of this paper, **Figures 4**, **5**,and **6**.

1. **Discussion**

Given the two implementations presented in this paper, the depth-first search algorithm is much faster at finding the target city; although, it did not produce the correct minimum route solution. This is because additional work needs to be done to improve upon the current optimal route as routes are finished. One avenue would be to focus on the vertices with edges to a finished vertex and determining which of these is optimal to precede the current finished vertex.

The breadth-first search algorithm was sufficiently augmented to produce the route of minimum distance. Additional work could be done to ensure it finishes without having to analyze the entire tree. If the target vertex had an id of 3, for instance, the algorithm would still finish the entire tree before returning the shortest path to 3.

1. **References**

Wikipedia, Traveling Salesman Problem - <https://en.wikipedia.org/wiki/Travelling_salesman_problem#History>

NumPy Documentation - <https://docs.scipy.org/doc/>

Pandas Documentation - <https://pandas.pydata.org/pandas-docs/stable/>

Matplotlib Documentation - <https://matplotlib.org/3.1.1/contents.html>

1. **Appendix**

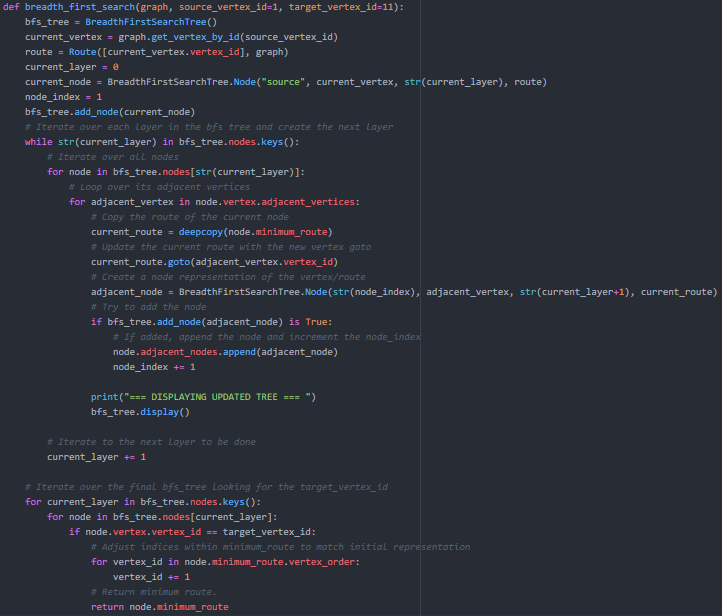
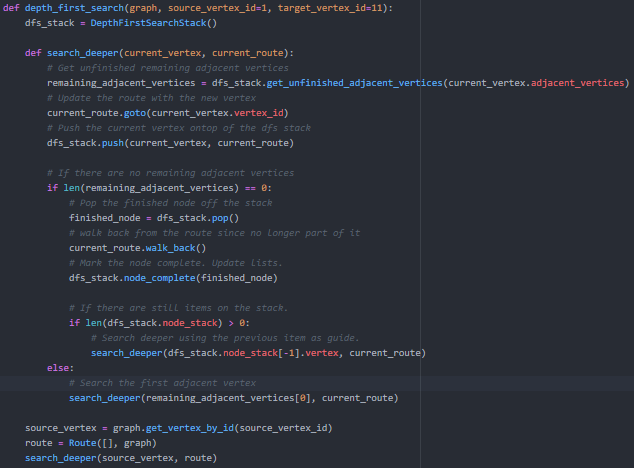


Figure : Breadth-First Search Algorithm

Figure : Depth-First Search AlgorithmA close up of a map

Description automatically generatedFigure : 11PointDFSBFS.tsp Input

A close up of a map

Description automatically generated

Figure : 11PointDFSBFS.tsp BFS Output

A close up of a map

Description automatically generated

Figure : 11PointDFSBFS.tsp DFS Output