Traveling Salesman Problem – Search with BFS and DFS

Karthik Malyala
Computer Science & Engineering
Speed School of Engineering
University of Louisville, USA
ksmaly01@louisville.edu

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

In this project, the Traveling Salesman Problem (TSP), a famous non-deterministic polynomial-complete (NP-Complete) problem, was explored using both Breadth-First Search and Depth-First Search approaches to find the shortest and most cost-efficient route for a salesman to be able to visit a target city from the starting city without having to visit every city and returning back to the starting city. These approaches provided as supplements to the Brute Force approach that was explored in the last project to analyze differences between the algorithm runtimes when dealing with complex TSP datasets where Brute Force would be inefficient as the number of cities increase, as seen in the last project. The above task was completed using Python 3.9 and its accompanying libraries in the PyCharms IDE using a Lenovo ThinkPad X390 Yoga with Core i7 processing and 16 GB RAM.

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

To solve this specific special case of TSP, two approaches were used: Breadth-First Search (BFS) and Depth-First Search (DFS). Both approaches are search algorithms that traverse in their respective manner and provide results. For this project's purpose, the path to be found is a path from origin city (City 1) to target city (City 11) using the coordinates from the given dataset (11PointDFSBFS.tsp). There can be many outputs, but this report showcases two of the possible outputs using modified algorithms for each approach.

2.1 Breadth-First Search (BFS)

This search algorithm focuses on visiting every node (city) in a level-by-level format. It causes a ripple effect as the nodes that you visit in every iteration slowly drift off from the origin node. In a binary tree format, this algorithm will first start with the root, then visit all the immediate children, and then the nodes belonging to each child. It utilizes the Queue data structure and follows a First in First Out (FIFO) traversal to cause the ripple effect which is very useful if the target city is very close to the origin city. For the BFS approach, there were two implementations/interpretation in this project. The first focuses on reaching the target city in the least number of nodes. The other modifies the approach to just output the cheapest route, the one with the least distance (cost), out of all possible paths to the target.

To find the least number of nodes, the BFS function takes in the euclidianPath matrix, the origin city, and the target city. Then, it loads the origin city onto a queue where it is then popped to visit its adjacent nodes. For every node, the current path and the queue are appended, and it then checks if the current node its dealing with is the target city. If so, then we calculate the distance and keep track of the shortest distance and best route by far. This implementation returns the first-found shortest path, rather than the cheapest.

To find the cheapest route, the CheapestBFS function takes in the euclidianPath tree, the origin city, and the target city. For this approach, everything remained the same as the least nodes implementation except for how the queue and solution were structured. The queue in this implementation contained tuples of the nodes in queue and the accompanying path it took to get to that node. In this modification, there isn't a visitedNodes list because we focus on backtracking algorithm paradigm to come up with the pathToGetThere and the target city. Once a node equaling the target city is found, the path is then appended onto a solution and the shortest distance and best route are tracked for every path involving the target city node for a final cheapest path output.

2.2 Depth-First Search (DFS)

This search algorithm focuses on going as deep as possible with regards to the levels of the nodes before moving onto the adjacent nodes from a root node. This approach is analogous to a corn maze where you try and find a route's dead end and return back to the original location to go another route. It uses a route-based technique with a Stack data structure and follows Last in First Out (LIFO) traversal to cause the maze effect which is very useful for finding paths to targets far from the root node. Unlike BFS, DFS explores its next-level neighbors before visiting its adjacent neighbors by using the backtracking algorithm paradigm heavily. For this DFS approach, there were also two implementations/interpretations in the project. The first focuses on going as deep as possible until it hits the target city in as many nodes and any distance that it takes while the second modifies that approach to output the cheapest path to the target city, 11.

To go as deep as possible into the levels, the DFS function just took in a stack, with the origin city (1), and used global variables for the euclidianPaths matrix and the visited nodes. The stack and visitedNodes were predefined in the main. It uses the LIFO traversal to pop and push the stack as needed. If a current node has already been visited, it pops it off the stack, in a LIFO manner. If not, then it appends it to the stack and visitedNodes to then calculate if it has met the target city. If no, then it recursively loops itself within the function to eventually find a deep path that does so. Similar to the BFS function, this DFS function also returns the first-found deep path to the target city.

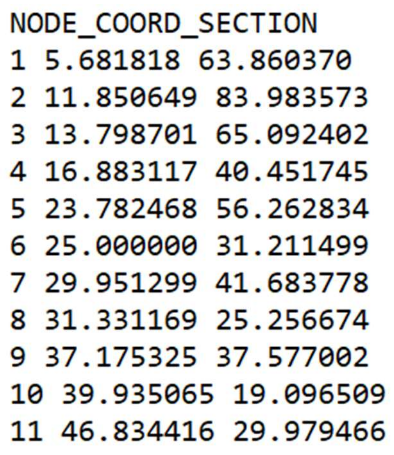
To find the cheapest route, the CheapestDFS function takes in the euclidianPath matrix, startCity, and an endCity. This implementation is very similar to the CheapestBFS function except for how the stack is being utilized. Given that the stack is LIFO, the last added path is being iterated first so that we go depth-first rather than breadth-first. Unlike the CheapestBFS function that utilizes FIFO, the CheapestDFS function validates the depth-first searching in visiting as deep as possible and returning back to the root node by using a stack and LIFO.

3. Results

Both approaches were significantly better than the Brute Force approach that was used in the last project, in terms of algorithm runtime. The DFS approach was slightly faster than the BFS approach for both interpretations, as seen in the runtime specifications below.

3.1 Data

The data being inputted was provided in the form of a .tsp file which had included a list of cities and their corresponding x and y coordinates. More specifically, there was only one dataset being used that included all coordinates for 11 different cities. Each file was split into a specification part and a data part. With the specification parts containing descriptive information of the datasets, the data part contained all the vital information like Node Coordinates under NODE_COORD_SECTION, as seen below (11 Cities):



```
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
```

To determine the connections between cities in the network, an Adjacency Matrix was provided showing us the Euclidian paths from one city to another city. This matrix was hardcoded into a dictionary showing origin cities and their paths to the respective cities to then apply both BSF and DSF searches to calculate the best path to the target city.

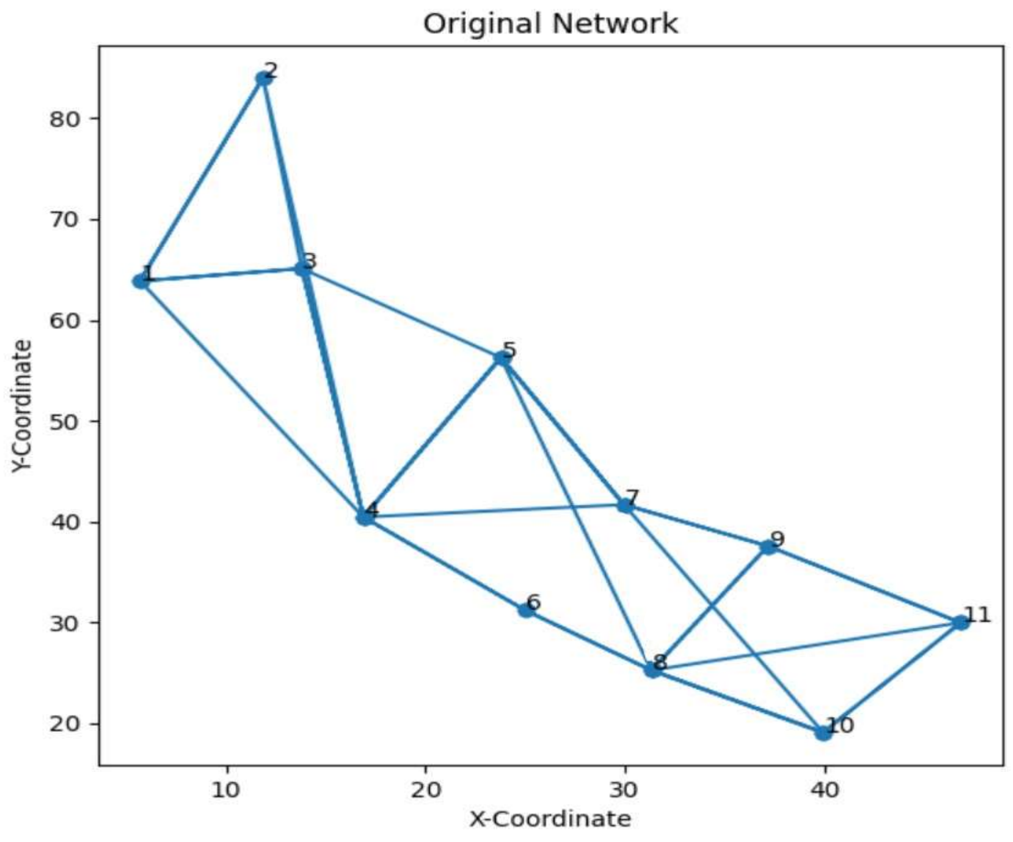
Table 1: Cities connected by a one-way path of Euclidian distance (left = from, top = to).

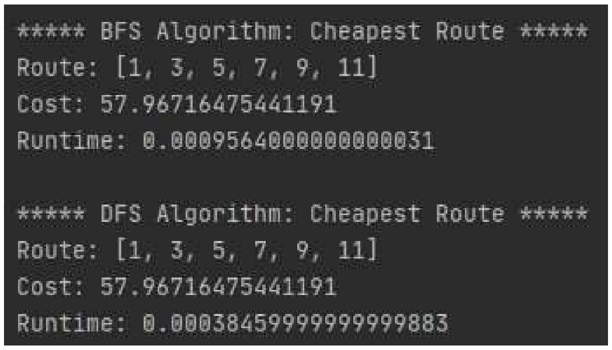
pt	1	2	3	4	5	6	7	8	9	10	11
1		x	x	x							
2			x								
3				x	x						
4					x	x	x				
5							x	x			
6								x			
7									x	x	
8									x	x	x
9											x
10											x

3.2 Results

Approach (Cheapest)	Optimal Route	Cost	Runtime
Breadth-First Search	1 -> 3 -> 5 -> 7 -> 9 -> 11	57.96716475	0.0009564
Depth-First Search	1 -> 3 -> 5 -> 7 -> 9 -> 11	57.96716475	0.0003846

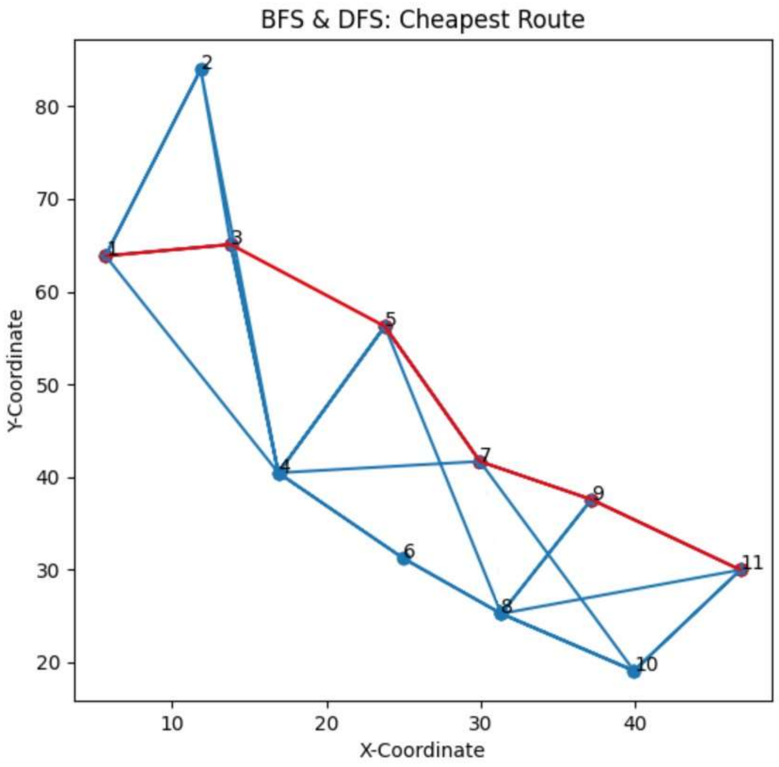
Approach (Quickest)	First Acceptable Route	Cost	Runtime
Breadth-First Search	1 -> 4 -> 7 -> 9 -> 11	59.6755207	0.0001771
Depth-First Search	1 -> 2 -> 3 -> 4 -> 5 -> 7 -> 9 -> 11	118.5519187	1.08E-05

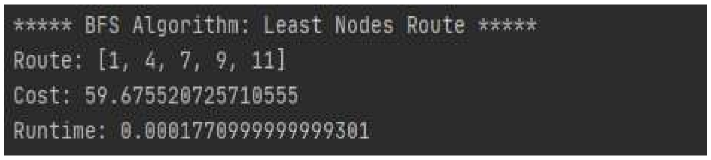




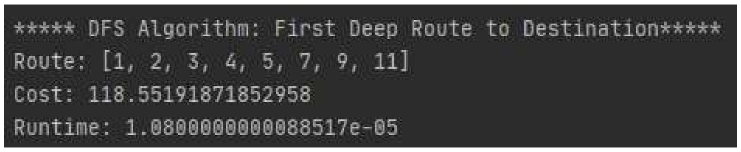
```
***** BFS Algorithm: Cheapest Route *****
Route: [1, 3, 5, 7, 9, 11]
Cost: 57.96716475441191
Runtime: 0.0009564000000000031

***** DFS Algorithm: Cheapest Route *****
Route: [1, 3, 5, 7, 9, 11]
Cost: 57.96716475441191
Runtime: 0.0003845999999999883
```

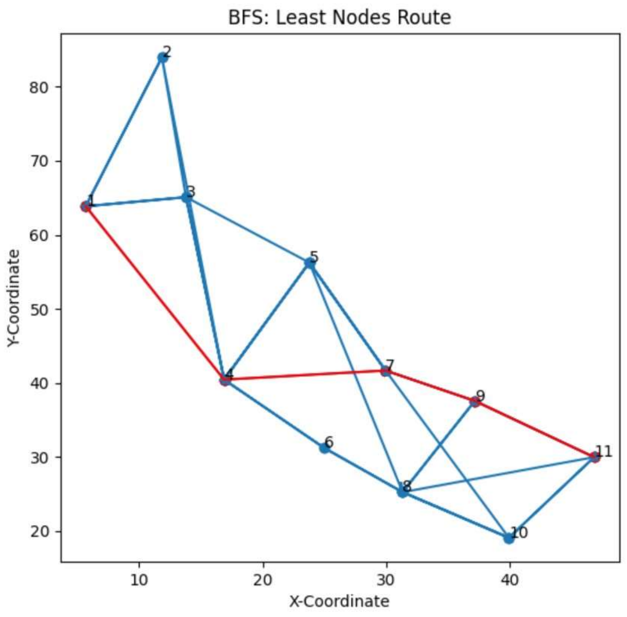


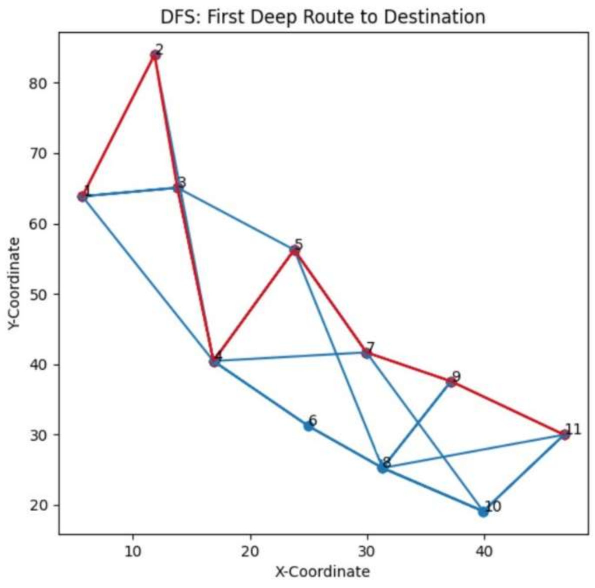


```
***** BFS Algorithm: Least Nodes Route *****
Route: [1, 4, 7, 9, 11]
Cost: 59.675520725710555
Runtime: 0.0001770999999999301
```



```
***** DFS Algorithm: First Deep Route to Destination*****
Route: [1, 2, 3, 4, 5, 7, 9, 11]
Cost: 118.55191871852958
Runtime: 1.0800000000088517e-05
```





4. Discussion

As seen above, the two interpretations showed how each of the search algorithms work in terms of finding the first path to the target city, shortest (least nodes and least transitions) in BFS and first deep route in DFS, versus finding the cheapest path overall. In both implementations, DFS proved to be slightly faster than BFS due to the LIFO/ Stack structure and theory behind the algorithms. Given that the target city, 11, is very far from the origin city, 1, for this particular assignment, its is proven that DFS works much better for target nodes that are far away from the root node, as discussed above. More specifically, the DFS approach was 0.0005718 seconds faster than the BFS approach in finding the cheapest route and 0.0000166 seconds faster than the BFS approach in finding the first acceptable route.

However, DFS does have its disadvantages in that it won't always output the most optimal path nor the accurate solution. It has a major disadvantage of not being complete due to indefinite paths or depth limit, despite its memory efficiency with b * d states at depth d. On the other hand, BFS has an advantage of being complete and guaranteed to find a solution, but requires a lot of memory with b^{d-1} states with depth d. Although the time difference for both the implementations is very minimal in this case, it can significantly vary when dealing with larger datasets.

5. References

Matplotlib Documentation - https://matplotlib.org/stable/index.html
Difference Between BFS and DFS : https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/