Project 2: Traveling Salesman Problem – Search with BFS and DFS

Nick York
CES Department
Speed School of Engineering
University of Louisville, USA
neyork01@louisville.edu

1. Introduction

The Traveling Salesman Problem ("the TSP") was introduced in Project 1. A quick recap: the TSP requires finding the minimum cost route ("best route") for a salesman to visit a list of cities without visiting a city more than once, except for the starting city. The salesman must return to their starting city at the end. Each city is represented by an xy-coordinate pair and numbered starting at one. The coordinates are provided by a .tsp file ("TSP file"). The cost to travel between two cities is the Euclidean distance between them, calculated as:

$$cost = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Project 2 introduces a variation of the TSP where the goal is to find the approximate minimum cost route ("optimal route") starting at city one and ending at city eleven. Each city has direct connections to a subset of all cities. Each direct connection is one-way. The goal of this project was to write a program ("the program") to find the optimal route for a given set of cities using two search algorithms: breadth-first search ("BFS") and depth-first search ("DFS").

In terms of Project 2, BFS visits each city and its direct connections first, then the next set of direct connections, and so on until there are no more cities to visit. Each city is marked after it is visited to avoid cycles, which may break the algorithm. On the other hand, DFS visits the next available direct connection (if any) until there are no more cities to visit. Once DFS has no more cities to visit, it returns to the previous city and visits the next available direct connection (if any). Like BFS, each city is marked after it is visited. This process repeats until all possible routes have been run. Both algorithms visit the least costly city out of available direct connections.

2. Approach

I wrote a Python program that allows the user to specify which TSP file to use. Next, the program reads and parses the xy-coordinate pairs in a TSP file. To implement the search algorithms, I decided to create a Map class and City class to represent the TSP. Map assigns a list of City objects to its *cities* attribute so methods can be invoked on them to find the optimal route. Map also tracks the optimal route and cost with *route* (initially empty) and *cost* (initially set to infinity) attributes.

City has attributes to both represent a city and implement the search algorithms. The basic attributes are *coords* and *num*, which store an xy-coordinate pair and city number, respectively. The *not_visited* attribute is a Boolean value that is set to True initially and set to False once it that City is visited. The *connections* attribute takes a list of City objects the City has a direct connection to. The cost for each direction connection is calculated and assigned to *cost* in the order of *connections*. Each City has a *prev_city* attribute that tracks the city before it in the current route to reconstruct the optimal route. The *curr_cost* attribute tracks the cost of the current route based on the previous city's *curr_cost* plus the cost to visit the next city.

The xy-coordinates are passed as an argument to the *map_TSP()* function to build each City object and add the direct connections to each. Then, the City objects are used to build a Map object. A Map object is returned from *map_TSP()*. For each algorithm, the specific class method is called on the Map object: *BFS()* and *DFS()*.

BFS() takes the starting city as an argument and enqueues it to *to_visit*, which is a Python List object used like a queue data structure to store the next city or cities to visit. While the queue is not empty, the next city to visit is dequeued and its direct connections are sorted by cost in ascending order. For each direct connection ("next city"), *not_visited* is set to False, the current city assigned as *prev_city*, and the cost to visit it from the current city calculated and added to the next city's *curr_cost*. Finally, the next city is enqueued.

Once any direct connections are enqueued, the algorithm checks whether the current city is the ending city. If it is, then the algorithm checks whether the current city's *curr_cost* is less than *Map.cost*. If it is, then *Map.cost* is set to *curr_cost* and *Map.route* is set to the optimal route reconstructed from *prev_city*. Once the queue is empty, the final values of *Map.cos*t and *Map.route* are the solution.

DFS() takes the starting city as an argument and calls a recursive function, *DFS_visit(),* with it. The starting city becomes the current city. Like *BFS()*, *DFS_visit()* sorts the current city's direct connections by cost in ascending order, marks each next city as visited, records each next city's previous city, and calculates the cost to visit each and adds it to their current cost. Unlike *BFS()*, the next step of the algorithm is to call DFS_visit() using next city. Once DFS_visit() is called and the current city has no direct connections, the algorithm checks whether the current city is the ending city. If the current city is city eleven, then the process of determining the optimal route proceeds the same as in *BFS()*. The base case is reached when the original branch is reached and no more connections can be called. *DFS_visit()* returns to *DFS()*. The final values of *Map.cost* and *Map.route* are the solution.

3. Results

3.1 Data

For this project the students were a single TSP file with eleven xy-coordinate pairs representing cities to test our program on.

3.2 Results

The results of DFS and BFS for the single TSP file are printed to the console during runtime (Fig 1). The optimal route for each algorithm is plotted and saved as a PNG file under a folder named

Results (Fig 2 and Fig 3).[1] Likewise, the results for each algorithm are also saved to a CSV file under Results, listing the algorithm used, the optimal route, cost, and runtime in seconds (Fig 4).

```
BFS Traveling Salesman Problem
Route: [1, 3, 5, 8, 11] with cost 69.656378
is better than [] with cost inf

Optiaml route for BFS: [1, 3, 5, 8, 11] with a cost of 69.656378

DFS Traveling Salesman Problem
Route: [1, 3, 5, 7, 9, 11] with cost 57.967164
is better than [] with cost inf

Optiaml route for DFS: [1, 3, 5, 7, 9, 11] with a cost of 57.967164
```

Fig 1. Output of running the program to solve all TSP files in the Files directory.

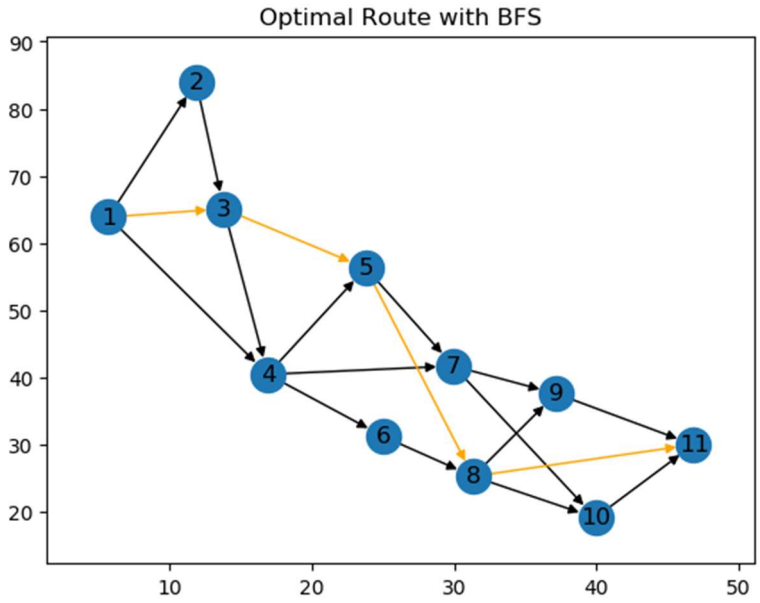

Fig 2. A digraph showing the optimal route for BFS in orange. Arrows represent direct connections.

[1] I replaced my plotting solution from Project 1 with a new one utilizing the NetworkX Python library. I use a directed graph to visualize cities as nodes and plot connections as edges. To add a title, axes, and multiple edge colors, I used two separate solutions I found on Stack Overflow (see References).

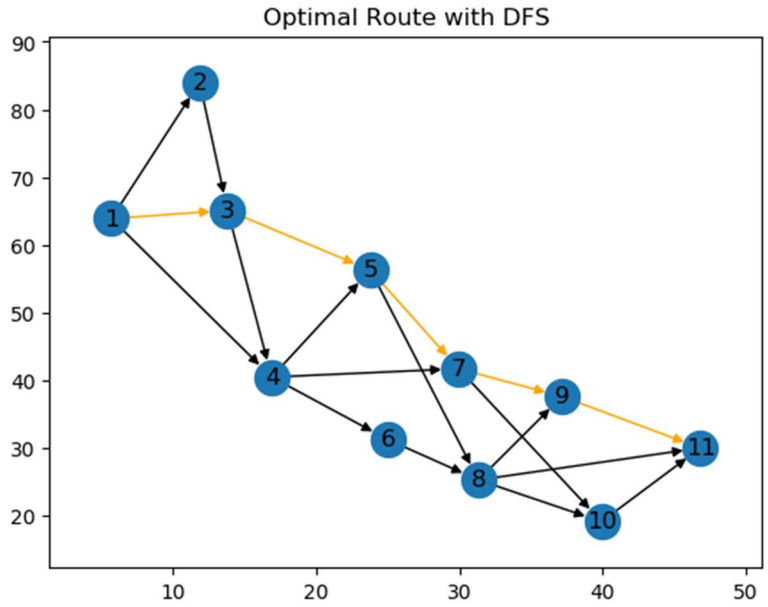

Fig 3. A digraph showing the optimal route for DFS. DFS finds a less costly route than BFS.

Algorithm	Optimal Route	Cost	Running Time (sec)
BFS	[1, 3, 5, 8, 11]	69.656378	0.001512766
DFS	[1, 3, 5, 7, 9, 11]	57.967164	0.000997066

Fig 4. Results for each TSP file provided by the professor.

Discussion

Unlike the brute force approach used in Project 1, search with BFS and DFS finds an optimal route and not the best route. BFS and DFS ran in fractions of a second for a TSP file with eleven cities, whereas brute-force took minutes to do the same. BFS took four transitions to reach the ending city for its optimal route compared to DFS which took five. However, BFS found a less costly route and ran in less time. Another benefit of BFS and DFS is that they are not susceptible to combinatorial explosion, which limited the brute-force approach to a handful of cities if the user wished for a solution in a reasonable amount of time. In short, BFS and DFS are useful heuristics for finding the optimal route for many cities quickly.

References

StackOverflow answer to "How to set colors for nodes in NetworkX?"

StackOverflow answer to "How to make x and y axes appear with networkx nx.draw()?"