**Solving TSP with Brute Force**

Michael Telahun

CSE

Speed School of Engineering

University of Louisville, USA

[m0tela01@louisville.edu](mailto:m0tela01@louisville.edu)

1. **Introduction**

In this project the goal was to solve the Traveling Salesperson problem via code without any sophisticated methods, specifically brute force was used. As a result, the intentions of this project were to show the difficulty in solving problems which can appear simple but grow exponentially are in fact not solvable without careful consideration. The simplicity of a “brute force” solution was meant to show the characteristic or perhaps nature of a baseline solution. It clearly serves are a horrible solution in many cases, specifically where exponential and factorial growth occurs. This project appears to be meant to get our feet wet and make sure everyone can program to some degree. The future projects appear to build upon this simple brute force implementation so there is a simple class with each of the functions that will be used in a future version of the project. Since it was stated that reusable code would be best, the code is about as recyclable as possible given such few constraints. An additional class for the TSP data representing the array of data that is read from the files as a class could have been created, but there was no need to at this stage and it may be overkill. Instead there are some functions so that when the time comes a new function in the future for calculating or traversing the locations can be added.

The development of this project was done in Python 3 and can be run with just few packages that are relatively common, specifically Numpy for arrays and Matplotlib for visualization.

**Running the code:**

***$ python Michael\_Telahun\_P1.py [number] --verbose***

**Example (1):** *$ python Michael\_Telahun\_P1.py 7 --verbose*

**Example (2):** *$ python Michael\_Telahun\_P1.py 8*

1. **Approach**

The algorithm for this project was rather lackluster since it was brute force. Essentially the algorithm is 3 steps:

1. Compute the permutations of all cities where any location can be the starting location, but it is also the final city as the traveler goes back to the start in the end. This takes n locations to n! permutations or O(n!).
2. For each permutation compute the Euclidean distance of the permutation from location to location and save it in an array. For each file there are n locations so O(n). Finding the Euclidean distance is just a constant operation O(1), but it must be done n! or m times. Therefore, this piece is O(m) where m is the number of permutations not number of locations.
3. Return the index of the minimum distance in the array and thereby retrieving the permutation with the shortest distance. This is just a simple search that takes O(m) time where again m is the number of permutations not number of locations.
4. **Results**

The result of this “algorithm” was bad since it was brute force. The solution is essentially eclipsed by the n! piece of the brute force approach in this problem. There is no way to find a minimum distance without first calculating all the distances within each permutation and summing the result. Overall, this solution is bad not just because it contains a factorial calculation in its runtime but because there is no real thought or design that goes into the logic. If this is a class about Artificial Intelligence this about as far ways from AI that we can get in a computer. With that said the algorithm did perform as intended. The small files ran well pretty face because there were a small number of permutations and the large files took a long time or did not complete.

The dataset provided for this was straightforward. It consisted of some meta data and the coordinates of the locations. The number of locations is specified in the meta data but that is the only information in the meta data that was used in this project.

As far as numerical results, the only meaningful results that may be of use were the runtimes for the each of the different files. With nine locations or fewer the time it took to compute the minimum cost can be measured in seconds. At ten and eleven locations it begins to take several minutes. With twelve locations the program fails to execute after hanging for several minutes.

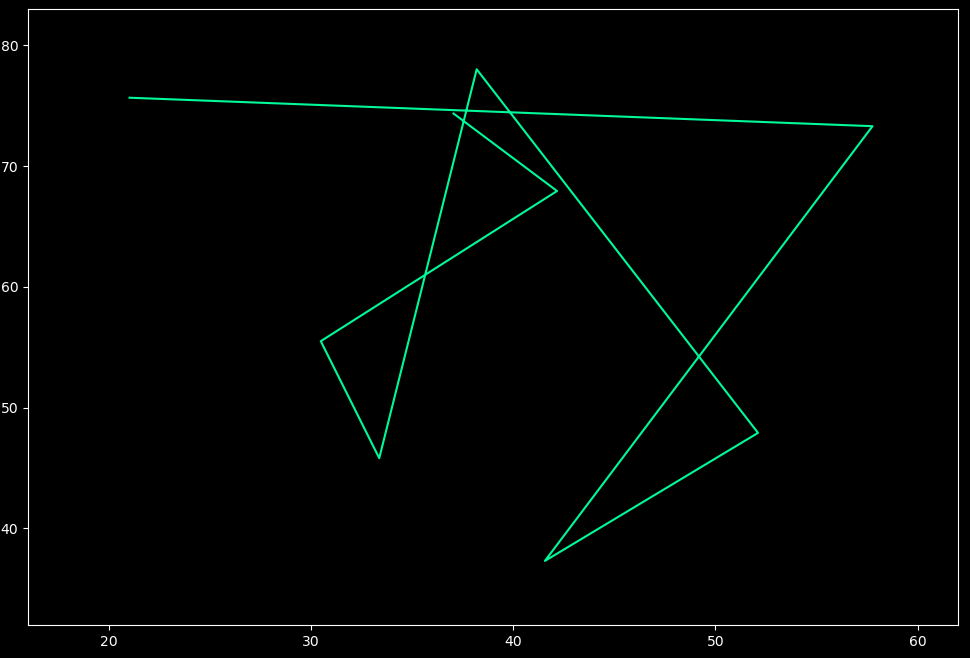


Figure1. Display of brute force traversal

1. **Discussion**

The results of this project were pretty much expected since brute force solutions are typically worst-case solutions and to be avoided if possible. Dr. Yampolskiy also stated “if you break your computer you have successfully completed this assignment,” which is what is implemented here. Running the brute force solution worked for 11 or fewer locations. The knowledge to be gained here is that for small sample sets/datasets brute force solutions are good enough. That said larger datasets can be extremely resource demanding and it is very easy to stumble into this issue.

It was an interesting introduction that begs the question in which way or with what methods should we improve the running time/complexity/space of the solution for this problem? For example, implementing Dijkstra’s algorithm would be one of the most obvious solutions. The code is commented and again can be run using the command line, see above introduction for how to run the file.

1. **References**

Aside from the lectures, no other references were used.