Traveling Salesman Problem - Brute Force

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

1. Introduction

In this project, the Traveling Salesman Problem (TSP), a famous non-deterministic polynomial-complete (NP-Complete) problem, was explored using a Brute Force approach to find the shortest and most cost-efficient route for a salesman to be able to visit a set of given cities and return to the starting point. This approach provided a very basic foundational insight into the complexities of the TSP and algorithm runtime trends based on the number of cities in a dataset, which can later be made more efficient using other methods. 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

To set up a Brute Force approach to solve the TSP, the coordinates were first read into a Python dictionary wherein each key resembles a city/node and its corresponding tuple value represent its x and y coordinates. Second, all permutations for the given number of cities were generated using the permutations function from itertools, which uses the backtracking algorithm paradigm to recursively find all possible order of cities, accompanied by the range function to build all n! possibilities. Backtracking is a technique wherein a set of solutions is built incrementally by using the depth-first search method and recursive calling.

Third, each route generated with the permutations function is then iterated through one by one wherein a second for loop comes into play in constantly calculating the distance between each of the cities, in the order of the particular permutation being iterated through, and adds to a total route distance for that specific route (permutation). When dealing with datasets with 11 and 12 cities, it was important to take each permutation (route) at a time with the first for loop rather than putting all possible routes in a list to avoid a MemoryError. Then, each of the total route distances are compared with each other and the smallest of them all is stored as the minimal cost and that current route (permutation) is stored as the best route. Since the Hamiltonian cycle rules that a path must be closed, the starting city is concatenated to the end of each permutation (path) to complete the path and illustrate it better when graphing. Finally, the results are graphed with matplotlib using its associated functions: subplots, scatter, annotate, and quiver. Once the optimized route is acquired, the graphing portion of the program connects each city with an arrow, in the order that the specific route suggests.

3. Results

Although the algorithm and approach produced optimized routes with the respective minimal costs, it resulted in highly inefficient results when dealing with datasets above 9 cities, as seen in the runtime specifications below.

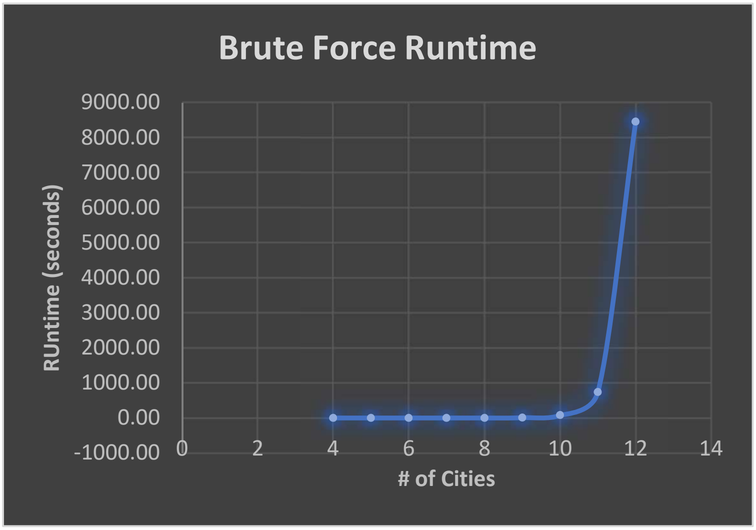
3.1 Data

The data being inputted was provided in the form of .tsp files which had included lists of cities and their corresponding x and y coordinates. More specifically, there were 9 total datasets with the number of cities ranging from 4 to 12. 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 (4 Cities):

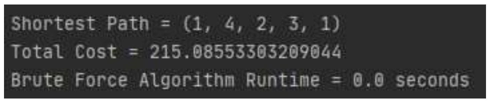
```
NODE_COORD_SECTION
1 87.951292 2.658162
2 33.466597 66.682943
3 91.778314 53.807184
4 20.526749 47.633290
```

3.2 Results

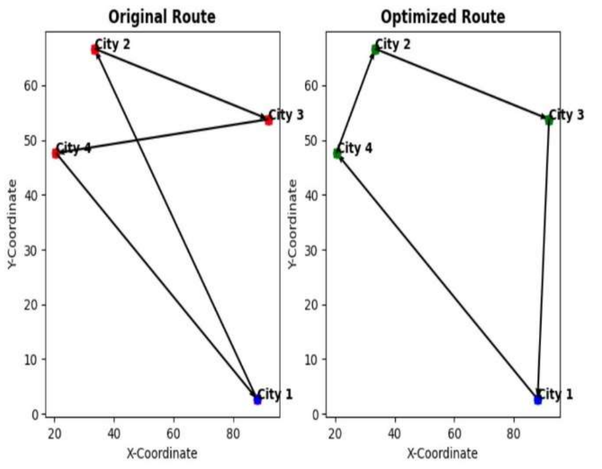
File	Number of Cities	Optimal Route	Total Cost (Distance)	Brute Force Runtime (secs)
Random4.TSP	4	1 -> 4 -> 2 -> 3 -> 1	215.085533	0.00
Random5.TSP	5	1 -> 2 -> 5 -> 3 -> 4 -> 1	139.133542	0.00196791
Random6.TSP	6	1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 1	118.968914	0.0079999
Random7.TSP	7	1 -> 2 -> 7 -> 3 -> 6 -> 5 -> 4 -> 1	63.863032	0.08100581
Random8.TSP	8	4 -> 5 -> 2 -> 3-> 7 -> 1 -> 6 -> 8 -> 4	310.9820797	0.69216776
Random9.TSP	9	6 -> 7 -> 1 -> 8 -> 4 -> 9 -> 2 -> 5 -> 3 -> 6	131.0283661	7.43036246
Random10.TSP	10	1 -> 2-> 7 -> 6 -> 8 -> 5 -> 9 -> 10 -> 4 -> 3 -> 1	106.7858202	79.32763791
Random11.TSP	11	1 -> 6 -> 10 -> 11 -> 8 -> 8 -> 9 -> 7 -> 5 -> 3 -> 4 -> 2 -> 1	252.6844345	737.418431
Random12.TSP	12	1 -> 8 -> 2 -> 3 -> 12 -> 4 -> 9 -> 5 -> 10 -> 6 -> 7 -> 11 -> 1	66.08484401	8451.450601



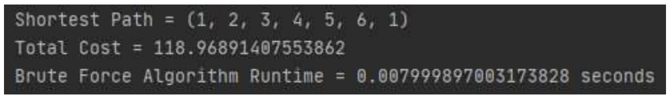
# of Cities = 4



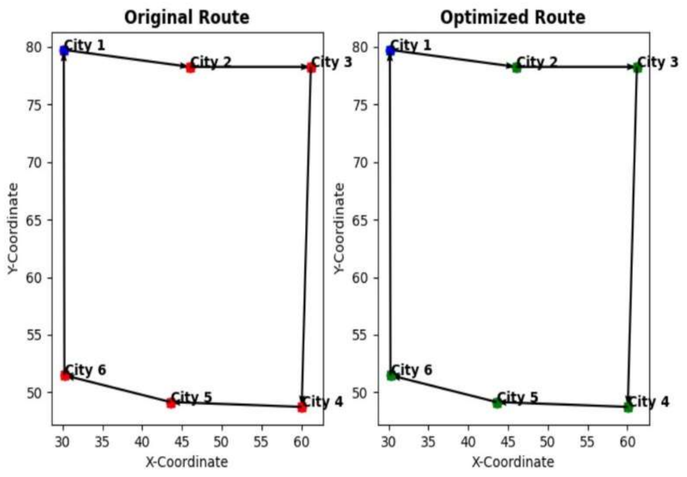
```
Shortest Path = (1, 4, 2, 3, 1)
Total Cost = 215.08553303209044
Brute Force Algorithm Runtime = 0.0 seconds
```



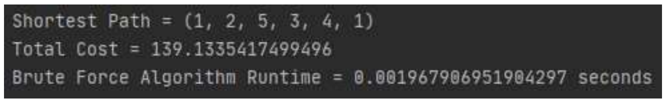
# of Cities = 6



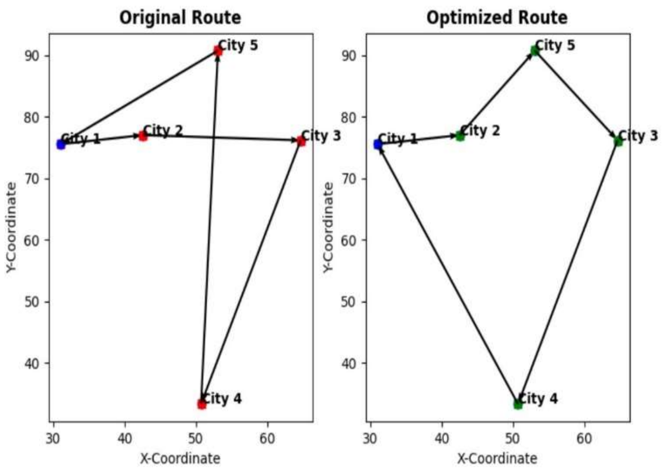
```
Shortest Path = (1, 2, 3, 4, 5, 6, 1)
Total Cost = 118.96891407553862
Brute Force Algorithm Runtime = 0.007999897003173828 seconds
```



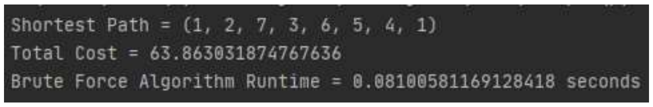
# of Cities = 5



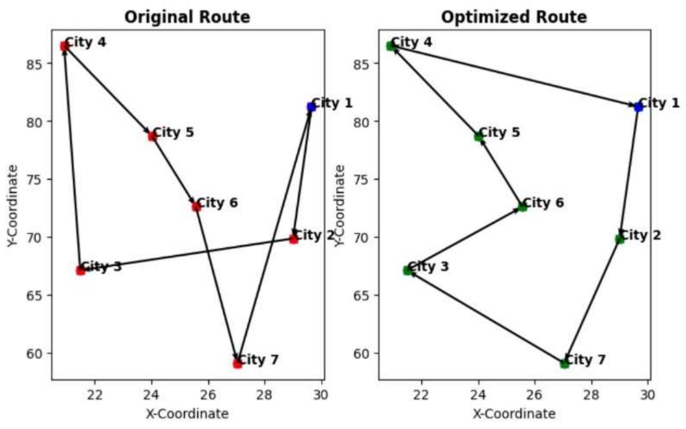
```
Shortest Path = (1, 2, 5, 3, 4, 1)
Total Cost = 139.1335417499496
Brute Force Algorithm Runtime = 0.001967906951904297 seconds
```



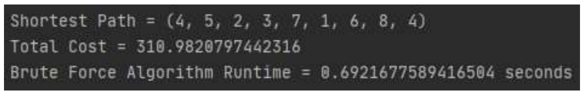
# of Cities = 7



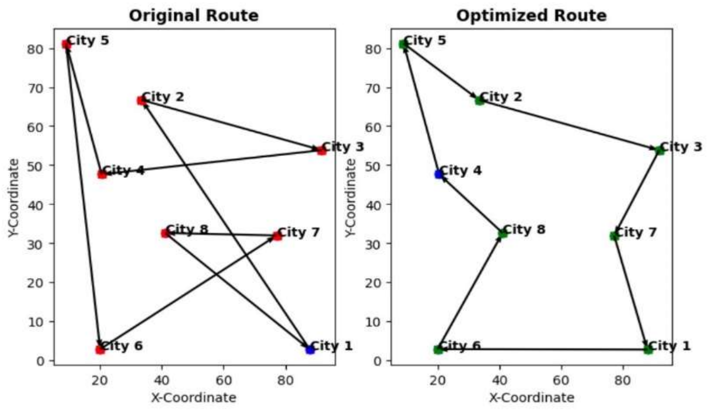
```
Shortest Path = (1, 2, 7, 3, 6, 5, 4, 1)
Total Cost = 63.863031874767636
Brute Force Algorithm Runtime = 0.08100581169128418 seconds
```



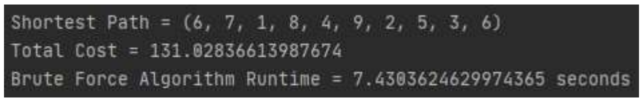
# of Cities = 8



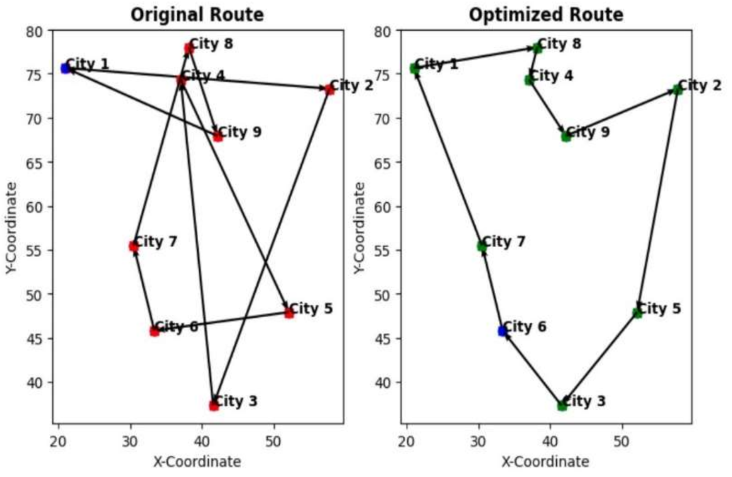
```
Shortest Path = (4, 5, 2, 3, 7, 1, 6, 8, 4)
Total Cost = 310.9820797442316
Brute Force Algorithm Runtime = 0.6921677589416504 seconds
```



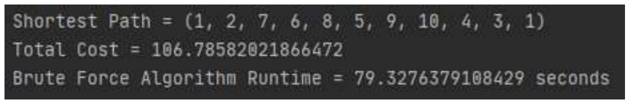
# of Cities = 9



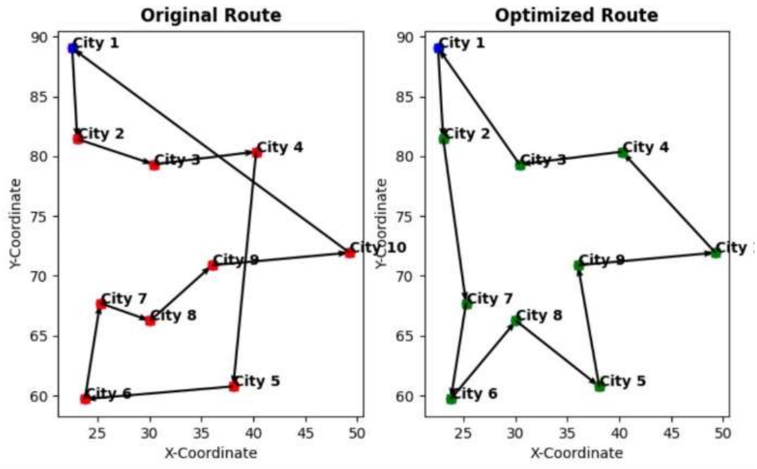
```
Shortest Path = (6, 7, 1, 8, 4, 9, 2, 5, 3, 6)
Total Cost = 131.02836613987674
Brute Force Algorithm Runtime = 7.4303624629974365 seconds
```



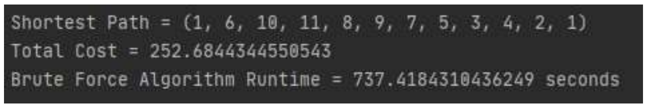
# of Cities = 10



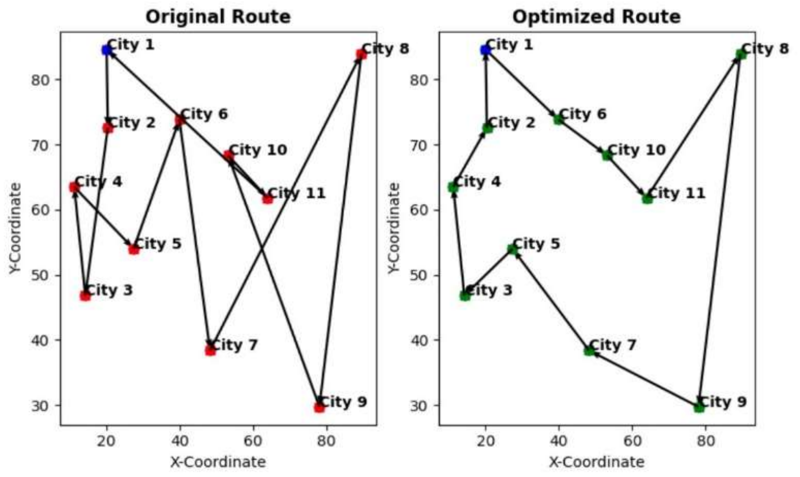
```
Shortest Path = (1, 2, 7, 6, 8, 5, 9, 10, 4, 3, 1)
Total Cost = 106.78582021866472
Brute Force Algorithm Runtime = 79.3276379108429 seconds
```



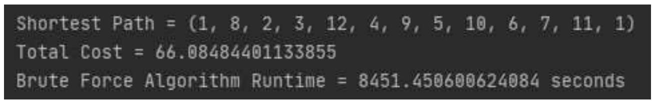
# of Cities = 11



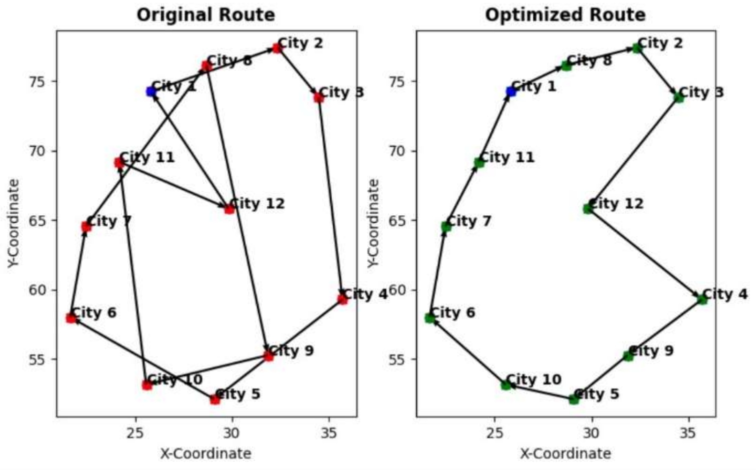
```
Shortest Path = (1, 6, 10, 11, 8, 9, 7, 5, 3, 4, 2, 1)
Total Cost = 252.6844344550543
Brute Force Algorithm Runtime = 737.4184310436249 seconds
```



# of Cities = 12

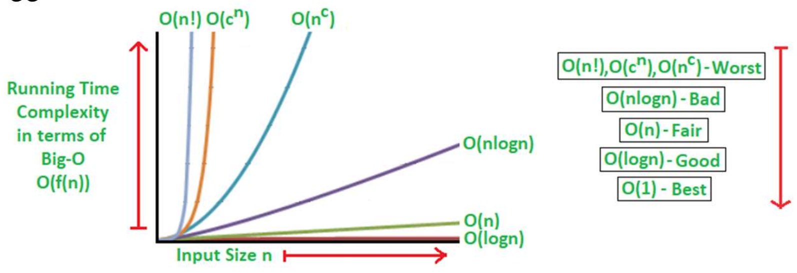


```
Shortest Path = (1, 8, 2, 3, 12, 4, 9, 5, 10, 6, 7, 11, 1)
Total Cost = 66.08484401133855
Brute Force Algorithm Runtime = 8451.450600624084 seconds
```



4. Discussion

As seen above, the algorithm gets significantly inefficient when reaching for datasets above 10 cities. With the 12-city dataset taking well over 2 hours to compute, it can be projected how the Brute Force approach is not the most viable option when dealing with large datasets. The factorial increase in runtime, as seen in the graph above, is due to the backtracking algorithm paradigm having a time complexity of O(n*n!) with n! permutations and O(n) time in dealing with each permutation at a time. Additionally, one can compare the similarity between the O(n!) factorial increase line shown below to the one from our program results to conclude that our brute force approach has the worst time complexity when dealing with bigger datasets.



The city datasets usually produced multiple optimized routes, but the program picks the first found optimized route, which explains why most routes started from City 1 (except for the datasets of 8 and 9 cities). Given that this specific program was run on a personal laptop, results may significantly vary when run on a more powerful PC with faster processing speed, more cores, and more RAM. However, the factorial graph trend will be synonymous. A more efficient approach in comparison would be with Dynamic Programming which would only store the optimal paths by far with memory and then pick out of those, but it does come with a disadvantage of approximations at times rather than a fully supported answer. Overall, this project served as a great introduction into solving the TSP using Brute Force approach and allowed for more creative thinking for projects dealing with TSP moving forward.

5. References

Itertools Documentation - https://docs.python.org/3/library/itertools.html
Matplotlib Documentation - https://matplotlib.org/stable/index.html
Backtracking Algorithm Paradigm Explanation - https://www.geeksforgeeks.org/write-a-c-program-to-print-all-permutations-of-a-given-string/
Time Complexity Information - https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/
Dynamic Programming vs Brute Force - https://towardsdatascience.com/obvious-dynamic-programming-part-i-553ae1740c67