# Shortest Drone Delivery Route Finding Algorithm

Data Structures and Algorithms Project Report

Aditi Satish Suvranil Ghosh

## Abstract:

As more and more smart service automations get adopted in different industries, there rises a need for faster and more efficient algorithms that form the backbones of such automation. Speaking of service automations, delivery using autonomous drones are getting popular by the day and big e-commerce companies like Amazon [2], Uber [3], etc. are in the testing phase of the mentioned. Drone routing can be really complicated depending on the location of the destinations and the number of drones involved. This paper takes into consideration a more dialed down version of drone routing and given a list of delivery locations, aims to find the shortest path for a single drone to complete all the deliveries using a rather slow solution to the Travelling Salesman Problem (TSP) [1].

## Introduction:

We want to start off by describing the Travelling Salesman Problem. It is a NP-complete problem where the challenge is, given a list of destinations, to find the shortest and most efficient route that a person could take to visit all the locations and come back to their origin. For example, if a person is in New Jersey and they want to do a road trip through Delaware, Pennsylvania, New York, and North Carolina, what is the shortest route they can take in order to visit all these states and then come back home to New Jersey? There exists no “quick” solution to this problem in that the complexity of calculating the best route can only increase with the number of destinations and hence TSP is also classified as a NP-hard problem.

Since even the most expensive drones have limited range (around 20 Km) when it comes to how far they can travel in one go, instead of taking up cities or locations that are far apart as inputs to our solution, we used a list of randomly generated location coordinates within the city of New Brunswick, NJ. This is more realistic since deliveries across states or even different cities/towns would not be carried out a single drone. In the bigger picture, a fleet of drones are deployed with each drone assigned a handful delivery locations. Our implementation is aimed towards the routing of one of these drones and can be scaled in a parallel manner to each drone in the fleet. The paper is structured as follows: we give a detailed description of the algorithm(s) used, then

we talk about implementation, evaluation and application of our project and thereafter end with a concluding statement.

## Algorithm:

In our attempt to find the shortest route for delivery drones, we use a rather naive solution to the TSP, where we iterate through all possible routes i.e., permutations of the provided delivery locations and find the shortest unique route. Considering we have $N$ delivery locations (including the origin - warehouse) for a drone to travel to and complete deliveries, there are $N-1$ possibilities for the choice of the first delivery location, $N-2$ for the second location after the first delivery is complete, $N-3$for the third location and so on. So, if there is a list of N delivery locations including the origin (specifically $N-1$delivery locations), we consider (and generate) all $(N-1)!$ permutations of the delivery locations and iterate over them in order to determine which permutation results in the lowest cost and hence is the shortest route. These permutations essentially represent all possible Hamiltonian cycles in the graph of delivery locations. Hamiltonian cycles are tours or paths in a graph where the path visits every node in the graph exactly once and comes back to the node where the path started from. One noteworthy fact to note is that since we consider displacements between locations as our path lengths, the graph representation we are going to use for implementation of this algorithm will be undirected (or bidirected) and hence we shall have single duplicates of each of $(N-1)!$ paths [4]. Therefore, a possible optimization of this algorithm would be to consider $(N-1)!/2$ instead of $(N-1)!$ paths. We used the former for ease of implementation and do one comparison and two assignments (each for storing the minimum weight and shortest path) per permutation hence resulting in a time complexity of $O(N!)$.

Now, there are a few assumptions we make while deploying this algorithm and a handful limitations as well. We assume that given the list of delivery locations, all the Hamiltonian cycles exist and so does a minimum weight Hamiltonian Cycle. This is a safe assumption since, given the aerial nature of drones, they can follow the displacement between locations A and B as their path. There still are certain limitations to this like path obstructions (high rises blocking the drone even if it is at its highest altitude), severe weather that can sway the drone off its ideal trajectory, etc. For our implementation, we consider an ideal scenario where the drone can move around freely and follow the straight-line paths between two locations.

## Implementation and Evaluation:

We implement our algorithm using Python and test it on different datasets that have been randomly generated using another Python script coordGenerator.py and output into text files,

all of which can be found in the Github repository here. Our implementation takes in a list of delivery locations represented by their coordinates in degrees (indexed 0 to N-1) as inputs and outputs the shortest path using corresponding indices of the locations and the length of the path in kilometers. The first entry in each input file is the origin of the drone (for example: an amazon warehouse) and the rest are the delivery locations. Here is a sample input file of length 4:

```
40.39101248761078 -74.58487406621235
40.46046455123439 -74.35689345994184
40.6339442677396 -74.44147785909529
40.43307662409769 -74.56588046623956
```

These are coordinates generated for the city of New Brunswick, NJ located at (40.48216° N, 74.451819° W) or (40.48216, -74.451819) degrees. The generator python script outputs M coordinate tuples within $\pm 0.15$ degrees (~16 Km) radius of New Brunswick where M is the length of the file (number of coordinates to process) ranging from 3 to 19. We do not consider files of length 1 or 2 since the solutions to those are trivial and unique.

These coordinates are then read from one such file of length M and fed into our algorithm. But since, our implementation of naive TSP solution takes in an adjacency matrix representation of a graph instead of raw coordinates alongside the source/origin of the path to be calculated, we do some data preprocessing in the form of calculating distances between each pair of vertices and inserting them into appropriate positions inside an adjacency graph. This data preprocessing step is done using the function `coordToGraph()` which takes in a list comprising line string of lines and returns a graph in the form of an adjacency graph where `graph[i][j]` denotes distance from vertex (location) `i` to vertex `j`. An optimization we applied to this function was to assign calculated distances between coordinates `i` and `j` to `graph[i][j]` and `graph[j][i]` at the same time since we assumed before that paths between two delivery locations are bidirectional. We also did a check of `i==j` while filling out the adjacency matrix in order to fill those with 0s assuming no negative distance between two coordinates.

To calculate the distance between two geographical coordinates, it is not as simple as finding the same for cartesian coordinates which can be found using the Pythagorean Theorem. This has to do with the fact that the surface of the earth is not flat but spherical. Hence, we use the Haversine Formula to calculate the distance between two geographical coordinates given in degrees. Given a pair of coordinates ($\varphi_1$,$\lambda_1$) and ($\varphi_2$,$\lambda_2$) in radians, the distance in kilometers between these two coordinates is calculated as follows [5]:

$$d = 2R * \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos(\varphi_1)\cos(\varphi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

Where latitudes are denoted by $\varphi$, longitudes by $\lambda$, and the average radius of the earth between the two coordinates by $R$ in kilometers. For simplicity, we use a constant value for $R$ i.e., 6378.1 Km, the global average radius of the earth.

Now that our data preprocessing is complete, and we are ready to run our algorithm on the adjacency matrix using a given source `s`. We set s to 0 for all our test runs due to the assumption we mentioned before that the first entry in the input file will always be the origin of the drone's delivery route and this origin is represented by vertex 0 in the graph. The shortest path and its weight is calculated by the function `shortestPath(s, V)` where the arguments `s` and V are the source and adjacency matrix respectively. This function starts off by storing all vertices other than the source vertex into a list `vertex[]` and declaring a couple more variables: `min_path_distance` initialized to infinity for storing the minimum route distance and `next_permutation` initialized to the return value of `permutations(vertex)`. `permutations()` is an inbuilt function in Python's `itertools` library that takes in a list of objects and returns all possible permutations of those objects as immutable tuples in a list-like data structure. Thereafter, the actual computation of the shortest path and its weight is straightforward: we iterate through the list of $(N-1)!$ permutations, calculate weights of each permutation, check if the weight is less than the current minimum and store the weight and permutation or discard accordingly, and finally return the deliverables i.e., shortest path and its corresponding weight. Each permutation here is a list of vertices and hence we use the returned permutation with the minimum weight in our driver code to print out the shortest path.

In the driver code that we use for evaluating our TSP solution implementation, we first process the file name passed in as a command line argument and throw an error if no such argument is found. Then we start a timer that calculates the time for data preprocessing and shortest path calculation and outputs the execution time on the screen and to a file under the './output/' directory. We also print out the shortest path using the permutation returned by the shortestPath() function with arrows in between each vertex. Below is an example of the output:

```
PS C:\Users\linar\Desktop\Spring 2021\DSA\Project> python .\droneRoute.py
deliveryLocs11.txt
Number of delivery locations:  11
Shortest Route: 0 -> 9 -> 7 -> 3 -> 8 -> 2 -> 1 -> 10 -> 4 -> 6 -> 5 -> 0
Shortest Route Distance: 96.255 Km
Execution time = 7.216 seconds
```

Relevant specifications of the system that our code was tested on (using Python version 3.9.1):

- Operating System:  Windows 10 Pro (64-Bit) Version: 20H2
- CPU: AMD Ryzen 9 5900x with 12 cores (24 logical) with clock speeds of up to 4.8GHz
- DRAM: 32 GB @3200Mhz

Following is plot of average (over 5 trials) execution time (in seconds) versus test dataset sizes of 4 through 16:

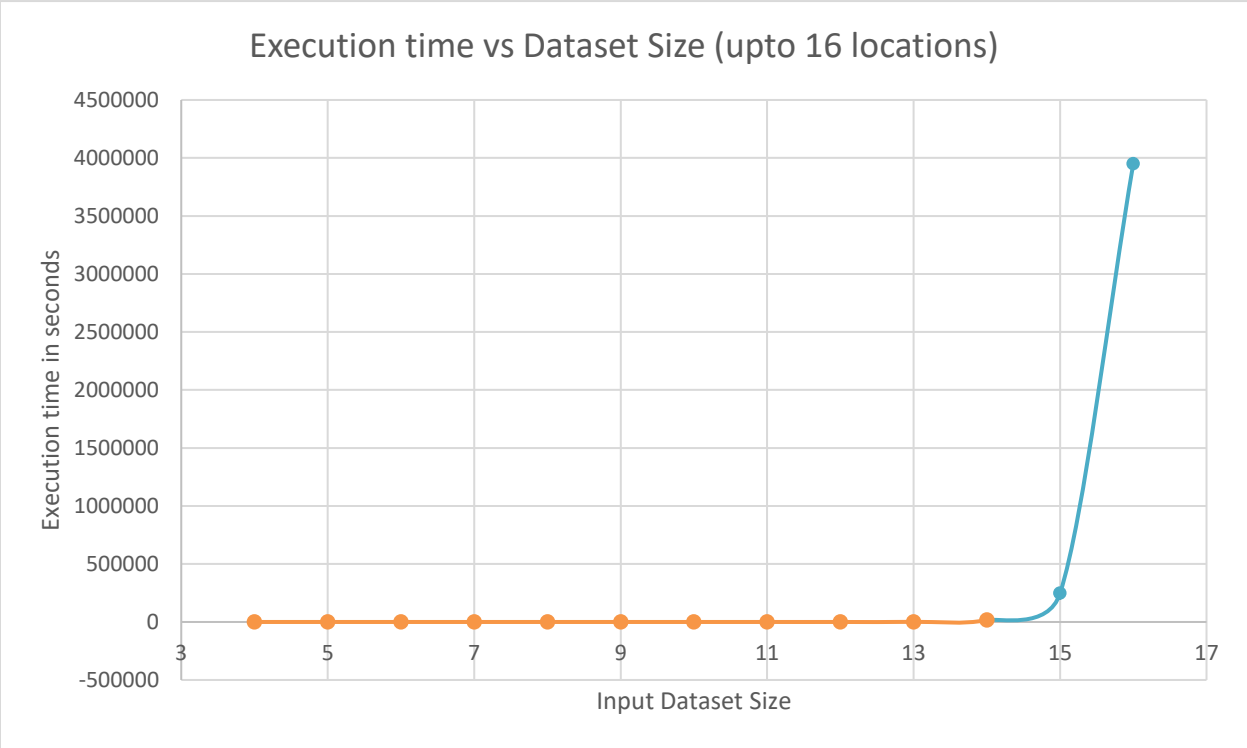


Figure 1

After analyzing the execution times corresponding to different dataset sizes, we observe that the execution time of our code on input data with $N$ elements is approximately $N$ times the execution time for $N-1$ elements. For example, for 14 elements (shortest path was of length ~96 Km), the average execution time was $16459$ seconds ($\sim 4.5$ hrs) which is approximately 14 times $1183$ seconds ($\sim 20$ minutes), the execution time for 13 elements. Testing for 15 and 16 elements was going to take days (nearly 3 days for 15 elements and 45 days for 16 elements) which is why we forecasted the execution times of those data points instead of experimentally acquiring the values and hence the blue line and data points in Figure 1 for 15 and 16. Our initial plan was to test datasets of size up to 19 but judging from the forecasts, it would take our code approximately 728 years to compute the shortest route for a dataset containing 19. That said, for the application of our implementation which is finding the shortest drone delivery routes, one drone would not have enough range to complete all 19 deliveries. 3 or 4 delivery locations sound a bit more reasonable in which case the code took around 1 milliseconds to compute the shortest route (distance was approximately 28 Km).

Now in figure 1, even though error bars using standard deviation were plotted (for data points other than the projected ones), the values of the standard deviation were too small to be visible

in this graph. Hence, we show more zoomed in versions of this graph in the following plots for up to 11, 10, 9 and even 8 elements or delivery locations:

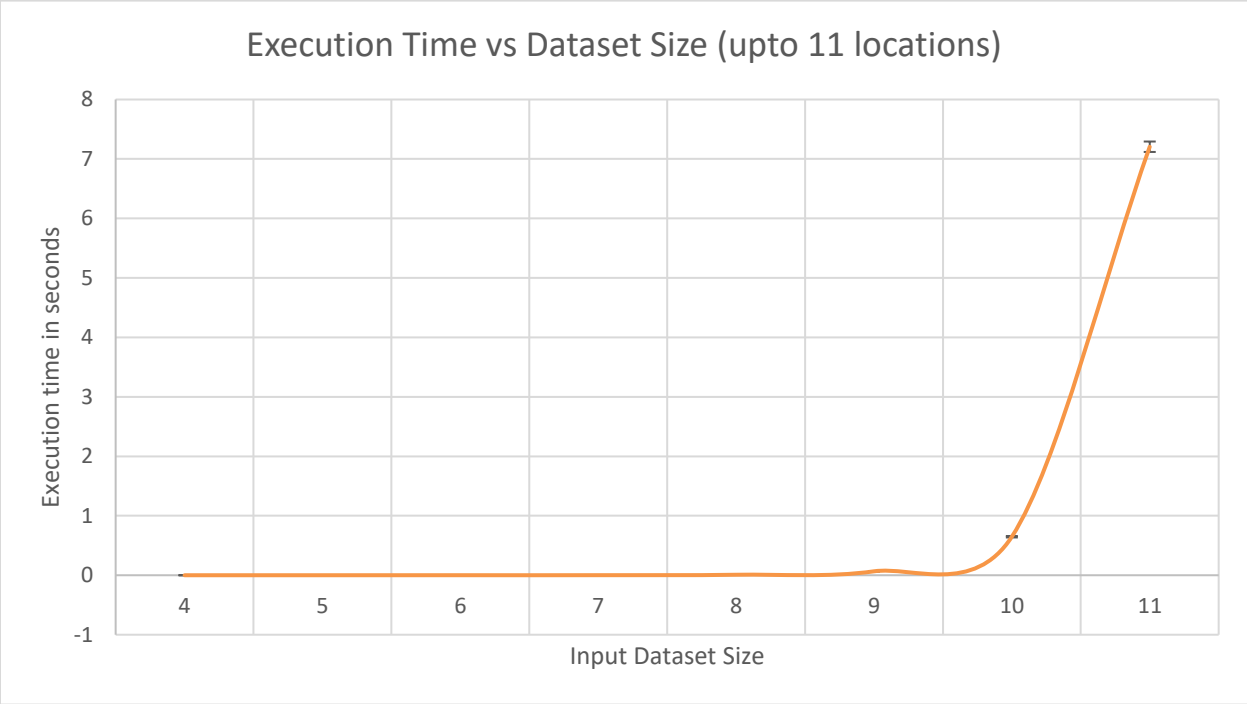


Figure 2

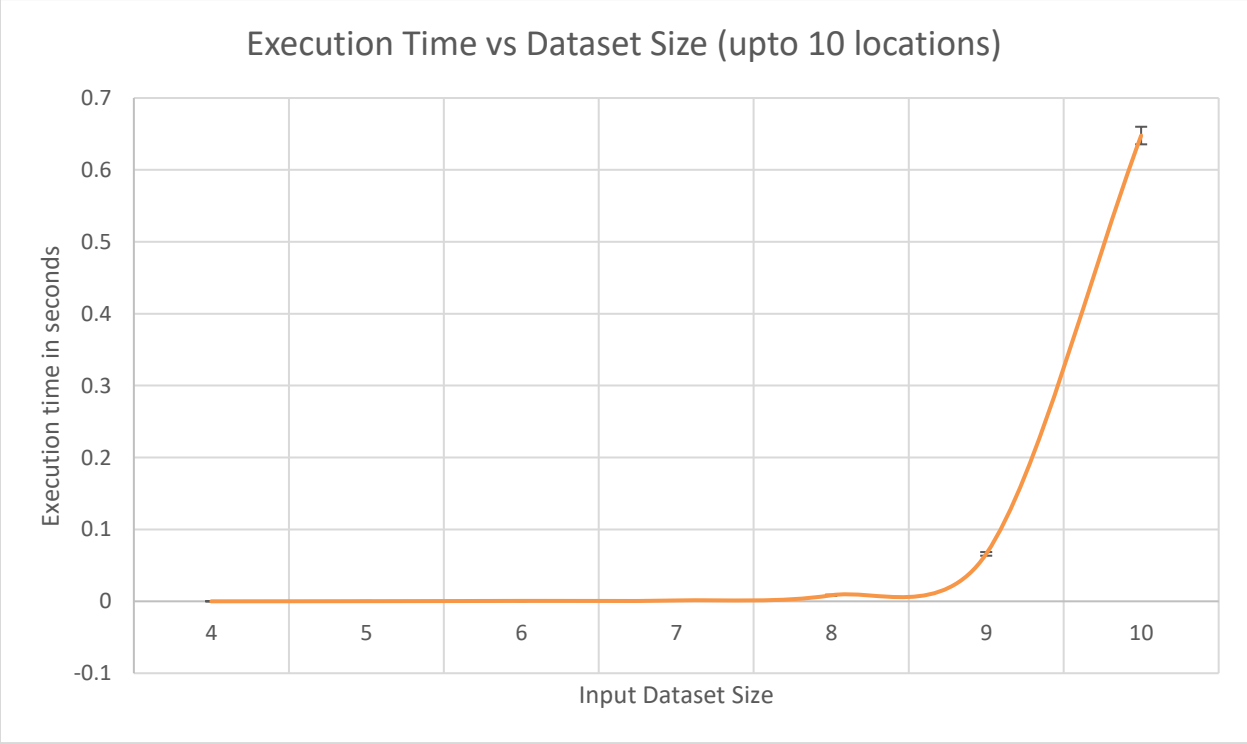


Figure 3

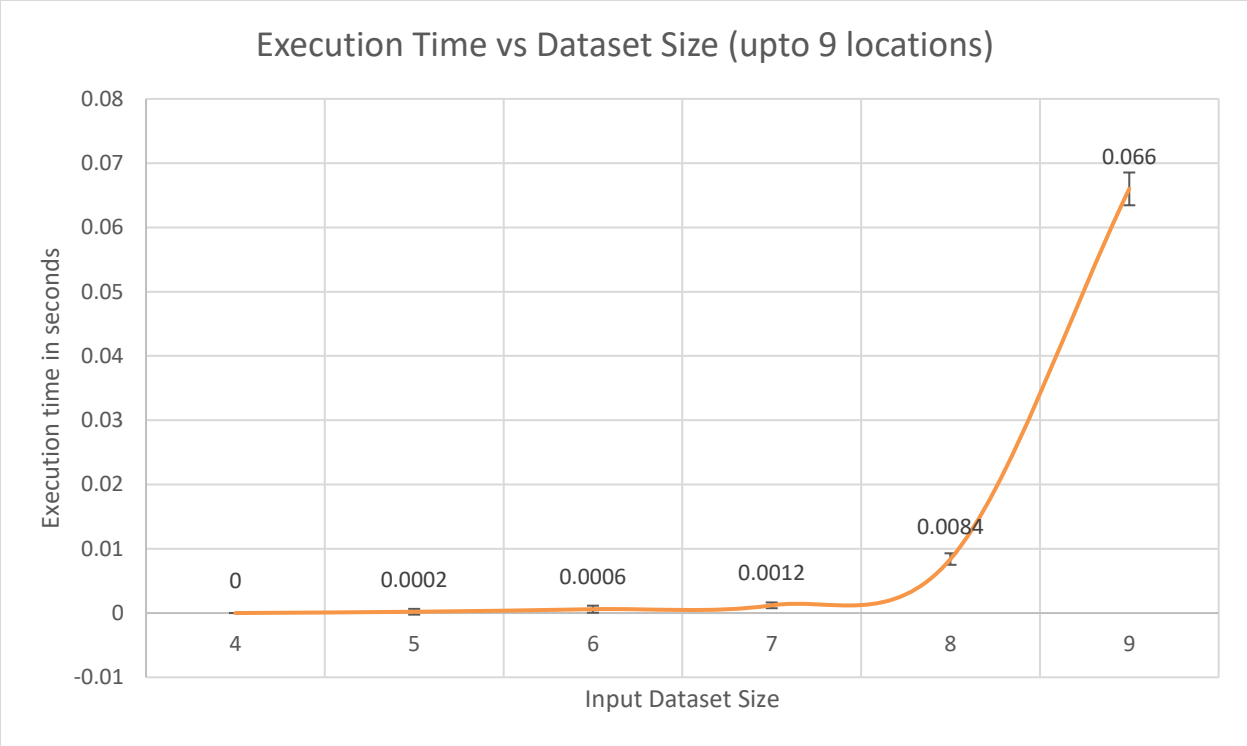


Figure 4

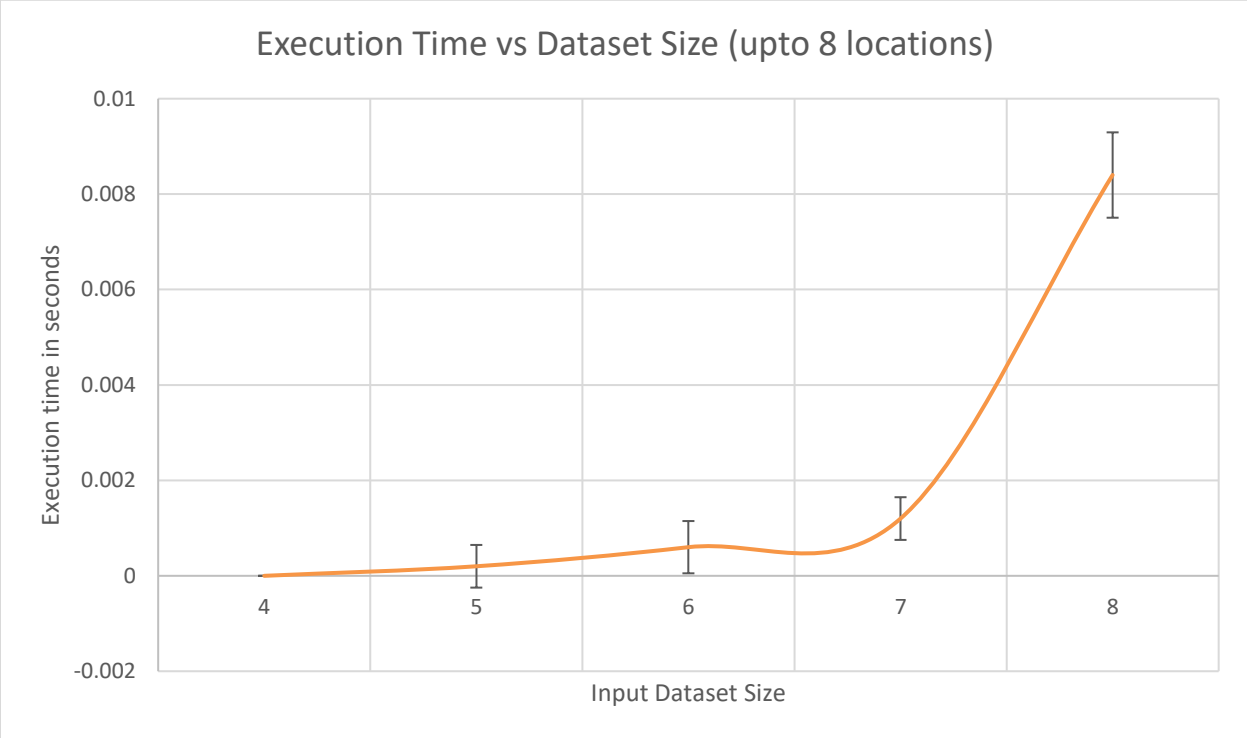


Figure 5

For exact values we also provide with the data recordings in the following table:

| Dataset Size | Execution Times (in seconds) | | | | | | |
|---|---|---|---|---|---|---|---|
| | Trial 1 | Trial 2 | Trial 3 | Trial 4 | Trial 5 | Mean | Standard Deviation |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0.001 | 0 | 0 | 0 | 0.0002 | 0.000447214 |
| 6 | 0.001 | 0 | 0 | 0.001 | 0.001 | 0.0006 | 0.000547723 |
| 7 | 0.001 | 0.001 | 0.001 | 0.002 | 0.001 | 0.0012 | 0.000447214 |
| 8 | 0.008 | 0.008 | 0.01 | 0.008 | 0.008 | 0.0084 | 0.000894427 |
| 9 | 0.07 | 0.063 | 0.066 | 0.066 | 0.065 | 0.066 | 0.00254951 |
| 10 | 0.649 | 0.668 | 0.636 | 0.644 | 0.642 | 0.6478 | 0.012214745 |
| 11 | 7.196 | 7.061 | 7.23 | 7.253 | 7.288 | 7.2056 | 0.087511714 |
| 12 | 86.412 | 88.081 | 88.822 | 87.599 | 85.085 | 87.1998 | 1.471429135 |
| 13 | 1183.98 | 1184.969 | 1180.655 | 1183.648 | 1184.55 | 1183.5604 | 1.702097911 |
| 14 | 16551.857 | 16435.198 | 16341.024 | 16403.838 | 16563.204 | 16459.0242 | 96.18811148 |
| 15 | | | | | **Projected:** | 246885.363 (69 hrs) | |
| 16 | | | | | **Projected:** | 3950165.808 (46 days) | |

As we can see from the charts and the table, our experiments of 5 trials for each dataset showed minimal error in form of standard deviation.

## Future Work:

In this project, we chose the simplest solution to TSP i.e., the naïve solution of iterating through all Hamiltonian cycles and finding the shortest one which resulted in a time complexity of $O(N!)$. There are a handful other faster solutions we want to implement as a part of our future work for this project:

- Nearest Neighbor, a greedy approach: time complexity of $O(N^2)$ [6]
- Christofides' Algorithm: time complexity of $O(N^4)$ [7]
- Dynamic programming approach: time complexity of $O(2^N * N^2)$ [8]

## Conclusion:

Given the short range of around 20 Km of modern drones, a realistic dataset for shortest route computation would contain 3 or 4 delivery locations depending on the distance between them (too far apart delivery locations would make it harder for the drone to complete all the listed deliveries). In our randomly generated dataset for New Brunswick, the shortest route for 3 delivery locations came out to be around 28 Km which is quite of stretch already and the

execution time for the same was around a millisecond. It is noteworthy to mention here, that the shortest route distance totally relies on the input dataset and how far apart the coordinates are. That said, we would have seen shorter routes if we chose to generate coordinates into our input dataset within a 5 Km radius instead of a 15 Km radius. Hence, we can safely say that our algorithm works and is fast enough for the given realistic application of finding the shortest route of drone delivery to a handful locations in a city or a town.

# References:


[1] S. Ma, "Understanding The Travelling Salesman Problem (TSP)", *Routific*, January 2020, https://blog.routific.com/travelling-salesman-problem

[2] Amazon Prime Air, https://www.amazon.com/Amazon-Prime-Air/b?ie=UTF8&node=8037720011

[3] B. Carson, "First Look: Uber Unveils New Design for Uber Eats Delivery Drone", *Forbes*, October 2019. https://www.forbes.com/sites/bizcarson/2019/10/28/first-look-uber-unveils-new-design-for-uber-eats-delivery-drone/?sh=13e437af78f2

[4] R. Mathew, D. Cherukupalli, K. Pusich, K. Zhao, "Travelling Salesman Algorithms from Naive to Christofide", Traveling Salesman Algorithms

[5] "Haversine Formula", *Wikipedia*, https://en.wikipedia.org/wiki/Haversine_formula

[6] F. A. Karkory, A. A. Abudalmola, "Implementation of Heuristics for Solving Travelling Salesman Problem Using Nearest Neighbour and Minimum Spanning Tree Algorithms", *International Journal of Mathematical, Computational, Physical, Electrical and Computer Engineering Vol:7, No:10,* 2013

[7] H.-C. An, R. Kleinberg, and D. B. Shmoys, "Improving Christofides' Algorithm for the s-t Path TSP", *CoRR, abs/1110.4604*, 2011

[8] M. Sniedovich, "A Dynamic Programming Algorithm for the Travelling Salesman Problem", *ACM SIGAPL APL Quote Quad,* June 1993


# Appendix:

Work division

- We decided on our project topic, chalked out our roadmap as well as learnt most of the material needed to carry out the project in a group study manner.
- On the implementation and evaluation side of things, Aditi worked on the data preprocessing, helper functions for main algorithm implementation and input data generation whereas Suvranil worked on the driver code, main algorithm implementation, data recording and plotting.

- For the project report and presentation, we again divided our work equally and completed both alongside the implementation well ahead of the deadline but added in more material after the first round of presentations catered towards addressing the concerns Professor Shantenu Jha had for the first five presenters.

Henceforth, it is fair to say that both of us worked on this project: Aditi contributed to 50% of the project and Suvranil 50%.