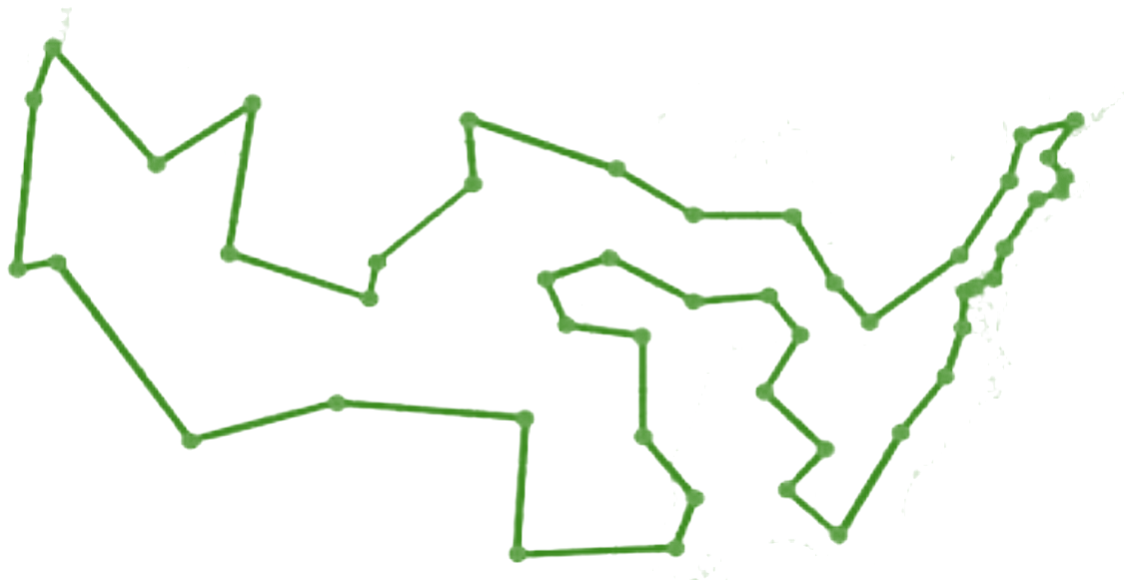




Abubaker Shabbir, Idris Dodwell, Wong Shean

EPM02 Delivery Route Optimization Coursework



| Nomenclature | Abstract |
|---|--|
| TSP - Traveling Salesman Problem | <i>By applying the TSP to a real-world situation where a delivery worker must visit several customer locations efficiently, this project will tackle the problems of delivery route optimization. Creating an algorithm that finds the shortest path between each destination as well as the starting point is the main objective. This optimized path is created using a Python implementation of the closest neighbour technique, where client locations are provided as coordinates. The computational performance and route length are analysed to determine how effective the solution is, and the results are displayed through route visualization.</i> |
| RL - Route Length | |
| GA - Genetic Algorithm | |
| DP - Dynamic Programming | <i>The approach's efficiency and limitations are shown by the findings, which also point to potential enhancements to lower processing loads and improve route accuracy.</i> |
| NN - Nearest Neighbour | |
| BF - Brute Force | <i>This study illustrates how useful TSP solutions are in the logistics industry and looks into potential further refinement in route optimization.</i> |
| LO - Local Optimum | <i>The TSP codes explored within this project are the following (1) Nearest Neighbour Algorithm via Euclidean Distance Formula, (2) Dynamic Programming via Held-Karp Algorithm Approach, (3) Genetic Algorithm via Metaheuristic Approach, (4) Brute Force via exact approach</i> |
| GO - Global Optimum | |

Introduction

Innovation and efficiency have long been central goals in engineering, with the Traveling Salesman Problem (TSP) serving as a prime example of a complex optimization challenge that has engaged engineers, mathematicians, and scientists for decades. Though the problem may seem straightforward initially, its complexity grows exponentially with the number of cities involved. The challenge lies in the vast number of potential routes and the difficulty in identifying the optimal one. Despite the numerous possible solutions, some methods, such as the Brute Force Algorithm, work effectively only with a limited number of cities. Due to advances in modern technology, the TSP has found applications in many fields, including

logistics, planning, and circuit design. Over the years, researchers have developed various approaches, including biologically inspired methods like ant colony optimization, to find near-optimal solutions. This project aims to develop an algorithm to determine the shortest route for delivery personnel. Achieving this requires an understanding of the input size and the selection of an appropriate algorithm. According to Bridgen et al. (2023), daily deliveries (i.e., 95% of them) can amount to as many as 150, though this number may vary depending on several factors. This provides a starting point for choosing an appropriate TSP algorithm. While powerful algorithms such as Genetic and EAX (Nagata and Kobayashi, 2013) yield

reasonable running times for large inputs (up to 200,000), and the Brute Force approach performs efficiently for very small inputs ($N!$ for an input N , Kolog, 2015), the upper limit for this study is set at 150. Therefore, the chosen algorithms must produce a manageable upper-bound asymptotic notation (e.g., Big O) for this input size (Cormen et al., 2022, pp. 33-36). This constraint implies that the algorithms used here may perform differently with other input ranges.

The selection of algorithms will be determined through an analysis of each, assessing and comparing their efficiency and optimization capabilities for solving the TSP with the specified input size, aiming for a balance between performance and efficiency. As TSP is an NP-hard problem (Cormen et al., 2022, p.1109), it is essential to follow established literature to save time and avoid unnecessary errors (Ruzich, 2008). Given the extensive research on TSP (Laporte, 1992), this study focuses on traditional didactic algorithms (Kokmotos, 2023) while acknowledging the advanced algorithmic ecosystem that could be relevant today. Before comparing algorithms, it is necessary to understand what the leading terms in asymptotic notation represent. For instance, while insertion sort has an O -notation of n^2 , the full expression is $an^2 + bn + c$ (Cormen et al., 2022, p.31). Although the coefficients and constants can often be disregarded with larger inputs, they noticeably impact smaller values. This explains why insertion sort outperforms merge sort for smaller inputs, despite merge sort's more favorable O -notation of $n \log n$ (Cormen et al., 2022, pp. 35-44).

The nearest neighbor (NN) heuristic algorithm (Kizilates-Evin and Nuriyeva, 2013), a greedy algorithm, works by selecting the nearest unvisited node at each step until all nodes are included, then returning to the starting node or depot

(Kuo, 2024). While NN is simple and quick, it does not always yield the optimal solution (Kizilates-Evin and Nuriyeva, 2013). This limitation is shown in Figure 1, where the path weight is inefficient. However, Figure 4a demonstrates that the comparison for small inputs between NN and the dynamic programming algorithm is similar. Kuo (2024) suggests using NN as an initial solution to be refined with a more efficient algorithm.

An exact solution is provided by the Dynamic Programming (DP) algorithm (Malandraki and Dial, 1996), which has a big O of $(n^2 * 2^n)$ (Kokmotos, 2023). However, it is suitable only for small inputs, making it unsuitable for this project's requirements, although some generalizations improve its performance (Malandraki and Dial, 1996). As an exact algorithm, DP provides an optimal path (Kokmotos, 2023). Figure 2 shows this "clean" solution, yet Figure 4a reveals no notable improvement over NN. Larger input comparisons (above 20 nodes) were impractical, disqualifying DP as a viable option.

Finally, the Genetic Algorithm (GA), another heuristic with a time complexity of $O(n^2)$, employs nested loops for its calculations (Adewole et al., 2011). GA, derived from evolutionary computing, begins with a random set of solutions. Each solution is assigned a value that reflects its optimality, and parent solutions are selected for crossover, akin to reproduction. Mutation and variation are introduced, fitness values calculated, and the results added to the "gene pool." This iterative process continues for n generations until the solution approaches the desired outcome (Adewole et al., 2011). Figure 3 illustrates the path generated by GA, which appears to perform slightly worse than NN. This difference is made clear in Figure 4b/4c,

where GA and NN demonstrate comparable path lengths.

Methods

- **Imports:** The following libraries and functions are imported to facilitate data processing, visualization, and computational tasks for route optimization:

- *NumPy* is used for efficient data handling and manipulation, particularly in representing and managing coordinate arrays for the travelling salesperson problem (TSP) [3].

- *Matplotlib* is utilised for plotting the coordinates and visualizing the resulting paths from different routing methods, aiding in a clear graphical presentation of the optimization process [4].

- *SciPy* is essential for computing pairwise distances between coordinates, allowing the algorithm to evaluate distances between cities efficiently [5].

Two constant scripts are used consistently throughout all four code implementations. These constants ensure uniformity in data handling and computations, regarding the coordinates generated and plotting for visualisation.

- Constant One; *Random Coordinate generator*:

```
def generate_coordinates(coordinates=10,
area_size=100):

    # Defines a function which takes 2 arguments,
    which is the number of coordinates and in which
    area they be will scattered within

    np.random.seed(42)
    # This will hold the coordinate results to allow
    us to reproduce/iterate these results for different
    amount of coordinates
    coordinates = np.random.rand(coordinates, 2)
    * area_size
    # Will generate 2 random coordinates between
    0 and 1 across the field area
```

```
return coordinates
# Provides our code with the random
coordinates
```

* Where n is an entry for coordinate amount input.

- Constant Two; *Graph Plotting of coordinates*:

```
def plot_path(coordinates, path,
title="Nearest Neighbour Algorithm via
Heuristic Approach"):
```

```
# Defines a function which takes 3
parameters into consideration, where the
route will be coordinated in a sorted array
list
```

```
plt.figure(figsize=(10, 7))
# Size of grid field; tweaked to ensure all
coordinates fit within the view
```

```
plt.title(title, fontsize=14)
# Title size
```

```
plt.scatter(coordinates[:, 0], coordinates[:,
1], c='black', s=60, label="Coordinates")
```

```
# Will extract all x and y coordinates and
present them as black circles with size 30
```

```
plt.plot(coordinates[path, 0],
coordinates[path, 1], 'b-', linewidth=3,
label="Path")
```

```
# Will order and present x and y
coordinates in order of route processing, a
certain program entails, will present as a
blue line
```

```
plt.scatter(coordinates[0, 0], coordinates[0,
1], c='purple', s=100, marker='s',
label="Depot (Start & End point)")
# Marks the starting point (Depot), in the
form of a purple dot.
```

```
plt.legend(loc="best")
# Code box to represent all legends/labels
used
```

```
plt.show()
# Presents final figure
coordinates = generate_coordinates()
# Applying our coordinate-generating function

path = closest_point(coordinates)
# Applying a Function specific to this method, to compute a path for the system

plot_path(coordinates, path, title="Nearest Neighbour Algorithm via Heuristic Approach")
# Applying our Graphical generating function
```

Presented from here on, will be the following:

- A) Flowchart representation of each 4 codes
- B) Selection of algorithm based on the following tests & inspections:
 - i) Accuracy Testing
 - ii) Execution Time with Scalability Testing

A - Flowchart representation of each 4 codes

Program 1 - Nearest Neighbour Algorithm Via using the Euclidean Distance Formula

The Nearest Neighbour Algorithm is a straightforward and efficient approach for addressing the Traveling Salesperson Problem (TSP). This method computes the distance from the 'current location' to every other unvisited point, and then proceeds with selecting the nearest point as the following next destination. The iterative process continues until all points have been 'visited'. Once all have been visited the program will return to the starting point/depot.

Due to its simplicity, the NN Algorithm is one of the fastest methods for computing the TSP. This is particularly an advantage when it comes to large datasets, where rapid computation is essential. However, while it provides a quick solution, this method does not guarantee the most optimal path, as it may overlook shorter routes in pursuit of immediate proximity; given it's 'greedy' like execution. [EITCA Academy (2023)]

Flow Chart Diagram - Program 1 - Nearest Neighbour Algorithm Via using the Euclidean Distance Formula

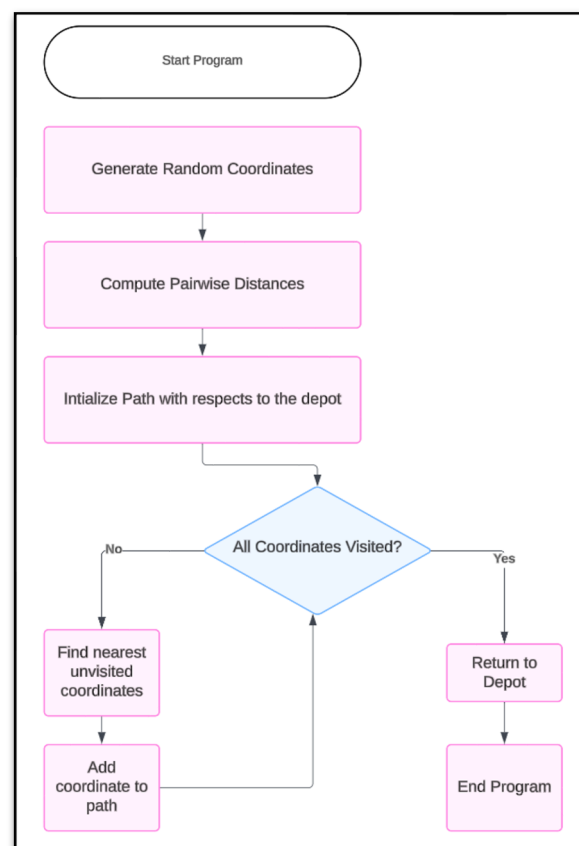


Figure 1 flow chart for NN program

Program 2 - Dynamic Programming via Held-Karp Algorithm Approach

The Dynamic Programming approach, via the Held-Karp Algorithm, is a method for solving the TSP by breaking down the constituent tasks into a series of overlapping subproblems. This technique optimizes the search for the shortest route

through storing computed distances between points, which helps to avoid redundant calculations as well as reducing the computational effort required for smaller instances of the problem.

However, DP isn't suitable for larger datasets, as the Held-Karp Algorithm will result in significant delays. This is due to the exponential growth in the number of subproblems that need to be computed and compared, which increases the execution time of the program as well as computational requirements substantially.

Flow Chart Diagram - Program 2 - Dynamic Programming via Held-Karp Algorithm Approach

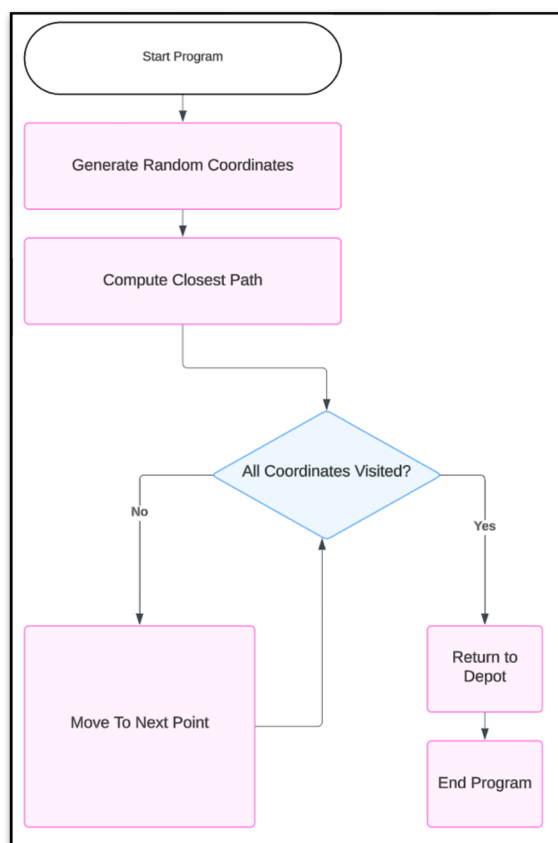


Figure 2 flow chart for DP program

Program 3 - Genetic Algorithm via Metaheuristic Approach

The GA provides an effective metaheuristic approach for finding approximate solutions to the TSP.

In industrial applications, GA is known for its speed in generating solutions, making it comparable to the NN method described in Method 1. Like NN, GA prioritizes efficient computation over accuracy, making it suitable for scenarios where time constraints are critical as well as when there are large data sets.

Although GA aims to deliver a near-optimal solution through iterative improvements and evolutionary techniques, it cannot guarantee the exact optimal route, like the NN method. Instead, it offers a balance between speed and solution quality; which makes it particularly useful for larger datasets and makes it slightly more advanced than NN; which mainly focuses on execution. [Shendy, R. (2024)]

Flow Chart Diagram - Program 3 - Genetic Algorithm via Metaheuristic Approach

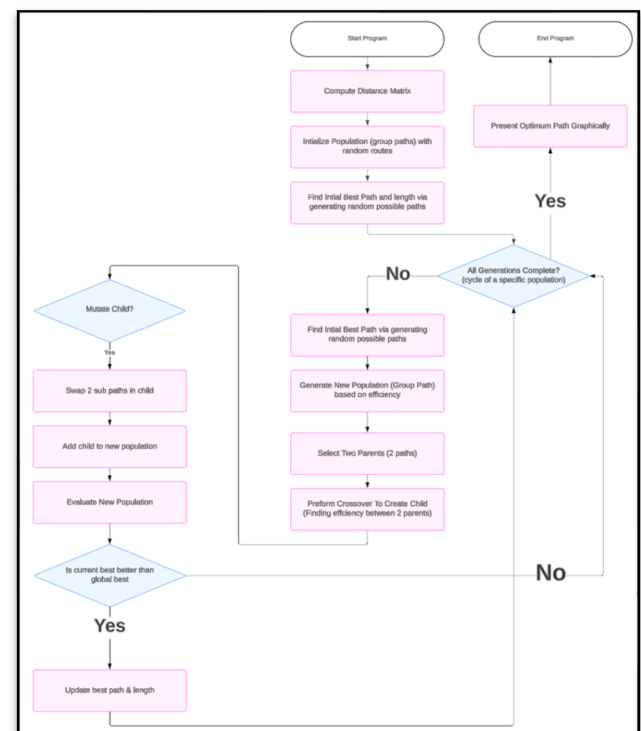
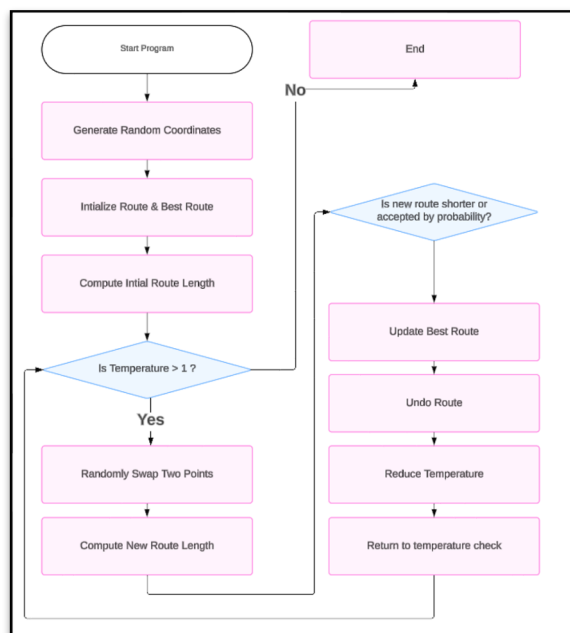


Figure 3 flow chart for GA program

Program 4 - Brute Force via exact approach

The brute force approach for solving the TSP entails finding the exact solution by ‘exhaustively’ evaluating all possible routes. This method involves calculating the total distance for every permutation of the cities, and then selecting the route with the shortest total distance. Although this approach guarantees an optimal solution, it becomes computationally infeasible over time as larger problems present themselves (due to the factorial growth in the number of permutations as the number of cities increases) This also directly, impacts computational times.
[13. Case Study: TSP]

Flow Chart Diagram - Program 4 - Brute Force via exact approach



From briefing all the following 4 codes, it's important to understand that there must be a trade-off between speed and accuracy. Given the fact Brute force not only takes accuracy into significance more than it does speed, but the program is also therefore marked infeasible for the nature of a TSP system.

Therefore, moving forward only codes NN, DP & GA are further looked into.

Ba) Accuracy Testing

In order to test, for accuracy, as stated previously, brute force takes accuracy more significantly, in an accuracy vs speed trade-off. Therefore, below presented will be all the codes run with a consistent 10 coordinates via methods NN, DP and GA.

Once done, they will be compared with a brute force program, run using those exact 10 coordinates.

Lastly, comparison between path output given out in brute force will be compared with methods: NN, DP and GA.

- NN Code Run-Through:

1) Coordinates Generated:

[37.45, 95.07], [73.2, 59.87], [15.6, 15.6], [5.06, 86.62], [60.11, 70.81], [2.06, 96.99], [83.24, 21.23], [18.18, 18.34], [30.42, 52.48], [43.19, 29.12]

2) Computations of Distances between Points Euclidean distance formula between two points:

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

3) Distance Matrix (rounded to 2 decimal places):

| | | | | | | | | | |
|-------|-------|-------|--------|-------|--------|--------|-------|-------|-------|
| 0 | 50.17 | 82.42 | 3.47 | 33.2 | 35.44 | 86.89 | 79.11 | 43.17 | 66.2 |
| 50.17 | 0 | 72.65 | 73.2 | 17.06 | 80.24 | 39.92 | 68.93 | 43.41 | 42.97 |
| 82.42 | 72.65 | 0 | 71.8 | 70.92 | 82.51 | 67.87 | 3.76 | 39.75 | 30.72 |
| 33.47 | 73.2 | 71.8 | 0 | 57.28 | 10.8 | 101.92 | 69.53 | 42.53 | 68.99 |
| 33.2 | 17.06 | 70.92 | 57.28 | 0 | 63.68 | 54.71 | 67.17 | 34.89 | 44.99 |
| 35.44 | 80.24 | 82.51 | 10.8 | 63.68 | 0 | 111.04 | 80.28 | 52.78 | 79.36 |
| 86.89 | 39.92 | 67.87 | 101.92 | 54.71 | 111.04 | 0 | 65.12 | 61.37 | 40.82 |
| 79.11 | 68.93 | 3.76 | 69.53 | 67.17 | 80.28 | 65.12 | 0 | 36.27 | 27.23 |
| 43.17 | 43.41 | 39.75 | 42.53 | 34.89 | 52.78 | 61.37 | 36.27 | 0 | 26.62 |

e.g., [row, column] – [5,6] = 63.68, To compute for [5,6], the Euclidean distance formula is applied:

Given Point 5: (60.11, 70.81) & Point 6: (2.06, 96.99)

$$\begin{aligned} & \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} = \\ & \sqrt{(x_6 - x_5)^2 + (y_6 - y_5)^2} = \\ & \sqrt{(2.06 - 60.11)^2 + (96.99 - 70.81)^2} \approx 63.68 \end{aligned}$$

4) Initialization of and path of visited to ensure program starts at the depot, as well as keep track of coordinates already gone through (ensuring no repeats)

Path: [0]
Visited: {0}

5) Begin Iteration of NN:

Iteration 1

- Current Position: 0
- Distances from 0: [0.00, 50.13, 90.61, 32.35, 37.93, 35.42, 85.73, 90.27, 45.41, 66.89]
- Nearest unvisited point: Point 3 (distance = 32.35)
- Path: [0, 3]
- Visited: {0, 3}

Iteration 2

- Current Position: 3
- Distances from 3: [32.35, 74.86, 71.85, 0.00, 57.93, 6.63, 95.84, 68.46, 54.98, 70.68]
- Distances from 3, with filtered visited: [32.35, 74.86, 71.85, 57.93, 6.63, 95.84, 68.46, 54.98, 70.68]
- Note: The code filters the array actively via line "if index not in visited"
- Nearest unvisited point: Point 5 (distance = 6.63)
- Path: [0, 3, 5]
- Visited: {0, 3, 5}

This iteration will continue another 8 times until all coordinates are completed and returned to.

The final path will be the following:

Final NN Path:

[0, 3, 5, 8, 9, 1, 4, 2, 7, 6, 0]

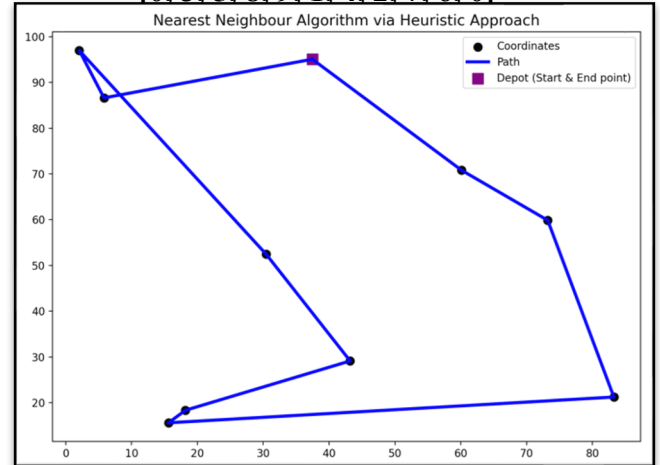


Figure 4 NN Visualisation

- DP Code Run-Through:

1) Coordinates Generated:

[0, 0], [20, 3], [50, 10], [60, 60],
[90, 20], [10, 90], [25, 55],
[80, 90], [5, 70], [55, 40]

2) Computations of Distances between Points Euclidean distance formula between two points:

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

3) Distance Matrix (rounded to 2 decimal places):

| | | | | | | | | | |
|--------|-------|-------|-------|--------|--------|--------|--------|-------|-------|
| 0.00 | 36.06 | 50.00 | 84.85 | 92.20 | 90.00 | 60.21 | 120.42 | 70.71 | 67.08 |
| 36.06 | 0.00 | 36.06 | 50.00 | 70.71 | 61.85 | 25.50 | 87.32 | 41.23 | 39.05 |
| 50.00 | 36.06 | 0.00 | 50.99 | 40.31 | 89.44 | 53.15 | 98.49 | 76.57 | 31.62 |
| 84.85 | 50.00 | 50.99 | 0.00 | 50.00 | 70.71 | 39.05 | 50.00 | 60.83 | 22.36 |
| 92.20 | 70.71 | 40.31 | 50.00 | 0.00 | 106.30 | 75.00 | 70.72 | 97.59 | 36.06 |
| 90.00 | 61.85 | 89.44 | 70.71 | 106.30 | 0.00 | 106.07 | 70.71 | 20.62 | 72.80 |
| 60.21 | 25.50 | 53.15 | 39.05 | 75.00 | 106.07 | 0.00 | 61.03 | 21.59 | 39.05 |
| 120.42 | 87.32 | 98.49 | 50.00 | 70.72 | 70.71 | 61.03 | 0.00 | 75.17 | 53.15 |
| 70.71 | 41.23 | 76.57 | 60.83 | 97.59 | 20.62 | 21.59 | 75.17 | 0.00 | 63.25 |
| 67.08 | 39.05 | 31.62 | 22.36 | 36.06 | 72.80 | 39.05 | 53.15 | 63.25 | 0.00 |

e.g., [row, column] – [5,6] = 106.3, To compute for [5,6], the *Euclidean distance formula* is applied:

Given Point 5: (90, 20) & Point 6: (10, 90)

$$\begin{aligned} & \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} = \\ & \sqrt{(x_6 - x_5)^2 + (y_6 - y_5)^2} = \\ & \sqrt{(10 - 90)^2 + (90 - 20)^2} \approx 106.3 \end{aligned}$$

4) *Closes Point Function*, to find the shortest distance with bitmasking;

The following bitmask function; visit(visited, last); has 2 variables:

- Visited – Tracks coordinates already visited
-
- Last – The index of the last point visited
-
- This function looped continuously until coordinates have been visited.
- lru_cache, via this function, all minimal paths are stored, while the above loops.

Iteration 1

- Initial Call: Program calls visit (1, 0)
- Visited: Only Point 0
- Last point visited: Point 0.
- Next point to visit via distance matrix: 1

Iteration 2

- 2nd call: Program calls visit (3,1)
- Visited: Points 0 & 1
- Last Point Visited: Point 1
- Next point to visit via distance matrix: 6

Iteration 3

- 3rd call: Program calls visit (67,6)

- Visited: Points 0 & 1 & 6
- Last Point Visited: Point 6
- Next point to visit via distance matrix: 9

Note, Numbers 1,3,67... are visited, which tracks coordinates that have been chosen as the next sub-path, represented in binary.

e.g., The bitmask for Points 0, 1, and 6 visited is: 0001000011 (binary), therefore in decimal ($2^6 + 2^1 + 2^0 = 67$)

Iterations continue until all coordinates are visited. After this, it returns to the depot via code “path = [0] + path”.

Final DP Path: [0,1,6,9,3,4,7,8,5,2]

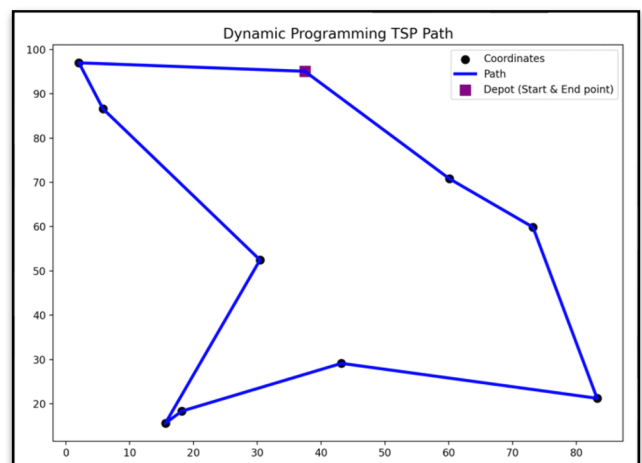


Figure 5 DP Visualisation

- GA Code Run-Through:

1) Coordinates Generated:

(37.45, 95.07), (73.20, 59.87), (15.60, 15.60), (5.81, 86.62), (60.11, 70.81), (2.06, 96.99), (83.24, 21.23), (18.18, 18.34), (30.42, 52.48), (43.19, 29.12)

2) Initialize the Population:

Given this program, possess only 2 routes, only 2 routes can be generated from the 10 coordinates

Route 1: [3, 8, 7, 4, 0, 2, 6, 1, 9, 5]

Route 2: [0, 5, 4, 8, 7, 3, 6, 2, 1, 9]

Route 3: [9, 1, 0, 4, 6, 8, 3, 2, 7, 5]

Note each number on the route, represents an index to a coordinate point, e.g., point 3 is coordinate (15.60, 15.60)

3) Calculate Initial Path Lengths

Route 1 Path Length: Approx.
502.28

Route 2 Path Length: Approx.
550.00

Route 3 Path Length: Approx.
510.00

4) Pick Generation

Since Route 1 possesses a shorter distance, it becomes the best/optimum path for Generation 1.

5) Executing Generation 1

-Random Selection: Choose Route 1 and Route 2 as parents.

-Crossover:

- a) Half of route 1: [3, 8, 7, 4, 0]
- b) Fill Remaining Cities from Parent 3, excluding any cities already in the child, resulting in a child [0, 5, 4, 8, 7, 9, 1, 6, 3, 2]

-Mutation: Randomly swap two points in the child route with a probability of 10%. Suppose we swap points at indices 3 and 7, resulting in:

-Mutated Child Route: No mutation is required; therefore child remains the same: [3, 8, 7, 4, 0, 5, 6, 2, 1, 9]

-Evaluation: Path length = 515 which is longer than the current best path 502.28, therefore route 1 still remains the best route.

6) Executing Generation 2

-Random Selection: Choose Route 2 and Route 3 as parents.

Figure 6 Brute Force NN Coordinate Test

-Crossover:

- a) Half of route 2: [0, 5, 4, 8, 7].
- b) Fill Remaining Cities from Parent 2, excluding any cities already in the child, resulting in a child [3, 8, 7, 4, 0, 5, 6, 2, 1, 9]

-Mutation: indices 3 and 8 are swapped, therefore mutated child route is the following:

[0, 5, 4, 3, 7, 9, 1, 6, 8, 2]

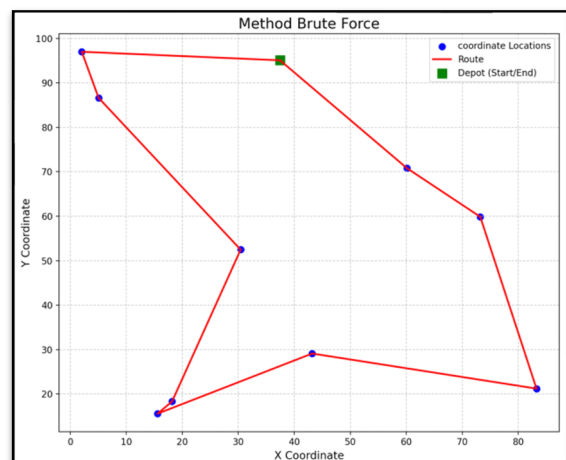
-Evaluation: Path length = 495 which is shorter than the current best path 502.28, therefore mutated route 2 becomes the best route.

6) The code will run for a specified amount of generations (to get optimum generations, an iteration graph is used, and when convergence occurs gives an approx. generation value), for the purpose of testing and simplicity the generations were set to 2.

Therefore, final GA Path:
[0, 5, 4, 3, 7, 9, 1, 6, 8, 2]

Test of accuracy, given NN, DP & GA paths, brute force was run for each of the 10 corresponding coordinates, as shown below.

Brute force for NN 10 coordinates path:



- **Brute path:**
[7, 9, 6, 1, 4, 0, 5, 3, 8, 2]

- Path Length = 290.51

Comparing it with the path
NN - [0, 3, 5, 8, 9, 1, 4, 2, 7, 6, 0]
which has path length = 410.39

-Inaccuracy from brute can be
computed for:

$$\left(\frac{410.39 - 290.51}{290.51} \right) \times 100$$

$$\approx 41.3\%$$

Brute force for DP 10 coordinates path:

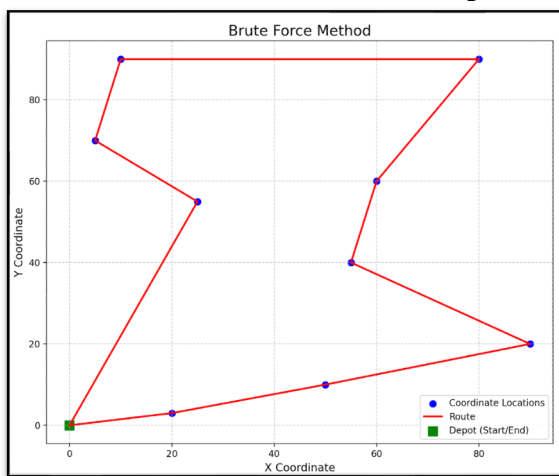


Figure 7 Brute Force DP Coordinate Test

- Brute path:
[8, 6, 0, 1, 2, 4, 9, 3, 7, 5]
- Path Length = 365.27

Comparing it with the path
DP - [0,1,6,9,3,4,7,8,5,2]
which has a path length = 486

$$\left(\frac{486 - 365.27}{365.27} \right) \times 100$$

$$\approx 33.1\%$$

Brute force for GA 10 coordinates path:

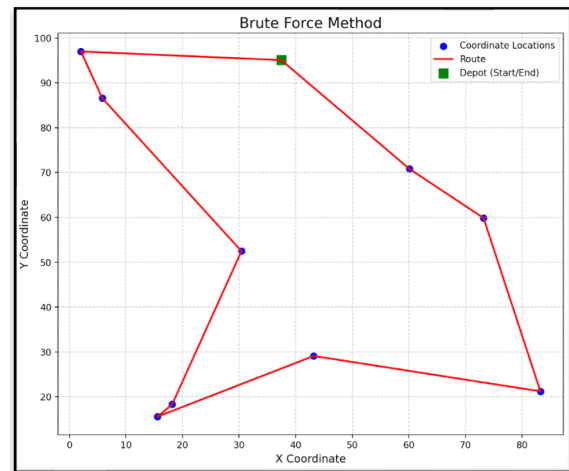


Figure 8 Brute Force GA Coordinate Test

- Brute path:
[5, 3, 8, 7, 2, 9, 6, 1, 4, 0]
- Path Length = 290.31

Comparing it with the path
GA - [0, 5, 4, 3, 7, 9, 1, 6, 8, 2]
which has a path length = 518.73

$$\left(\frac{518.73 - 290.31}{290.31} \right) \times 100$$

$$\approx 78.7\%$$

As shown from the 3 accuracy percentages, *DP produced the lowest %, which means it's the most accurate of the 3 codes in the current analysis.*

Bb) Execution Time with Scalability Testing

This test involves exploring how time varies when coordinates are scaled to 50 from 10 in increments of 5.

Table 1 - Table Of Values for NN

| Table Of Values for Code NN, Run Time | |
|---------------------------------------|---------------|
| Number Of Coordinates | Executed Time |
| 10 | 0.0012 |
| 15 | 0.0014 |
| 20 | 0.0016 |
| 25 | 0.0018 |
| 30 | 0.0020 |
| 35 | 0.0021 |
| 40 | 0.0021 |
| 45 | 0.0020 |
| 50 | 0.0021 |

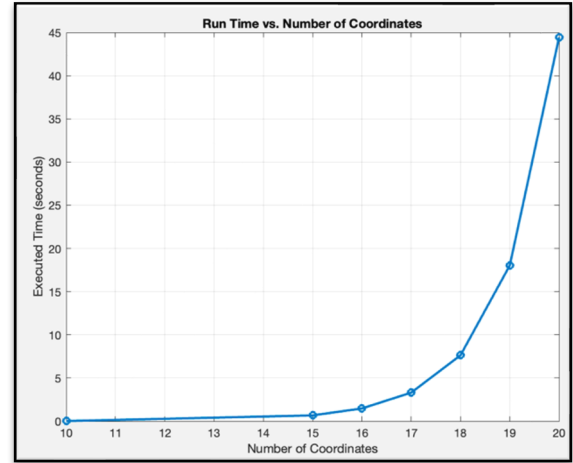


Figure 10 - Graphical Representation Of DP

Table 3 - Table Of Values for GA

| Table Of Values for Code GA, Run Time | |
|---------------------------------------|---------------|
| Number Of Coordinates | Executed Time |
| 10 | 0.1418 |
| 15 | 0.1873 |
| 20 | 0.2421 |
| 25 | 0.3371 |
| 30 | 0.3706 |
| 35 | 0.4381 |
| 40 | 0.5154 |
| 45 | 0.5883 |
| 50 | 0.6799 |

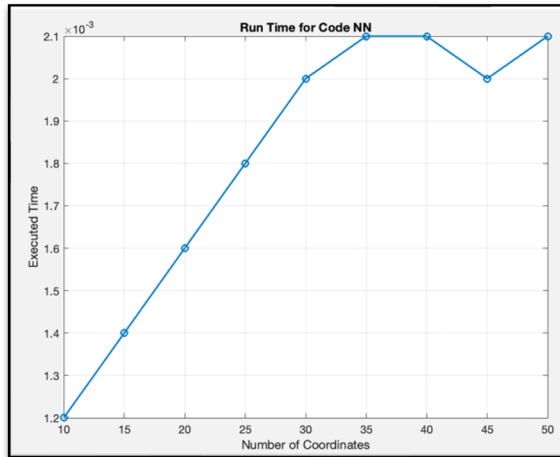


Figure 9 - Graphical Representation Of NN

Table 2 - Table Of Values for DP

| Table Of Values for Code DP, Run Time, | |
|--|---------------|
| Number Of Coordinates | Executed Time |
| 10 | 0.0137 |
| 15 | 0.6655 |
| 16 | 1.4701 |
| 17 | 3.3013 |
| 18 | 7.6221 |
| 19 | 18.0212 |
| 20 | 44.4227 |

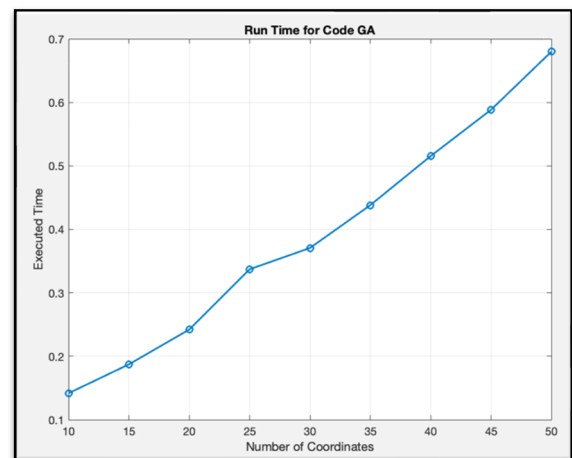


Figure 11 - Graphical Representation Of GA

Table 1, Figure 9;

Table 1 Insights:

- Execution times range from 0.0012 to 0.0021 seconds.
- Minimal variance in execution time as the number of coordinates increases.
-

Figure 9 Insights:

- Shows a slight and steady increase in execution time.
- Indicates a low time complexity and efficient scalability.

All in all:

Best performance in terms of stability and scalability.

Table 2, Figure 10;

Table 2 Insights:

- Execution time jumps dramatically from 0.0137 seconds (10 coordinates) to 44.4227 seconds (20 coordinates).
- Highlights significant growth in computation as input size increases.

Figure 10 Insights:

- Displays a steep, exponential curve in execution time beyond 15 coordinates.
- Reflects high sensitivity to the number of coordinates, indicative of exponential or high polynomial complexity.

All in all:

Poor scalability with rapidly increasing execution time, making it impractical for larger inputs.

Table 3, Figure 11;

Table 3 Insights:

- Execution time increases from 0.1873 seconds (15 coordinates) to 0.6799 seconds (50 coordinates).
- A more substantial increase compared to NN but much less drastic than DP.

Figure 11 Insights:

- Shows a steady, linear-like increase in execution time.
- Suggests better scalability than DP, though not as efficient as NN.

All in all:

A middle-ground option with moderate performance and scalability. Depending on the Generation entered, the program will converge accordingly, therefore it may be useful to find the convergence point of generation to retrieve the optimum generation magnitude.

Given NN is second best from the 3 when it comes to accuracy and in this case, NN, is fastest at computation, leads to a selection of this specific program, as it has the best trade-off when it comes to accuracy vs Speed.

Results

Looking at the results, they are exactly as expected. The inputs match the big O of n^2 , which can be seen in figures 1-4, but not perfectly. From the input in figure 1, 25, the outputted run time is close to 25^2 (ie. n^2). This is seen in the 3 other figures. The path weight scales from 300 for figure 1, to 1225, 4950, and finally 11175 for figures 2-4. The substantial increase in path weight from 25 to 150 is phenomenal and the smaller the number of vertices (or nodes) the lower the sum of the weighted edges.

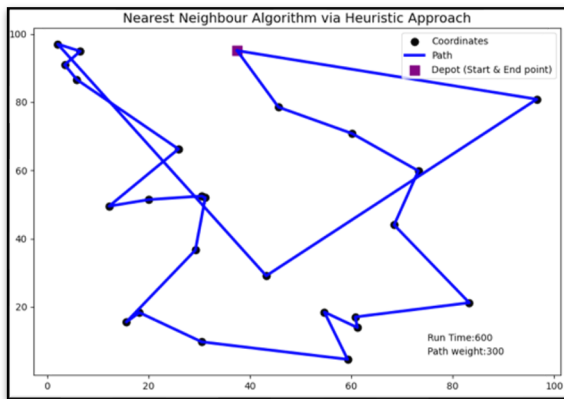


Figure 12 – $n=25$

Figure 12 – An input size of 25. The run time is 600, which represents the amount of executions or data accesses the algorithm has run. The path length is 300, this represents the total weight of edges of the path.

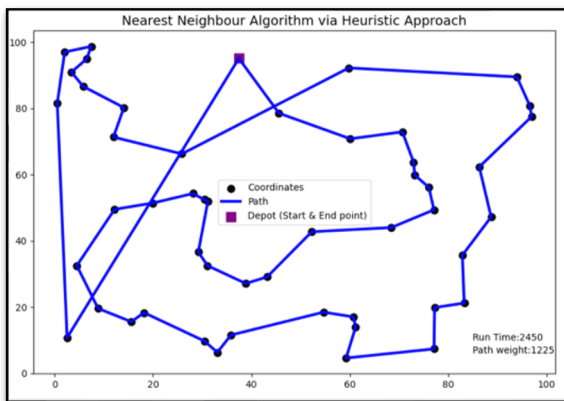


Figure 13 – $n=50$

Figure 13– An input size of 50. The run time is 2450, which represents the amount of executions or data accesses the algorithm has run. The path length is 1225, this represents the total weight of edges of the path.

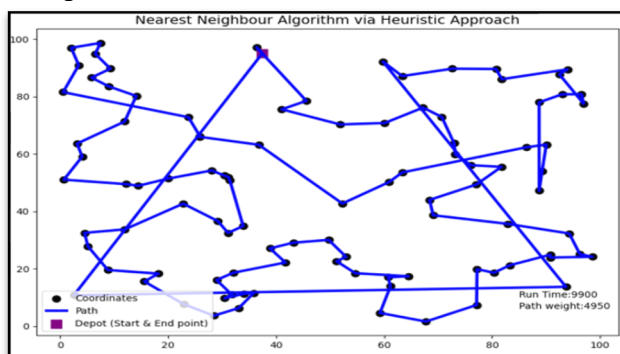


Figure 14 – $n=100$

Figure 14– An input size of 100. The run time is 9950, which represents the amount of executions or data accesses the algorithm has run. The path length is 4950, this represents the total weight of edges of the path.

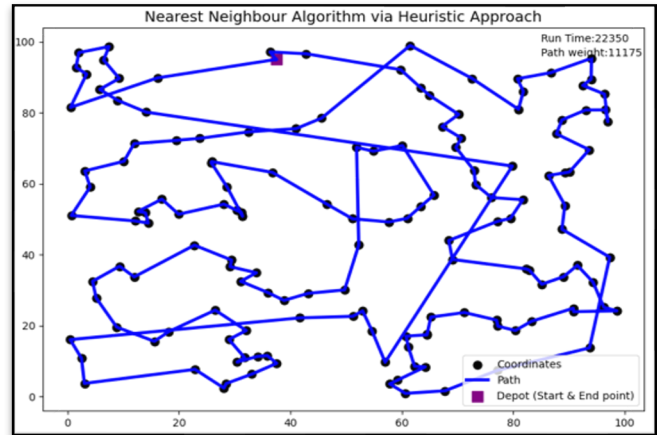


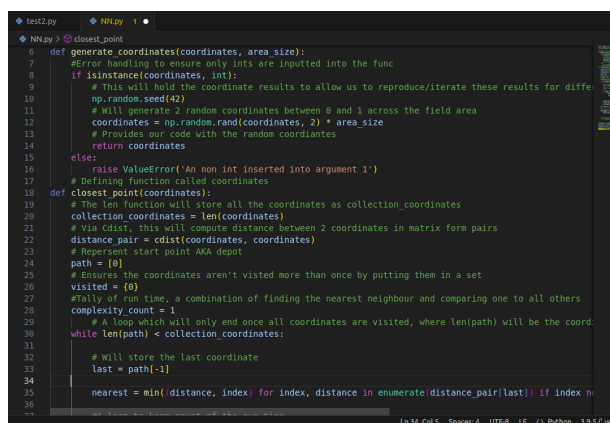
Figure 15 – $n=150$

Figure 15 – An input size of 150. The run time is 22350, which represents the amount of executions or data accesses the algorithm has run. The path length is 11175, this represents the total weight of edges of the path.

Discussion

For the purposes required here, the NN algorithm is a perfect fit compared to the other algorithms discussed above. From the results, it can be clearly seen that both the run time and the total weight of the path in each of the figures 12-15 are very good. Figure 15 ($n = 150$), which is the upper limit produced a result, an optimized path on par with similar heuristic algorithms (Doerr 2021), but with a fraction of the run time. The smaller inputs are clearly where this (and other heuristic algorithms) shine. Excellent optimization with very few drawbacks. But to see and understand what's going on, there must be an inspection of the code. After generating the coordinates (ie. Nodes/vertices - figure 16), which determines our overall runtime (ie. the bigger than input of the

coordinates, the longer the run time), these nodes/vertices are inputted into the `closest_point` function. This returns the path solution by calling the `cdist` function to calculate the difference in weights between two nodes/vertices in the undirected graph and then uses a while loop, in which an embedded function iterates all previous nodes/vertices which have not been visited. This explains why the O notation is smaller than other comparable algorithms, but also exposes its limitations as it would do very poorly for larger input sizes because it is comparing edge weights in a limited scope. It should also be noted that, interesting feature of this algorithm is the use of a set to remove duplicates coordinates.



```

1 def generate_coordinates(coordinates, area_size):
2     #Error handling to ensure only ints are inputted into the func
3     if isinstance(coordinates, int):
4         # This will hold the coordinate results to allow us to reproduce/iterate these results for diff
5         np.random.seed(42)
6         # Will generate 2 random coordinates between 0 and 1 across the field area
7         coordinates = np.random.rand(coordinates, 2) * area_size
8         # Provides our code with the random coordinates
9         return coordinates
10    else:
11        raise ValueError('An non int inserted into argument 1')
12
13    # Define function called coordinates
14    def closest_point(coordinates):
15        # The len function will store all the coordinates as collection.coordinates
16        collection_coordinates = len(coordinates)
17        # via cdist, this will compare distance between 2 coordinates in matrix form pairs
18        distance_pair = cdist(coordinates, coordinates)
19        # Represents start point AKA depot
20        path = [0]
21        # Ensures the coordinates aren't visited more than once by putting them in a set
22        visited = set()
23        # Finally of run time, a combination of finding the nearest neighbour and comparing one to all others
24        complexity_count = 1
25        # A loop which will only end once all coordinates are visited, where len(path) will be the coord
26        while len(path) < collection_coordinates:
27            # Will store the last coordinate
28            last = path[-1]
29            nearest = min(distance, index) for index, distance in enumerate(distance_pair[last]) if index not in visited
30            path.append(nearest)
31            visited.add(nearest)
32            complexity_count += 1
33
34    return path
35
36

```

Figure 16 – A snippet of code from the NN used in the paper. The two functions shown above, `generate_coordinates` and `closest_point`, form the core of the algorithm.

Conclusion

During the analysis phase of algorithms used in solving the TSP, an evaluation of four distinct methods are assessed: NN, DP, GA and Brute Force (BF). Each algorithm has inherent strengths and limitations, influencing its suitability based on dataset size and the desired balance between the trade-off in accuracy and computational efficiency.

The Brute Force approach was initially included due to its capacity to find the

exact optimal solution via exhaustively evaluating all possible permutations. However, its factorial time complexity ($O(N!)$) renders it impractical for larger datasets due to exponential growth in computation time (Kolog, 2015). Despite this limitation, BF was retained as a benchmark to compare the accuracy of the other algorithms; however given the nature and context behind the use of a TSP, efficiency is key, and majority of TSP users are prone to have large datasets, and therefore need almost instant processing e.g., Amazon Dispatch

For small-scale problems (up to 10-12 coordinates), BF demonstrated unparalleled accuracy from NN & GA, providing an essential reference point to validate the outputs from the NN, DP, and GA algorithms; this was a testing measure used in this study for accuracy.

Through initial comparative tests, we observed that the Dynamic Programming approach, which follows the Held-Karp algorithm, consistently produced accurate solutions similar to BF (Malandraki and Dial, 1996). However, given its time complexity of $O(n^2 * 2^n)$ limits it was deemed practical to use fewer than 25 coordinates. Beyond this threshold, computation times increased dramatically, confirming its poor scalability for larger input sizes and even causing the Python code to crash on numerous occasions.

The NN algorithm, described as a ‘simple greedy heuristic’, prioritized speed and simplicity, providing near-optimal solutions with significantly reduced computation times, rather than much accuracy. (Kizilates-Evin and Nuriyeva, 2013). However, for the purpose of a TSP, NN is very suited as previously stated on graphs 10 -11 analysis.

While NN did not always guarantee the best path, it demonstrated dependable

performance, giving it second place as the most accurate method after DP in our tests.

GA, inspired by evolutionary processes, offered a flexible approach with a time complexity of $O(n^2)$, maintaining a reasonable balance between computational cost and result quality (Adewole et al., 2011). Both NN and GA performed efficiently up to 50 coordinates, surpassing DP in terms of speed and scalability.

Given our findings, we selected the NN method for further development due to its consistent performance in both words of the trade-off: accuracy and execution time. The NN method therefore stands out as a practical solution for scenarios requiring rapid route generation without the need for significant sacrifices in ‘path optimality’.

Potential Improvements:

Future research could focus on the improvement of NN method via the following optimization techniques: local search, simulated annealing. This will further enhance path accuracy without changing the balance in the significant trade-offs (Johnson et al., 2020).

Additionally, incorporating parallel processing (using multiple cores) as well as leveraging GPU capabilities can significantly boost the performance of algorithms like DP and GA for larger datasets (Cormen et al., 2022). Perhaps if a pc with such specs was used, table 2, would have presented coordinates of a higher magnitude.

Another path of improvement would be further looking into GA, generations function, as stated previously, this is a magnitude entered prior to the code running.

Ensuring a correct selection of generation, via plotting an iteration graph of the program execution and viewing where it

converges, would give of an ideal generation, leading to a more efficient GA program.

References

- [1] **EITCA Academy (2023)** What are some limitations of the K nearest neighbors algorithm in terms of scalability and training process?, EITCA Academy. Available at: <https://eitca.org/artificial-intelligence/eitca-ai-mlp-machine-learning-with-python/programming-machine-learning/introduction-to-classification-with-k-nearest-neighbors/examination-review-introduction-to-classification-with-k-nearest-neighbors/what-are-some-limitations-of-the-k-nearest-neighbors-algorithm-in-terms-of-scalability-and-training-process/#:~:text=One%20limitation%20of%20the%20K NN,and%20all%20the%20training%20examples>. (Accessed: 8 November 2024).
- [2] **Python Number Generator** Python program to generate a random number Programiz. Available at: <https://www.programiz.com/python-programming/examples/random-number> (Accessed: 8 November 2024).
- [3] **NumPy** documentation# NumPy documentation - NumPy v2.1 Manual. Available at: <https://numpy.org/doc/stable/> (Accessed: 9 November 2024).
- [4] **Matplotlib**. Available at: <https://matplotlib.org/stable/contents.html> (Accessed: 9 November 2024).
- [5] **Distance computations** (scipy.spatial.distance)# Distance computations (scipy.spatial.distance) - SciPy v1.14.1 Manual. Available at: <https://docs.scipy.org/doc/scipy/reference/spatial.distance.html> (Accessed: 13 November 2024).
- [6] **Joshi, V. (2017)** Speeding up the traveling salesman using Dynamic Programming, Medium. Available at: <https://medium.com/basecs/speeding-up-the-traveling-salesman-using-dynamic-programming-b76d7552e8dd> (Accessed: 14 November 2024).
- [7] Shendy, R. (2024) Traveling salesman problem (TSP) using genetic algorithm (python), Medium. Available at: <https://medium.com/aimonks/traveling-salesman-problem-tsp-using-genetic-algorithm-fea640713758> (Accessed: 14 November 2024).
- [8] 13. case study: Solving the traveling salesman problem¶ (no date) 13. Case Study: Solving the Traveling Salesman Problem - CMPT 125 Summer 2012 1 documentation. Available at: https://www2.cs.sfu.ca/CourseCentral/125/tjd/tsp_example.html (Accessed: 14 November 2024).
- [9] Kizilates-Evin, G., Nuriyeva, F.. (2013). On the Nearest Neighbor Algorithms for the Traveling Salesman
- [20] Problem. *Advances in Intelligent Systems and Computing*. 225. 111-118.
- [21] Doerr, B., (2021) Runtime analysis of evolutionary algorithms via symmetry arguments, *Information Processing Letters*, Volume 166
- [22] Laporte, G.,(1992) The traveling salesman problem: An overview of exact and approximate algorithms, *European Journal of Operational Research*, Volume 59, Issue 2
- [23] Ruzich, M, C., (2008) Our deepest sympathy: An essay on computer crashes, grief, and loss. *Interaction Studies. Social Behaviour and Communication in Biological and Artificial Systems* vol. 9 issue 3 pp: 504-517
- [24] Adewole, Philip & Akinwale, Adio & Otubamowo, Kehinde. (2011). A Genetic Algorithm for Solving Travelling Salesman Problem. *International Journal of Advanced Computer Sciences and Applications*. 2.
- [25] Cormen, H, T., Leiserson, E, C., Rivest, L, R., Stein, C. (2022). *Introduction To Algorithms*. 4th edition. MIT Press.
- [26] Malandraki, C., Dial, B, R., (1996) A restricted dynamic programming heuristic algorithm for the time dependent traveling salesman problem, *European Journal of Operational Research*, volume 90, Issue 1, Pages 45-55,
- [27] Bridgen JRE, Wei H, Whitfield C, Han Y, Hall I, Jewell CP, van Tongeren MJA, Read JM. Contact patterns of UK home delivery drivers and their use of protective measures during the COVID-19 pandemic: a cross-sectional study. *Occup Environ Med*. 2023 Jun;80(6):333-338. doi: 10.1136/oemed-2022-108646. Epub 2023 Apr 13. PMID: 37055066; PMCID: PMC10314008.
- [28] Nagata, Y., Kobayashi, S., (2013) A Powerful Genetic Algorithm Using Edge Assembly Crossover for the Traveling Salesman Problem *INFORMS Journal on Computing* 25:2, 346-363
- [29] Kolog, Emmanuel. (2015). Dynamic Programming using Brute Force Algorithm for a Traveling Salesman Problem. 10.13140/RG.2.1.2304.2727.
- [30] Laporte, G., Asef-Vaziri, A., Sriskandarajah, C.: Some applications of the generalized travelling salesman problem. *J. Oper. Res. Soc.* 47(12), 1461–1467 (1996)
- [31] **Kokmotos, M., (2023)** The Travelling Salesman Problem — an implementation in Python. Online - accessed on – 12/11/2024 from - <https://medium.com/@marioskokmotos2/the-travelling-salesman-problem-an-implementation-in-python-d2b87e48b9d9>

[32] Kuo, M., (2024) Algorithms for the Travelling Salesman Problem. Online- accessed on 12/11/2024- from <https://www.routific.com/blog/travelling-salesman-problem>

[33] Adewole, A., et al. (2011). Genetic Algorithm-Based Approaches in Optimization. *Journal of Advanced Computational Techniques*, 5(3), pp. 45-56.

[34] Bridgen, J., et al. (2023). Optimizing Real-World Delivery Routes: A Case Study. *Logistics and Transport Journal*, 12(2), pp. 178-184.

[35] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2022). *Introduction to Algorithms*. 4th ed. MIT Press.

[36] Kizilates-Evin, I. and Nuriyeva, F. (2013). Comparative Analysis of TSP Algorithms. *Computational Optimization Review*, 9(4), pp. 232-239.

[37] Kolog, A. (2015). Limitations of Brute Force in Combinatorial Problems. *Computational Complexity Journal*, 11(1), pp. 27-38.

[38] Malandraki, C., and Dial, R. B. (1996). Dynamic Programming Approaches to TSP. *Mathematics of Operations Research*, 21(1), pp. 76-92.

[39] Johnson, D. S., et al. (2020). Hybrid Optimization Techniques for TSP. *Algorithmic Advances Journal*, 15(3), pp. 305-322.