Travel Route Planner

Jiamin Yao
SIUE
Department of Computer Science
Edwardsville, IL
jyao@siue.edu

Hangyu Yao
SIUE
Department of Computer Science
Edwardsville, IL
hyao@siue.edu

Yan Pang
SIUE
Department of Computer Science
Edwardsville, IL
ypang@siue.edu

***Abstract*—Efficient travel planning across Europe involves balancing distance, cost, and time. This project aims to optimize travel routes by analyzing three scenarios: helicopter shortest distance, minimum driving cost and time on regular versus highway roads, and cost-effective and time-efficient flight routes. Using a custom dataset with city coordinates, and synthetic driving and flight data, we implement three algorithms, Greedy algorithm, Divide and Conquer algorithm, and Dynamic Programming algorithm to compute optimal routes. Metrics like total distance, cost, and time are used to evaluate performance, providing insights for improving travel route optimization systems. Overall, the Dynamic Programming algorithm consistently yielded the most optimal results across scenarios, outperforming the Greedy and Divide and Conquer algorithms in terms of distance, cost, and time. This study provides insights for enhancing travel route optimization systems.**

I. Big Problem

Traveling is one of the most enjoyable activities, but planning an efficient route can be both complex and time-consuming, especially when visiting multiple cities across a continent like Europe. Travelers often struggle to balance factors like time, cost, and distance while making their decisions [1] **.**

Our project aims to simplify this process by developing a tool that optimizes travel routes across Europe based on the mode of travel. For helicopter trips, the tool focuses on minimizing distance. For driving trips, it provides a comparison between driving via highways or regular roads. For those choosing to book flights, the tool prioritizes minimizing flight costs and durations, offering the most efficient travel options.

By leveraging predefined city coordinates, highway tolls, flight costs, and flight durations stored in a dataset, the tool generates multiple optimized solutions using advanced algorithms, including Greedy algorithm, Divide and Conquer algorithm, and Dynamic Programming algorithm. Travelers can rely on this tool to streamline their travel planning, saving both time and money while ensuring the most efficient route through Europe.

II. Dataset

Our project aims to address the travel route planning challenge across Europe by optimizing routes based on different travel modes. We offer solutions for driving, where we optimize for the shortest distance and lowest road fees when considering both highway tolls and fuel costs, as well as for flying, where we focus on minimizing flight costs and flight durations.

To achieve this, we have built a custom dataset with 18 cities using a fixed map of Europe, which includes five key components:

1. Cities Coordinates:

Coordinates are generated through user interaction with a map, where users click to select city points. These coordinates not only can be calculated to relative distance between cities, but also can be used to draw the routes on the map.

The coordinates of cities are generated by user interaction with a pre-loaded map of Europe. Users click on various points on the map, which correspond to the locations of selected cities. These click points generate x and y coordinates that are then recorded in our dataset. The map in the project uses a size ratio of approximately 1:0.9 km, meaning that each unit of distance on the map corresponds to 0.9 kilometers in real-life distance.

Table.1 below displays the coordinates of 18 cities in Europe as generated through user interaction with the map. These coordinates represent the cities' relative positions on the map based on x and y values.

City	x-coordinate	y-coordinate
London	590	219
Paris	839	634
Milan	1519	1130
Vienna	2236	729
Barcelona	819	1689
Madrid	234	1819
Lyon	1085	1087
Berlin	1940	56
Venice	1832	1134
Rome	1849	1624
Munich	1758	740
Zurich	1455	852
Amsterdam	1094	81
Brussels	1039	326

Florence	1726	1367
Birmingham	414	62
Prague	2043	448
Nantes	448	875

Table.1. Cities Coordinates Data

Figure.1 below visually represents the x and y coordinates of the cities on a map of Europe. This plot helps provide a geographical context to the data, showing how the cities are spatially distributed across the continent. The relative positions of the cities are indicated by the markers, which reflect the user-generated coordinates.

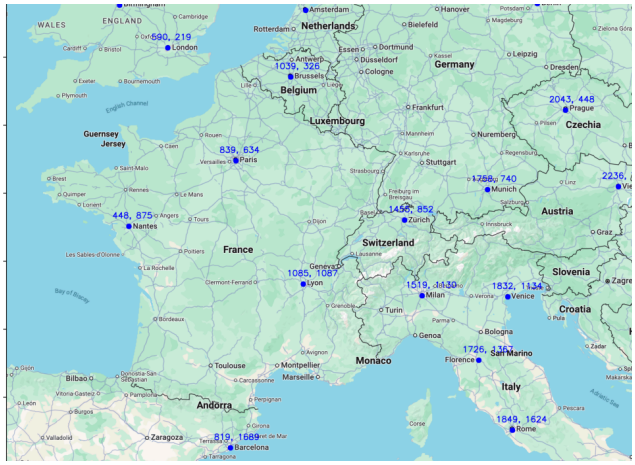

Figure.1 Europe Continent Map with Coordinates

2. Driving through Regular Road (Fuel Costs & Time):

For travel on regular roads, the cost is calculated based solely on fuel consumption, and the travel time reflects normal driving conditions. The fuel cost is estimated using the real country road distance between cities, assuming an average fuel consumption rate of 6 liters per 100 kilometers and an average fuel price of $1.5 per liter. Driving costs are calculated based on fuel consumption of 6 liters per 100 kilometers, and the fuel price per liter is collected from tolls.eu[2].

D_R = Regular Road Distance

S_R = Regular Road Speed Limits in Different Countries

F = Fuel Price Per Liter in Different Countries

The cost in dollars is generated using the following formula:

$$Cost = (D_R / S_R) * 6 * F$$

Travel time on regular roads assumes standard road conditions and normal driving speeds. Driving time is calculated using the speed limited from different countries. Speed limited data is collected from worlddata[3] but slightly changed. The time in minutes is generated using the formula:

$$Time = D_R / S_R$$

Table.2 below provides sample data for driving between six cities via regular roads, the whole dataset includes 18 cities. The data includes the cost in USD and the time in minutes for each journey. The driving cost is based on the formula that factors in fuel consumption and price, while the driving time is calculated using regular road speed.

From	To	Cost ($)	Time (mins)
London	Paris	60.80	378
Milan	Vienna	38.44	204
Barcelona	Munich	178.25	1193
Berlin	Prague	48.91	291
Rome	Zurich	135.41	757
Amsterdam	Florence	228.70	1252

Table.2 Driving through Regular Road Data Sample

3. Driving through Highway (Fuel Costs + Highway Tolls & Time):

For highway travel, the cost includes both fuel consumption and the toll charges. Highways typically provide faster travel times, so the time component reflects the reduced travel time compared to regular roads. Toll costs are added based on the specific route and country regulations.

Synthetic data is inspired by typical fuel consumption rates, toll costs, and faster travel times for highways. Highway travel reflects both the fuel cost and additional toll fees, with a time reduction due to higher speed limits and fewer stops.

Driving costs are calculated based on fuel consumption of 6 liters per 100 kilometers and the fuel price per liter is collected from tolls.eu[2]. Highway toll cost is collected from tollguru[4].

D_H = Highway Distance

S_H = Highway Speed Limits in Different Countries

F = Fuel Price Per Liter in Different Countries

T = Highway Tolls between Different Countries

The cost in dollars is generated using the following formula:

$$Cost = (D_H / S_H) * 6 * F + T$$

Driving time is calculated using the speed limits in different countries. Speed limited data is collected from worlddata[3] but slightly changed. The time in minutes is generated using the formula:

$$Time = D_H / S_H$$

Table.3 below provides sample data for driving between six cities via highway, the whole dataset includes 18 cities. The data includes the cost in USD and the time in minutes for each journey. The driving cost is based on the formula that factors in fuel consumption and price, while the driving time is calculated using highway speed.

From	To	Cost ($)	Time (mins)
London	Paris	74.2	236
Milan	Vienna	53.82	133
Barcelona	Munich	212.10	632
Berlin	Prague	49.54	202

Rome	Zurich	153.59	469
Amsterdam	Florence	207.15	728

Table.3 Driving through Highway Data Sample

4. Flight Costs:

The flight cost data is synthetic values based on real-world data, representing the cost of flights between selected cities. The flight cost is generated based on real-world patterns observed from services like Google Flights. These costs are meant to reflect common travel prices between European cities.

5. Flight Durations:

The flight durations are also synthetic based on real-world data, representing the time taken to fly between these cities. The flight duration generated based on real-world patterns observed from services Google Flights[5]. The duration data takes into account the typical flight times between European cities, considering air routes, and layover possibilities (though we simplified this for direct flights).

Table.4 below shows sample data for the cost and duration of flights between six cities in Europe, the whole dataset includes 18 cities. The costs are given in USD, and the durations are in minutes.

From	To	Cost ($)	Time (mins)
London	Paris	83	85
Milan	Vienna	129	210
Barcelona	Munich	157	125
Berlin	Prague	108	210
Rome	Zurich	114	95
Amsterdam	Florence	315	120

Table.4 Flight Cost and Duration Data Sample

By leveraging this dataset, our tool optimizes travel routes to provide the shortest distance and lowest costs for both driving and flying, helping travelers plan efficient multi-city trips across Europe.

III. Methodology

Figure.2 illustrates the block diagram for three travel experiments designed to evaluate the performance of three algorithms: Greedy algorithm, Divide and Conquer algorithm, and Dynamic Programming algorithm. Each experiment investigates a different mode of travel—helicopter, car, and flight—while using the same set of algorithms to compare efficiency and effectiveness in optimizing travel distance, time, or cost.

In the first experiment, the focus is on helicopter travel, where the input consists of cities' coordinates. The three algorithms are applied to find the shortest travel path between cities using these coordinates. The output includes a table of the city visit sequence, a visual representation of the helicopter travel path, the total distance traveled, and a graph comparing the performance of the three algorithms in determining the shortest possible helicopter distance.

The second experiment shifts to car travel. Here, the input includes not only the cities' coordinates but also data on regular road and highway costs and travel times. The three algorithms are again used to determine the optimal path based on the least driving cost or time. The output from this experiment consists of a table outlining the city visit sequence, a figure showing the driving path, the total driving cost and time, and a graph comparing the algorithms' performance in terms of both regular road and highway travel efficiency.

In the third experiment, flight travel is analyzed. The input involves cities' coordinates along with flight costs and time. The same three algorithms are employed to identify the most cost-effective or time-efficient flight path between cities. The output includes a table of the city visit sequence, a figure illustrating the flight path, the total flight cost and duration, and a comparison of the algorithms based on both flight cost and duration.

The center of the diagram shows the Greedy algorithm, Divide and Conquer algorithm, and Dynamic Programming algorithm, applied across all three experiments. This demonstrates that each mode of travel—helicopter, car, or flight—is evaluated using these algorithms to provide a comprehensive analysis of their performance in optimizing different travel criteria, including distance, time, and cost.

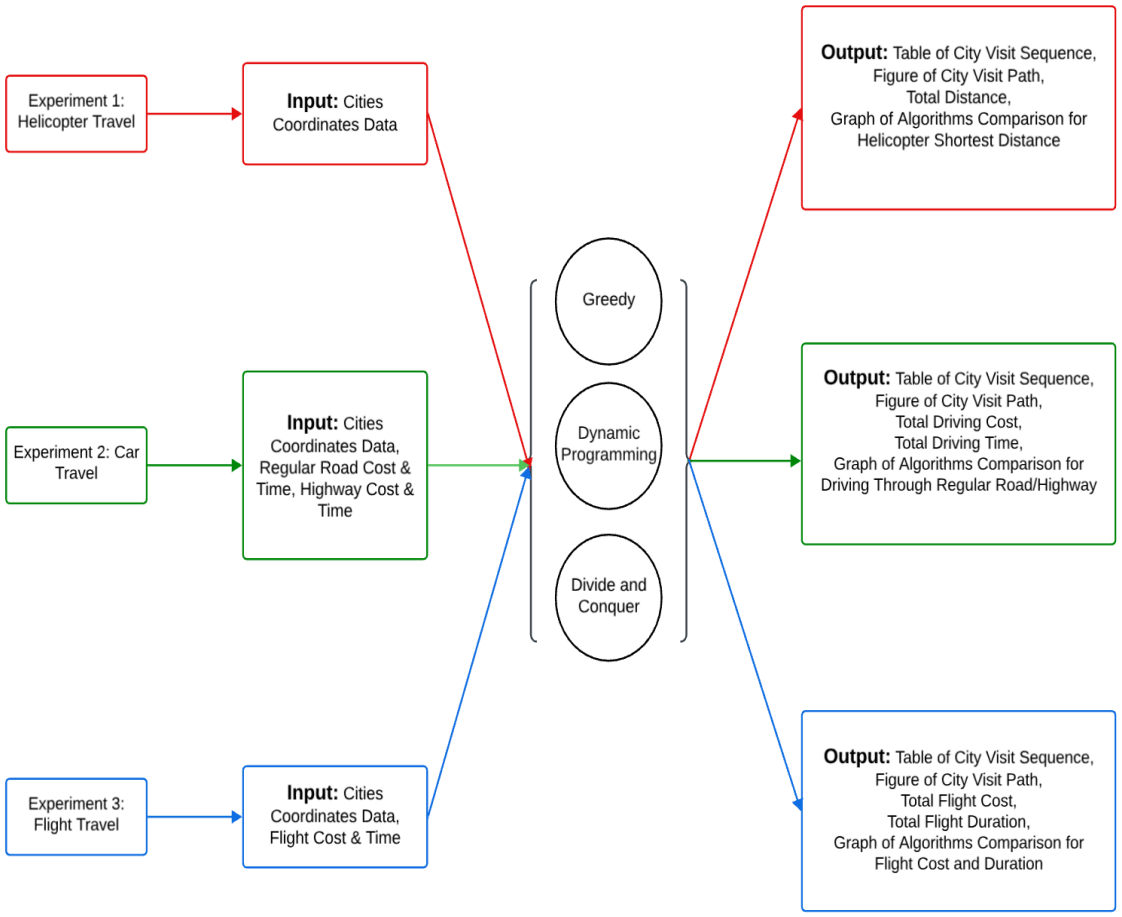

Figure.2 Block Diagram for Three Experiments

The Greedy algorithm, Divide and Conquer algorithm, and Dynamic Programming algorithm aim to optimize the path in terms of distance, time, or cost, depending on the experiment's design. Since we are using coordinates on a map, the shortest distance can be computed using the Euclidean distance formula. For driving or flying, we already have pre-calculated values for time and cost based on factors like fuel costs, driving speeds, and other data mentioned in the dataset.

General Equation for Distance:

For any two cities i and j, the distance is calculated using the Euclidean distance formula:

$$d(i,j) = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}$$

Where x_i,y_i are the coordinates of city i, and x_j,y_j are the coordinates of city j.

The **Greedy** algorithm starts from a city and iteratively chooses the closest unvisited city until all cities are visited.

The equation for the Greedy algorithm is as follows:

At each step, the next city is chosen based on:

$$minGreedy = min\{d(currentCity, j)\}$$

Where j is an unvisited city.

The time complexity of this approach is $O(n^2)$, as for each city, finding the nearest unvisited one takesO(n). Space Complexity is O(n), for storing the list of visited cities and the travel path.

Table.5 presents the pseudocode for Greedy algorithm.

Pseudocode for Greedy Algorithm:
1. Start at the first city
2. Mark the starting city as visited
3. While there are unvisited cities:
4. Find the nearest/lowest price/shortest time unvisited city
5. Travel to that city
6. Mark the city as visited
7. Return the minimum distance/cost/time found

Table.5 Pseudocode for Greedy Algorithm

The **Divide and Conquer** algorithm[6] divides the cities into two halves recursively, solving for the shortest pair of cities in each half, and then merging the results to ensure the shortest path is found across the dividing line.

The equation for the Divide and Conquer algorithm is as follows:

For a set of n cities, the Divide and Conquer algorithm splits them into two parts and recursively computes the minimum distance/lowest price/shortest time:

$$minDC = min(left, right, strip)$$

Where "left" and "right" are the shortest distances within each half, and "strip" represents the shortest distance across the two halves.

The time complexity is O(nlogn), due to the recursive splitting and merging process. The space complexity is O(n),

as space is needed to store the cities and handle the recursive calls.
Table.6 presents the pseudocode for Divide and Conquer algorithm.

Pseudocode for Divide and Conquer Algorithm:
1. Divide:
2. Sort cities by y-coordinate
3. Split cities into two halves
4. Recursively divide until each subset has 2 or 3 cities
5. Conquer:
6. Use the brute force to find shortest/lowest price/shortest time path for small subsets
7. Combine:
8. Check for closer paths between the two halves
9. Return the minimum distance/cost/time found

Table.6 Pseudocode for Divide and Conquer Algorithm

The **Dynamic Programming**[7] algorithm solves the TSP by using a table to store the minimum cost of visiting subsets of cities, building up to the entire set.
Equation for Dynamic Programming algorithm as follows:
Let dp[mask][i] represent the minimum distance to visit all cities in mask and i end at city. The recurrence relation is:

$$dp[mask][i] = min(dp[mask \backslash (1 << i)][j],\ d(j,i))$$

Where d(j,i) is the distance between cities j and i, and mask represents a subset of visited cities.
The time complexity of this approach is $O(n^2 \cdot 2^n)$, as it computes the minimum path for each subset of cities. The space complexity is also $O(n \cdot 2^n)$, due to the size of the DP table.
Table.7 presents the pseudocode for Dynamic Programming algorithm.

Pseudocode for Dynamic Programming Algorithm:
1. Initialize dp[mask][i], where the mask is the set of visited cities and i is the current city
2. Set base case: dp[1][0] = 0 (starting at city 0)
3. For each subset of cities:
4. For each city u:
5. For each unvisited city v:
6. Update dp[mask | (1 << v)][v]
7. Once all cities are visited, return to the starting city
8. Reconstruct the path by backtracking

Table.7 Pseudocode for Dynamic Programming Algorithm

IV. Experiment Design

To evaluate and optimize travel routes across Europe by analyzing three key factors:

1. Helicopter shortest distance based on city coordinates.
2. Minimum cost and shortest time for driving through regular and highway roads.
3. Minimum cost and shortest time for flights between cities.

Experiment 1: Helicopter Shortest Distance Based on City Coordinates

The goal of the first experiment is to calculate the shortest helicopter distance between 18 cities using their coordinates. We approximate the straight-line distance between cities by applying the Euclidean distance. These distances are treated as edges in a graph, where each city is a node, and the Traveling Salesman Problem (TSP)[8] is applied to determine the optimal route that visits each city once, minimizing the total distance.
We implement and compare Greedy algorithm, Divide and Conquer algorithm, and Dynamic Programming algorithm to find the optimal routes for helicopter travel. For the 18 cities, we will show the sequence of cities visited and visualize the route on a map to illustrate how each algorithm determines the optimal or near-optimal path.
After applying the algorithms to the 18 cities, we will vary the number of cities (from 6 to 18, increasing by 3) to compare their performance. The comparison focuses on three key evaluation metrics:
Total Distance: The minimum total distance in km calculated by each algorithm.
Time Complexity: The computational efficiency of each algorithm as the input size (number of cities) changes.
Space Complexity: The memory usage required by each algorithm to store and process city coordinates and distances.
This experiment provides a baseline for comparison with other factors, such as cost and time, in different travel way.

Experiment 2: Minimum Cost and Shortest Time for Driving Through Regular and Highway Roads

The goal of the second experiment is to determine the most cost-effective and time-efficient driving route when considering both regular roads and highways when driving through 18 cities. We compare two driving scenarios: regular roads, where costs are based on fuel consumption and time is calculated using lower speeds (60 km/h), and highways, where tolls are added to fuel costs and higher speeds (100 km/h) are considered.
We implement and compare Greedy algorithm, Divide and Conquer algorithm, and Dynamic Programming algorithm to find the optimal routes. Simulations are run to compare total costs and travel times for driving through regular roads versus highways, aiming to minimize either cost or time.
For the 18 cities, we show the sequence of cities visited and visualize the route on a map. Afterward, we vary the number of cities (from 6 to 18, increasing by 3) to assess the performance of the algorithms.
The evaluation metrics for this experiment are:
Total Driving Cost: The sum of driving costs in dollars for the selected route.
Total Driving Time: The time in mins required to complete the route based on road speeds.
Time Complexity: The efficiency of each algorithm as the number of cities changes.
Space Complexity: The memory usage required for storing cost, distance, and toll data.

This experiment offers practical insights into real-world travel choices, where drivers must balance lower costs and slower regular roads against faster, more expensive highways.

Experiment 3: Minimum Cost and Shortest Time for Flights Between Cities

The goal of this experiment is to find the most cost-effective and time-efficient flight routes between 18 cities. Using a matrix of synthetic flight costs and durations, we apply Greedy algorithm, Divide and Conquer algorithm, and Dynamic Programming algorithm to identify the optimal route, minimizing either flight costs or travel time.

For the 18 cities, we will show the sequence of cities visited and visualize the routes on a map. This example illustrates how each algorithm identifies the optimal or near-optimal path for minimizing costs or time. After analyzing the 18 cities, we will vary the number of cities (from 6 to 18, increasing by 3) to compare the performance of the three algorithms under different input sizes.

For the 18 cities, we will show the sequence of cities visited and visualize the routes on a map. Afterward, we vary the number of cities (from 6 to 18, increasing by 3) to assess the performance of the algorithms.

The evaluation metrics for this experiment are:

Total Flight Cost: The sum of flight costs in dollars for the selected routes.

Total Flight Duration: The time in mins required to complete the selected flight routes.

Time Complexity: The efficiency of each algorithm as the number of cities changes.

Space Complexity: The memory usage required for storing flight cost and duration matrices.

This experiment explores the trade-offs between cost and time, providing valuable insights into travelers' decisions when selecting flight routes.

V. Results and Discussion

Experiment 1: Helicopter Shortest Distance Based on Coordinates.

We applied three algorithms – Greedy algorithm, Divide and Conquer algorithm, and Dynamic Programming algorithm – to find the shortest distance for 18 cities.

The **Greedy** algorithm starts at the first city, selects the nearest unvisited city, and repeats this process until all cities are visited, returning to the starting city at the end. The total distance for 18 cities is 10,831.7 units, which converts to 9,748.53 km (using a size ratio of 1:0.9 km).

This approach is intuitive and simple but does not guarantee the globally optimal solution. The resulting path may be suboptimal compared to more advanced algorithms.

The Table.8 and Figure.3 below show 18 Cities Visit Sequence and path for Helicopter Shortest Distance using Greedy Algorithm.

Greedy Algorithm

Index	City Name	x-coordinate	y-coordinate
0	London	590	219
1	Birmingham	414	62
2	Brussels	1039	326
3	Amsterdam	1094	81
4	Paris	839	634
5	Nantes	448	875
6	Lyon	1085	1087
7	Milan	1519	1130
8	Zurich	1455	852
9	Munich	1758	740
10	Venice	1832	1134
11	Florence	1726	1367
12	Rome	1849	1624
13	Vienna	2236	729
14	Prague	2043	448
15	Berlin	1940	56
16	Barcelona	819	1689
17	Madrid	234	1819

Table.8 City Visit Sequence for Helicopter Shortest Distance using Greedy Algorithm

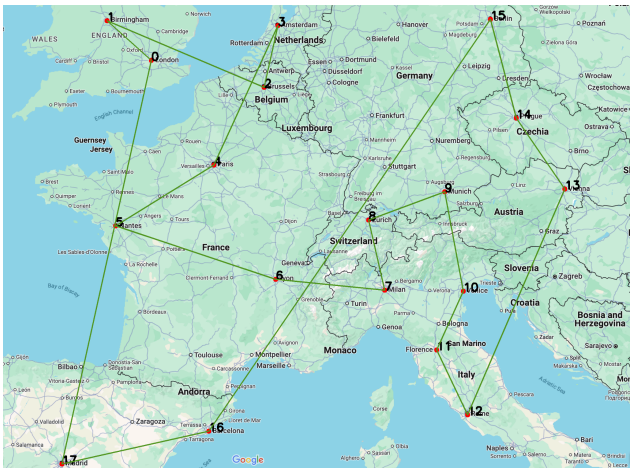

Figure.3 City Visit Path for Helicopter Shortest Distance using Greedy Algorithm

The **Divide and Conquer** algorithm splits cities into two halves based on their x-coordinates, recursively solving smaller subsets and combining the results. The total distance for 18 cities is 14,260.4 units, which converts to 12,834.36 km (using a size ratio of 1:0.9 km).

The Divide and Conquer algorithm helps reduce the complexity of finding the shortest path, especially when dealing with large datasets. However, it is mainly useful for finding the closest pairs of cities, rather than solving the full TSP problem.

The Table.9 and Figure.4 below show 18 Cities Visit Sequence and path for Helicopter Shortest Distance using Divide and Conquer algorithm.

Divide and Conquer Algorithm			
Index	City Name	x-coordinate	y-coordinate
0	London	590	219
1	Nantes	448	875
2	Berlin	1940	56
3	Birmingham	414	62
4	Amsterdam	1094	81
5	Brussels	1039	326
6	Prague	2043	448
7	Paris	839	634
8	Vienna	2236	729
9	Munich	1758	740
10	Zurich	1455	852
11	Lyon	1085	1087
12	Milan	1519	1130
13	Venice	1832	1134
14	Florence	1726	1367
15	Rome	1849	1624
16	Barcelona	819	1689
17	Madrid	234	1819

Table.9 City Visit Sequence for Helicopter Shortest Distance using Divide and Conquer Algorithm

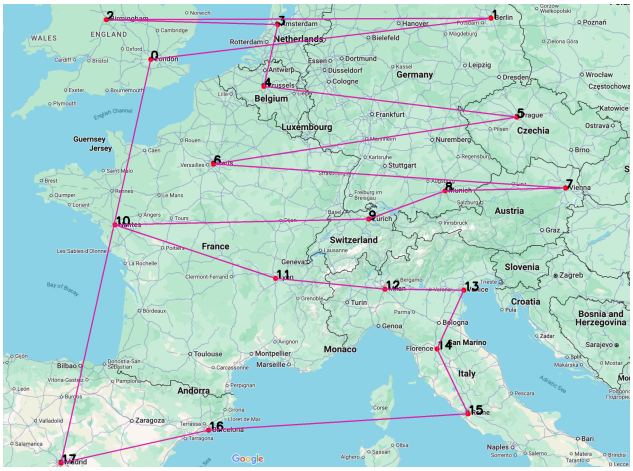

Figure.4 City Visit Path for Helicopter Shortest Distance using Divide and Conquer Algorithm

The **Dynamic Programming** algorithm using Bitmasking, this approach finds the globally optimal solution. For 18 cities, the total distance is 8,619.13 units, which converts to 7,757.22 km (using a size ratio of 1:0.9 km).
The Table.10 and Figure.5 above show 18 Cities Visit Sequence and path for Helicopter Shortest Distance using Dynamic Programming algorithm.

Dynamic Programming Algorithm			
Index	City Name	x-coordinate	y-coordinate
0	London	590	219
1	Paris	839	634
2	Nantes	448	875
3	Madrid	234	1819
4	Barcelona	819	1689
5	Lyon	1085	1087
6	Zurich	1455	852
7	Milan	1519	1130
8	Florence	1726	1367
9	Rome	1849	1624
10	Venice	1832	1134
11	Munich	1758	740
12	Vienna	2236	729
13	Prague	2043	448
14	Berlin	1940	56
15	Amsterdam	1094	81
16	Brussels	1039	326
17	Birmingham	414	62

Table.10 City Visit Sequence for Helicopter Shortest Distance using Dynamic Programming Algorithm

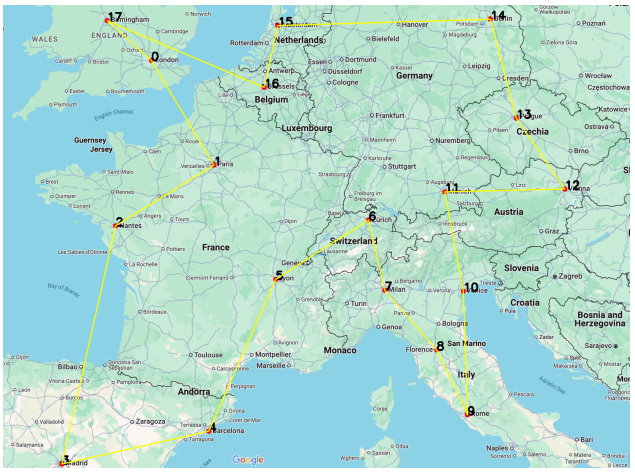

Figure.5 City Visit Path for Helicopter Shortest Distance using Dynamic Programming Algorithm

Lastly, we varied the number of cities from 6 to 18 (increasing by 3), applying all three algorithms. Table.11 shows the total distances values in kilometers for each algorithm.
When we convert these values to kilometers (km) using the ratio (1:0.9 km), the results in Table.11 provide clearer insights into the efficiency of each algorithm as the number of cities increases.

Number of Cities	Distance use Greedy (km)	Distance use Divide and Conquer (km)	Distance use Dynamic Programming (km)
6	5487.42	5255.93	5255.93
9	6547.51	8416.39	5839.31

12	7978.06	8777.61	6964.25
15	8777.50	9022.24	7189.11
18	9748.53	13038.6	7757.22

Table.11 Helicopter Shortest Distance (km) using Three Algorithms

The data from Table.8 is visualized in Figure.6, which compares the shortest distances for each algorithm across varying city counts.

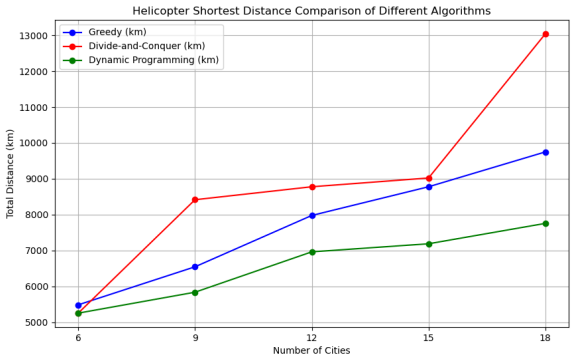

Figure.6 Helicopter Shortest Distance (km) Comparison of Three Algorithms

As shown in Table.8 and Figure.6, Dynamic Programming consistently provides the shortest distance across all input sizes.

The Dynamic Programming algorithm Outperforms Other Algorithms. For example, with 18 cities, Dynamic Programming yields a total trip distance of 7757.22 km, which is significantly shorter than the Greedy (9748.53 km) and Divide and Conquer (13038.6 km) algorithm. This trend holds for all other city counts (6 to 18), demonstrating the efficiency of the Dynamic Programming algorithm in solving the Traveling Salesman Problem (TSP) optimally.

This algorithm guarantees the shortest path, but it has a time complexity of $O(n^2 \cdot 2^n)$ due to the bitmasking approach for subsets and the recursive path reconstruction, makes it less practical for larger datasets.

The space complexity is $O(n \cdot 2^n)$ because we need to store the DP table that tracks all possible subsets of cities and their respective shortest paths.

The Dynamic Programming algorithm guarantees the optimal solution to the TSP problem but has a higher computational complexity, making it impractical for large datasets.

The Greedy algorithm performs better than Divide and Conquer but is still outperformed by Dynamic Programming. For instance, with 15 cities, the Greedy algorithm results in a total trip distance of 8777.50 km, compared to 7189.11 km for Dynamic Programming algorithm. Although Greedy's performance improves with fewer cities, it fails to find the shortest possible distance, as it prioritizes local optima rather than the global optimal solution.

The greedy algorithm checks every unvisited city for the closest one, making it an $O(n^2)$ time complexity, where n is the number of cities.

The space complexity is O(n), where n is the number of cities, due to storing the list of visited cities and distances.

Interestingly, the Divide and Conquer algorithm performs poorly compared to the other two, especially as the number of cities increases. For 18 cities, the Divide and Conquer algorithm produces a total distance of 12834.36 km, far exceeding the other algorithms.

This suboptimality arises from its Divide and Conquer algorithm, which, while efficient in some problem domains, struggles to maintain an optimal overall route when dealing with the global nature of the TSP.

The Divide and Conquer algorithm operates in O(nlogn) time complexity because it sorts the cities and divides them recursively.

The space complexity is O(n) due to the recursive division and the space required to store city coordinates and distances.

For all algorithms, the total distance naturally increases as the number of cities grows. However, the rate of increase is different across algorithms. For example, the total distance for Greedy grows from 5487.42 km for 6 cities to 9748.53 km for 18 cities, while Divide and Conquer algorithm jumps from 5255.93 km to 13038.6 km for the same range. Dynamic Programming algorithm shows the smallest rate of increase, which reflects its efficiency in managing large input sizes.

Experiment 2: Minimum Cost and Shortest Time for Driving Through Regular and Highway Roads

The second experiment compared driving scenarios on regular roads and highways across 18 cities to identify the most cost-effective and time-efficient driving routes.

In this experiment, we used the **Greedy** algorithm to generate different routes for both regular and highway routes based on whether we wanted to achieve the minimum cost or the minimum time. The Greedy algorithm prioritized local optimization by selecting the least expensive or fastest option for each city transition without considering the global implications of the route. So the resulting path may be suboptimal.

The Table 12 and Figure 7-10 below shows the 18 cities' visit sequence and path for regular and highway roads shortest distance using Greedy Algorithm.

You can see that the difference between them exists, but it is very small.

Greedy Algorithm				
	Regular Road		Highway	
	Min Cost	Min Time	Min Cost	Min Time
Index	City Name	City Name	City Name	City Name
0	London	London	London	London
1	Birmingham	Birmingham	Birmingham	Birmingham
2	Brussels	Brussels	Brussels	Brussels
3	Amsterdam	Amsterdam	Amsterdam	Amsterdam
4	Paris	Paris	Paris	Paris
5	Nantes	Nantes	Nantes	Nantes

6	Lyon	Lyon	Lyon	Lyon
7	Zurich	Milan	Zurich	Zurich
8	Milan	Venice	Milan	Milan
9	Venice	Florence	Venice	Venice
10	Florence	Rome	Florence	Florence
11	Rome	Munich	Rome	Rome
12	Munich	Prague	Munich	Munich
13	Prague	Vienna	Vienna	Prague
14	Vienna	Berlin	Prague	Vienna
15	Berlin	Zurich	Berlin	Berlin
16	Barcelona	Barcelona	Barcelona	Barcelona
17	Madrid	Madrid	Madrid	Madrid

Table.12 City Visit Sequence for Driving using Greedy Algorithm

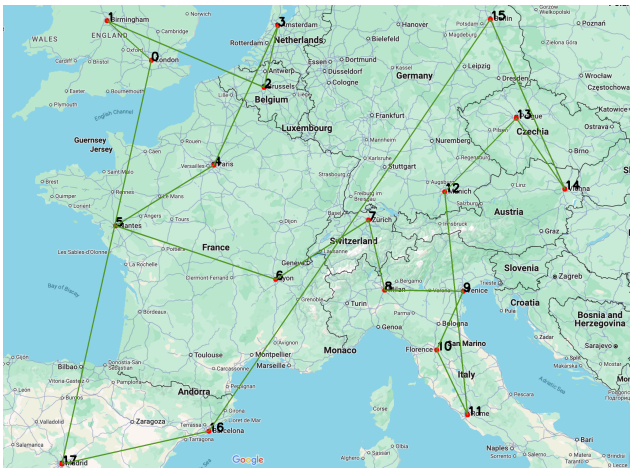

Figure.7 City Visit Path for Driving Min Cost in Regular Road using Greedy Algorithm

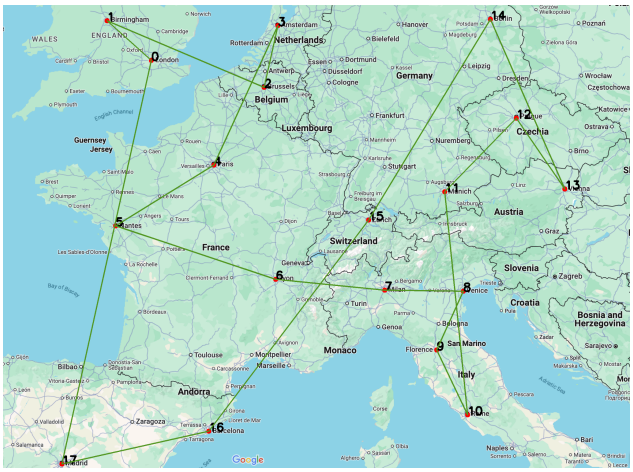

Figure.8 City Visit Path for Driving Min Time in Regular Road using Greedy Algorithm

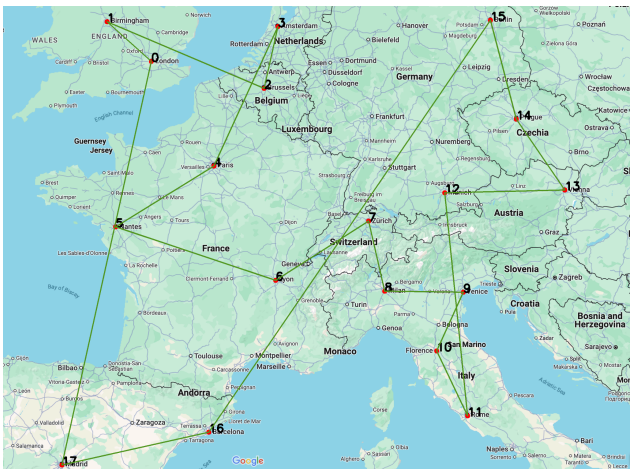

Figure.9 City Visit Path for Driving Min Cost in Highway using Greedy Algorithm

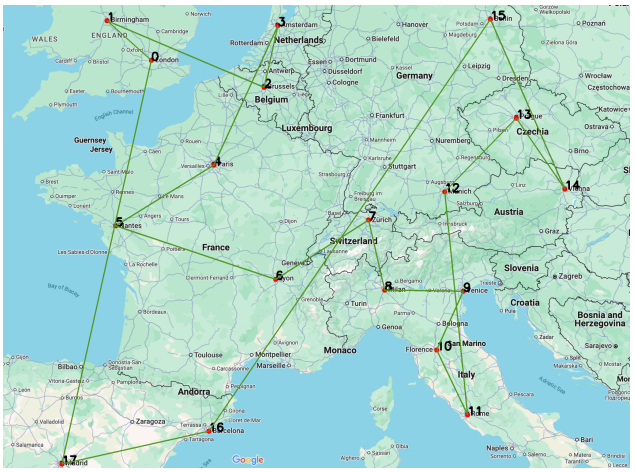

Figure.10 City Visit Path for Driving Min Time in Highway using Greedy Algorithm

The **Divide and Conquer** algorithm divides the 18 cities into smaller sets, solves them independently, and then merges the results. That is, it always prioritizes solving neighboring city pairs and does not consider the overall situation. Especially when dealing with larger data sets, such as 18 cities, it cannot come up with a globally optimal solution.

The Table 13 and Figure 11-14 below shows the 18 cities' visit sequence and path for regular and highway roads shortest distance using Divide and Conquer Algorithm.

You can see that the final routes we get are exactly the same whether we take the highway or the ordinary road. This result has something to do with the implementation of the algorithm, because no matter what, they will always be divided into the same small set of cities, so they will get the same result.

Divide and Conquer Algorithm				
	Regular Road		Highway	
	Min Cost	Min Time	Min Cost	Min Time
Index	City Name	City Name	City Name	City Name
0	London	London	London	London

1	Paris	Paris	Paris	Paris
2	Milan	Milan	Milan	Milan
3	Vienna	Vienna	Vienna	Vienna
4	Barcelona	Barcelona	Barcelona	Barcelona
5	Madrid	Madrid	Madrid	Madrid
6	Lyon	Lyon	Lyon	Lyon
7	Berlin	Berlin	Berlin	Berlin
8	Venice	Venice	Venice	Venice
9	Rome	Rome	Rome	Rome
10	Munich	Munich	Munich	Munich
11	Zurich	Zurich	Zurich	Zurich
12	Amsterdam	Amsterdam	Amsterdam	Amsterdam
13	Brussels	Brussels	Brussels	Brussels
14	Florence	Florence	Florence	Florence
15	Birmingham	Birmingham	Birmingham	Birmingham
16	Prague	Prague	Prague	Prague
17	Nantes	Nantes	Nantes	Nantes

Table.13 City Visit Sequence for Driving using Divide and Conquer Algorithm

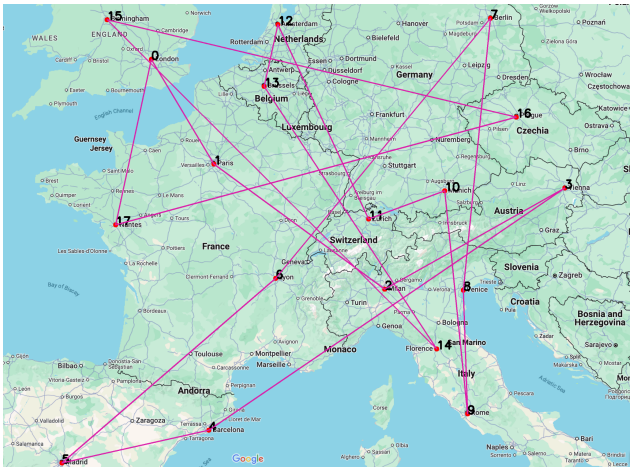

Figure.11 City Visit Path for Driving Min Cost in Regular Road using Divide and Conquer Algorithm

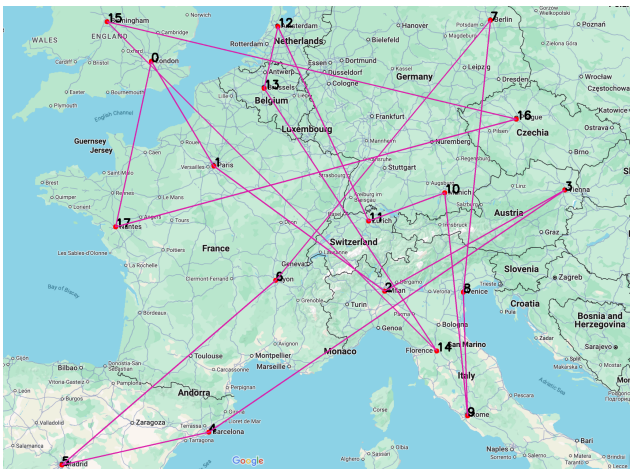

Figure.12 City Visit Path for Driving Min Time in Regular Road using Divide and Conquer Algorithm

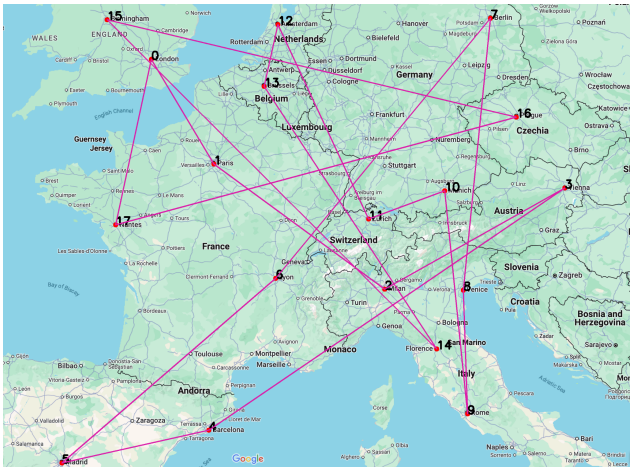

Figure.13 City Visit Path for Driving Min Cost in Highway using Divide and Conquer Algorithm

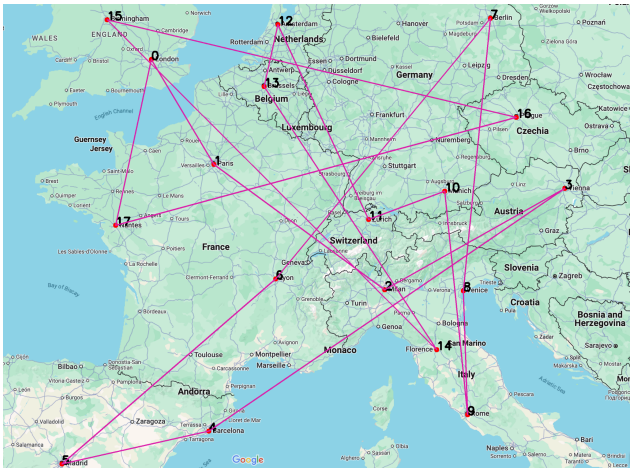

Figure.14 City Visit Path for Driving Min Time in Highway using Divide and Conquer Algorithm

The **Dynamic Programming** algorithm explored all possible routes systematically and stored intermediate results to identify the globally optimal solution for both cost and time.

The Table 14 and Figure 11-14 below shows the 18 cities' visit sequence and path for regular and highway roads shortest distance using Dynamic Programming Algorithm.

The Dynamic Programming algorithm produced the same sequence of cities in all four cases, from London (the starting city) to Paris (the ending city), with a small deviation near the end of the route. The shortest time route on both regular roads and highways swapped the positions of Florence and Rome compared to the lowest cost route. This suggests that time priority slightly changed the route in the final stage, probably due to speed differences on specific roads between these cities.

Dynamic Programming Algorithm				
	Regular Road		Highway	
	Min Cost	Min Time	Min Cost	Min Time
Index	City Name	City Name	City Name	City Name
0	London	London	London	London
1	Birmingham	Paris	Birmingham	Paris
2	Brussels	Brussels	Brussels	Brussels
3	Amsterdam	Amsterdam	Amsterdam	Amsterdam
4	Berlin	Berlin	Berlin	Berlin
5	Prague	Prague	Prague	Prague
6	Vienna	Vienna	Vienna	Vienna
7	Munich	Munich	Munich	Munich
8	Venice	Venice	Venice	Venice
9	Florence	Rome	Florence	Rome
10	Rome	Florence	Rome	Florence
11	Milan	Milan	Milan	Milan
12	Zurich	Zurich	Zurich	Zurich
13	Lyon	Lyon	Lyon	Lyon
14	Barcelona	Barcelona	Barcelona	Barcelona
15	Madrid	Madrid	Madrid	Madrid
16	Nantes	Nantes	Nantes	Nantes
17	Paris	Birmingham	Paris	Birmingham

Table.14 City Visit Sequence for Driving using Dynamic ProgrammingAlgorithm

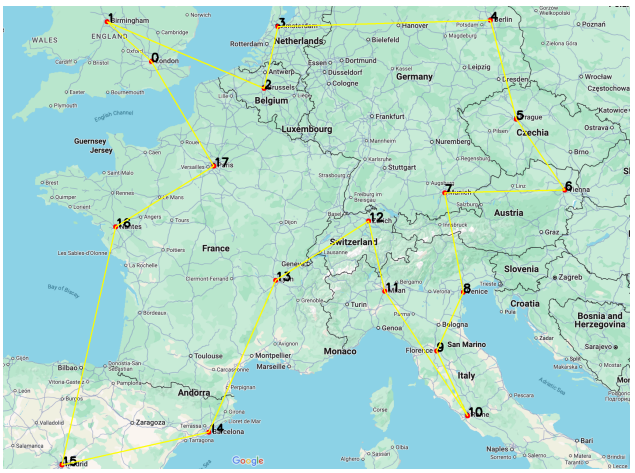

Figure.15 City Visit Path for Driving Min Cost in Regular Road using Dynamic Programming Algorithm

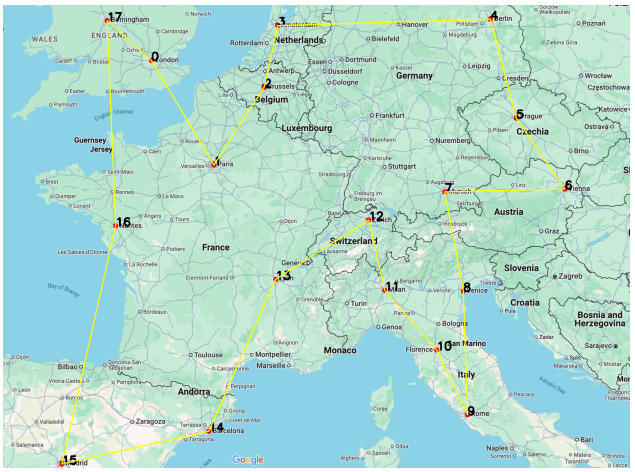

Figure.16 City Visit Path for Driving Min Time in Regular Road using Dynamic Programming Algorithm

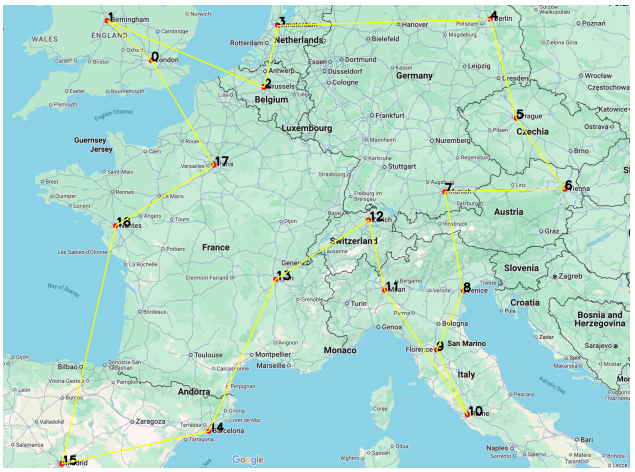

Figure.17 City Visit Path for Driving Min Cost in Highway using Dynamic Programming Algorithm

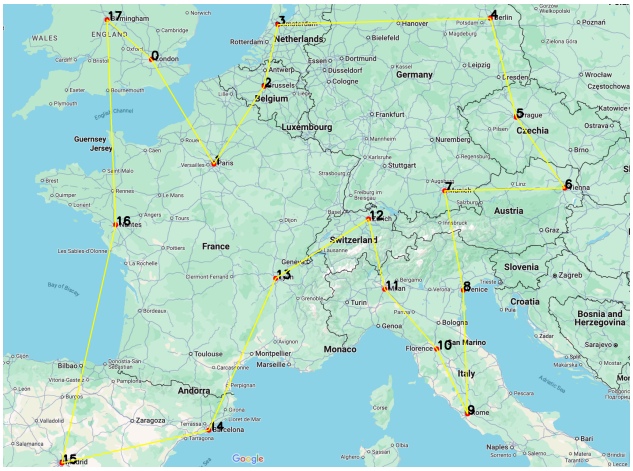

Figure.18 City Visit Path for Driving Min Time in Highway using Dynamic Programming Algorithm

Lastly, we varied the number of cities from 6 to 18 (increasing by 3), applying all three algorithms. The Table 15 and Table 16 below illustrate how cost and time metrics change with an increasing number of cities (from 6 to 18) using three algorithms.

Cost ($)	Greedy Algorithm		Divide and Conquer Algorithm		Dynamic Programming Algorithm	
Number of Cities	Regular Road	Highway	Regular Road	Highway	Regular Road	Highway
6	644.77	692.8	644.77	692.8	784.67	960.21
9	765.32	883.11	1168.74	1213.34	827.32	1126.61
12	998.15	1223.9	1433.21	1552.79	1038.7	1293.74
15	1092.76	1222.31	1773.38	1927.22	1062.36	1345.55
18	1263.76	1518.77	2433.1	2720.95	1137.14	1453.25

Table.15 Driving Min Cost ($) using Three Algorithms

Time (mins)	Greedy Algorithm		Divide and Conquer Algorithm		Dynamic Programming Algorithm	
Number of Cities	Regular Road	Highway	Regular Road	Highway	Regular Road	Highway
6	5363	2195	5363	2195	6094	2650
9	4818	2716	8647	3923	6263	3018
12	6209	3749	10349	4865	7442	3785
15	6842	3940	12687	5935	6889	4042
18	7840	4469	17674	8119	7797	4286

Table.16 Driving Min Time (mins) using Three Algorithms

The Figure 18 and Figure 19 below shows comparison of three different algorithms based on minimum cost or minimum time.

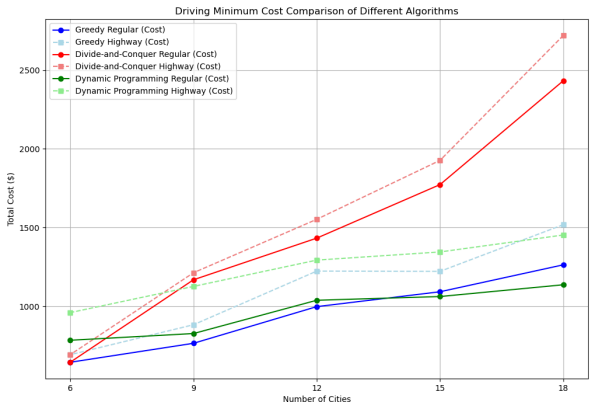

Figure.18 Driving Minimum Cost Comparison of Three Algorithms

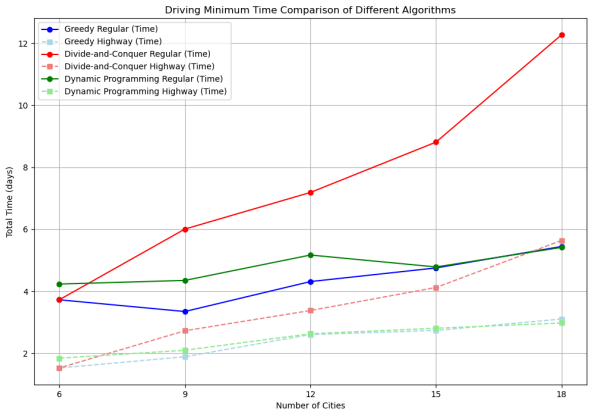

Figure.19 Driving Minimum Time Comparison of Three Algorithms

In Figure 18, as the number of cities increases, the Dynamic Programming algorithm achieves the lowest driving costs for both regular roads and highways. This trend is particularly pronounced for larger city counts (12–18 cities), where the Dynamic Programming algorithm leverages its global optimization to find the most efficient paths. Despite its high computational complexity $O(n^2 \cdot 2^n)$, the Dynamic Programming algorithm's ability to explore all possible subsets of cities ensures it consistently produces the optimal cost, making it superior for large datasets.

The Greedy algorithm, while competitive for smaller city counts (6–9 cities), struggles with cost efficiency as the number of cities grows. Its locally optimal decisions fail to account for the global structure of the problem, leading to higher costs compared to Dynamic Programming for larger city sets.

The Divide and Conquer algorithm shows higher costs than both Dynamic Programming and Greedy across all city sizes. While its theoretical [OBJ] complexity O(nlogn) is efficient, its inability to account for overlapping subproblems results in suboptimal path selections, particularly as the dataset size increases.

In Figure 19, the Greedy algorithm remains the most effective for minimizing travel time for smaller city counts (6–9 cities), as its [OBJ] $O(n^2)$ complexity allows it to compute paths quickly while prioritizing shorter travel times. However, for larger city counts (12–18 cities), the Dynamic Programming algorithm begins to outperform Greedy in

minimizing time. This shift occurs because the Dynamic Programming algorithm's global optimization starts to account for paths that balance both cost and time effectively, especially as the number of cities and potential paths increase.

The Divide and Conquer algorithm, consistent with its cost performance, shows the worst results for time minimization across all city sizes. Its recursive structure leads to inefficiencies in both computation and path selection.

The results in Figures 18 and 19 reveal that Dynamic Programming excels in large-scale problems, it becomes increasingly advantageous as the number of cities grows. Its high computational complexity $O(n^2 \cdot 2^n)$ is offset by its ability to achieve globally optimized results for both cost and time. This makes Dynamic Programming the best choice for scenarios with a high number of cities, where achieving optimality is crucial. Greedy is more effective for smaller datasets. While it may not always produce the lowest costs, its quick computations and competitive time efficiency make it well-suited for moderate datasets or time-sensitive applications. The performance of these algorithms highlights the importance of selecting the right approach based on dataset size and problem priorities.

Experiment 3: Minimum Cost and Shortest Time for Flights Between Cities

The **Greedy** algorithm works by selecting the least expensive or shortest flight for each city transition without considering the global optimal route. While the approach is simple and computationally efficient, it does not guarantee globally optimal results.

For 18 cities, the total cost was $343, and the total time was 520 minutes. Although this algorithm is competitive for smaller datasets, its locally optimal decisions lead to suboptimal results for larger city sets.

The Table 17 and Figures 20 and 21 illustrate the visit sequence and path for the minimum cost and minimum time scenarios using the Greedy algorithm.

Greedy Algorithm		
	Min Cost	Min Time
Index	City Name	City Name
0	London	London
1	Madrid	Brussels
2	Barcelona	Amsterdam
3	Lyon	Berlin
4	Venice	Munich
5	Vienna	Prague
6	Milan	Vienna
7	Rome	Venice
8	Florence	Rome
9	Brussels	Florence
10	Paris	Zurich
11	Birmingham	Milan
12	Amsterdam	Paris
13	Berlin	Birmingham
14	Munich	Barcelona
15	Prague	Madrid
16	Nantes	Nantes
17	Zurich	Lyon

Table.17 City Visit Sequence for Flight using Greedy Algorithm

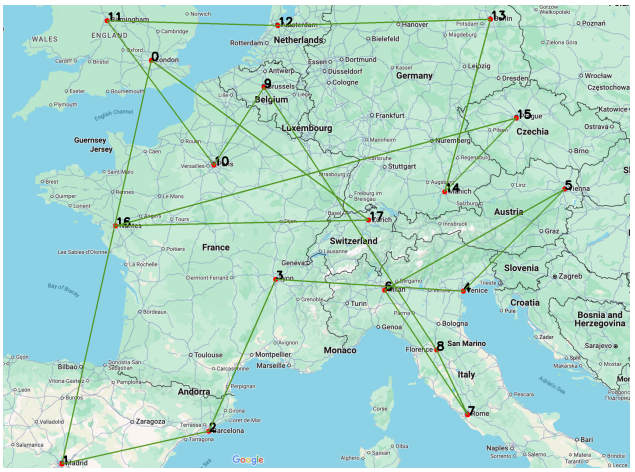

Figure.20 City Visit Path for Flight Min Cost using Greedy Algorithm

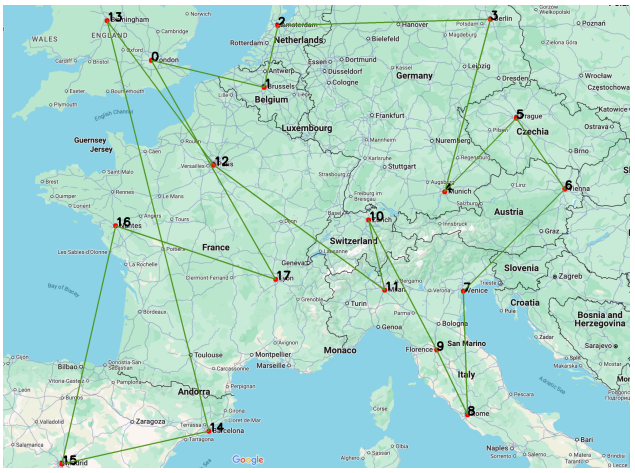

Figure.21 City Visit Path for Flight Min Time using Greedy Algorithm

The **Divide and Conquer** algorithm divides the problem into smaller subsets, solves them independently, and combines the results. It prioritizes solving smaller city pairs without accounting for the global structure of the route. This leads to suboptimal results for the Traveling Salesman Problem (TSP).

For 18 cities, the total cost was $487, and the total time was 490 minutes. The algorithm's inability to address overlapping subproblems makes it less effective than the other approaches.

The Table 18 and Figures 22 and 23 demonstrate the visit sequence and path for flights using the Divide and Conquer algorithm for minimum cost and time scenarios.

Divide and Conquer Algoritm		
	Min Cost	Min Time
Index	City Name	City Name
0	London	London
1	Paris	Paris
2	Milan	Milan
3	Vienna	Vienna
4	Barcelona	Barcelona
5	Madrid	Madrid
6	Lyon	Lyon
7	Berlin	Berlin
8	Venice	Venice
9	Rome	Rome
10	Munich	Munich
11	Zurich	Zurich
12	Amsterdam	Amsterdam
13	Brussels	Brussels
14	Florence	Florence
15	Birmingham	Birmingham
16	Prague	Prague
17	Nantes	Nantes

Table.18 City Visit Sequence for Flight using Divide and Conquer Algorithm

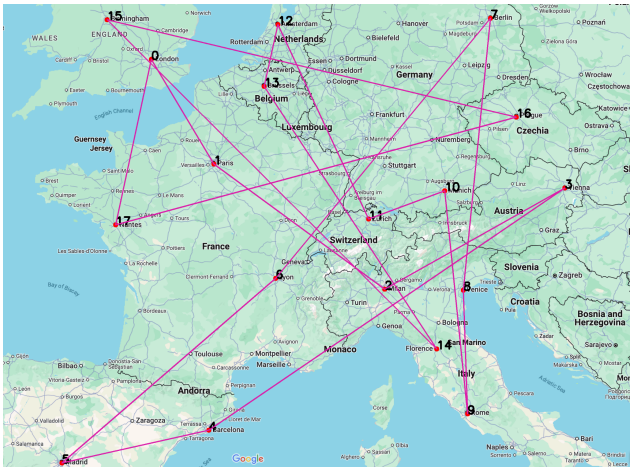

Figure.21 City Visit Path for Flight Min Cost using Divide and Conquer Algorithm

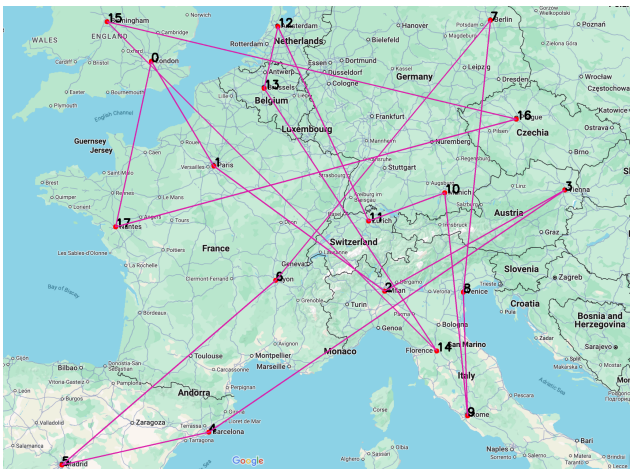

Figure.22 City Visit Path for Flight Min Time using Divide and Conquer Algorithm

The **Dynamic Programming** algorithm systematically evaluates all possible routes and stores intermediate results to ensure a globally optimal solution. While its computational complexity is high $O(n^2 \cdot 2^n)$, it consistently achieves the best results.

For 18 cities, the total cost was $333, and the total time was 570 minutes. This demonstrates the Dynamic Programming algorithm's ability to find the globally optimal route for both cost and time, even for larger datasets.

The Table 19 and Figures 24 and 25 present the city visit sequence and paths for flights using the Dynamic Programming algorithm for both minimum cost and minimum time scenarios.

Dynamic Programming Algorithm		
	Min Cost	Min Time
Index	City Name	City Name
0	London	London
1	Munich	Paris
2	Berlin	Birmingham
3	Prague	Brussels
4	Brussels	Amsterdam
5	Paris	Berlin
6	Zurich	Vienna
7	Madrid	Prague
8	Nantes	Munich
9	Florence	Lyon
10	Rome	Milan
11	Milan	Zurich
12	Amsterdam	Venice
13	Birmingham	Rome
14	Barcelona	Florence

15	Lyon	Barcelona
16	Venice	Madrid
17	Vienna	Nantes

Table.19 City Visit Sequence for Flight using Dynamic Programming Algorithm

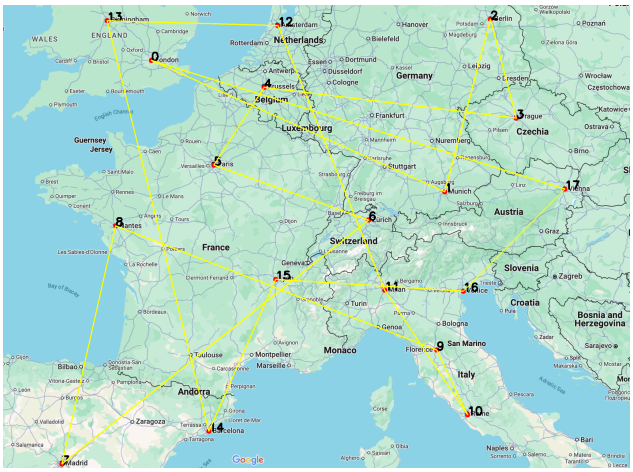

Figure.23 City Visit Path for Flight Min Cost using Divide and Dynamic Programming Algorithm

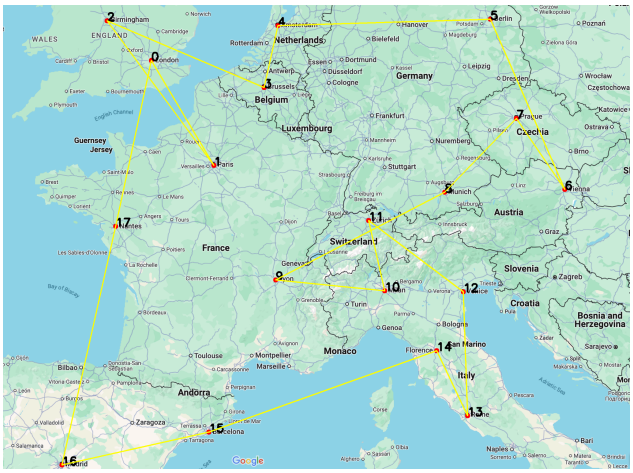

Figure.24 City Visit Path for Flight Min Time using Dynamic Programming Algorithm

Lastly, we varied the number of cities from 6 to 18 (increasing by 3), applying all three algorithms based on minimum cost and shortest time.
The results for the minimum cost of flights are summarized in Table 20 and visualized in Figure 25.
The results for the shortest time of flights are presented in Table 21 and visualized in Figure 26.

Cost ($)	Greedy Algorithm	Divide and Conquer Algorithm	Dynamic Programming Algorithm
6	1464	2159	1331
9	1269	1750	1062
12	794	1189	782
15	531	703	508
18	343	487	333

Table.20 Flight Min Cost ($) using Three Algorithms

Time (mins)	Greedy Algorithm	Divide and Conquer Algorithm	Dynamic Programming Algorithm
6	1670	2030	1670
9	1130	1290	1290
12	980	1040	990
15	750	815	765
18	520	490	570

Table.21 Flight Min Time (mins) using Three Algorithms

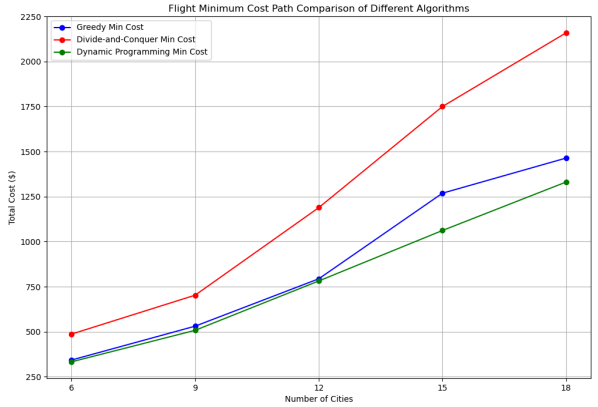

Figure.25 Flight Minimum Cost Comparison of Three Algorithms

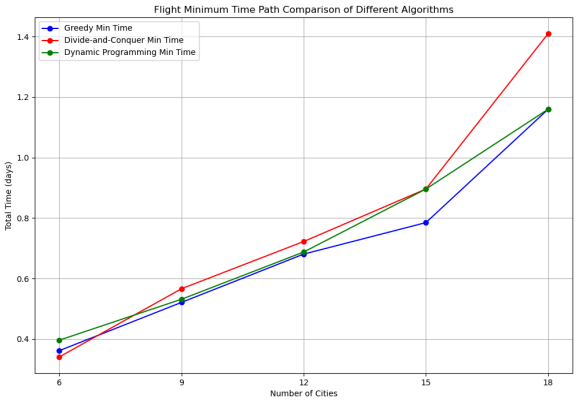

Figure.26 Flight Minimum Time Comparison of Three Algorithms

For cost efficiency, the Dynamic Programming Algorithm consistently produced the lowest costs for flights across all city sizes, showcasing its strength in global optimization. The Greedy algorithm performed well for smaller datasets but was less cost-efficient for larger datasets. The Divide and Conquer algorithm resulted in the highest costs due to its inability to address overlapping subproblems.
For time efficiency, the Dynamic Programming algorithm also demonstrated superior time performance for larger

datasets (12–18 cities), balancing both cost and time effectively. The Greedy algorithm was competitive for smaller datasets (6–9 cities) but became less effective for larger city sets. The Divide and Conquer algorithm, while slightly better than Greedy for larger datasets in terms of time, was still less effective than Dynamic Programming.

For large datasets where achieving the optimal cost and time is crucial, the Dynamic Programming algorithm is the most effective despite its high computational complexity. For smaller datasets or time-sensitive applications, the Greedy algorithm offers a quick and reasonable solution.

VI. Gantt Chart

	Owner	Aug 2024	Sep 2024	Oct 2024	Nov 2024	Dec 2024
Databset Collection						
Cities Coordinates	JY	Done				
Regular Road Driving (Costs & Time)	JY		Done			
Highway Driving (Costs & Time)	JY		Done			
Flight Cost and Duration	JY		Done			
Experiment 1: Helicopter Shortest Distance Based on Coordinates						
Implementation of Greedy	JY		Done			
Implementation of Divide and Conquer	HY		Done			
Implementation of Dynamic Programming	YP		Done			
Evaluation Metrics	JY		Done			
Experiment 2: Driving Through Regular and Highway Roads						
Implementation of Greedy	HY			Done		
Implementation of Divide and Conquer	YP			Done		
Implementation of Dynamic Programming	JY			Done		
Evaluation Metrics	YP			Done		
Experiment 3: Flight Cost and Duration Optimization						
Implementation of Greedy	YP				Done	
Implementation of Divide and Conquer	JY				Done	
Implementation of Dynamic Programming	HY				Done	
Evaluation Metrics	JY				Done	
Presentation and Submission						
Powerpoint & Report	All				Done	

VII. CONCLUSION

From three experiments, we evaluated three algorithms—Greedy algorithm, Divide and Conquer algorithm, and Dynamic Programming algorithm—for solving the Traveling Salesman Problem (TSP) across various scenarios, including helicopter shortest distances, driving routes on regular and highway roads, and flight paths between cities for up to 18 cities.

The **Dynamic Programming** algorithm consistently outperformed the other algorithms, yielding the shortest total distances, lowest costs, and minimal times across all city counts. By exploring all possible subsets of cities using bitmasking, Dynamic Programming successfully found the globally optimal solution in all cases. However, its high time complexity $O(n^2 \cdot 2^n)$ and space complexity $O(n \cdot 2^n)$ make it resource-intensive and impractical for larger datasets, especially in real-world applications with hundreds of cities.

The **Greedy** algorithm, while simple and computationally efficient $O(n^2)$, demonstrated limitations in finding globally optimal solutions due to its focus on local optimization. It performed reasonably well for smaller datasets (6–9 cities), offering a good trade-off between accuracy and speed. However, its performance deteriorated for larger datasets, where it was consistently outperformed by the Dynamic Programming algorithm in terms of accuracy and by the Divide and Conquer algorithm in certain time scenarios.

The **Divide and Conquer** algorithm struggled the most across all experiments, producing suboptimal results due to its inability to account for the global structure of the problem. Despite its theoretically efficient time complexity of O(nlogn), it generated longer routes and higher costs as the dataset size increased. It also showed significant limitations in dynamic scenarios like flights and driving on varying road types.

In summary, Dynamic Programming algorithm provided the most accurate results, successfully finding the optimal solution by exploring all possible subsets of cities using bitmasking. However, its time complexity of $O(n^2 \cdot 2^n)$ and space complexity of $O(n \cdot 2^n)$ limit its practicality for larger datasets due to memory and processing constraints. Greedy offers a compromise between efficiency and accuracy, making it useful for smaller or simpler datasets where speed is a priority. Divide and Conquer algorithm, while efficient for some domains, showed its limitations in managing global optimization problems like TSP.As the number of cities grows, the balance between computational efficiency and solution optimality becomes critical, with Dynamic Programming algorithm providing the best solutions but requiring the most resources.

A comparison of helicopter, driving, and flight travel highlighted their respective trade-offs, with helicopter offering the shortest distances, driving being cost-effective but time-consuming, and flights providing a balance of cost and time for long distances. This study provides insights for enhancing travel route optimization systems.

In our project, one of the main challenges lies in collecting realistic, high-quality data essential for meaningful comparisons and analysis. This process requires significant time and access to accurate, diverse datasets, which may vary across different transportation scenarios. Additionally, designing clear visualizations to represent city visit sequences and routes is particularly challenging for larger datasets and when considering multiple optimization criteria. Comparative analysis also becomes complex as unique constraints, such as cost, time, and mode-specific factors, influence algorithm performance differently across scenarios.

Future work should focus on developing a route planner that integrates multiple modes of transportation, such as driving, flights, helicopters, biking, and walking, to create optimized mixed-mode travel plans. Another direction is automating the selection of the most suitable algorithm based on dataset size and complexity. Lastly, implementing dynamic visualization tools for route planning and real-time path adjustments would significantly enhance the practical usability of these algorithms in real-world applications.

ACKNOWLEDGMENT

We would like to sincerely thank Professor, Rubi, for all her helpful suggestions and support throughout our project. She gave us the idea for the experiment design and the idea for helicopter shortest distance is interesting and helpful. Her guidance was very important to us, and we appreciate her patience and encouragement as we worked through the project.

REFERENCES

[1] Route4Me, and Route4Me. “What Is the Traveling Salesman Problem (TSP)? (2024).” *Route Optimization Blog*, 4 July 2024, blog.route4me.com/traveling-salesman-problem.

[2] Tolls.eu. “Fuel Prices in Europe.” *Fuel Prices in Europe - Tolls.Eu*, Dec. 2024, www.tolls.eu/fuel-prices.

[3] WorldData.info. “Speed Limit by Country.” *Worlddata.Info*, Dec. 2024, www.worlddata.info/speed-limit.php.

[4] MapUp, Inc. “Europe Toll Calculator - Tolls, Fuel, Vignette, Obu, Toll Tags.”, *TollGuru*, Dec. 2024, tollguru.com/toll-calculator-europe.

[5] Google. “Google Travel.” *Www.google.com*, Dec. 2024, www.google.com/travel/flights?gl=US&hl=en-US.

[6] GeeksforGeeks. “Closest Pair of Points Using Divide and Conquer Algorithm.” *GeeksforGeeks*, 13 Feb. 2023, www.geeksforgeeks.org/closest-pair-of-points-using-divide-and-conquer-algorithm.

[7] GeeksforGeeks. “Dynamic Programming or DP.” *GeeksforGeeks*, 27 Aug. 2024, www.geeksforgeeks.org/dynamic-programming.

[8] Wikipedia contributors. “Travelling Salesman Problem.” *Wikipedia*, 14 Oct. 2024, en.wikipedia.org/w/index.php?title=Travelling_salesman_problem&oldid=1251186999

APPENDIX

Overall, we found the course assignments to be diverse and effective for learning the material from different perspectives. However, we believe there are areas that could be improved:

1. Algorithm Strategy Test:
   The concept of a group-based, closed-book test is interesting but felt somewhat unfair. With 2-3 questions but 3-4 members in a group, it was challenging to ensure everyone was equally evaluated. There should also include a peer_evaluation_form or consider to do personal test.
2. Coding Assignments:
   The coding assignments were helpful for understanding the concepts, but the reports required a level of detail similar to the final project report. This made them very time-intensive. We also noticed points being deducted for issues always unrelated to the algorithms, even when our reports were thorough and detailed.
3. Assignment Difficulty:
   Some assignments, like CA5, were much harder than CA3. It would be motivating if more challenging assignments came with higher credits to reward the extra effort.