Flight Route Optimization with The Travelling Salesman Problem and Graph Theory

Moh. Hafizh Irham Perdana - 13524025
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
E-mail: hafizhirham06@gmail.com, 13524025@std.stei.itb.ac.id

***Abstract*—Optimizing flight routes is a critical challenge in air transportation, especially when considering the Earth's spherical geometry. This paper applies the Travelling Salesman Problem (TSP) within the framework of graph theory to model and determine the most efficient route that visits a set of airports exactly once and returns to the starting point. By treating each airport as a node and the great-circle distances between them as weighted edges, we implement both heuristic and exact algorithms—such as Nearest Neighbor and Brute Force—to compare performance and route quality. The study also presents visualizations of the calculated routes based on a spherical Earth model, emphasizing the significance of using spherical geometry in realistic distance calculations. This approach demonstrates the power of computational optimization in real-world geospatial problems and highlights the trade-offs between accuracy and efficiency in algorithmic solutions.**

***Keywords*—Traveling Salesman Problem, flight route, graph theory.**

I. Introduction

The global aviation industry is a cornerstone of modern transportation, facilitating the rapid movement of people and goods across countries and continents. As air traffic continues to grow, the demand for route planning that is not only economically efficient but also geographically accurate, has become increasingly critical. Airlines and logistics companies continuously seek to optimize their flight schedules to reduce fuel consumption, lower operational costs, and increase passenger satisfaction. One of the most fundamental computational problems underlying these efforts is the optimization of multi-stop flight routes — a problem that, at its core, mirrors the classical Travelling Salesman Problem (TSP).

The Travelling Salesman Problem is a well-known combinatorial optimization problem in theoretical computer science. It asks: "Given a list of cities and the pairwise distances between them, what is the shortest possible route that visits each city exactly once and returns to the starting city?" Despite its seemingly simple formulation, the TSP is classified as NP-hard, meaning there is no known polynomial-time algorithm to solve all instances of the problem efficiently. The number of possible routes increases factorially with the number of cities, making brute-force methods computationally infeasible beyond small inputs. As such, practical solutions often rely on heuristics or approximation algorithms that balance computational efficiency with solution quality.

In this paper, we model the problem of flight route optimization as a specific instance of the TSP. Each airport is treated as a node in a graph, and each direct flight path between two airports is represented as an edge weighted by the geographical distance between them. To accurately reflect the Earth's surface, which is approximately spherical, the distances between airports are not computed using flat (Euclidean) geometry. Instead, we employ spherical geometry, specifically the great-circle distance formula, which calculates the shortest path between two points on the surface of a sphere.

The resulting mathematical model is a complete-weighted-undirected graph, in which each node (airport) is connected to every other node by an edge whose weight corresponds to the great-circle distance. This model captures the key characteristics of real-world flight networks, where airlines often have the option to fly directly between any pair of major airports, and the cost of that flight is influenced primarily by distance.

To solve the TSP in this context, we implement and analyze two algorithmic approaches with different computational trade-offs:

1. Brute-Force Search, which enumerates all possible permutations of airport visits to find the exact shortest route. This method guarantees the optimal solution but is computationally expensive and limited to smaller instances (typically fewer than 10 airports).
2. Nearest Neighbor Heuristic, a greedy approximation algorithm that builds a tour by repeatedly selecting the nearest unvisited airport. Although it does not always yield the optimal solution, this method offers much faster computation and remains practical for larger datasets.

II. Basic Theory

A. Graph Fundamentals

Graph is a type of data structure consisting of vertices and edges. It is useful in many fields such as mathematics, data science, computer science, or even economics. Graph data structure can be used to model various real-world scenarios, analyze correlation between packs of objects, and understand the relation between variables or parameters. In this context, considering flight routes or networks, nodes represent points of the airport locations while edges describe the path of the flights.

A graph can be expressed as an ordered pair of two sets vertices(V) and edges(E). Formally, graph G is defined as:

$$G = (V, E)$$

- V is set of vertices, defined as nodes that can be labeled or unlabeled as a representation of objects or elements being connected.

$$V = \{v_1, v_2, v_3, \dots, v_n\}$$

- E is a set of edges that are the connections between pairs of vertices. They represent the relationships or associations between the objects. These two are the fundamental components of graph data structures.

$$E = \{e_1, e_2, e_3, \dots, e_n\}$$

Defined graph must consist of a minimum of one vertices with no edge's requirements, meaning that it can be consist without edges or many edges.

A graph can also be represented visually with the point or node as the vertices and line as the edges.

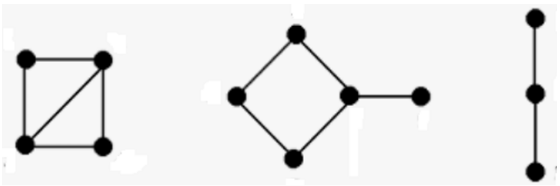

Figure 1. Simple graph (https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf)

A graph can be weighted or unweight depending on its purpose and objectives.

- Unweighted graph is a graph where all edges are considered equal. By this definition, there is no numerical value (weight) assigned to any edge. This graph focuses on the connections, not the cost, distance, or time between them, without any measure of how strong or long those connections are. These types of graphs are used many to represent social networks and network connectivity.

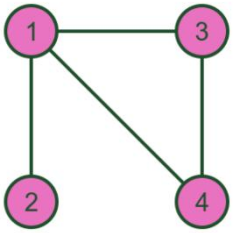

Figure 2. Unweighted graph (https://graphicmaths.com/computer-science/graph-theory/graphs/)

- Weighted graph is a graph where each edge has a numerical value (called a weight). This weight can represent distance, cost, time, energy, or any measurable factor. This graph not only focuses on how objects are connected, but how strong or weak those connections are. This graph has much more complexity but offers more information and analysis.

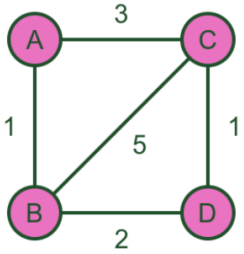

Figure 3. Weighted graph (https://graphicmaths.com/computer-science/graph-theory/graphs/)

Weighted graphs frequently serve to represent tangible entities and the connections among them. An instance of this can be found within the framework of Google Maps, where cities represent nodes, roads stand as edges, and the edge weights signify the time or distance of two cities. An example of the implementation is to find the shortest path between objects, optimal delivery cost, or network routing. In this paper case it is to optimize flight route considering its distance.

Besides the weight, a graph may be directed or undirected depending on its purpose.

- Undirected have edges that do not have direction. The edges indicate a *two-way* relationship, in that each edge can be traversed in both directions. The figure below shows an example of the undirected graph with five nodes and six edges.

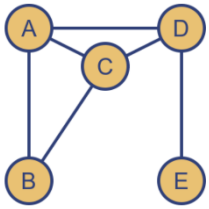

Figure 4. Undirected graph (https://graphicmaths.com/computer-science/graph-theory/graphs/)

- Directed graph offers directional information. The edges indicate a *one-way* relationship, in that each edge can only be traversed in a single direction. The figure below shows a simple directed graph with four nodes and five edges.

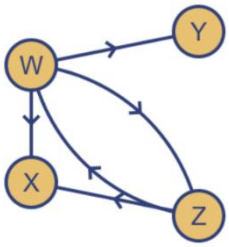

Figure 5. Directed graph (https://graphicmaths.com/computer-science/graph-theory/graphs/)

B. Travelling Salesman Problem

The Traveling Salesman Problem (TSP) is a famous algorithmic challenge in computer science and operations research that seeks to identify the most efficient route, typically the shortest one, for a salesperson. This route begins with a specified origin, visits a given set of cities (nodes), and may or may not include a designated destination. The problem has significant practical relevance, particularly in optimizing logistics and delivery operations. In theoretical computer science, the TSP has commanded so much attention because it's so easy to describe yet so difficult to solve. The TSP is classified as a combinatorial optimization problem, recognized as NP-hard problem, indicating the quantity of potential solution sequences escalates exponentially with the increasing number of cities(or nodes). As a result, computer scientists have yet to discover an algorithm capable of solving TSP instances in polynomial time, necessitating the use of approximation algorithms to explore numerous permutations and identify the shortest, most cost-effective route.

Let us consider an illustrative example of the Traveling Salesman Problem (TSP). We are given four cities: A, B, C, D, and E, along with the distances separating them. Figure 6 visually represents these cities and their inter-city distances. For this scenario, distinct routes can be generated. The route A→D→C→B→E→A is identified as the optimal solution for this particular problem with total of 19 distance.

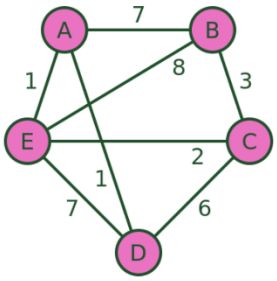

Figure 6. TSP graph simulation (https://graphicmaths.com/computer-science/graph-theory/graphs/)

Numerous heuristic techniques, such as the greedy method, ant algorithms, simulated annealing, tabu search, and genetic algorithms, have been employed to find efficient solutions to this problem. However, as the number of cities grows, the computational effort required to find a solution becomes challenging. Despite this computational difficulty, approaches like genetic algorithms and tabu search can provide near-optimal solutions for problems involving thousands of cities. This paper aims to provide an overview of some methods utilized for solving the Traveling Salesman Problem.

C. Haversine Formula

Haversine formula is an essential equation in the finding straight line distance between two coordinates on earth using latitude and longitude parameters. The haversine formula calculates by using trigonometry applied to a round shape. This formula discusses the shapes of sides and angles in spherical triangle. The haversine method is commonly used in the world of aviation to calculate the distance of an aircraft with the coordinates of the destination. Following is the haversine algorithm calculation:

$$hav(\theta) = hav(\Delta\varphi) + \cos(\varphi_1)\cos(\varphi_2)\, hav(\Delta\lambda)$$

Where,

- θ is the central angle between two points(measured from the earth reference)
- φ_1, φ_2 are the latitude of point 1 and point 2
- λ_1, λ_2 are the longitude of point 1 and point 2
- Δφ, Δλ are the latitude difference and longitude difference

Then the haversine function itself is applied to each variable above resulting in this following equation,

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

The variable d is the distance between two points along a great circle and R is the Earth radius (6371 km). The haversine formula accounts for Earth's spherical shape and remains particularly well-conditioned for numerical computation even at small distances.

D. Brute Force Algorithm

The Brute Force algorithm is a basic problem-solving method that relies on exhaustively checking all possible solutions, using computational power rather than optimization techniques to enhance efficiency. It typically operates based on the problem's explicit definition and requirements. This method addresses the task in a direct, straightforward, and intuitive manner. In this TSP case, it simply calculates the total distance for every possible route and selects the shortest one.

E. Nearest Neighbor Algorithm

To implement the nearest algorithm, begin at a randomly selected starting point. And then, the algorithm finds the closest unvisited node and adds it to the sequence. Then, we move to the next node and repeat the process of finding the nearest unvisited node until all nodes are included in the tour. Finally, we returned to the starting airport to complete the cycle. Although the nearest neighbor method is simple to grasp and performs efficiently, it often fails to produce the optimal solution for the traveling salesman problem. The resulting route may be considerably longer than the shortest possible path, particularly in larger or more complex cases. However, this algorithm remains a useful initial approach, offering a fast and reasonably effective solution when exact optimization is not essential.

III. IMPLEMENTATION

As stated previously, the author's analysis will focus on making the shortest flight route and comparing the algorithmic performance between the straightforward brute force algorithm and heuristic nearest-neighbor algorithm.

A. Obtaining Location Data

The data is collected from airports dataset openflights.org. For the purposes of this study, the airport data is grouped into five regional categories: Asia, Europe, America, Africa, and Australia & Oceania.

```python
airports_asia = {
    "CGK": (-6.1256, 106.6559),   # Soekarno-Hatta International Airport, Indonesia
    "PEK": (40.0801, 116.5849),   # Beijing Capital International Airport, China
    "DXB": (25.2527, 55.3643),    # Dubai International Airport, UEA
    "ICN": (37.4691, 126.4509),   # Incheon International Airport, South Korea
    "DEL": (28.5665, 77.1031),    # Indira Gandhi International Airport, India
    "SIN": (1.3501, 103.9940),    # Singapore Changi Airport, Singapore
    "KUL": (2.7456, 101.7099),    # Kuala Lumpur International Airport, Malaysia
    "BKK": (13.6810, 100.7470),   # Suvarnabhumi Airport, Thailand
    "MNL": (14.5086, 121.0198),   # Ninoy Aquino International Airport, Philippines
    "HKG": (22.3080, 113.9150),   # Hong Kong International Airport, Hong Kong
    "TPE": (25.0777, 121.2333),   # Taiwan Taoyuan International Airport, Taiwan
    "HND": (35.5522, 139.7799),   # Tokyo Haneda Airport, Japan
}
airport_names = list(airports_asia.keys())
```

Figure 7. Asia Airport Dataset

```python
airports_europe = {
    "LHR": (51.4706, -0.4619),    # London Heathrow Airport, United Kingdom
    "CDG": (49.0127, 2.55),       # Charles de Gaulle Airport, France
    "AMS": (52.3086, 4.7638),     # Amsterdam Schiphol Airport, Netherland
    "FRA": (50.0333, 8.5705),     # Frankfurt Main Airport, Germany
    "IST": (40.9768, 28.8146),    # Istanbul Airport, Turkiye
    "MAD": (40.4719, -3.5626),    # Adolfo Suárez Madrid-Barajas Airport, Spain
    "ZRH": (47.4646, 8.5491),     # Zurich Airport, Swizz
    "MUC": (48.3538, 11.7861),    # Munich Airport, Germany
    "FCO": (41.8002, 12.2388),    # Leonardo da Vinci-Fiumicino Airport, Italy
    "CPH": (55.6179, 12.6560),    # Copenhagen Airport, Denmark
    "ARN": (59.6519, 17.9186),    # Stockholm Arlanda Airport, Sweden
    "OSL": (60.121, 11.0502),     # Oslo Gardermoen Airport,  Norway
    "SVO": (55.9725, 37.4146),    # Sheremetyevo International Airport, Russia
}
airport_names = list(airports_europe.keys())
```

Figure 8. Europe Airport Dataset

```python
airports_america = {
    "ATL": (33.6367, -84.4281),   # Hartsfield-Jackson Atlanta Intl Airport, USA
    "JFK": (40.6398, -73.7789),   # John F. Kennedy International Airport, USA
    "LAX": (33.9425, -118.4079),  # Los Angeles International Airport, USA
    "YYZ": (43.6772, -79.6305),   # Toronto Pearson International Airport, Canada
    "MEX": (19.4363, -99.0720),   # Mexico City International Airport, Mexico
    "GRU": (-23.4355, -46.4730),  # São Paulo/Guarulhos International Airport, Brazil
    "EZE": (-34.8222, -58.5358),  # Ministro Pistarini International Airport, Argentina
    "MIA": (25.7931, -80.2906),   # Miami International Airport, USA
    "LIM": (-12.0219, -77.1143),  # Jorge Chávez International Airport, Peru
    "SCL": (-33.3930, -70.7857),  # Comodoro Arturo Merino Benítez Intl Airport, Chile
    "PTY": (9.0713, -79.3834),    # Tocumen International Airport, Panama
    "SAL": (13.4409, -89.0557),   # El Salvador International Airport, El Salvador
}
airport_names = list(airports_america.keys())
```

Figure 9. America Airport Dataset

```python
airports_africa = {
    "JNB": (-26.1392, 28.246),    # OR Tambo Johannesburg International Airport, South Africa
    "CAI": (30.1219, 31.4055),    # Cairo International Airport, Egypt
    "CMN": (33.3675, -7.5899),    # Mohammed V International Airport, Morocco
    "ADD": (8.9778, 38.7993),     # Addis Ababa Bole International Airport, Ethiopia
    "NBO": (-1.3192, 36.9277),    # Jomo Kenyatta International Airport, Kenya
    "CPT": (-33.9648, 18.6016),   # Cape Town International Airport, South Africa
    "LOS": (6.5773, 3.3211),      # Murtala Muhammed International Airport, Nigeria
    "ABJ": (5.2613, -3.9262),     # Félix-Houphouët-Boigny International Airport, Ivory Coast
    "DKR": (14.7397, -17.4902),   # Blaise Diagne International Airport, Senegal
    "ALG": (36.691, 3.2154),      # Houari Boumediene Airport, Algeria
}
airport_names = list(airports_africa.keys())
```

Figure 10. Africa Airport Dataset

```python
airports_oceania = {
    "SYD": (-33.946, 151.1770),   # Sydney Kingsford Smith Airport, Australia
    "MEL": (-37.6733, 144.843),   # Melbourne Airport, Australia
    "BNE": (-27.384, 153.1170),   # Brisbane Airport, Australia
    "AKL": (-37.008, 174.792),    # Auckland Airport, New Zealand
    "PER": (-31.9402, 115.9670),  # Perth Airport, Australia
    "WLG": (-41.3272, 174.8049),  # Wellington International Airport, New Zealand
    "CNS": (-16.8857, 145.755),   # Cairns Airport, Australia
    "NAN": (-17.7553, 177.4429),  # Nadi International Airport, Fiji
    "POM": (-9.4433, 147.220),    # Jacksons International Airport, Papua New Guinea
    "HNL": (21.32062, -157.9242), # Daniel K. Inouye International Airport, USA(Hawaii)
}
airport_names = list(airports_oceania.keys())
```

Figure 11. Oceania Airport Dataset

B. Calculating the Distance

The following Python function implements the Haversine formula, which is used to compute the shortest distance between two points on the surface of a sphere (i.e., the Earth), given their latitude and longitude.

```python
# Define Haversine formula
def haversine_distance(coord1, coord2, radius=6371):
    lat1, lon1 = map(math.radians, coord1)
    lat2, lon2 = map(math.radians, coord2)
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = math.sin(dlat/2)**2 + math.cos(lat1)*math.cos(lat2)*math.sin(dlon/2)**2
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
    return radius * c
```

Figure 12. Haversine Formula

Input Parameters:

- coord1 and coord2: Tuples representing the latitude and longitude of two geographical points (in degrees).
- radius: The radius of the Earth in kilometers (6,371 km).

Process:

- Convert degrees to radian, since trigonometric functions in Python use radians, while the geographic coordinates uses degree.
- Calculate the differences between two points.
- Apply the Haversine Formula and finally compute the distance.

```python
# Distance matrix
n = len(airport_names)
distance_matrix = np.zeros((n, n))
for i in range(n):
    for j in range(n):
        if i != j:
            distance_matrix[i][j] = haversine_distance(airports[airport_names[i]], airports[airport_names[j]])
```

Figure 13. Distance Matrix

The code generates a distance matrix between all pairs of airports using the previously defined Haversine formula.

- Provides the "cost" or "distance" between any two nodes.
- After execution, the distance_matrix[i][j] holds the geographic distance (in km) between airport i and airport j. This matrix is then used as the input for the algorithms.

C. Implementing Algorithms

The following code implements the Nearest Neighbor (NN) algorithm. A greedy heuristic that constructs a tour by always selecting the nearest unvisited airport at each step.

- Retrieves the number of airports, based on the size of the distance matrix.
- The tour starts at the first airport (index 0), which is marked as visited. Loops through the remaining unvisited cities to build the tour, while also calculating the distance.

```python
# Nearest Neighbor Algorithm
def tsp_nearest_neighbor(matrix):
    n = len(matrix)
    visited = [False] * n
    path = [0]
    visited[0] = True
    total_cost = 0
    current = 0
    for _ in range(n - 1):
        next_city = min((i for i in range(n) if not visited[i]), key=lambda i: matrix[current][i])
        total_cost += matrix[current][next_city]
        path.append(next_city)
        visited[next_city] = True
        current = next_city
    total_cost += matrix[current][0]
    path.append(0)
    return path, total_cost
```

Figure 14. Nearest Neighbor Algorithm

```python
# Brute Force Algorithm
def tsp_brute_force(matrix):
    n = len(matrix)
    min_path = float('inf')
    best_route = []
    for perm in permutations(range(1, n)):
        route = [0] + list(perm) + [0]
        cost = sum(matrix[route[i]][route[i+1]] for i in range(len(route)-1))
        if cost < min_path:
            min_path = cost
            best_route = route
    return best_route, min_path
```

Figure 15. Brute Force Algorithm

- Evaluate every possible tour of the airports by generating all possible permutations of cities to visit, excluding the starting airport (index 0). This is because the salesman always starts and ends at the first airport.
- Build a complete route by adding the starting airport (index 0) at the beginning and end.
- Calculates the total travel distance. Then, it returns to the one with the lowest total travel cost.

```python
start_time = time.time()
if algorithm == "nn":
    path, cost = tsp_nearest_neighbor(distance_matrix)
    algo_name = "NearestNeighbor"
elif algorithm == "bf":
    path, cost = tsp_brute_force(distance_matrix)
    algo_name = "BruteForce"
end_time = time.time()
elapsed = end_time - start_time
path_coords = [airports[airport_names[i]] for i in path]
```

Figure 16. Execution Time

It records the time before starting the algorithm. Records the time again after the function finishes execution. Then calculates the total time elapsed by subtracting the start time from the end time.

D. Visualizing the Flight Route

Converts the indices of the cities in the nn_path (also for the brute force respectively) into actual geographic coordinates (latitude and longitude). Then creating a new plot using Basemap.

```python
plt.figure(figsize=(12, 6))
region_map_center = {
    "asia": {"lon_0": 110, "lat_0": 20},
    "europe": {"lon_0": 10, "lat_0": 50},
    "america": {"lon_0": -80, "lat_0": 20},
    "africa": {"lon_0": 20, "lat_0": 0},
    "oceania": {"lon_0": 150, "lat_0": -25}
}
lon_center = np.mean([lon for lat, lon in path_coords])
center = region_map_center[region]

m = Basemap(projection='mill', lon_0=center["lon_0"], lat_0=center["lat_0"])
m.drawcoastlines()
m.drawcountries()
m.drawmapboundary(fill_color='lightblue')
m.fillcontinents(color='lightgray', lake_color='lightblue')
m.drawparallels(np.arange(-90, 91, 30), labels=[1,0,0,0])
m.drawmeridians(np.arange(-180, 181, 60), labels=[0,0,0,1])
```

Figure 17. Plot Visualization

Labelling the Node

- The coordinates are separated into latitudes and longitudes. These are converted into 2D map coordinates using m().
- Each airport on the path is labeled with its visiting order and code.

```python
lats, lons = zip(*path_coords)
x, y = m(lons, lats)
m.plot(x, y, marker='o', color='red', linewidth=2)

# Labels
for i, (lon, lat) in enumerate(zip(lons, lats)):
    plt.text(x[i]+100000, y[i]+100000, f"{i+1}. {airport_names[path[i]]}", fontsize=9, color='darkblue')
plt.title(f"TSP Flight Route using {algo_name} Algorithm ({region.capitalize()})")
route_labels = ' → '.join([airport_names[i] for i in path])
plt.figtext(0.5, 0.13, f"Flight Route: {route_labels}", ha='center', fontsize=9, color='black')
plt.figtext(0.5, 0.1, f"Total Distance: {cost:.2f} km", ha='center', fontsize=9, color='black')
plt.figtext(0.5, 0.07, f"Execution Time: {elapsed:.4f} s", ha='center', fontsize=9, color='black')

# Save
filename = f"TSP_{region}_{algo_name}.png"
plt.tight_layout()
plt.savefig(filename, dpi=300)
plt.show()
print(f"Map saved as {filename}")
```

Figure 18. Route and Label

E. Implementation Results

The results are Travelling Salesman Problem graph in each region respectively where NN is a flight route using Nearest Neighbor Algorithm and BF is using Brute Force algorithm.

a. Asia Group
   - NN distance: 26265.29 km
   - BF distance: 22269.81 km

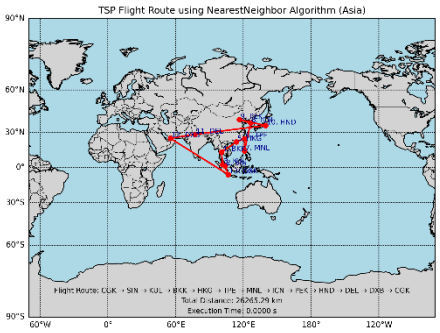

Figure 19. NN Asia Route

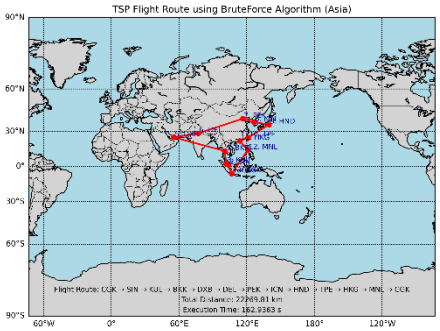

Figure 20. BF Asia Route

b. Europe Group
- NN: 12187.50 km
- BF: 10114.82 km

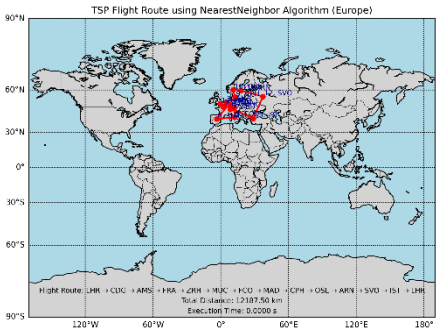

Figure 21. NN Europe Route

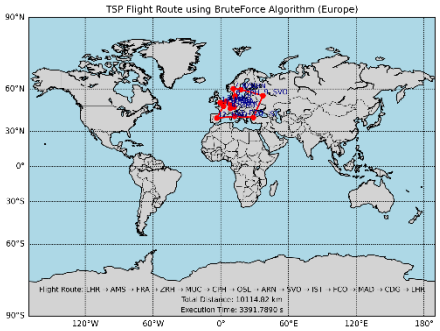

Figure 22. BF Europe Route

c. America Group
- NN: 26689.64 km
- BF: 25414.92 km

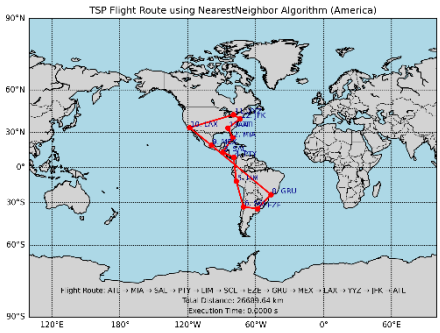

Figure 23. NN America Route

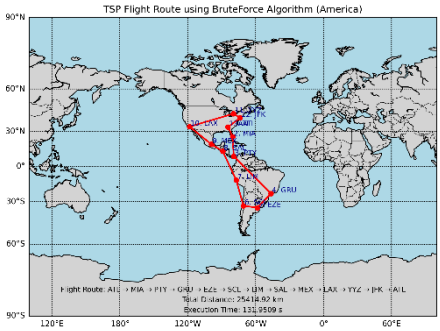

Figure 24. BF America Route

d. Africa Group
- NN: 22221.94 km
- BF: 21296.35 km

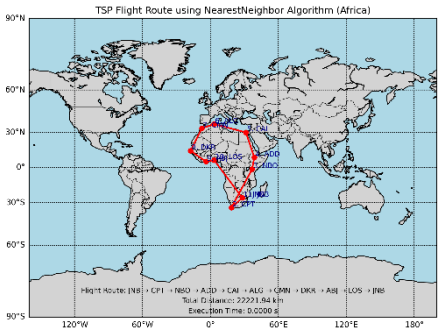

Figure 25. NN Africa Route

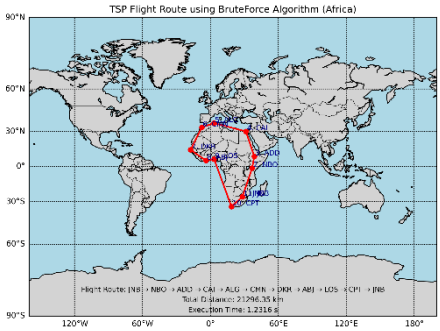

Figure 26. BF Africa Route

e. Oceania Group
- NN: 29201.02 km
- BF: 22033.65 km

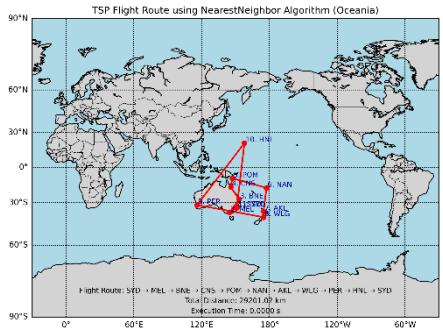

Figure 27. NN Oceania Route

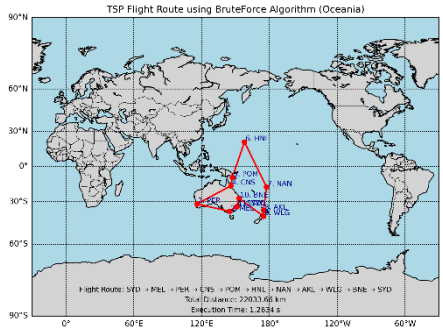

Figure 28. NN Oceania Route

IV. Conclusion

The implementation evaluates two algorithms—Nearest Neighbor (NN) and Brute Force (BF)—for solving the Travelling Salesman Problem (TSP) across five regional datasets: Asia, Europe, America, Africa, and Oceania. Each dataset contains 10 to 13 major international airports. The results are summarized in Table 1 and analyzed further below.

Dataset	Airports	NN Distance	NN Time (s)	BF Distance	BF Time (s)	Difference	NN/BF Ratio	Deviation (%)
Asia	12	26265,29 ~		22269,81	162,94	3995,48	1,179	17,9%
Europe	13	12187,50 ~		10114,82	3391,79	2072,68	1,205	20,5%
America	12	26689,64 ~		25414,92	131,95	1274,72	1,050	5,0%
Africa	10	22221,94 ~		21296,35	1,23	925,59	1,043	4,3%
Oceania	10	29201,02 ~		22033,65	1,26	7167,37	1,325	32,5%

Table 1. Data Analysis

A. Flight Route Accuracy

In terms of accuracy, the Brute Force algorithm consistently yields the shortest total travel distances, as it exhaustively explores all possible permutations. The Nearest Neighbor algorithm, on the other hand, provides faster but sub-optimal solutions. Table 1 shows that the average NN/BF ratio is greater than 1, confirming that NN algorithm consistently produces longer flight routes compared to BF algorithm, which guarantees the optimal solutions.

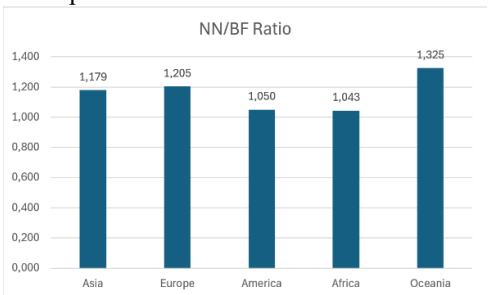

Figure 29. NN/BF Ratio

Geographic complexity and airports distribution likely influence the performance of heuristic algorithms like NN. The more scattered or irregular the region, the more prone NN is to suboptimality. The highest deviation occurs in Oceania, where NN produces a path 32.53% longer than BF. In contrast, Africa shows the smallest deviation, with only 4.35% longer path, suggesting that NN performs better when airport locations are relatively well-aligned geographically. These observations highlight the sensitivity of the NN heuristic to spatial layout. For example, NN performs poorly in regions like Oceania and Asia, where the distributions are more scattered or non-linear.

B. Execution Time

Execution time reveals a stark contrast between the two algorithms. which can be directly attributed to their computational complexity:

- Nearest Neighbor: O(n^2)
  NN visits each airport once, selecting the nearest unvisited airport at each step making it quadratic in complexity. Table 1 shows that NN execution time is near zero.
- Brute Force: O(n!)
  Brute Force evaluates every possible permutation of the cities, making it grow extremely fast with increasing n. Table 1 shows that BF execution time is immense.

The application of both algorithms, nearest neighbor and brute force, has successfully made the flight route, even though each algorithm has its upsides and downsides. The application of approximation algorithm, which is nearest neighbor, does not produce optimal results, but it capable of delivering solutions that fall within acceptable or near-optimal ranges in a short time. While brute force always produces optimal results even though it consumes a lot of time.

Attachment

- GitHub Repository: https://github.com/hafizhperdana/MakalahMatdis25/tree/main/MakalahMatdis
- YouTube Video: https://youtu.be/0USvvii8eBw

Acknowledgment

The author wishes to express deep gratitude to God Almighty for His continuous blessings and guidance, which made it possible to conduct this research and complete the paper smoothly. Heartfelt thanks are also directed at the IF1220 course lecturer, Dr. Ir. Rinaldi Munir, M.T., for his unwavering dedication and outstanding mentorship. His valuable insights, teaching, and support have significantly aided the author in grasping the course material and finishing this assignment. May the knowledge and kindness he has imparted continue to bring meaningful impact.

REFERENCES

[1] K. Rai, L. Madan, and K. Anand, "Research Paper on Travelling Salesman Problem and It's Solution Using Genetic Algorithm," *International Journal of Innovative Research in Technology*, Vol. 1, Issue 1, pp. 103–114, 2014.

[2] D. A. Prasetya, P. T. Nguyen, R. Faizullin, I. Iswanto, and E. F. Armay, "Resolving the Shortest Path Problem using the Haversine Algorithm," *Journal of Critical Reviews*, Vol. 7, Issues 1, pp. 62–64, 2020.

[3] C. Rego, D. Gamboa, F. Glover, and C. Osterman, "Traveling salesman problem heuristics: Leading methods, implementations and latest advances," *European Journal of Operational Research*, pp. 427–441, 2011.

[4] GraphicMaths, "Travelling Salesman Problem," available online: https://graphicmaths.com/computer-science/graph-theory/travelling-salesman-problem/, [accessed: 17 June 2025].

[5] Routific, "Algorithms for the Travelling Salesman Problem," available online: https://www.routific.com/blog/travelling-salesman-problem, [accessed: 18 June 2025].

[6] GeeksForGeeks, "Introduction to Graph Data Structure," available online: https://www.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/, [accessed: 18 June 2025].

[7] Matlab, "Directed and Undirected Graphs," available online: https://www.mathworks.com/help/matlab/math/directed-and-undirected-graphs.html, [accessed: 17 June 2025].

[8] OpenFlights, "Airport, airline and route data," available online: https://github.com/jpatokal/openflights/blob/master/data/airports.dat, [accessed: 18 June 2025].

[9] MovableType, "Calculate Distance, Bearing and More Between Latitude/Longitude Points," available online: https://www.movable-type.co.uk/scripts/latlong.html, [accessed: 19 June 2025].

[10] R. Munir, *Graf (Bagian 1),* Program Studi Teknik Informatika, STEI-ITB, available online: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf, , [accessed: 17 June 2025].

[11] R. Munir, Graf (Bagian 2), Program Studi Teknik Informatika, STEI-ITB, available online: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/21-Graf-Bagian2-2024.pdf, , [accessed: 17 June 2025].

[12] R. Munir, Graf (Bagian 3), Program Studi Teknik Informatika, STEI-ITB, available online: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/22-Graf-Bagian3-2024.pdf, [accessed: 17 June 2025].

STATEMENT

Hereby, I declare that this paper I have written is my own work, not a reproduction or translation of someone else's paper, and not plagiarized.

Bandung, 20 June 2025

Moh. Hafizh Irham Perdana (13524025)