

# Graph-Based Pathfinding Algorithm Project



## **Team 03**

Hyunsu Kim (21900217)

Dayoung Shim (22100421)

Eunji Hwang (22100809)

Enow Divine Akong (22347018)

## Introduction

This project focuses on optimizing flight paths for Korean Airlines between Jeju International Airport (CJU) and Gimpo International Airport (GMP). We employ the A\* Algorithm to find the shortest routes under clear and severe weather conditions. The goal is to minimize travel time, fuel consumption, and operational costs by estimating realistic weight values for flight routes and incorporating severe weather impacts.

## Project Deliverables

### 1. *How do you estimate weight values for all the edges of the graph? Note that the estimated weight values should closely reflect the actual scenario.*

To estimate the weight values for all edges of the graph, we used the actual distances between nodes. Specifically, the haversine function was used to calculate the distances between nodes on the Earth's surface. The haversine function takes the latitude and longitude of two nodes as input and computes the great-circle distance between them, which is the shortest distance over the Earth's surface. The formula for the haversine function is as follows:

$$a = \sin^2(\Delta\phi / 2) + \cos(\phi_1) \cdot \cos(\phi_2) \cdot \sin^2(\Delta\lambda / 2)$$
$$c = 2 \cdot \operatorname{atan2}(a^{0.5}, (1-a)^{0.5})$$
$$d = R \cdot c$$

where  $\phi_1$  and  $\phi_2$  are the latitudes of the points (in radians),  $\lambda_1$  and  $\lambda_2$  are the longitudes of the points (in radians),  $\Delta\phi$  is  $\phi_2 - \phi_1$ ,  $\Delta\lambda$  is  $\lambda_2 - \lambda_1$ ,  $R$  is the Earth's radius (mean radius = 6,371 km), and  $d$  is the distance between the two points.

Using this method, we calculated the actual distance between each pair of nodes, and this distance was used as the weight for the edges. By using actual distances as weights, the graph more accurately reflects real-world scenarios.

	Name	Start Longitude (deg)	Start Latitude (deg)	End Longitude (deg)	End Latitude (deg)	Weight
0	A582	127.348575	36.838358	127.500925	36.714367	19.344404
1	A582	127.500925	36.714367	127.819172	36.453422	40.612232
2	A582	127.819172	36.453422	128.029970	36.279140	27.051894
3	A582	128.029970	36.279140	128.590770	35.809835	72.562918
4	A582	128.590770	35.809835	128.774025	35.503350	37.888564

### 2. *How do you represent the graph? Provide the results and reasoning behind your decision.*

Among various methods to represent the graph, we chose to use a dictionary structure to implement the adjacency list. Because the dataset that we were using contains many nodes but comparatively fewer edges relative to the number of nodes, we chose an adjacency list representation that is suitable for such datasets. In this approach, each node is a key, and the value is another dictionary containing the neighboring nodes and the corresponding edge weights. The graph is represented as follows:

```
{(127.348575, 36.83835833): [(127.500925, 36.71436667), 19.344403897482156),
  ((127.0316667, 37.09436944), 40.038619195066055),
  ((127.500925, 36.71436667), 19.344403897482156),
  ((127.0316667, 37.09436944), 40.038619195066055)],
(127.500925, 36.71436667): [(127.348575, 36.83835833), 19.344403897482156),
  ((127.8191722, 36.45342222), 40.612231737684354),
  ((127.348575, 36.83835833), 19.344403897482156),
  ((127.8191722, 36.45342222), 40.612231737684354)],
(127.8191722, 36.45342222): [(127.500925, 36.71436667), 40.612231737684354),
  ((128.0299697, 36.27914028), 27.05189393964307),
  ((127.500925, 36.71436667), 40.612231737684354),
  ((128.0299697, 36.27914028), 27.05189393964307)],
(128.0299697, 36.27914028): [(127.8191722, 36.45342222), 27.05189393964307),
  ((128.5907703, 35.80983494), 72.56291773343138),
  ((127.8191722, 36.45342222), 27.05189393964307),
  ((128.5907703, 35.80983494), 72.56291773343138),
  ((127.9913598, 36.56138491), 31.573739041449645)],
(128.5907703, 35.80983494): [(128.0299697, 36.27914028), 72.56291773343138),
  ((128.7740249, 35.50335003), 37.88856389676699),
  ((127.8185057, 35.51765317), 76.96041659739552),
  ((127.6144444, 35.863475), 88.2119645680769),
  ((129.0811111, 35.90349722), 45.40103138042188),
  ((128.5609923, 35.47970869), 36.80685210509806),
  ((128.5609923, 35.47970869), 36.80685210509806),
  ((127.8185057, 35.51765317), 76.96041659739552),
```

This method of graph representation offers several advantages in terms of space efficiency, accessibility, and flexibility. It is space-efficient because it does not store unnecessary edges, making memory usage efficient. It provides easy access to the nodes connected to a particular node and their weights, facilitating data exploration. Additionally, it allows for dynamic addition or removal of nodes and edges. These characteristics make it intuitive to understand the structure of the graph.

However, if we represent the graph with latitude and longitude information, it might become difficult to identify where each node is located when we see that graph representation again later. For this reason, we decided to represent the adjacency list using a dictionary where instead of inserting latitude and longitude information directly, we inserted the names of the nodes. To achieve this, we assigned names to each node sequentially, and based on these assigned node names, we constructed a new adjacency list.

<table><thead><tr><th>latitude</th><th>longitude</th><th>Node</th></tr></thead><tbody><tr><td>127.348575</td><td>36.838358</td><td>Node0</td></tr><tr><td>127.500925</td><td>36.714367</td><td>Node1</td></tr><tr><td>127.819172</td><td>36.453422</td><td>Node2</td></tr><tr><td>128.029970</td><td>36.279140</td><td>Node3</td></tr><tr><td>128.590770</td><td>35.809835</td><td>Node4</td></tr><tr><td>...</td><td>...</td><td>...</td></tr><tr><td>127.952743</td><td>36.843677</td><td>Node163</td></tr><tr><td>128.029970</td><td>36.279140</td><td>Node164</td></tr><tr><td>128.135278</td><td>37.606944</td><td>Node165</td></tr><tr><td>128.135278</td><td>37.606944</td><td>Node166</td></tr><tr><td>126.624120</td><td>33.384605</td><td>Node167</td></tr></tbody></table>	latitude	longitude	Node	127.348575	36.838358	Node0	127.500925	36.714367	Node1	127.819172	36.453422	Node2	128.029970	36.279140	Node3	128.590770	35.809835	Node4	...	...	...	127.952743	36.843677	Node163	128.029970	36.279140	Node164	128.135278	37.606944	Node165	128.135278	37.606944	Node166	126.624120	33.384605	Node167	<pre>{'Node132': [('Node133', 19.3444), ('Node140', 40.0386), ('Node133', 19.3444), ('Node140', 40.0386)], 'Node133': [('Node132', 19.3444), ('Node134', 40.6122), ('Node132', 19.3444), ('Node134', 40.6122)], 'Node134': [('Node133', 40.6122), ('Node164', 27.0519), ('Node133', 40.6122), ('Node164', 27.0519)], 'Node164': [('Node134', 27.0519), ('Node136', 72.5629), ('Node134', 27.0519), ('Node136', 72.5629), ('Node61', 31.5737)], 'Node136': [('Node164', 72.5629), ('Node137', 37.8886), ('Node121', 76.9604), ('Node123', 88.212), ('Node125', 45.401), ('Node151', 36.8069), ('Node151', 36.8069), ('Node121', 76.9604),</pre>
latitude	longitude	Node																																			
127.348575	36.838358	Node0																																			
127.500925	36.714367	Node1																																			
127.819172	36.453422	Node2																																			
128.029970	36.279140	Node3																																			
128.590770	35.809835	Node4																																			
...	...	...																																			
127.952743	36.843677	Node163																																			
128.029970	36.279140	Node164																																			
128.135278	37.606944	Node165																																			
128.135278	37.606944	Node166																																			
126.624120	33.384605	Node167																																			
Designate the node name for each node	New adjacency list																																				

```
{'Node132': [('Node133', 19.3444),
 ('Node140', 40.0386),
 ('Node133', 19.3444),
 ('Node140', 40.0386)],
 'Node133': [('Node132', 19.3444),
 ('Node134', 40.6122),
 ('Node132', 19.3444),
 ('Node134', 40.6122)],
 'Node134': [('Node133', 40.6122),
 ('Node164', 27.0519),
 ('Node133', 40.6122),
 ('Node164', 27.0519)],
 'Node164': [('Node134', 27.0519),
 ('Node136', 72.5629),
 ('Node134', 27.0519),
 ('Node136', 72.5629),
 ('Node61', 31.5737)],
 'Node136': [('Node164', 72.5629),
 ('Node137', 37.8886),
 ('Node121', 76.9604),
 ('Node123', 88.212),
 ('Node125', 45.401),
 ('Node151', 36.8069),
 ('Node151', 36.8069),
 ('Node121', 76.9604),
```

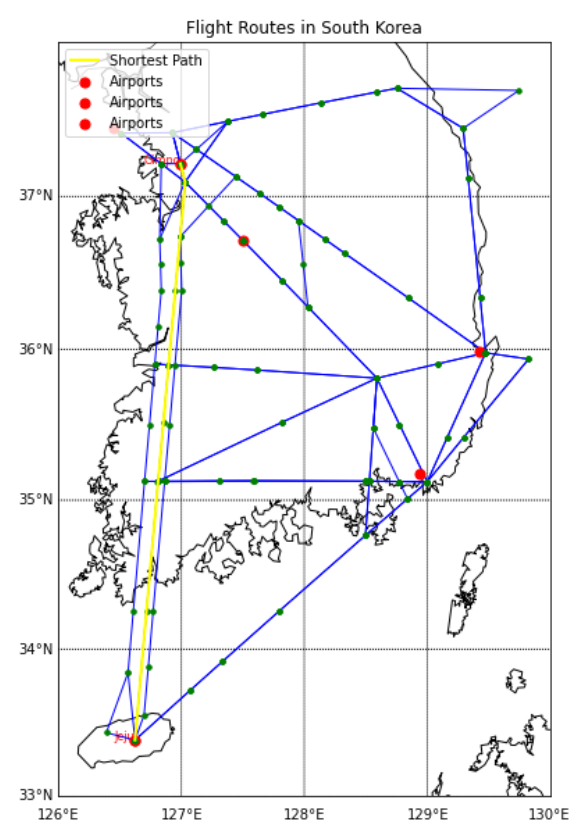
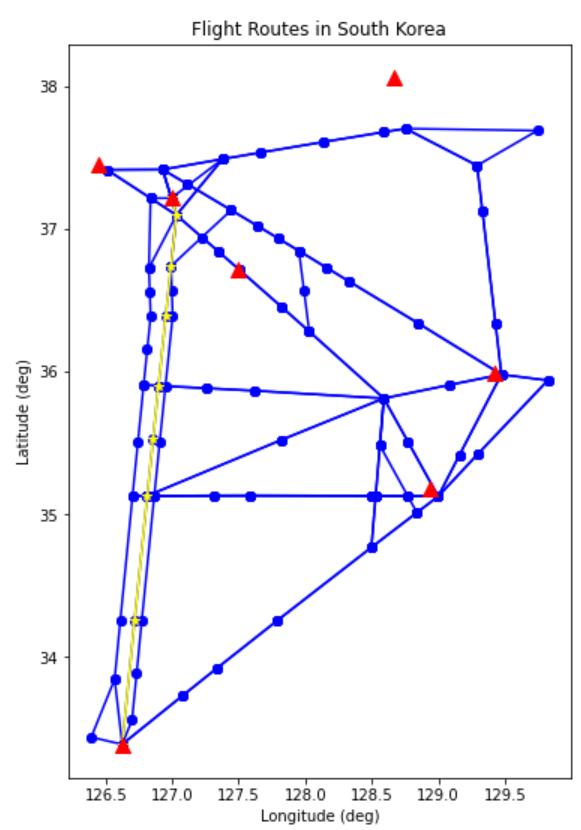
Designate the node name for each node

Additionally, we marked the location of each node on a map to provide information about where each node is actually positioned.

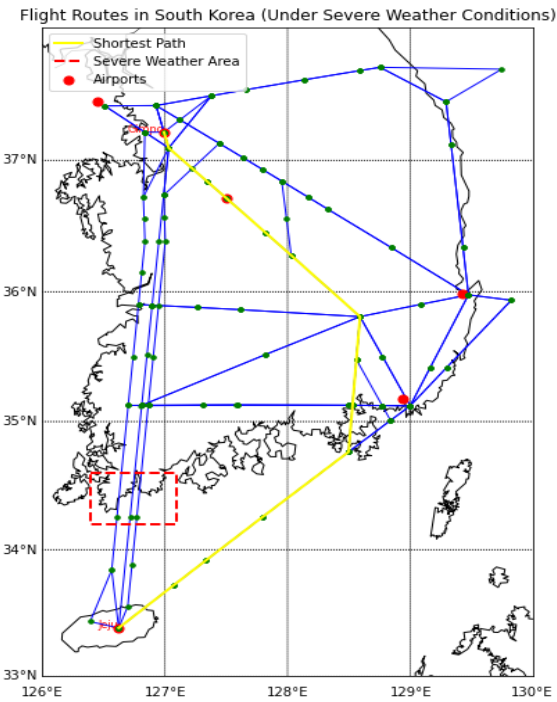
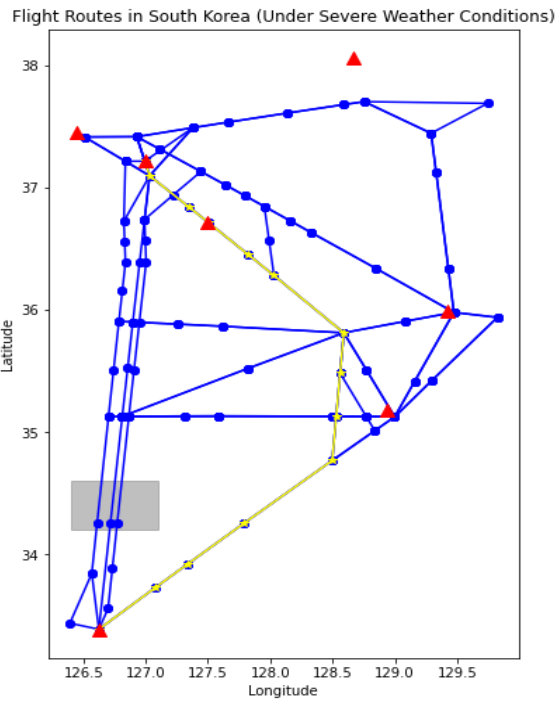
3. Consider selecting the A\* algorithm to determine the shortest path for this problem. What is the shortest path under clear weather conditions? What is the shortest path under severe weather conditions?

Our team obtained the shortest path with two versions of the code. We obtained the total distance for each and performed a cross-check. The result is as follows.

a) The shortest path under clear weather conditions

Code ver1 (main code)	Code ver2
	
<pre>[(126.9931833, 37.21361389), (127.0316667, 37.09436944), (126.9909903, 36.73705139), (126.9514333, 36.38625278), (126.8969425, 35.89779972), (126.8553833, 35.52115), (126.8121194, 35.12623581), (126.7170492, 34.25407056), (126.6241196, 33.38460522)]</pre>	<pre>[(126.9931833, 37.21361389), (127.0316667, 37.09436944), (126.9909903, 36.73705139), (126.9514333, 36.38625278), (126.8969425, 35.89779972), (126.8553833, 35.52115), (126.8121194, 35.12623581), (126.7170492, 34.25407056), (126.6241196, 33.38460522)]</pre>
Total distance: 427.85374155469265 km	Total distance (code2): 427.85374155469265 km

**b) The shortest path under severe weather conditions**

Code ver1 (team code)	Code ver2
	
<pre>[(126.9931833, 37.21361389), (127.0316667, 37.09436944), (127.348575, 36.83835833), (127.500925, 36.71436667), (127.8191722, 36.45342222), (128.0299697, 36.27914028), (128.5907703, 35.80983494), (128.5609923, 35.47970869), (128.529722, 35.1266665), (128.4976922, 34.76535483), (127.792902, 34.25466028), (127.331378, 33.91454919), (127.0730556, 33.72222222), (126.6241196, 33.38460522)]</pre>	<pre>[(126.9931833, 37.21361389), (127.0316667, 37.09436944), (127.348575, 36.83835833), (127.500925, 36.71436667), (127.8191722, 36.45342222), (128.0299697, 36.27914028), (128.5907703, 35.80983494), (128.5609923, 35.47970869), (128.529722, 35.1266665), (128.4976922, 34.76535483), (127.792902, 34.25466028), (127.331378, 33.91454919), (127.0730556, 33.72222222), (126.6241196, 33.38460522)]</pre>
Total distance: 560.7155840514827 km	Total distance (code2): 560.7155840514827 km

The shortest path result graph of the two codes and the path list of each code are exactly the same. In addition, the total distance in each code is the same. So, we concluded that the shortest path result of each code is actually the shortest path.

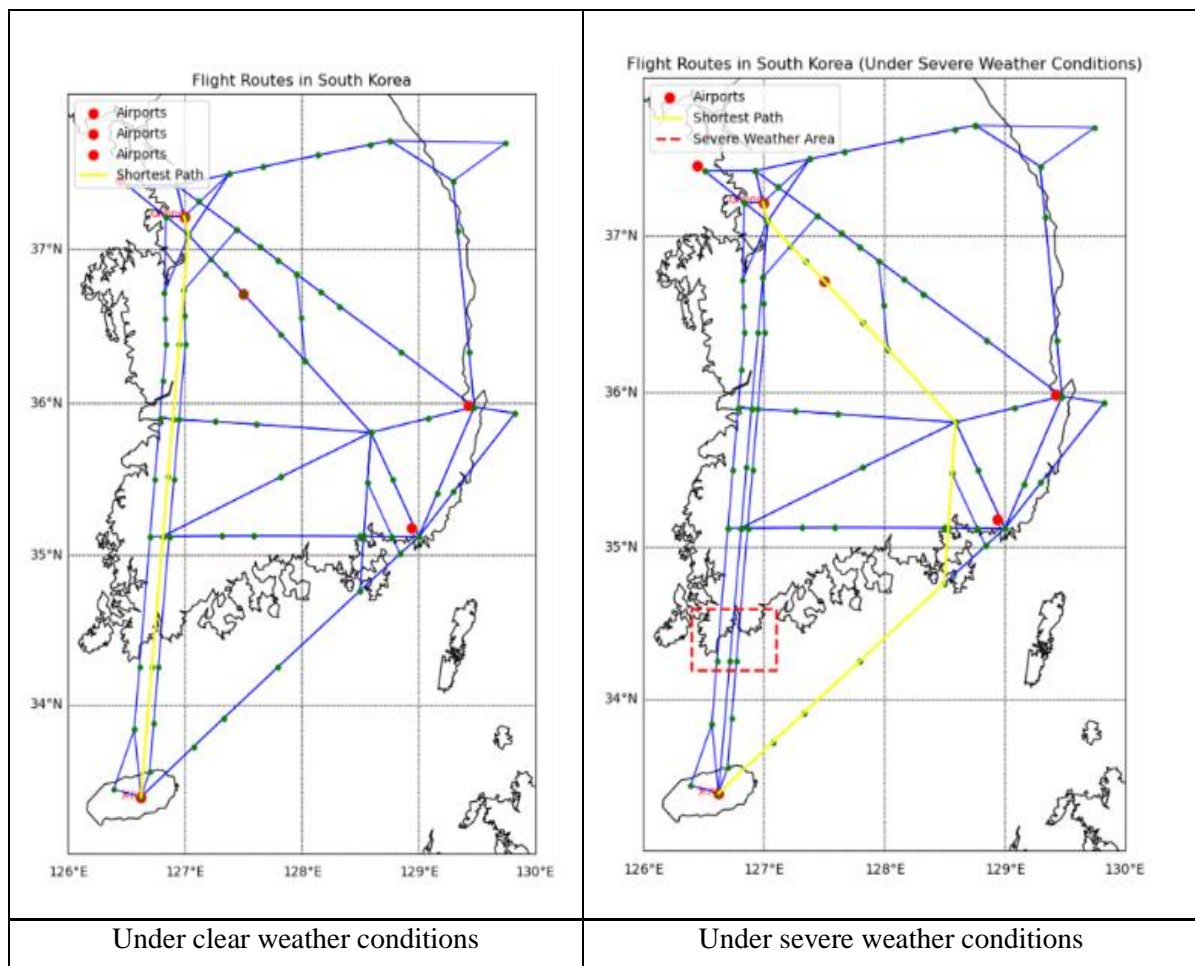


4. *Do you believe that the results for the shortest paths are the same for both clear and severe weather conditions? Provide an explanation for why they may be identical or different.*

Naturally, the shortest route differs in clear weather and severe weather conditions. The severe weather interval is on the shortest route in clear weather. Since the plane should not pass that interval, it will return to another route, which means it differs from the shortest route in clear weather.

5. *What is your approach to incorporating the impact of severe weather conditions into the implementation?*

[Figure 1. Graphs Showing Routes with and without Severe Weather Conditions]



Our approach involves handling three nodes within the severe weather conditions polygon. By excluding these three nodes, we aim to secure the possible shortest path. The nodes inside the polygon are:  $[(126.7170492, 34.25407056), (126.7715867, 34.25392406), (126.6101618, 34.25428772)]$ .

[Figure 2. Shortest path using A\* algorithm, with and without severe weather conditions]

<pre> [(126.9931833, 37.21361389), (127.0316667, 37.09436944), (126.9909903, 36.73705139), (126.9514333, 36.38625278), (126.8969425, 35.89779972), (126.8553833, 35.52115), (126.8121194, 35.12623581), (126.7170492, 34.25407056), (126.6241196, 33.38460522)] </pre>	<pre> [(126.9931833, 37.21361389), (127.0316667, 37.09436944), (127.348575, 36.83835833), (127.500925, 36.71436667), (127.8191722, 36.45342222), (128.0299697, 36.27914028), (128.5907703, 35.80983494), (128.5609923, 35.47970869), (128.529722, 35.1266665), (128.4976922, 34.76535483), (127.792902, 34.25466028), (127.331378, 33.91454919), (127.0730556, 33.72222222), (126.6241196, 33.38460522)] </pre>
Under clear weather conditions	Under severe weather conditions

In detail, first examining the route without severe weather conditions as shown in Figure 2, the path starts from Gimpo [126.9931833, 37.21361389], traverses through various nodes, and reaches Jeju [126.6241196, 33.38460522]. Additionally, it was verified that the coordinates [126.7170492, 34.25407056], marked with a red box, lie within the severe weather conditions polygon.

Therefore, to avoid the severe weather conditions area, our team removed the three nodes within the polygon. This results in a different shortest path from Gimpo to Jeju. Consequently, excluding the three nodes within the severe weather conditions area, [(126.7170492, 34.25407056), (126.7715867, 34.25392406), (126.6101618, 34.25428772)], increases the number of nodes in the shortest path from 9 to 14, thereby increasing the total distances, which are approximately 427.9 km and 560.7 km.

## 6. *Do you think that the shortest path identified by your algorithm represents a global optimum?*

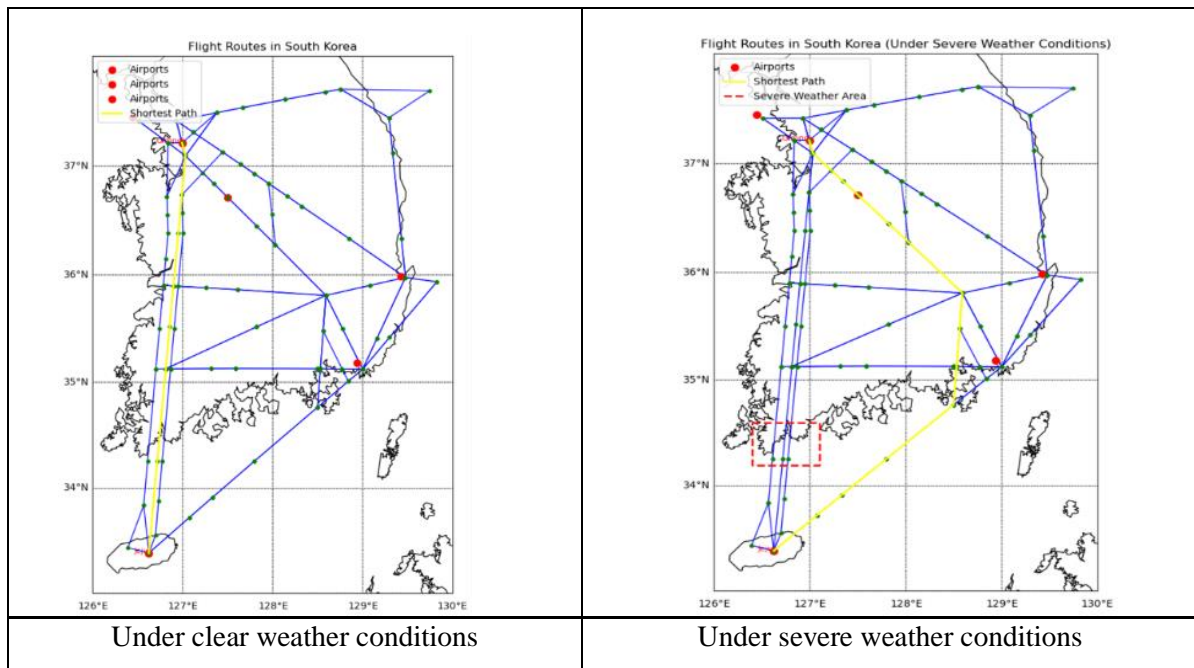
Yes, the shortest path identified by our algorithm represents the global Optimum because of the following: A\* implementation verifies Completeness, Admissibility and Consistency conditions and we also used the DIJKSTRA algorithm to verify the total distance provided by the A\* algorithm and the distances were the same. See details below:

### a) **A\* Implementation is at Global Optimum when the following are verified: Completeness, Admissibility and Consistency.**

#### i. **Completeness:** A\* is complete if it always finds a solution when one exists.

For both cases, that is, without weather conditions and with weather condition implementation using A\*, we observe that the yellow line starts at one red triangle (the start airport—GIMPO) and ends at another red triangle (the goal airport—JEJU). Therefore, the algorithm is complete, as illustrated below the graph.





**ii. Admissibility:** A\* is admissible if the heuristic function used is never overestimating the cost to reach the goal.

We can see that the yellow line which is the shortest path follows a logical and reasonably direct route between nodes, with no unnecessary long detours. It's therefore Admissible.

**iii. Consistency (Monotonicity):** The distance between x and the goal; minus the distance between some other node y satisfies the following:  $h(x) \leq \text{Actual} + h(y)$ .

The heuristic function used is Haversine distance which estimates the straight-line distance between two points on the Earth's surface. The consistency property is inherently satisfied because the Haversine distance follows the triangle inequality: the direct distance between two points is never more than the sum of the distances when passing through an intermediate point.

The path (**yellow line**) is smooth and direct, without any sudden large jumps or backtracking, indicating that each step taken is consistent with the heuristic estimate. The path avoids the gray forbidden zone (For weather conditions), which means it respects the constraints and still finds the shortest route.

**b) Comparing the Distance produced by the A\* algorithm and the distance produced by the DIJKSTRA algorithm.**

As seen in the table below:

No	Type of Algorithm	Distance under clear weather conditions	Distance under severe weather conditions
1	DIJKSTRA	427.85374	560.7155
2	A*	427.85374	560.7155

**7. Can you explain the step-by-step procedure of the pseudo-code you implemented to tackle this project?**

**a) Preparation and Setup for Flight path optimization**

In the preliminary phase of the project, we focused on three core areas: outlining the project's objectives, visualizing flight paths, and calculating weights Using the Haversine Formula. Initially, we defined our goal to develop a tool for calculating optimal flight routes. Next, visualizing flight routes and airport locations in South Korea territory by using Python's Matplotlib library and the Basemap toolkit for geographical mapping. We then used the Haversine formula to compute the great-circle distances between airways, providing accurate weights for the edges in our graph-based model. Pseudo code for these steps are as follows:

```
# Phase 1: Project Initialization
# Step 1: Define project objectives
project_objectives = [
    "Develop a tool for calculating optimal flight routes",
    "Visualize flight paths and airport locations in South Korea",
    "Calculate weights using the Haversine Formula"]
# Step 2: Visualize flight paths and airport locations
# Use Matplotlib and Basemap toolkit for geographical mapping in Python
visualize_flight_paths()
# Step 3: Calculate weights using Haversine Formula
# Compute great-circle distances between airways
compute_weights_with_haversine()
```

**b) Flight path optimization using a graph-based pathfinding algorithm**

In the second phase of our project, we applied the A\* algorithm to identify the shortest flight paths under two distinct scenarios. In clear weather conditions, we implemented the A\* algorithm, using a heuristic function which is already defined via the Haversine formula based on the straight-line distance between nodes. We further optimized the search process by incorporating a KDTree for nearest-neighbor searches, which significantly enhanced our ability to quickly identify the closest neighboring nodes. The final route was calculated for total optimized distance, and then visualized on the map. Pseudo code for these steps are as follows:

```
# Phase 2: Applying A* Algorithm for Shortest Flight Paths
# Step 1: Define scenarios
scenarios = [ "Clear weather conditions"]
# Step 2: Implement A* algorithm with heuristic function
if scenario == "Clear weather conditions":
    implement_a_star_algorithm_with_heuristic()
# Step 3: Optimize search process using KDTree
# Step 4: Calculate final route for optimized distance
# Step 5: Visualize final route on the map
```

In severe weather conditions, we adapted weather information defined by polygon coordinates. Specifically, we modified the graph to exclude any nodes and edges that fell within the severe weather polygon, a key step in rerouting the flight paths to avoid areas affected by adverse conditions.

Subsequently, we reapplied the A\* algorithm using these adjustments, which led to the identification of a different set of shortest paths specifically tailored to navigate around the severe weather. The outcomes were visualized, effectively highlighting the impact of severe weather on flight routes, and also calculating the total distance. Pseudo code for these steps are as follows:

```
# Phase 3: Adapting to Severe Weather Conditions
# Step 1: Define severe weather polygon coordinates
# Step 2: Modify graph to exclude nodes and edges within severe weather polygon
# Step 3: Reapply A* algorithm with adjusted graph
# Step 4: Visualize outcomes of adjusted flight paths
# Step 5: Calculate total distance of adjusted flight paths
```

### c) Verification

In the verification phase of our project, we implement **Dijkstra's algorithm** alongside the A\* algorithm to ensure that our solutions were optimal under both clear and severe weather conditions. By comparing the outcomes of both algorithms, we observed that the total optimization distances were the same for each method. Additionally, we confirmed that while Dijkstra's algorithm provided accurate path calculations, the A\* algorithm was superior in terms of efficiency, processing paths more quickly due to its heuristic-based decision-making. This comparative analysis validated the robustness of the A\* algorithm and its suitability for real-time flight path optimization across varying air travel conditions. Pseudo code for these steps are as follows:

```
# Phase 4: Verification with Dijkstra's Algorithm
# Step 1: Implement Dijkstra's algorithm
# Step 2: Implement A* algorithm
# Step 3: Compare optimization distances of both algorithms
# Step 4: Analyze efficiency of algorithms
# Step 5: Validate A* algorithm for real-time flight path optimization
```

### References

- OpenAI. (2024). ChatGPT [Large language model]. <https://chat.openai.com>
- Rachmawati, D., & Gustin, L. (2020). Analysis of Dijkstra's algorithm and A\* algorithm in shortest path problem. *Journal of Physics. Conference Series*, 1566(1), 012061. <https://doi.org/10.1088/1742-6596/1566/1/012061>
- Ramee, Coline & Kim, Junghyun & Deguignet, Marie & Justin, Cedric & Briceno, Simon & Mavis, Dimitri. (2020). Aircraft Flight Plan Optimization with Dynamic Weather and Airspace Constraints