Flight Path Optimization Using Dijkstra’s Algorithm

김 주 호, 신 의 진

(Joo-Ho Kim[1], Yi-Jin Sin[2])

[1]School of Mechanical & Control Engineering, Handong Global University

[2]School of Computer Science & Electrical Engineering, Handong Global University

Abstract: The complex problems around us can be solved more efficiently using algorithms. In this report, one of various algorithms, Dijkstra's algorithm, is applied to flight path optimization system. The flight path optimization system provides users with the fastest and safest path based on the location and path information of each airport. According to the results attached to the report, it is expected that this system will show good performance without any problems.

Keywords: Dijkstra’s Algorithm, Flight Path Optimization, Shortest path, CCW Algorithm

I. Introduction

Polygon information = ([(128.7, 36.6), (128.7, 37.6), (129.3, 37.6), (129.3, 36.6)])

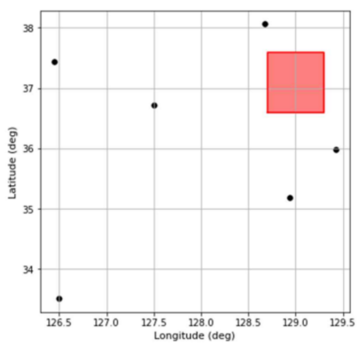


Fig 1. Visualization of coordinates

In this report, it was discussed how to find the safest and shortest flight path based on the location information of each airport and the location information of convective areas. The Dijkstra algorithm was used as an flight path optimization method. This system calculates weights for paths between each airport based on the Haversine formula. So, it provides a path from the starting point to the destination with the lowest weight. It also receives information about the convective area and provides a path to avoid the area. Therefore, it is expected that users will be provided with the optimal path efficiently and conveniently.

II. Program composition and theory

2.1 Program description

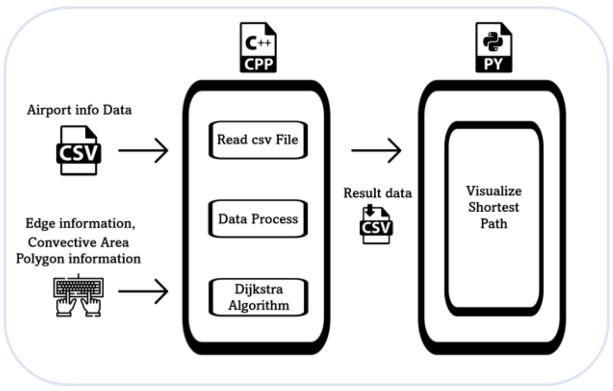


Fig 2. Visualization of program description

The composition of the program can be divided into four operations. Most of the operations are performed in C++, and Python is used for graph visualization. First, a csv file containing information of each airport is read and stored in an appropriate data type. Next, weather information and path information of each airport are inputted by the user and stored. In the third step, the Dijkstra algorithm is applied based on the stored data. The program outputs the optimal path by dividing it into a case where weather information is considered and a case where it is not. Finally, the resulting data is saved as a csv file and visualized through Python. Fig 3. represents the program consists of three code files.

- AdjacencyList.h
- Main.cpp
- AL_project1_visualize_plot.ipynb

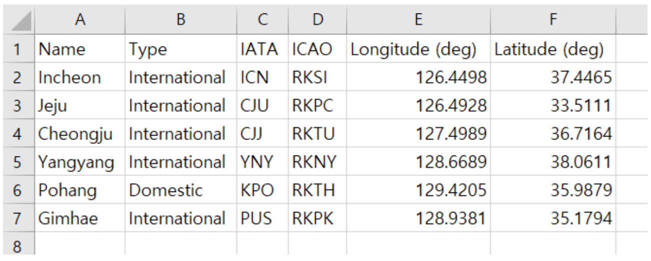
Fig 3. List of program codes

AdjacencyList.h file contains class AdjList. The class AdjList creates an undirected, weighted graph using the adjacency list, and the class AdjList includes graph-related operation functions. It also includes the Dijkstra function that finds the shortest path based on the weights of the graph.

Main.cpp file is the main file of the program. It reads csv data and creates a graph using class AdjList of AdjacencyList.h. Provide a menu to allow the user to select an action. The user can request the shortest path that does not consider the region and the shortest path that does not consider the region by inputting the desired origin, destination, and one or more bad weather regions. Also, the user can request all edge information in the current graph. In addition, it includes a function to determine an edge passing through a bad weather region and a function to save the calculated shortest path to a csv file.

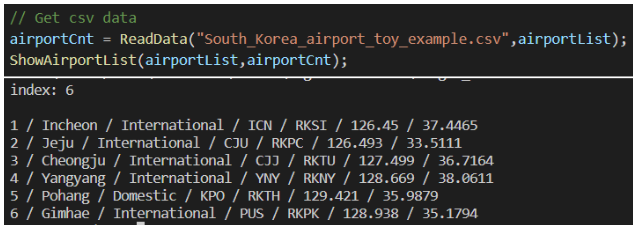
AL_project1_visualize_plot.ipynb file contains the code to visualize the graph of the initial condition or the graph of the result for the shortest path. How to use the program is included in the **README.md** file.

2.2 Reading csv file and data (C++)



	A	B	C	D	E	F
1	Name	Type	IATA	ICAO	Longitude (deg)	Latitude (deg)
2	Incheon	International	ICN	RKSI	126.4498	37.4465
3	Jeju	International	CJU	RKPC	126.4928	33.5111
4	Cheongju	International	CJJ	RKTU	127.4989	36.7164
5	Yangyang	International	YNY	RKNY	128.6689	38.0611
6	Pohang	Domestic	KPO	RKTH	129.4205	35.9879
7	Gimhae	International	PUS	RKPK	128.9381	35.1794
8						

Fig 4. The csv file example

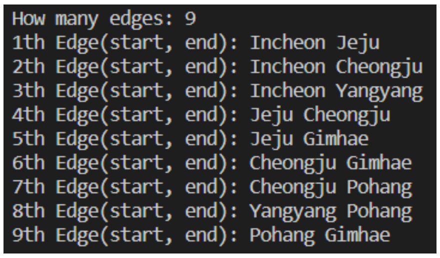


```
// Get csv data
airportCnt = ReadData("South_Korea_airport_toy_example.csv",airportList);
ShowAirportList(airportList,airportCnt);
```

```
index: 6

1 / Incheon / International / ICN / RKSI / 126.45 / 37.4465
2 / Jeju / International / CJU / RKPC / 126.493 / 33.5111
3 / Cheongju / International / CJJ / RKTU / 127.499 / 36.7164
4 / Yangyang / International / YNY / RKNY / 128.669 / 38.0611
5 / Pohang / Domestic / KPO / RKTH / 129.421 / 35.9879
6 / Gimhae / International / PUS / RKPK / 128.938 / 35.1794
```

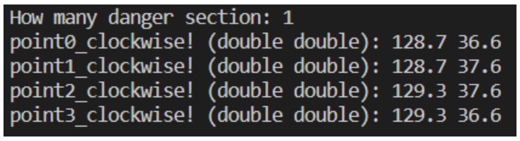
Fig 5. Output of **ReadData** function in the code

The given csv file is structured as shown in Fig 4. Each data is stored separately in the **airportList** for each row through the **ReadData** function. As data, the longitude and latitude information of each airport is mainly used, and the distance between airports obtained using the Haversine Formula is used as a weight. Fig 5. is the output of the csv file read through **ReadData** function.



```
How many edges: 9
1th Edge(start, end): Incheon Jeju
2th Edge(start, end): Incheon Cheongju
3th Edge(start, end): Incheon Yangyang
4th Edge(start, end): Jeju Cheongju
5th Edge(start, end): Jeju Gimhae
6th Edge(start, end): Cheongju Gimhae
7th Edge(start, end): Cheongju Pohang
8th Edge(start, end): Yangyang Pohang
9th Edge(start, end): Pohang Gimhae
```

Fig 6. The path information for each airport

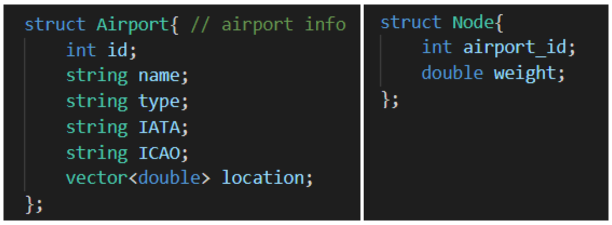


```
How many danger section: 1
point0_clockwise! (double double): 128.7 36.6
point1_clockwise! (double double): 128.7 37.6
point2_clockwise! (double double): 129.3 37.6
point3_clockwise! (double double): 129.3 36.6
```

Fig 7. The csv file example

Path information for each airport inputs the number of paths and the name of the airport in the order of start and end. Next, enter the number of convective areas, and enter each coordinate of the area in the order of longitude and latitude.

2.3 Data structure description



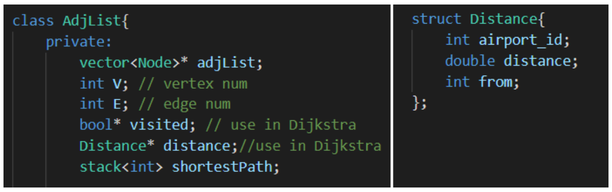
```
struct Airport{ // airport info
    int id;
    string name;
    string type;
    string IATA;
    string ICAO;
    vector<double> location;
};
```

```
struct Node{
    int airport_id;
    double weight;
};
```

Fig 8. The structure Airport and Node

For the data structure, undirected, weighted graph implemented as an Adjacency List was used. The airport is the vertex of the graph, and the path between the airports is the edge of the graph.

Unlike the Adjacency matrix, in Adjacency List, node can be copied multiple times and stored in a graph. Therefore, using a node containing all airport information as a vertex in the graph is expected to consume a lot of memory, so the node used in the graph is a structure that stores only the weight of the airport's id and edge.

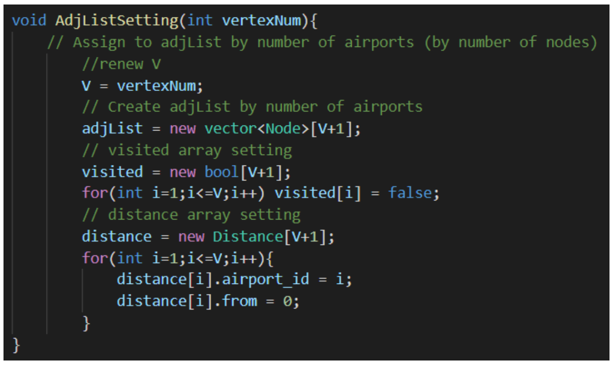


```
class AdjList{
    private:
        vector<Node>* adjList;
        int V; // vertex num
        int E; // edge num
        bool* visited; // use in Dijkstra
        Distance* distance;//use in Dijkstra
        stack<int> shortestPath;
```

```
struct Distance{
    int airport_id;
    double distance;
    int from;
};
```

Fig 9. The class AdjList & structure Distance

The graph is implemented in AdjList class. These are the private variables in the AdjList class. After reading the csv file from **Main.cpp**, declaring the AdjList class object to create the Adjacency list for the number of airports.



```
void AdjListSetting(int vertexNum){
    // Assign to adjList by number of airports (by number of nodes)
        //renew V
        V = vertexNum;
        // Create adjList by number of airports
        adjList = new vector<Node>[V+1];
        // visited array setting
        visited = new bool[V+1];
        for(int i=1;i<=V;i++) visited[i] = false;
        // distance array setting
        distance = new Distance[V+1];
        for(int i=1;i<=V;i++){
            distance[i].airport_id = i;
            distance[i].from = 0;
        }
}
```

Fig 10. The setting of adjacency list

After reading the csv file from **Main.cpp**, declaring the AdjList class object to create the adjacency list creates the adjacency list for the number of airports. Fig 11. is as follows. (Adjacency list when there are 6 airports)

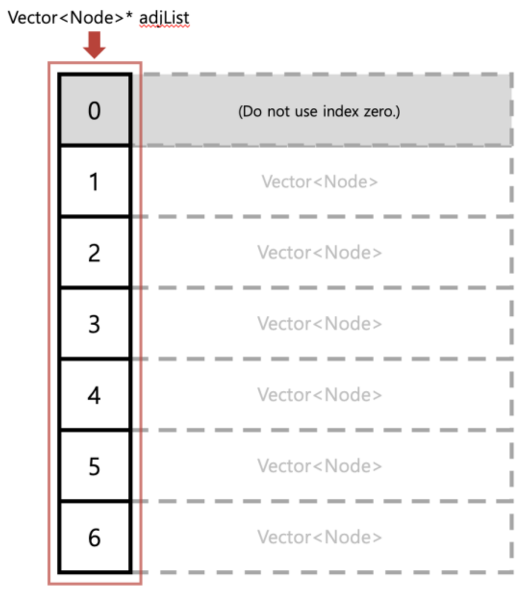
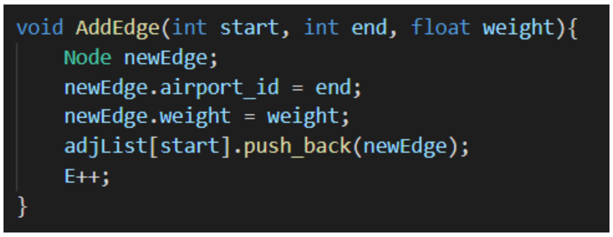


Fig 11. Visualization of the vector<node>* **adjList**



```
void AddEdge(int start, int end, float weight){
    Node newEdge;
    newEdge.airport_id = end;
    newEdge.weight = weight;
    adjList[start].push_back(newEdge);
    E++;
}
```

Fig 12. Definition of the function for adding edges

Declare the AdjList class in **Main.cpp**, and enter the id of the start airport and the id of the end airport through the **GetEdge()** function, and pass it to the **AddEdge()** function of the AdjList class, and add the corresponding edge to the adjacency list.

2.4 Haversine Formula

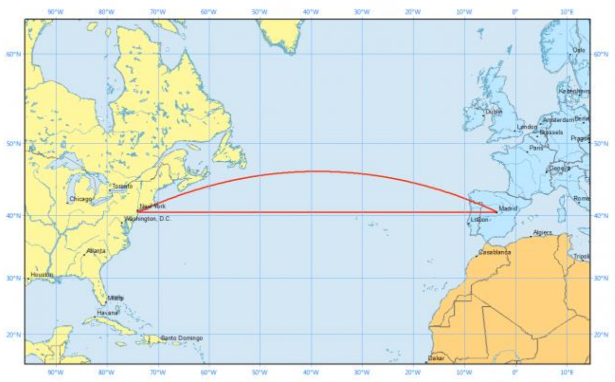


Fig 13. Distance by Harversine formula

In this study, each airport is defined as a node, and the length between each airport is defined as the edge weight.

In this case, the weight can be obtained using the Haversine Formula.

Because the earth exists in the form of a sphere rather than a plane, the curvature must be considered when calculating the distance between two points on the earth. If you simply use the "Euclidean distance" method to find the distance between two points, a large error will occur due to the curvature. Therefore, in this case, the Haversine Formula can be used to obtain the distance considering the curvature.

$$\Theta = \frac{d}{r}$$

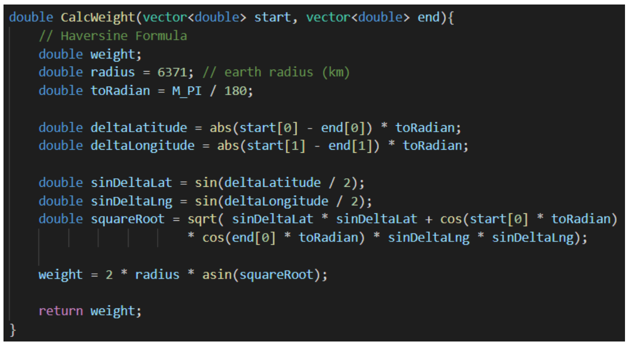
$$\operatorname{hav}(\theta) = \sin^2\left(\frac{\theta}{2}\right) = \frac{1-\cos(\theta)}{2}$$

$$\operatorname{hav}(\Theta) = \operatorname{hav}(\varphi_2 - \varphi_1) + \cos(\varphi_1)\cos(\varphi_2)\operatorname{hav}(\lambda_2 - \lambda_1)$$

$$d = r\operatorname{archav}(h) = 2r\arcsin\left(\sqrt{h}\right)$$

$$\begin{aligned} d &= 2r\arcsin\left(\sqrt{\operatorname{hav}(\varphi_2 - \varphi_1) + \cos(\varphi_1)\cos(\varphi_2)\operatorname{hav}(\lambda_2 - \lambda_1)}\right) \\ &= 2r\arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos(\varphi_1)\cos(\varphi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right) \end{aligned}$$

Fig 14. Haversine formula

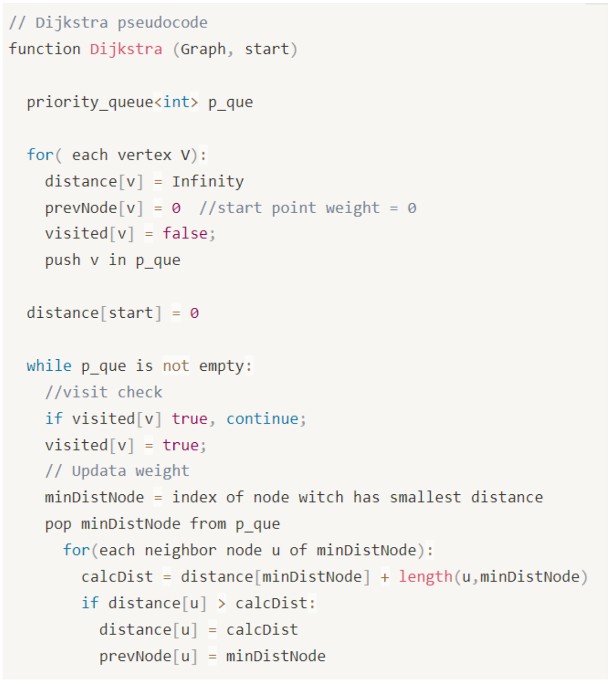


```
double CalcWeight(vector<double> start, vector<double> end){
    // Haversine Formula
    double weight;
    double radius = 6371; // earth radius (km)
    double toRadian = M_PI / 180;

    double deltaLatitude = abs(start[0] - end[0]) * toRadian;
    double deltaLongitude = abs(start[1] - end[1]) * toRadian;

    double sinDeltaLat = sin(deltaLatitude / 2);
    double sinDeltaLng = sin(deltaLongitude / 2);
    double squareRoot = sqrt( sinDeltaLat * sinDeltaLat + cos(start[0] * toRadian)
                    * cos(end[0] * toRadian) * sinDeltaLng * sinDeltaLng);

    weight = 2 * radius * asin(squareRoot);

    return weight;
}
```

Fig 15. Definition of function to apply Haversine formula

θ is the central angle of the arc connecting two points, γ is the radius of the earth ($6371km$), d is the distance between the two points, φ is latitude (rad), and λ is longitude (rad). The code below implements the Haversine Formula.

2.5 Dijkstra Algorithm description



```
// Dijkstra pseudocode
function Dijkstra (Graph, start)

  priority_queue<int> p_que

  for( each vertex V):
    distance[v] = Infinity
    prevNode[v] = 0  //start point weight = 0
    visited[v] = false;
    push v in p_que

  distance[start] = 0

  while p_que is not empty:
    //visit check
    if visited[v] true, continue;
    visited[v] = true;
    // Updata weight
    minDistNode = index of node witch has smallest distance
    pop minDistNode from p_que
      for(each neighbor node u of minDistNode):
        calcDist = distance[minDistNode] + length(u,minDistNode)
        if distance[u] > calcDist:
          distance[u] = calcDist
          prevNode[u] = minDistNode
```

Fig 16. Pseudocode for Dijkstra algorithm

The initial distance of each vertex (airport) is initialized to 0 for the start airport(distance[0]) and others to an infinite value. The priority queue pop based on the cumulative minimum distance of each airport, and the distance update of the poped airport is given priority. Update the visit status of the poped vertex (airport) and avoid repeating the subsequent process for already visited airports. Updates the distance for the neighbor nodes of the start airport, where the distance[u] of each neighbor node is updated if it finds a smaller distance value and updates the id of the current node to find the shortest path.

The shortest path output is possible by referring to the prevNode array. The prevNode [v] stores the vertex before v in the shortest path from start vertex to v.

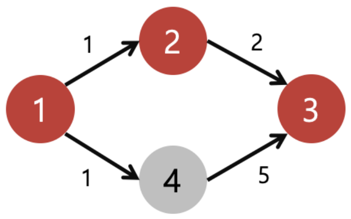
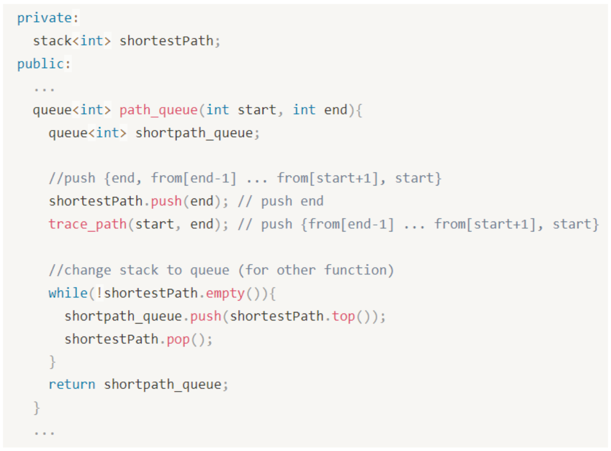


Fig 17. Example for explaining the process of Dijkstra algorithm

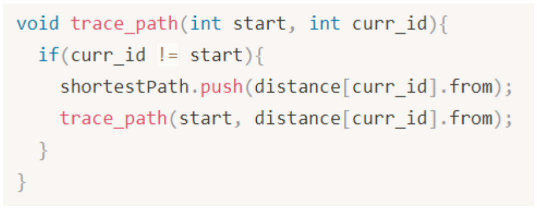
For example, when 1 is a start vertex, the shortest path from 1 to 3 is 1→2→3. At this point, prevNode[3] is 2. In shortest path array, start → prevNode[start] → ... → prevNode[prevNode[end]] → prevNode[end] → end vertex should be stored in the order. This was implemented using recursion.



```
private:
  stack<int> shortestPath;
public:
  ...
  queue<int> path_queue(int start, int end){
    queue<int> shortpath_queue;

    //push {end, from[end-1] ... from[start+1], start}
    shortestPath.push(end); // push end
    trace_path(start, end); // push {from[end-1] ... from[start+1], start}

    //change stack to queue (for other function)
    while(!shortestPath.empty()){
      shortpath_queue.push(shortestPath.top());
      shortestPath.pop();
    }
    return shortpath_queue;
  }
  ...
```

Fig 18. The process of converting a stack to a queue

First, put end vertex in the stack, and then push using the trace_path() recursive function in the order of prevNode[end] , prevNode[start], and start. Next, pop nodes one by one in the stack and store them in the queue. Through this, the order is rearranged from start to end.



```
void trace_path(int start, int curr_id){
  if(curr_id != start){
    shortestPath.push(distance[curr_id].from);
    trace_path(start, distance[curr_id].from);
  }
}
```

Fig 19. Function to trace the shortest path

The trace_path function uses recursion to search the prevNode until the node u to search in the prevNode is equal to the start node, and it pushes to the shortest path array. If prevNode[i] = start, the shortestPath is pushed from the end to the start node. Therefore, if u = start, the recursive stops.

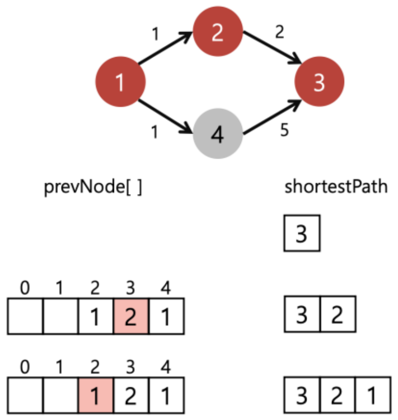


Fig 20. The process to find shortest path

After this process, the shortest path from the start node to the end node is stored in the queue. Return it to the Main.cpp function and print it.

2.6 Discrimination implementation for bad weather region

Bad weather region must be avoided unconditionally during flight. Therefore, we want to find the shortest path without using the edge if there is any overlap with the bad weather region.

The main idea is as follows. Determine whether the line segments that make up the bad weather region and the edge segments intersect. If there is any line segment crossing the edge, the edge is determined to pass the bad weather region and gives INT_MAX (a very large value) to the weight. This prevents the Dijkstra from putting the edge into the shortest path.

2.6.1 CCW algorithm

The CCW algorithm can be used as a mothod to determine whether the edges and segments that make up the severe weather zone intersect. Counterclockwise is an algorithm that determines the direction of three points through the external product of two Vector composed of three points.

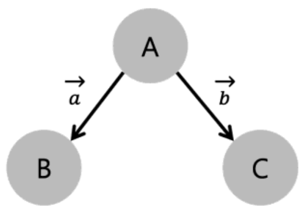


Fig 21. Three vector points in CCW

As shown in Fig 21., there are Vector *a* and *b* between the three points. Since the Vector product does not have the exchange law, the external product $a \times b$ and $b \times a$ of the two Vector have different values.

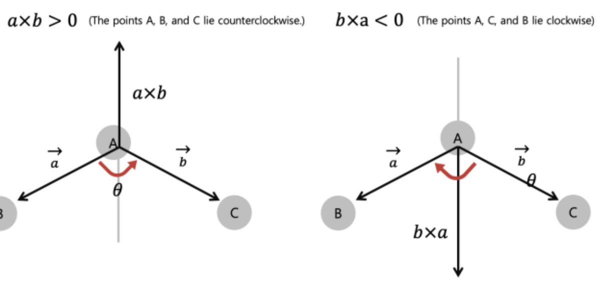

$a \times b > 0$ (The points A, B, and C lie counterclockwise.)
$b \times a < 0$ (The points A, C, and B lie clockwise)
$a \times b$
$b \times a$
θ
A
B
C

Fig 22. Cross product of two vectors

The direction of $a \times b$ is perpendicular to a and b and has a positive direction by the right-hand law. The direction of $b \times c$ is perpendicular to a and b, and similarly has a negative direction by the right-hand law. Looking at the directionality of the three points, it can be seen that $a \times b$ has a counterclockwise relationship (A→B→C) and $b \times a$ has a clockwise (A→C→B) relationship.

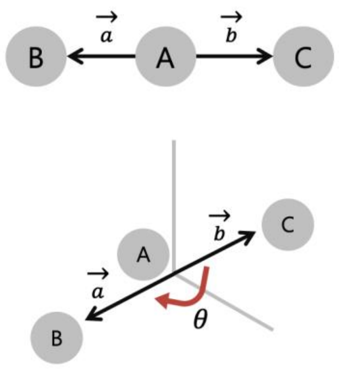


Fig 23. Cross product of two vectors of parallel case

When the three points are parallel, since theta value becomes 0 or 180, both $a \times b$ and $b \times a$ become 0.

$$A = (x_1, y_1, 0), \qquad B = (x_2, y_2, 0), \qquad C = (x_3, y_3, 0)$$

$$\underset{AB}{\rightarrow} = \underset{a}{\rightarrow} = (x_2 - x_1, y_2 - y_1, 0), \qquad \underset{AC}{\rightarrow} = \underset{b}{\rightarrow} = (x_3 - x_1, y_3 - y_1, 0)$$

$$\underset{a}{\rightarrow} \times \underset{b}{\rightarrow} = \begin{vmatrix} 1 & 1 & 1 \\ x_2 - x_1 & y_2 - y_1 & 0 \\ x_3 - x_1 & y_3 - y_1 & 0 \end{vmatrix}$$

$$= \begin{vmatrix} y_2 - y_1 & 0 \\ y_3 - y_1 & 0 \end{vmatrix} 1 - \begin{vmatrix} x_2 - x_1 & 0 \\ x_3 - x_1 & 0 \end{vmatrix} 1 + \begin{vmatrix} x_2 - x_1 & y_2 - y_1 \\ x_3 - x_1 & y_3 - y_1 \end{vmatrix} 1$$

$$= (0,0,(x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1))$$

Fig 24. The calculation for cross product of vectors

```
//Main.cpp
int ccw(pair<int, int> a, pair<int, int> b, pair<int, int> c) {
    double external = (b.first - a.first)*(c.second - a.second)
                    - (b.second - a.second)*(c.first - a.first);
    if (external > 0)return 1; // counter clockwise
    else if (external == 0)return 0; // pararall
    else return -1; // clockwise
}
```

Fig 25. The code for calculation for CCW

2.6.2 Apply CCW algorithm

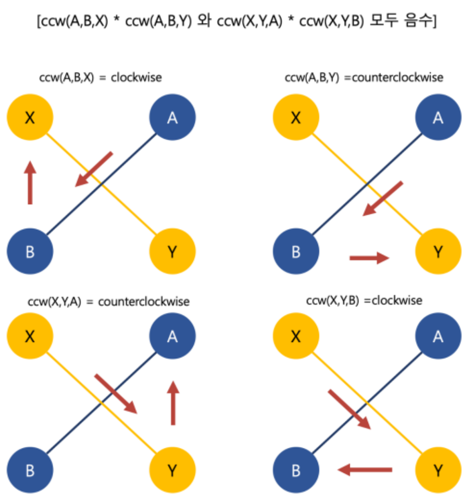

[ccw(A,B,X) * ccw(A,B,Y) 와 ccw(X,Y,A) * ccw(X,Y,B) 모두 음수]
ccw(A,B,X) = clockwise
ccw(A,B,Y) =counterclockwise
ccw(X,Y,A) = counterclockwise
ccw(X,Y,B) =clockwise
X
A
B
Y

Fig 26. The method for calculating the directionality of two intersecting line segments

When there are two line segments AB and *XY*, calculate the directions of points *A*, *B*, *X* and *A*, *B* and *Y* using CCW. The calculation results are negative in the clockwise direction, positive in the counterclockwise direction, and zero in parallel.

If *A*, *B*, *X* and *A*, *B*, *Y* are different directions, and *X*, *Y*, *A* and *X*, *Y*, *B* are different directions, the two line segments *AB* and *XY* intersect.

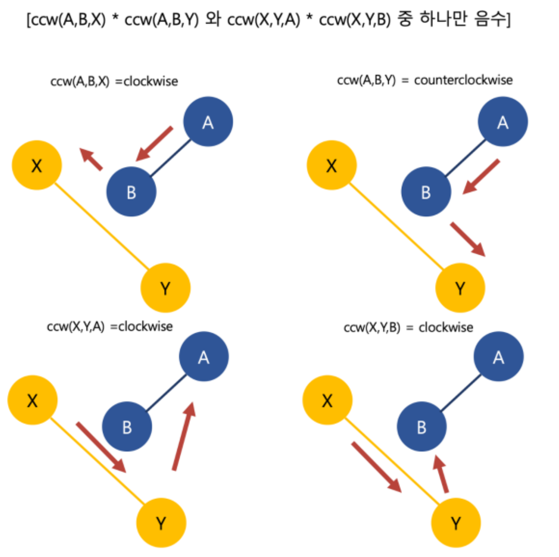

[ccw(A,B,X) * ccw(A,B,Y) 와 ccw(X,Y,A) * ccw(X,Y,B) 중 하나만 음수]
ccw(A,B,X) =clockwise
ccw(A,B,Y) = counterclockwise
ccw(X,Y,A) =clockwise
ccw(X,Y,B) = clockwise
A
B
X
Y

Fig 27. The method for calculating the directionality of two non-intersecting line segments

If *A*, *B*, *X* and *A*, *B*, *Y* have different directions, and *X*, *Y*, *A* and *X*, *Y*, *B* have the same directions, there may be cases where the two lines do not intersect.

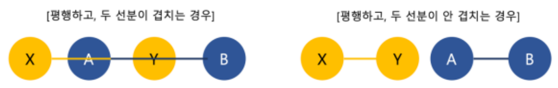
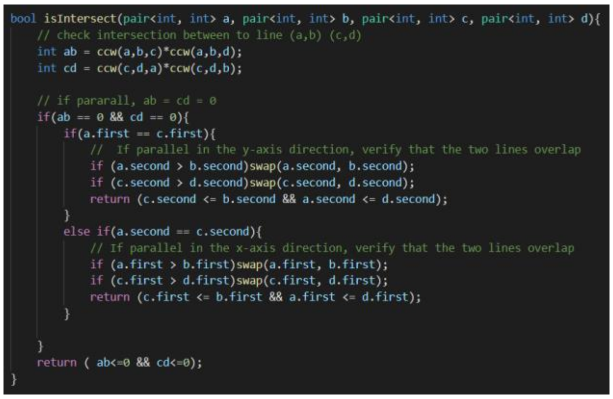

[평행하고, 두 선분이 겹치는 경우]
[평행하고, 두 선분이 안 겹치는 경우]
X
A
Y
B

Fig 28. The method for calculating the directionality of two parallel line segments.

If *A*, *B*, *X* and *A*, *B*, *Y* are parallel and at the same time *X*, *Y*, *A* and *X*, *Y*, *B* are parallel, the two lines may be parallel but not intersect, or parallel and intersect. Therefore, in this case, the position of the points that make up each line segment must be considered. This means that if *A* is less than *Y* and *X* is less than *B* at the same time, the two lines overlap, otherwise they do not overlap.

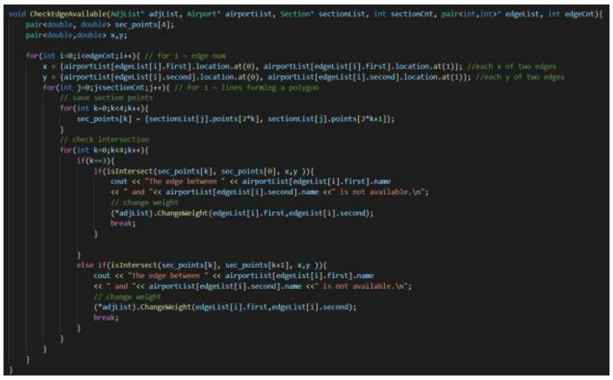


```
bool isIntersect(pair<int, int> a, pair<int, int> b, pair<int, int> c, pair<int, int> d){
    // check intersection between to line (a,b) (c,d)
    int ab = ccw(a,b,c)*ccw(a,b,d);
    int cd = ccw(c,d,a)*ccw(c,d,b);

    // if pararall, ab = cd = 0
    if(ab == 0 && cd == 0){
        if(a.first == c.first){
            //  If parallel in the y-axis direction, verify that the two lines overlap
            if (a.second > b.second)swap(a.second, b.second);
            if (c.second > d.second)swap(c.second, d.second);
            return (c.second <= b.second && a.second <= d.second);
        }
        else if(a.second == c.second){
            // If parallel in the x-axis direction, verify that the two lines overlap
            if (a.first > b.first)swap(a.first, b.first);
            if (c.first > d.first)swap(c.first, d.first);
            return (c.first <= b.first && a.first <= d.first);
        }

    }
    return ( ab<=0 && cd<=0);
}
```

Fig 29. The code for determining whether two line segments intersect.

To sum up, The **isintersect** function is based on CCW algorithm can be determined that the two lines AB and XY have intersections, and in other cases, there is no intersection. If the two lines intersect, return true, otherwise return false.

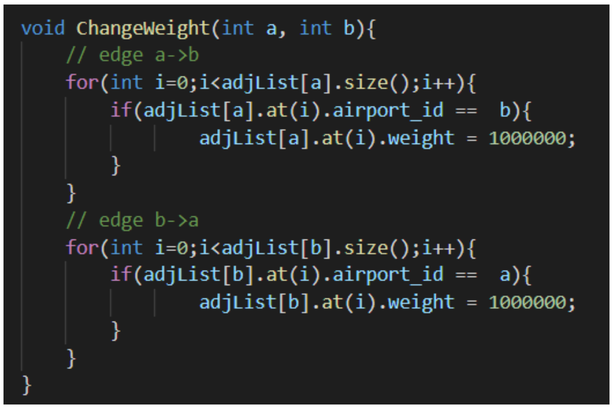
2.6.3 Implementation for solving problem



```
void CheckEdgeAvailable(AdjList* adjList, Airport* airportList, Section* sectionList, int sectionCnt, pair<int,int>* edgeList, int edgeCnt){
    pair<double, double> sec_points[4];
    pair<double,double> x,y;

    for(int i=0;i<edgeCnt;i++){ // for 1 ~ edge num
        x = {airportList[edgeList[i].first].location.at(0), airportList[edgeList[i].first].location.at(1)}; //each x of two edges
        y = {airportList[edgeList[i].second].location.at(0), airportList[edgeList[i].second].location.at(1)}; //each y of two edges
        for(int j=0;j<sectionCnt;j++){ // for 1 ~ lines forming a polygon
            // save section points
            for(int k=0;k<4;k++){
                sec_points[k] = {sectionList[j].points[2*k], sectionList[j].points[2*k+1]};
            }
            // check intersection
            for(int k=0;k<4;k++){
                if(k==3){
                    if(isIntersect(sec_points[k], sec_points[0], x,y )){
                        cout << "The edge between " << airportList[edgeList[i].first].name
                        << " and "<< airportList[edgeList[i].second].name <<" is not available.\n";
                        // change weight
                        (*adjList).ChangeWeight(edgeList[i].first,edgeList[i].second);
                        break;
                    }

                }
                else if(isIntersect(sec_points[k], sec_points[k+1], x,y )){
                    cout << "The edge between " << airportList[edgeList[i].first].name
                    << " and "<< airportList[edgeList[i].second].name <<" is not available.\n";
                    // change weight
                    (*adjList).ChangeWeight(edgeList[i].first,edgeList[i].second);
                    break;
                }
            }
        }
    }
}
```

Fig 30. The function for check if edges are available

First, The coordinates of the four points that make up the bad weather region are input in clockwise order. Find the four lines that make up the region through the input point. For all edges, determine whether the edge has intersections with the four lines forming regions by using CCW. If there is one intersection, the weight of the edge is given a very large value. In this method, when there are n edges and m sections, the time complexity is O(n×m).

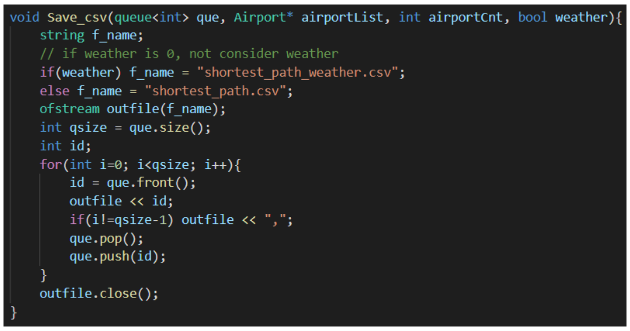


```
void ChangeWeight(int a, int b){
    // edge a->b
    for(int i=0;i<adjList[a].size();i++){
        if(adjList[a].at(i).airport_id ==  b){
            adjList[a].at(i).weight = 1000000;
        }
    }
    // edge b->a
    for(int i=0;i<adjList[b].size();i++){
        if(adjList[b].at(i).airport_id ==  a){
            adjList[b].at(i).weight = 1000000;
        }
    }
}
```

Fig 31. The function for updates for unavailable paths

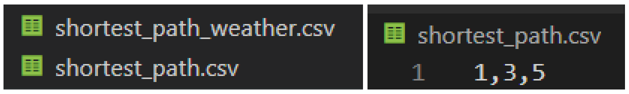
The function to change the weight is included in the AdjList class. Since it is an undirected graph, both edge(a,b) and edge(b,a) must be changed.

2.7 Saving the result as a csv file



```
void Save_csv(queue<int> que, Airport* airportList, int airportCnt, bool weather){
    string f_name;
    // if weather is 0, not consider weather
    if(weather) f_name = "shortest_path_weather.csv";
    else f_name = "shortest_path.csv";
    ofstream outfile(f_name);
    int qsize = que.size();
    int id;
    for(int i=0; i<qsize; i++){
        id = que.front();
        outfile << id;
        if(i!=qsize-1) outfile << ",";
        que.pop();
        que.push(id);
    }
    outfile.close();
}
```

Fig 32. The function for saving the results as a csv file



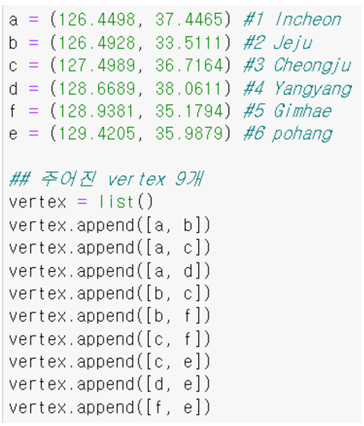
```
shortest_path_weather.csv
shortest_path.csv
```

```
shortest_path.csv
1    1,3,5
```

Fig 33. Saved csv files and components of file

The function **Save_csv** is a function that pops the values in the queue where the information on the shortest path is stored and saves it as a csv file. The file name is saved differently depending on whether or not the weather information is applied. The data for the shortest path is the id (int) value corresponding to each airport and is saved as a csv file according to the path. The saved csv file is visualized through Python.

2.8. Visualization of csv file (Python)



```
a = (126.4498, 37.4465) #1 Incheon
b = (126.4928, 33.5111) #2 Jeju
c = (127.4989, 36.7164) #3 Cheongju
d = (128.6689, 38.0611) #4 Yangyang
f = (128.9381, 35.1794) #5 Gimhae
e = (129.4205, 35.9879) #6 pohang

## 주어진 vertex 9개
vertex = list()
vertex.append([a, b])
vertex.append([a, c])
vertex.append([a, d])
vertex.append([b, c])
vertex.append([b, f])
vertex.append([c, f])
vertex.append([c, e])
vertex.append([d, e])
vertex.append([f, e])
```

Fig 34. The information about axis of airport and path

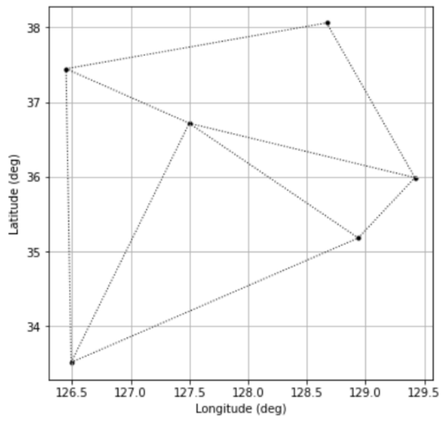


Fig 35. The graph visualized in Python code.

The python code reads the csv file and visualizes it as a graph. First, based on the given csv file as an example, the location of each airport and information of the given route were set as shown in Fig 35., and each vertex and edge were visualized as a graph.

Visualization of the shortest path first reads the csv file stored as the id of each airport for the shortest path. And the latitude and longitude information of the airport corresponding to the id is listed separately, and only the shortest path is displayed as a blue dotted line on the graph.

III. Program execution results

3.1 Visualization of Adjacency Lists

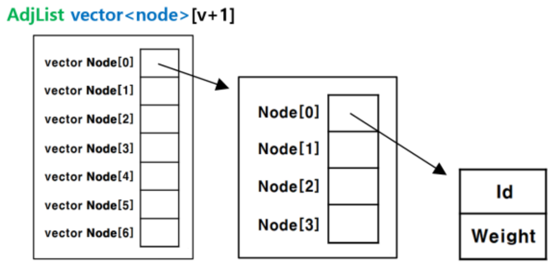
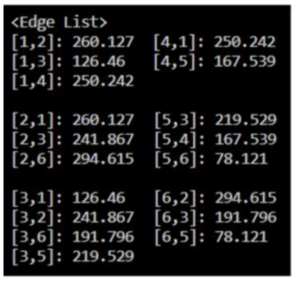
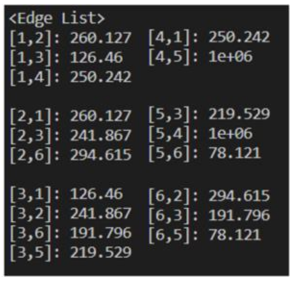


Fig 36. Visualization for data structure of AdjList

Fig 36. visualizes the adjacency list. Each adjacency list skips 0 and creates each vector node from 1 according to the id of the departure airport. In each vector node, Id and weight information of the destination airport is stored as node data.



```
<Edge List>
[1,2]: 260.127  [4,1]: 250.242
[1,3]: 126.46   [4,5]: 167.539
[1,4]: 250.242

[2,1]: 260.127  [5,3]: 219.529
[2,3]: 241.867  [5,4]: 167.539
[2,6]: 294.615  [5,6]: 78.121

[3,1]: 126.46   [6,2]: 294.615
[3,2]: 241.867  [6,3]: 191.796
[3,6]: 191.796  [6,5]: 78.121
[3,5]: 219.529
```

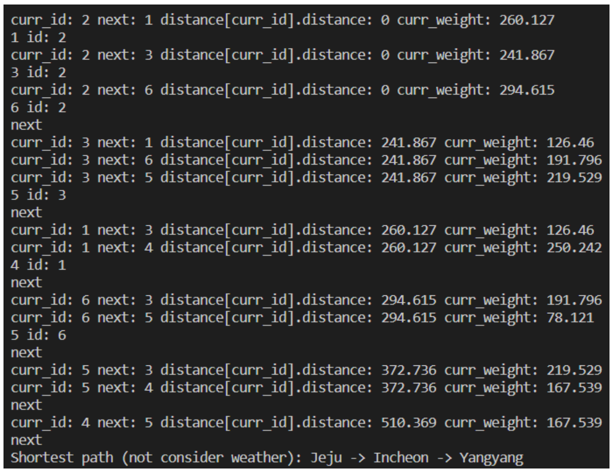


```
<Edge List>
[1,2]: 260.127  [4,1]: 250.242
[1,3]: 126.46   [4,5]: 1e+06
[1,4]: 250.242

[2,1]: 260.127  [5,3]: 219.529
[2,3]: 241.867  [5,4]: 1e+06
[2,6]: 294.615  [5,6]: 78.121

[3,1]: 126.46   [6,2]: 294.615
[3,2]: 241.867  [6,3]: 191.796
[3,6]: 191.796  [6,5]: 78.121
[3,5]: 219.529
```

Fig 37. Output of edge data

Fig 37. is the output of each adjacency list data. The left image shows the weight of each path when the convective area is not considered, and the right image shows the weight of each path when the convective area is not considered. Each adjacency list is output in the order of the stored airport id. And the id information of the destination airport and the weight of the route are output from each start airport.

3.2 The shortest path under clear weather condition



```
curr_id: 2 next: 1 distance[curr_id].distance: 0 curr_weight: 260.127
1 id: 2
curr_id: 2 next: 3 distance[curr_id].distance: 0 curr_weight: 241.867
3 id: 2
curr_id: 2 next: 6 distance[curr_id].distance: 0 curr_weight: 294.615
6 id: 2
next
curr_id: 3 next: 1 distance[curr_id].distance: 241.867 curr_weight: 126.46
curr_id: 3 next: 6 distance[curr_id].distance: 241.867 curr_weight: 191.796
curr_id: 3 next: 5 distance[curr_id].distance: 241.867 curr_weight: 219.529
5 id: 3
next
curr_id: 1 next: 3 distance[curr_id].distance: 260.127 curr_weight: 126.46
curr_id: 1 next: 4 distance[curr_id].distance: 260.127 curr_weight: 250.242
4 id: 1
next
curr_id: 6 next: 3 distance[curr_id].distance: 294.615 curr_weight: 191.796
curr_id: 6 next: 5 distance[curr_id].distance: 294.615 curr_weight: 78.121
5 id: 6
next
curr_id: 5 next: 3 distance[curr_id].distance: 372.736 curr_weight: 219.529
curr_id: 5 next: 4 distance[curr_id].distance: 372.736 curr_weight: 167.539
next
curr_id: 4 next: 5 distance[curr_id].distance: 510.369 curr_weight: 167.539
next
Shortest path (not consider weather): Jeju -> Incheon -> Yangyang
```

Fig 38. Output of process to find the shortest path under clear weather

Fig 38. is the operation process to find the shortest path using the Dijkstra algorithm. The id values of the departure airport (curr_id) and arrival airport (next) are displayed. Next, the sum of the weight of the current node and the weight of the next node is displayed. When the weight update condition is satisfied, the previous airport id value of the arrival airport is displayed so that the user can know how the route of each vertex (airport) is selected for each process. You can finally find the shortest path by tracing from the arrival airport to the previous airport.

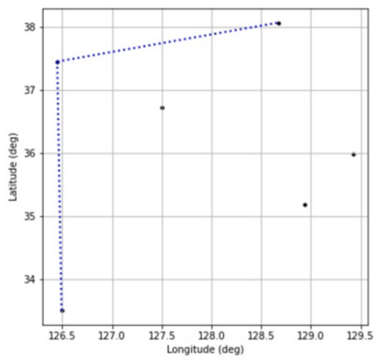
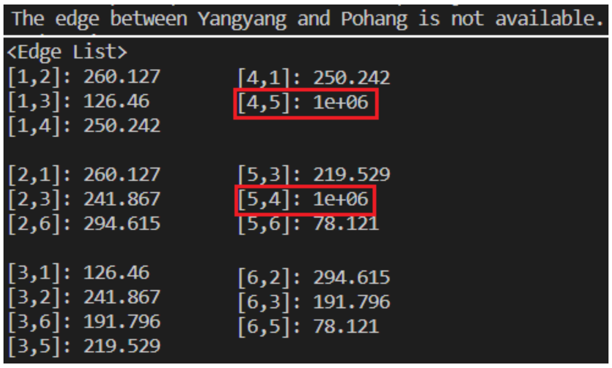


Fig 39. Visualization for Shortest path from Jeju to Yangyang airport under clear weather

As an example, for the given edge information, the shortest path of Yangyang Airport at Jeju Airport is Jeju → Incheon → Yangyang if the convective area is not considered.

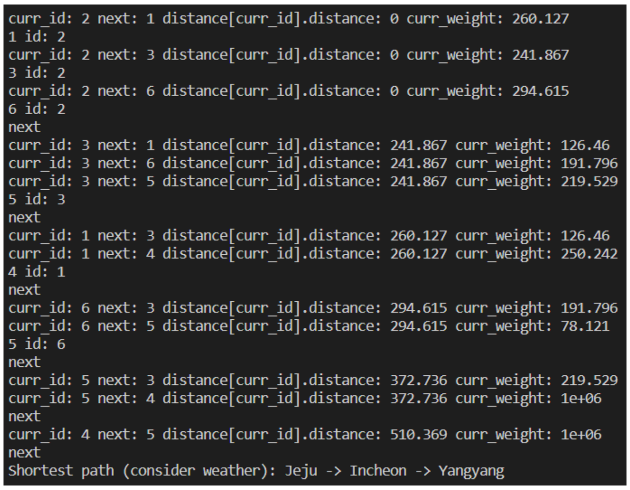
3.3 The shortest path under the severe weather condition



```
The edge between Yangyang and Pohang is not available.
<Edge List>
[1,2]: 260.127          [4,1]: 250.242
[1,3]: 126.46           [4,5]: 1e+06
[1,4]: 250.242

[2,1]: 260.127          [5,3]: 219.529
[2,3]: 241.867          [5,4]: 1e+06
[2,6]: 294.615          [5,6]: 78.121

[3,1]: 126.46           [6,2]: 294.615
[3,2]: 241.867          [6,3]: 191.796
[3,6]: 191.796          [6,5]: 78.121
[3,5]: 219.529
```

Fig 39. The weight information of each Edges

Fig 39. is the result of finding the edge that intersects the convective area using the **CheckEdgeAvailable** function. Fig 39. is the result of updating the weight to a very large value for the edge intersecting the convective area. In the Dijkstra algorithm, edges with high weights are not adopted as the shortest path, so updating these weights avoids the path passing through the convective area being adopted as the shortest path.



```
curr_id: 2 next: 1 distance[curr_id].distance: 0 curr_weight: 260.127
1 id: 2
curr_id: 2 next: 3 distance[curr_id].distance: 0 curr_weight: 241.867
3 id: 2
curr_id: 2 next: 6 distance[curr_id].distance: 0 curr_weight: 294.615
6 id: 2
next
curr_id: 3 next: 1 distance[curr_id].distance: 241.867 curr_weight: 126.46
curr_id: 3 next: 6 distance[curr_id].distance: 241.867 curr_weight: 191.796
curr_id: 3 next: 5 distance[curr_id].distance: 241.867 curr_weight: 219.529
5 id: 3
next
curr_id: 1 next: 3 distance[curr_id].distance: 260.127 curr_weight: 126.46
curr_id: 1 next: 4 distance[curr_id].distance: 260.127 curr_weight: 250.242
4 id: 1
next
curr_id: 6 next: 3 distance[curr_id].distance: 294.615 curr_weight: 191.796
curr_id: 6 next: 5 distance[curr_id].distance: 294.615 curr_weight: 78.121
5 id: 6
next
curr_id: 5 next: 3 distance[curr_id].distance: 372.736 curr_weight: 219.529
curr_id: 5 next: 4 distance[curr_id].distance: 372.736 curr_weight: 1e+06
next
curr_id: 4 next: 5 distance[curr_id].distance: 510.369 curr_weight: 1e+06
next
Shortest path (consider weather): Jeju -> Incheon -> Yangyang
```

Fig 40. Output of process to find the shortest path considering convective area

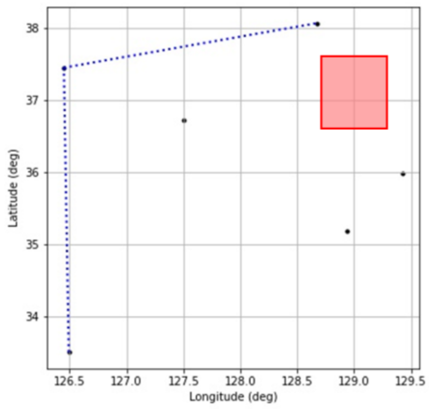


Fig 41. Visualization for Shortest path from Jeju to Yangyang airport considering convective area

As an example, if the shortest path from Jeju Airport to Yangyang Airport is obtained for the given edge information, it does not significantly affect the shortest path from Jeju to Yangyang despite the existence of a convective area. Therefore, the shortest path is Jeju → Incheon → Yangyang, the same as when the convective area is not considered.

IV. Discussion

4.1 Comparison of results for different conditions

In discussion, the results of the shortest path from Jeju Airport to Yangyang Airport are compared depending on whether the convective area is considered. As a result, the shortest path in both cases is Jeju → Incheon → Yangyang.

A given convective area exists between Pohang and Yangyang in terms of coordinates. It was confirmed through the CheckEdgeAvailable function that the edge between Pohang and Yangyang passed the convective area, and the edge weight was updated to a very high value.

In the process of finding the shortest path using the Dijkstra algorithm, each vertex is updated with the cumulative weight of the smallest value, and the edge at that time is adopted as the path. Since the weight between Pohang and Yangyang is very large, it can be seen that the edge from Yangyang to Pohang is not selected as a path.

4.2 Methodology for safe edge identification

Three methods were considered for discriminating an edge passing through a bad weather region, and finally, a discriminating method using the CCW algorithm was selected.

4.2.1 Square range method

Polygon information = ([(128.7, 36.6), (128.7, 37.6), (129.3, 37.6), (129.3, 36.6)])

Flight path area = ([(128.6, 36.0), (128.6, 38.1), (129.5, 38.1), (129.5, 36.0)])

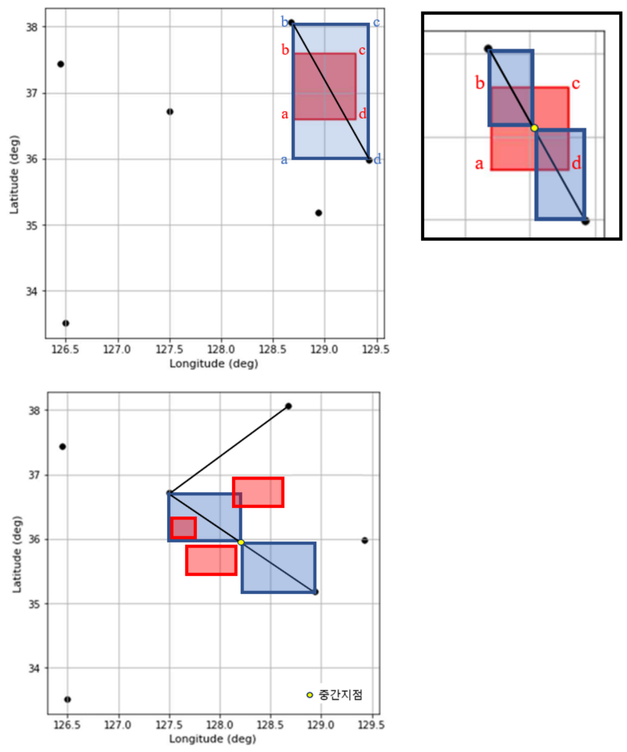


Fig 42. Visualization of Square range method

The longitude and latitude ranges of each edge are set in the range of the blue rectangle, and when the latitude and longitude of the bad weather region satisfies the range of the blue rectangle, we thought of a method to determine that the edge passes through the bad weather region. However, this method is not a good method because it incorrectly judges the case where the bad weather region does not cross the edge and exists within the blue rectangle.

In the first case, I thought that the possibility of making a mistake was due to the large range of the rectangle through the start and end points of the edge, so I further divided the range of the blue rectangle through the midpoint of the edge. However, in this case as well, although the frequency of misjudgment could be reduced compared to the first method, it was not a perfect method. Also, if the rectangle is further subdivided, the accuracy will increase, but the amount of computation will increase significantly, and this method was not chosen because it is not a perfect method.

4.2.2 Method using linear equations and simultaneous equations

First, find the linear equation of the line segments and edge segments constituting the bad weather region, and check whether a solution exists through the system. If a solution exists, check whether the solution is a point in the bad weather region. If included in the region, the edge belongs to the bad weather region. Otherwise, the corresponding edge can be used.

This method is the easiest to think of and can detect all cases without exception. However, implementing this in code requires a lot of computation. Therefore, it was not chosen as a method.

4.2.3 CCW (Counterclockwise) Algorithm

The CCW algorithm is an algorithm that can determine whether two line segments intersect by using the cross product to determine the direction of each of the two end points of the other line segment based on the two end points.

Unlike the above methods, this method has the advantage that the amount of computation is small by determining the intersection using only the endpoints of the edge to be compared. Also, when a given bad weather region is a polygon, this method was finally selected because it is an algorithm that can perfectly determine whether two line segments intersect.

V. Conclusion

This project proposes efficient route optimization by finding the shortest and safest air routes through the Dijkstra algorithm. The distance of each path was calculated through the Haversine formula in consideration of the curvature of the earth. A graph was constructed with each airport as vertex, the route between airports as edge, and the distance between airports as weight. Several methodologies have been proposed as a way to find a path avoiding the existing coherent area, but the CCW(counter clockwise) algorithm was finally applied through discussion. Through the implemented program, the shortest path considering the presence or absence of a coherent area was found, visualized with Python, and the shortest path was saved as an image file. As a result, the planned program was successfully implemented.

VI. Reference

[1] "C++ 에서 전화번호부 만들기(txt 읽어 csv 만들기)", Naver blod, Last modified May 29,2019, accessed Aprill 2, 2022, https://blog.naver.com/PostView.naver?blogId=tgsbj9807&logNo=221549399956

[2] "[알고리즘] Dijkstra(다익스트라)", Naver blog, last modified July 28, 2021, accessed Aprill 2, 2022, https://m.blog.naver.com/hjs8482/222448060931

[3] "최단거리 구하기, 하버사인 공식(Haversine Formula)", github. Io, last modified Sep 19, 2019, accessed Aprill 2, 2022, https://kayuse88.github.io/haversine/

[4] "Matpoltlib 에서 임의의 선 그리기", DelftStack, last modified May 12, 2021, accessed Aprill 3, 2022, https://www.delftstack.com/ko/howto/matplotlib/matplotlib-draw-an-arbitrary-line/#matplotlib matplotlib.collections.linecollection%25EC%259D%2584%25EC%2582%25AC%25EC%259A%25A9%25ED%2595%2598%25EC%2597%25AC-%25EC%259E%2584%25EC%259D%2598%25EC%259D%2598-%25EC%2584%25A0-%25EA%25B7%25B8%25EB%25A6%25AC%25EA%25B8%25B0

[5] "CCW 와 CCW 를 이용한 선분 교차 판별", tistory, last modified Oct 9, 2019, accessed Aprill 4, 2022, https://jason9319.tistory.com/358

[6] "c++ 두 선분의 교차 여부", tistory, last modified Mar 24, 2020, accessed Aprill 4, 2022, https://newdeal123.tistory.com/45

[7] " c++ 다익스트라 알고리즘 구현과 다익스트라 경로 추적", github, last modified Nov 5, 2020, accessed Aprill 1, 2022, https://techbless.github.io/2020/11/05/C-%EB%8B%A4%EC%9D%B5%EC%8A%A4%ED%8A%B8%EB%9D%BC-%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98-%EA%B5%AC%ED%98%84%EA%B3%BC-%EB%8B%A4%EC%9D%B5%EC%8A%A4%ED%8A%B8%EB%9D%BC-%EA%B2%BD%EB%A1%9C-%EC%B6%94%EC%A0%81-%EB%B0%A9%EB%B2%95/