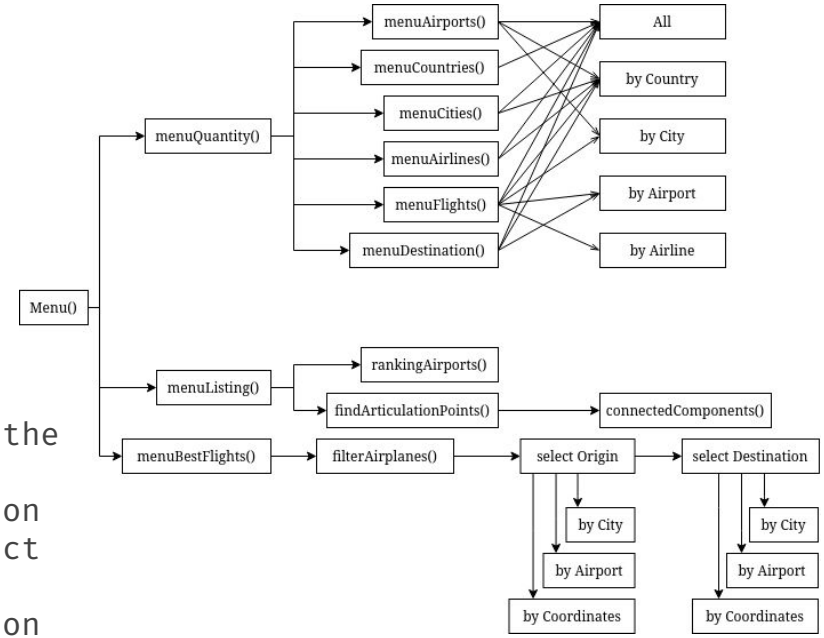
Air Travel Flight Management System

Made by: Clarisse Carvalho e Eduardo Oliveira.

Class Diagram/Sections

> Main Class: Represents the entry point of the application.
File included: ‘Menu.h’
> Menu Class:
Manages the menu and user interaction.
Files included: ‘dbairport.h’, ‘read.h’
> Database Airport Classe
Handle input processing, data management, statistics and output operations.
Files included: ‘Airline.h’, ‘Airport.h’, ‘Graph.h’
> Graph Class: Manages the structure and operations related to the graph, and includes the Edge Class and Vertex Class.
> Airport Class: Represents detailed information about an airport, and includes the Flight Struct and Ranking Struct
> Airline Class: Represents detailed information about an airline.



Read Database (1)

```cpp
Graph<Airport> readAirports(std::string folder)
{
  // Inicialiate Graph
  Graph<Airport> airports = Graph<Airport>();

  // Open csv file
  std::ifstream file("../dataset/" + folder + "/airports.csv", ios::in);
  if (!file.is_open())
  {
    std::cout << "Error opening file" << std::endl;
    return airports;
  }

  // Read and ignore header line
  std::string line;
  if (!std::getline(file, line))
  {
    std::cout << "Error reading header line" << std::endl;
    file.close();
    return airports;
  }
```

```cpp
  // Read each line and create an Airport object
  while (std::getline(file, line))
  {
    std::istringstream iss(line);
    std::string code, name, country, city;
    double latitude, longitude;

    std::getline(iss, code, ',');
    std::getline(iss, name, ',');
    std::getline(iss, city, ',');
    std::getline(iss, country, ',');

    iss >> latitude;
    iss.ignore();
    iss >> longitude;

    Airport airport(code, name, country, city, latitude, longitude);

    airportsHash.insert({code, airport});
    citiesHash.insert(city);
    countriesHash.insert(country);

    // airport.display();
    airports.addVertex(airport);
  }

  // Close file and return graph of airports
  file.close();
  return airports;
}
```

The *readAirports* function is designed to read airport information from a specified CSV file and construct a graph of airports. The function initializes an empty graph, opens the CSV file, and processes each line to create individual Airport objects. These airport objects are then added to the graph, resulting in a comprehensive representation of airports. in addition, 3 public hashTables are created to be used in other files. The airportsHash, citiesHash and countriesHash. We did this to achieve constant consultation

Read Database (2)

The ***readFlights*** function enhances the airport graph with flight details from a designated CSV file. It starts by utilizing *readAirports* to fetch the existing airport graph. After opening the flights CSV, it checks for errors, skips the header, and processes each line. Extracting source, target airports, and airline info, it adds edges to represent flight routes. The function closes the file, delivering the graph enriched with flight connections.

```
extern unordered_map<string, Airport> airportsHash;
extern unordered_set<string> citiesHash;
extern unordered_set<string> countriesHash;

unordered_map<string, Airline> readAirlines(std::string folder);
Graph<Airport> readAirports(std::string folder);
Graph<Airport> readFlights(std::string folder);
```

The ***readAirlines*** we use a hash table *(unordered_map<string, Airline>)* to read the *airlines.csv* file. With this, we save the airline code associated with the Airline class. Resulting in constant consultation.

Graphs

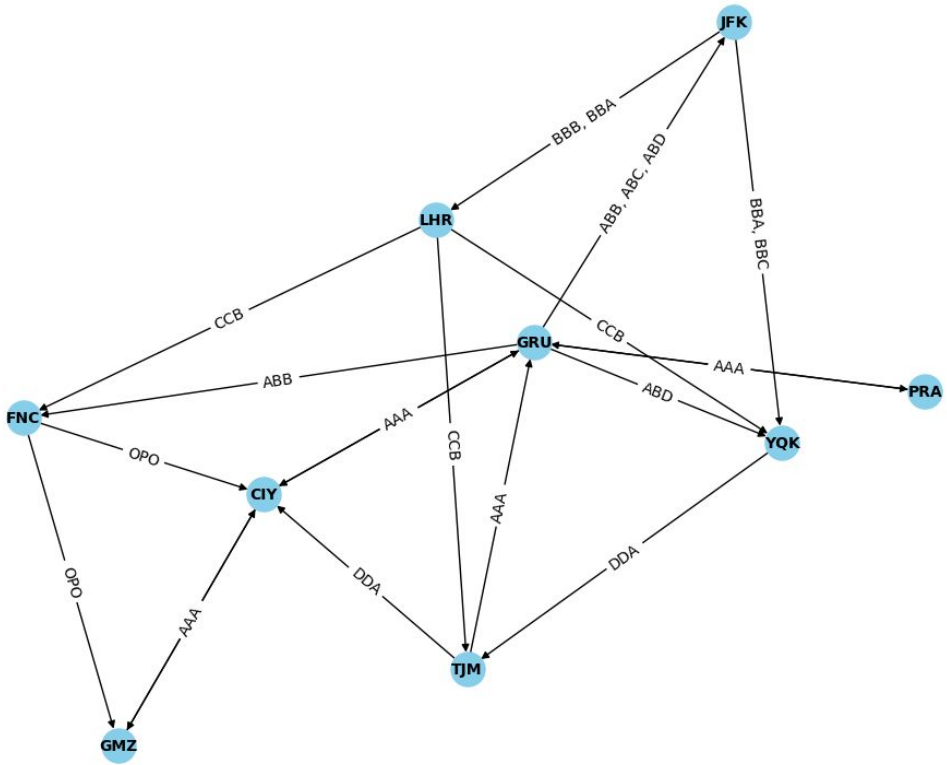
Data Visualization
Our dataset is represented as a graph, employing the Graph class template, with vertices of type Airport. This visualization offers insights into the structure and connectivity of the airport network.

Graph Representation:
Type of Graph: *Graph<Airport>*

Vertex Information:
Each airport node encapsulates crucial details, including airport code, name, country, and location. This comprehensive representation allows for a detailed analysis of the airport network.

Graph Properties:
Exploring graph properties such as *indegree* and *outdegree* enhances our understanding of the network's connectivity.

Graphs Airports



We created a fake database inspired by the original database provided, but with a reduced size, making it easier to read and test. Much of the algorithm code we created was based on the algorithm made on pencil and paper.

Airport
string code string name string country string city double latitude double longitude
display() void

User Interface (1)

```
// Function to display menu
void Menu(std::string folder);
void MenuQuantity();
void menuListing();
void menuAirports();
void menuCountries();
void menuCities();
void menuAirlines();
void menuFlights();
void menuDestinations();
void menuBestFlights();

// Function to handle the process of finding the best flight
vector<string> filterAirplanes();
int selectType(std::string arg);
std::string typeAirport(std::string type, int flag);
pair<std::string, std::string> typeCity(std::string type, int flag);
pair<double, double> typeCoordinates(std::string type, int flag);
```

Functions:

Menu Functionality:
> Functions to display menus;
Flight Operations:
> Functions to handle the process of finding the best flight;
Airplane Filtering:
> Function to filter airplanes;
Auxiliary Database Functions
(from dbairport.h)
Auxiliary Graph Operations
(from Graph.h)
Auxiliary Airport Operations
(from Airport.h)

User Interface (2)

```cpp
// Function Menu Main
void Menu(std::string folder) {
  int flag;

  if (!folder.empty())
    airports = readFlights(folder);

  system(command: "clear");
  std::cout << "Welcome to Travel Management:" << std::endl;
  std::cout << "-----------------------------------------------" << std::endl;
  std::cout << "1. Quantity calculation" << std::endl;
  std::cout << "2. Listing" << std::endl;
  std::cout << "3. Bests flights" << std::endl;
  std::cout << "0. Exit" << std::endl;
  std::cout << "-----------------------------------------------" << std::endl;

  std::cin >> flag;
  while (std::cin.fail()) {
    std::cin.clear();
    std::cin.ignore(n: std::numeric_limits<std::streamsize>::max(), delim: '\n');
    std::cout << "Invalid input. Please enter a valid number: " << std::endl;
    Menu(folder: "");
  }

  switch (flag) {
  case 1:
    quantity();
    break;
  case 2:
    listing();
    break;
  case 3:
    bestFlights();
    break;
  case 0:
    exit(status: 0);
    break;
  default:
    std::cout << "Invalid input. Please enter a valid number: " << std::endl;
    Menu(folder: "");
    break;
  }
}
```

Function “Template”:

- [Occasionally] Input data;
- [Occasionally] Read relevant information;
- **Display** menu options;
- **Read** user's choice from the console;
- If the input is not a valid number, prompt the user to re-enter their choice;
- Utilize a **switch statement** based on the user's choice:
  - Case n: Execute the corresponding function or action;
  - Case 0: Navigate back or exit the current context;
  - Default: Display an error message and return menu.

Functionalities & Algorithms (1)

Statistics

1. Number of Airports	
All airports	O(1)
Airports by Country	O(n)
Airports by City	O(n)

2. Number of Countries	
All countries	O(1)

3. Number of Cities	
All cities	O(1)
Cities by Country	O(n)

4. Number of Airlines	
All airlines	O(1)
Airlines by Country	O(n)

5. Number of Flights	
All flights	O(1)
Flights by Airport	O(n)
Flights by Country	O(n)
Flights by City	O(n)
Flights by Airline	O(n+e)

6. Number of Destinations	
Unlimited Stops	O(n+e)
Unlimited Stops	O(n+e)
Limited Stops	O(n+e)
Max Destinations	O(n*(n+e))

```
int quantityAirportsCountry(Graph<Airport> airports, std::string country)
{
    int count = 0;
    for (auto v : airports.getVertexSet())
    {
        Airport a = v->getInfo();
        if (a.getCountry() == country)
        {
            count++;
        }
    }
    return count;
}
```

```
int quantityAirlinesCountry(unordered_map<string, Airline> airlines, std::string country)
{
    int count = 0;
    for (auto v : airlines)
    {
        if (v.second.getCountry() == country)
        {
            count++;
        }
    }
    return count;
}
```

Functionalities & Algorithms (2)

Statistics

```
int quantityDestinationMax(Graph<Airport> airports)
{
    vector<string> path, maxPath;

    for (auto v : airports.getVertexSet())
    {
        // std::cout << "Consultando... " << v->getInfo().getCode() << std::endl;
        resetVisited(airports);
        path.clear();
        dfsMax(v, path, maxPath);
    }

    std::cout << "\nStarting in: " << maxPath[0] << std::endl;
    for (int i = 0; i < maxPath.size() - 1; i++)
    {
        std::cout << maxPath[i] << " -> ";
    }
    std::cout << "Ending in: " << maxPath[maxPath.size() - 1] << std::endl;
    return maxPath.size() - 1;
}
```

```
void dfsMax(Vertex<Airport> *v, std::vector<std::string> &path, std::vector<std::string> &maxPath)
{
    v->setVisited(true);
    path.push_back(v->getInfo().getCode());
    auto adjs = v->getAdj();

    if (path.size() > maxPath.size())
    {
        maxPath = path;
    }

    for (auto &e : adjs)
    {
        auto w = e.getDest();

        if (!w->isVisited())
        {
            dfsMax(w, path, maxPath);
        }
    }
    path.pop_back();
}
```

This function uses a modified DFS, where we have a path (Current) and maxPath (Largest). The `visited` attribute of each vertex is reset and then applied to DFS. Each time the current path is greater than the largest, this is saved. Therefore, at the end of the stack, we have the longest possible path.

Functionalities & Algorithms (3)

Listing

Ranking Airports Functions:

rankingAirports()
calculateIndegree()

Connecting Airports Functions:

findArticulationPoints()
resetVisited()
dfsConnectedComponents()
dfsArtc()
connectedComponents()

```
// Complexity: O(n * log(n) + n * e), where n is the number of airports and e
// is the number of flights in the graph (assuming sorting has a time
void rankingAirports(Graph<Airport> airports, int arg) {
  std::vector<Ranking> vec;

  calculateIndegree(&: airports);

  for (auto v: Vertex<Airport> * : airports.getVertexSet()) {
    int total = v->getIndegree() + v->getAdj().size();
    Ranking rank = {.code=v->getInfo().getCode(), .count=total};
    vec.push_back(x: rank);
  }

  std::sort(first: vec.begin(), last: vec.end(), comp: comparator);

  int i = 0;

  for (auto v: Ranking : vec) {

    if (i < arg) {
      std::cout << "Code: " << v.code << " / ";
      std::cout << "Total: " << v.count << std::endl;
      i++;
    }
  }
}
```

```
// Complexity: O(n^2), where n is the number of airports in the graph
void findArticulationPoints(Graph<Airport> &airports) {
  int connected = connectedComponents(&: airports);
  for (auto v: Vertex<Airport> * : airports.getVertexSet()) {
    // std::cout << " Analisando o vertice: " << v->getInfo().getCode() <<
    // std::endl;
    resetVisited(&: airports);
    int component = 0;
    for (auto w: Vertex<Airport> * : airports.getVertexSet()) {
      // std::cout << "\tAnalisando o vertice: " << w->getInfo().getCode()<<
      // std::endl;
      if (!w->isVisited() && w->getInfo().getCode() != v->getInfo().getCode()) {
        // std::cout << "\t\tChamando o DFS component++" << std::endl;
        dfsArtc(v, w);
        component++;
      }
    }
    if (component > connected) {
      std::cout << v->getInfo().getCode() << std::endl;
    }
  }
}
```

```
// Complexity: O(n + e), where n is the number of airports and e is the number
// of flights in the graph
void calculateIndegree(Graph<Airport> &airports) {
  for (auto v: Vertex<Airport> * : airports.getVertexSet()) {
    v->setIndegree(0);
  }
  for (auto v: Vertex<Airport> * : airports.getVertexSet()) {
    for (auto e: Edge<Airport> : v->getAdj()) {
      e.getDest()->setIndegree(e.getDest()->getIndegree() + 1);
    }
  }
}
```

```
// Complexity: O(n + e), where n is the number of airports and e is the
// number of flights in the graph
void dfsConnectedComponents(Graph<Airport> &airports, Vertex<Airport> *v) {
  v->setVisited(v: true);
  auto adjs: vector<Edge<Airport>> = v->getAdj();

  for (auto &e: Edge<Airport> & : adjs) {
    auto w: Vertex<Airport> * = e.getDest();
    if (!w->isVisited()) {
      dfsConnectedComponents(&: airports, v: w);
    }
  }
}
```

Functionalities & Algorithms (4)

Best Flights

Functions in Menu Class:
filterAirplanes()
SelectType()
typeAirport()
typeCity()
typeCoordinates()
Functions in Database Airport Class:
findBestFlights()
 > Airport to (Airport,City,Coordinates)
 > City to (Airport,City,Coordinates)
 > Coordinates to (Airport,City,Coordinates)
resetVisited()
showPath()
ComparatorPath()
findAirports()
bfsPath()
getPath()

```
pair<double, double> typeCoordinates(std::string type, int flag) {
  double arg1;
  double arg2;
  switch (flag) {
  case (3):
    std::cout << "Type Latitude of " << type << ": " << std::endl;
    std::cin >> arg1;
    std::cout << "Type Longitude of " << type << ": " << std::endl;
    std::cin >> arg2;
    break;
  }
  return make_pair(&x: arg1, &y: arg2);
}
```

```
// Complexity: O(n + e), where n is the number of airports and e is the
// number of flights in the graph
void findBestFlights(Graph<Airport> &airports, string countrySrc,
                     string citySrc, string countryDest, string cityDest,
                     vector<string> &airplanes) {
  vector<Vertex<Airport> *> src;
  vector<Vertex<Airport> *> dest;
  vector<vector<Flight>> paths;

  src = findAirports(&: airports, country: countrySrc, city: citySrc);
  dest = findAirports(&: airports, country: countryDest, city: cityDest);

  showPath(&: airports, source: src, dest, paths, &: airplanes);
}
```

Functionalities & Algorithms (5)

Best Flights

```
// Complexity: O(n), where n is the number of airports in the graph
vector<Vertex<Airport> *> findAirports(Graph<Airport> &airports, string country,
                                       string city) {
  vector<Vertex<Airport> *> vec;
  for (auto v: Vertex<Airport> * : airports.getVertexSet()) {
    if (v->getInfo().getCountry() == country &&
        v->getInfo().getCity() == city) {
      vec.push_back(x: v);
    }
  }
  return vec;
}
```

```
// Complexity: O(n + e), where n is the number of airports and e is the
// number of flights in the graph
void showPath(Graph<Airport> &airports, vector<Vertex<Airport> *> source,
              vector<Vertex<Airport> *> dest, vector<vector<Flight>> paths,
              vector<string> &airplanes) {
  vector<vector<Flight>> flights;
  for (auto s: Vertex<Airport> * : source) {
    // std::cout << "Source: " << s->getInfo().getCode() << std::endl;
    for (auto d: Vertex<Airport> * : dest) {
      // std::cout << "Destination: " << d->getInfo().getCode() <<
      // std::endl;
      std::string tgt = d->getInfo().getCode();
      resetVisited(&: airports);
      flights = bfsPath(v: s, &: tgt, &: airplanes);
      paths.insert(position: paths.end(), first: flights.begin(), last: flights.end());
    }
  }
```

```
if (paths.empty()) {
  std::cout << "-----------------------------------------------"
            << std::endl;
  std::cout << "Sorry, but there is no result with those inputs."
            << std::endl;
  return;
} else {
  std::sort(first: paths.begin(), last: paths.end(), comp: comparatorPath);
  const int min = paths[0].size();
  for (auto &p: vector<Flight> & : paths) {
    if (p.size() > min) {
      continue;
    }
    std::cout << "-----------------------------------------------"
              << std::endl;
    for (auto &f: Flight & : p) {
      if (!f.airline.empty()) {
        std::cout << f.code << " -(" << f.airline << ")"
                  << "-> ";
      } else {
        std::cout << f.code;
      }
    }
    std::cout << std::endl;
  }
}
```

Best Features (1)

Finding Best Flights Between Two Points on Earth:

Function:

findBestFlights(Graph<Airport> &airports, double latOrigin, double longOrigin, double latDest, double longDest, int distMax, vector<string> &airplanes)

Why: This function involves complex geographical calculations, converting degrees to radians and calculating distances between points on the Earth. It showcases a practical application of the project by enabling users to find the best flights between two locations based on geographical proximity. Demonstrates your ability to integrate domain-specific knowledge (geography) into your technical solution.

```
// Complexity: O(n + e), where n is the number of airports and e is the
// number of flights in the graph
void findBestFlights(Graph<Airport> &airports, double latOrigin,
                     double longOrigin, double latDest, double longDest,
                     int distMax, vector<string> &airplanes) {
  vector<Vertex<Airport> *> src =
      findAirports(&: airports, lat: latOrigin, lon: longOrigin, distMax);
  vector<Vertex<Airport> *> dest =
      findAirports(&: airports, lat: latDest, lon: longDest, distMax);
  vector<vector<Flight>> paths;

  showPath(&: airports, source: src, dest, paths, &: airplanes);
}
```

The findAirports function stores all the airports close to the given point in a vector and then the showPath function is called, which calculates all the smallest equivalent paths between each of the source - destinations possibilities. Displaying to the user only the best equivalents with the least number of stops

Main Difficulties

Time Management: Balancing project work during the holidays, including Christmas and New Year's, while preparing for upcoming exams.

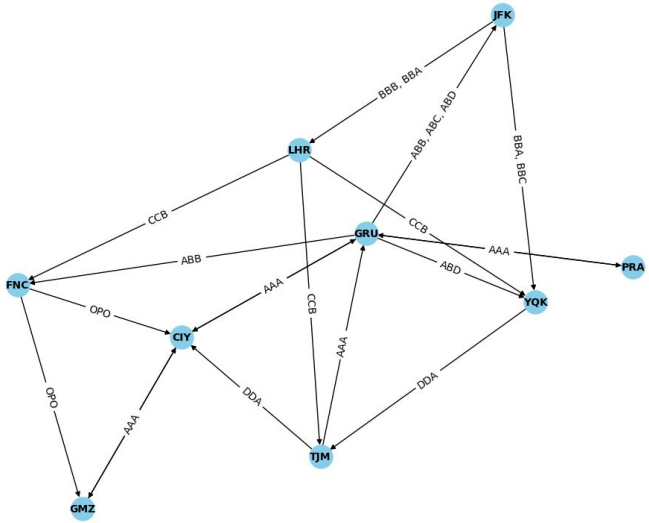
Find the longest path: in the graph by analyzing all points. This is an NP-Hard problem, and we managed to create a functional algorithm, but not fast.

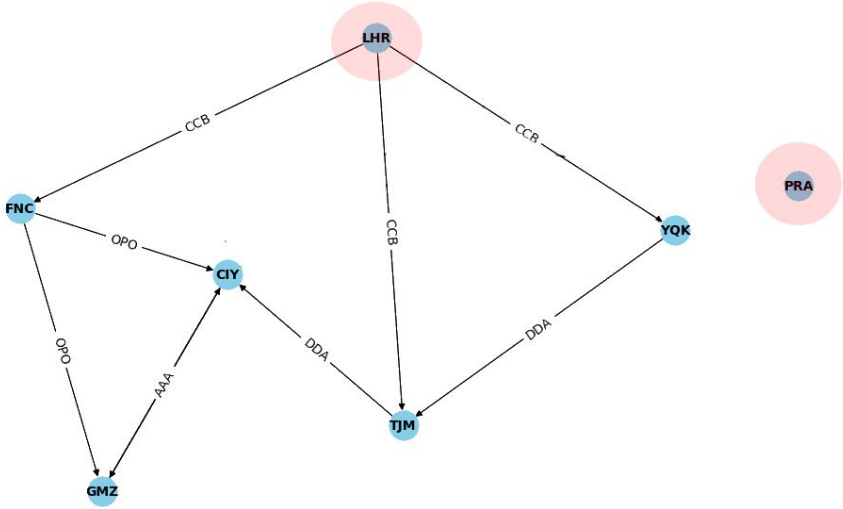
Find the essential points: those that, if removed, leave certain areas of the map inaccessible. We managed to create a solution, but not so fast O(n^2).

Graph Diameter: Dealing with high time complexity in the function calculating the diameter of the graph, and the challenge of optimizing its efficiency.

Extra (1)

Essential Points



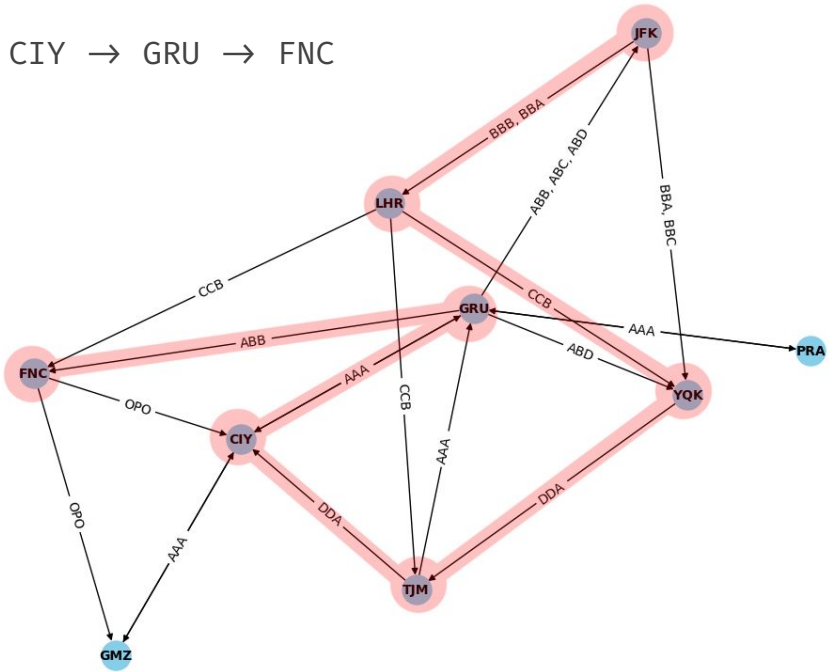


Our algorithm for analyzing the essential points will return JFK and GRU. This is because, when removing JFK, it becomes impossible to access the LHR. And when removing the GRU, the PRA is completely isolated, making it impossible to leave or enter.

Extra (2)

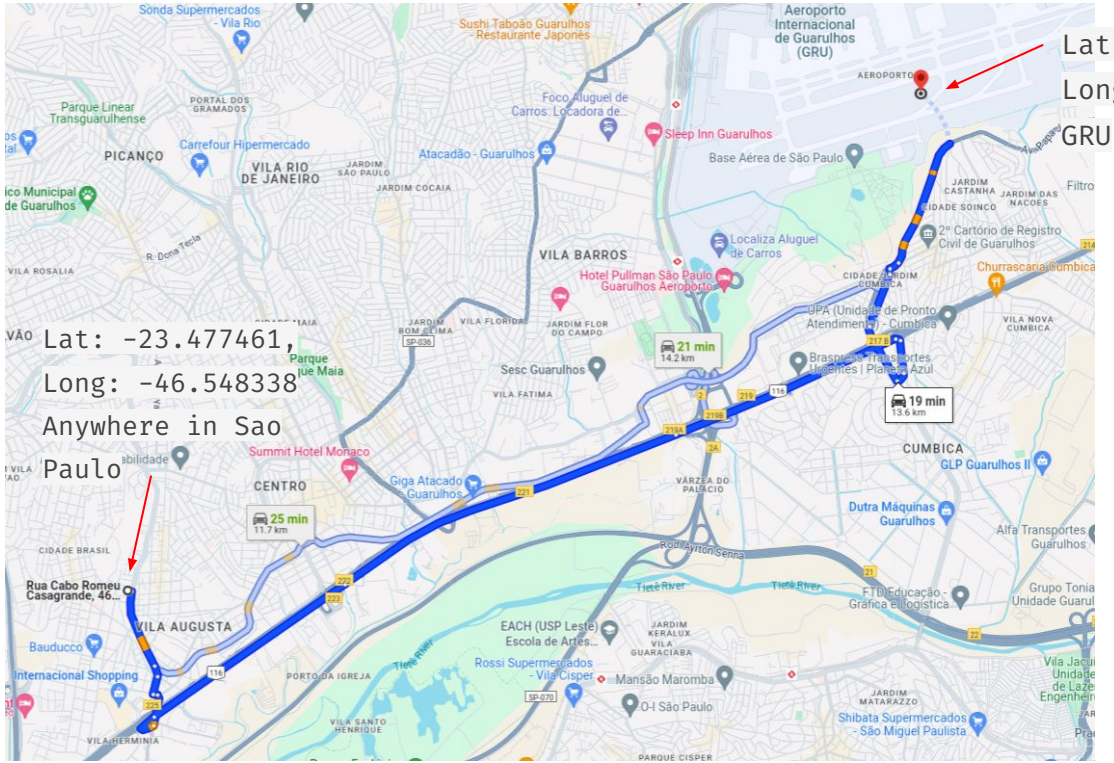
Max Path

Starting in: JFK
LHR → YQK → TJM → CIY → GRU → FNC
Ending in: GMZ



Extra (3)

Find best flights with different origins (Point to Airport)



Lat: -23.432075,
Long: -46.469511
GRU

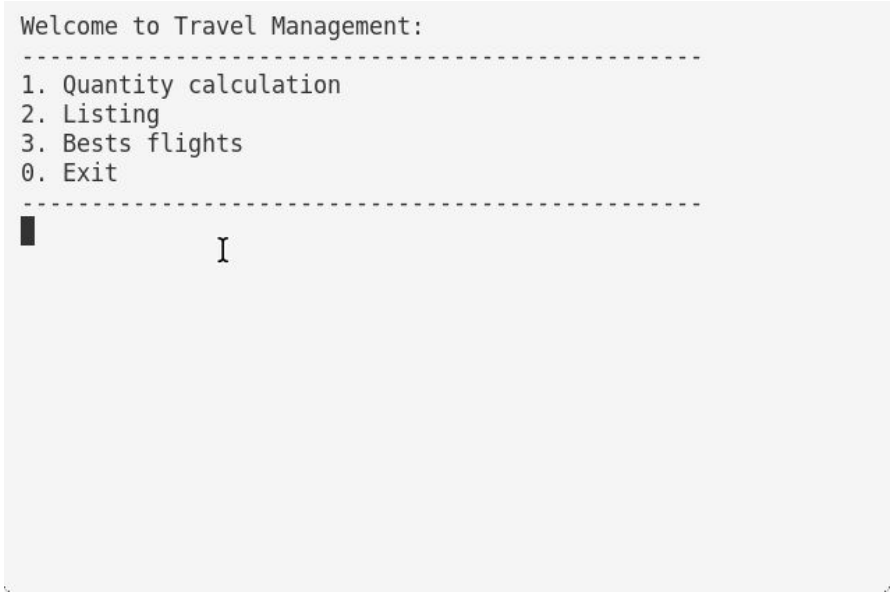
```
Welcome to Travel Management:
-------------------------------------------------
1. Quantity calculation
2. Listing
3. Bests flights
0. Exit
-------------------------------------------------
```

Extra (4)

Filter Airplanes

To make it more optimized and easier to use, we ask the user to enter all the airlines they would like to filter. We read this line and put each airline in a vector<string>.

We accept one company, none, or a specific set. When you enter none, you will be searched without any filter



Collaboration

(1/2) - Clarisse Carvalho, up202008444
(1/2) - Eduardo Silva, up202301394