Flight Informer

Gonçalo Barroso
Joaquin Aguirre
Diogo Neves

Project Description

Second project for the UC AED. Aim of the project is to make a user-friendly program that can be used to see information about worldwide airports, airlines and flights while using the graph data structure.

Classes

WorldGraphManager

- FileManager _vectors
- vector<Airline> _airlines
- vector<Flight> _flights
- Graph<Airport> _world

Airport

- string _code
- string _name
- string _city
- string _country
- float _latitude
- float _longitude

App

- WorldGraphManager _worldGraph

Flight

- string _source
- string _target
- string _airline

Airline

- string _code
- string _name
- string _callsign
- string _country

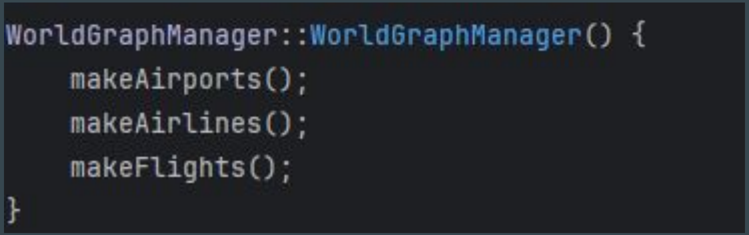
FileManager

- vector<string> _airportsfile
- vector<string> _airlinesfile
- vector<string> _flightsfile

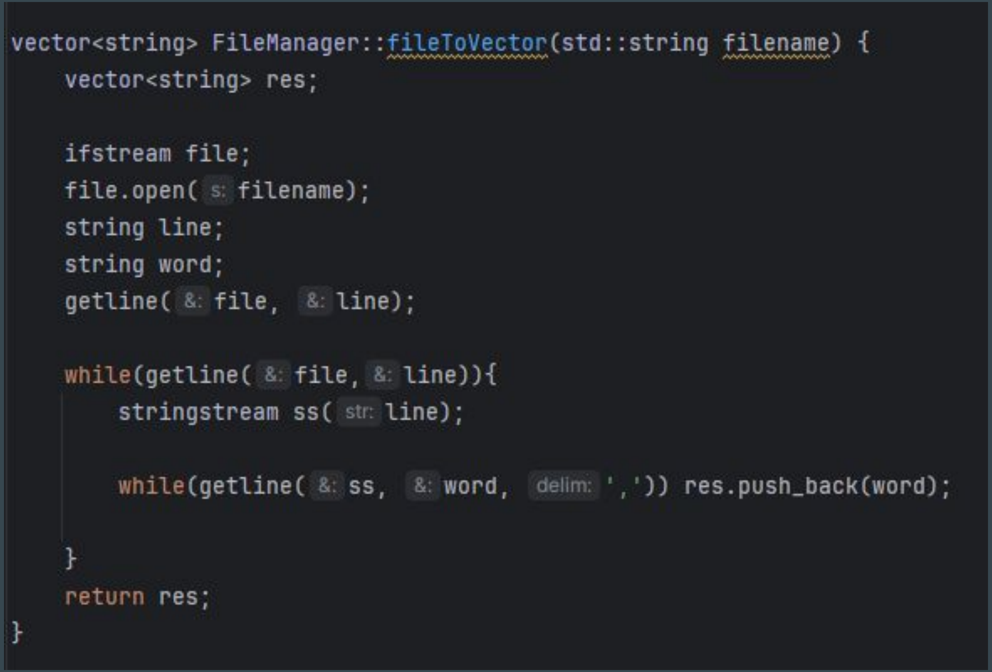
Data Set Reading

In main(), the App function run() is called. In here, when its WorldGraphManager value gets declared, by default it calls the methods that read the data in the file.

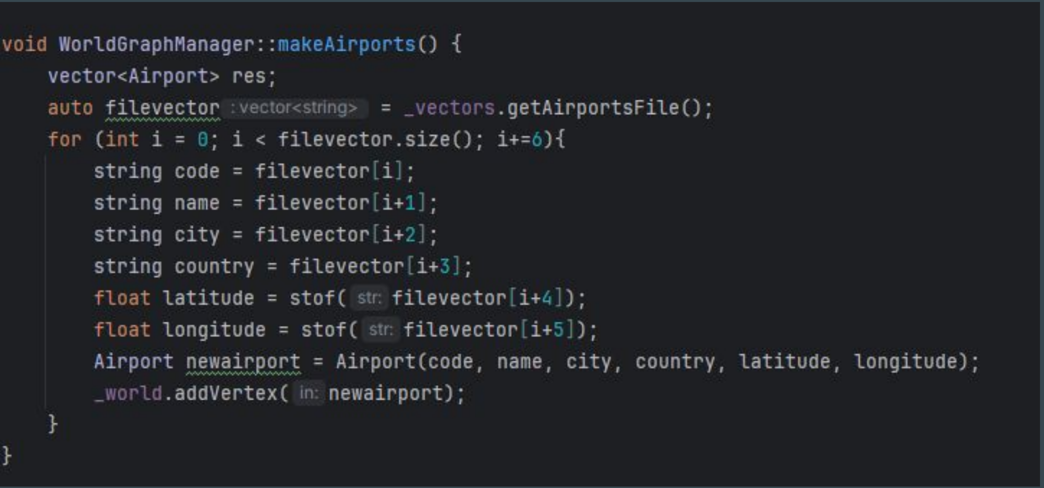
The FileManager class, contains the data of each of the three files in form of 3 vectors. The WorldGraphManager then turns the data in this vectors into the proper data structure, the graph.



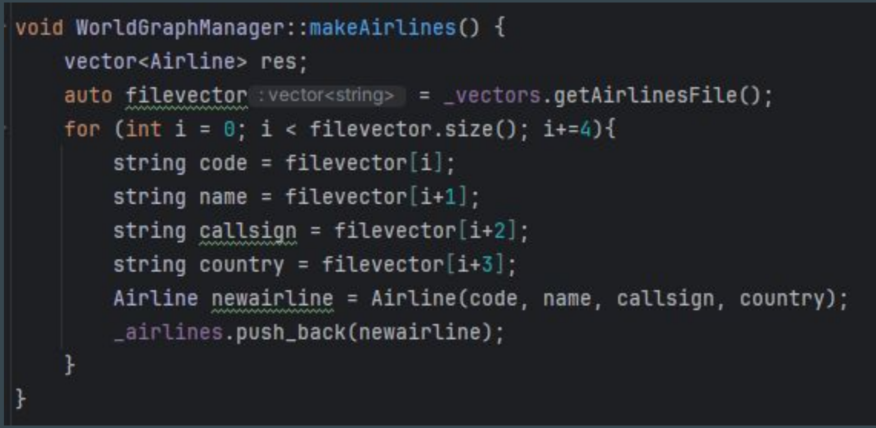
```
WorldGraphManager::WorldGraphManager() {
    makeAirports();
    makeAirlines();
    makeFlights();
}
```



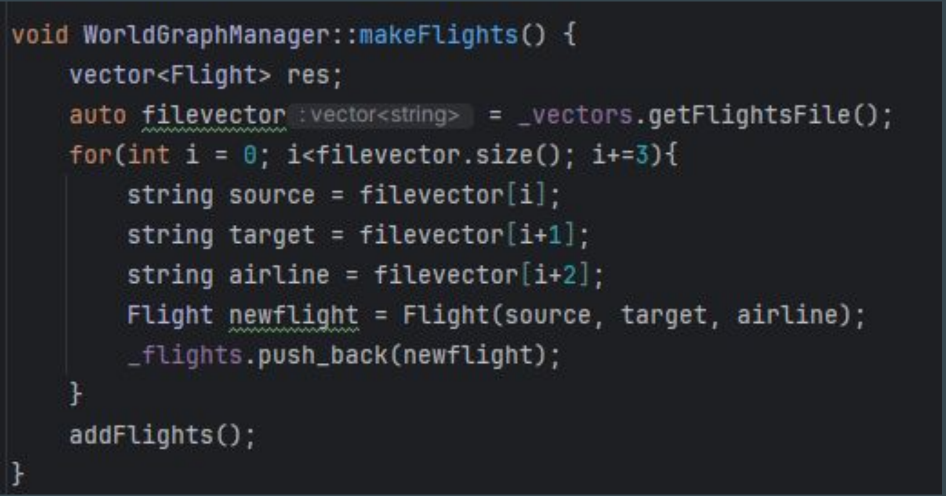
```
vector<string> FileManager::fileToVector(std::string filename) {
    vector<string> res;

    ifstream file;
    file.open( s: filename);
    string line;
    string word;
    getline( &: file,  &: line);

    while(getline( &: file, &: line)){
        stringstream ss( str: line);

        while(getline( &: ss,  &: word,  delim: ',')) res.push_back(word);

    }
    return res;
}
```



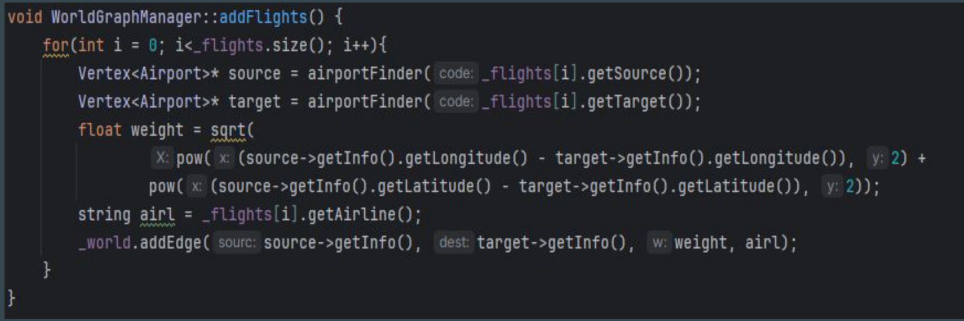
```
void WorldGraphManager::makeAirports() {
    vector<Airport> res;
    auto filevector :vector<string> = _vectors.getAirportsFile();
    for (int i = 0; i < filevector.size(); i+=6){
        string code = filevector[i];
        string name = filevector[i+1];
        string city = filevector[i+2];
        string country = filevector[i+3];
        float latitude = stof( str: filevector[i+4]);
        float longitude = stof( str: filevector[i+5]);
        Airport newairport = Airport(code, name, city, country, latitude, longitude);
        _world.addVertex( in: newairport);
    }
}
```



```
void WorldGraphManager::makeAirlines() {
    vector<Airline> res;
    auto filevector :vector<string> = _vectors.getAirlinesFile();
    for (int i = 0; i < filevector.size(); i+=4){
        string code = filevector[i];
        string name = filevector[i+1];
        string callsign = filevector[i+2];
        string country = filevector[i+3];
        Airline newairline = Airline(code, name, callsign, country);
        _airlines.push_back(newairline);
    }
}
```



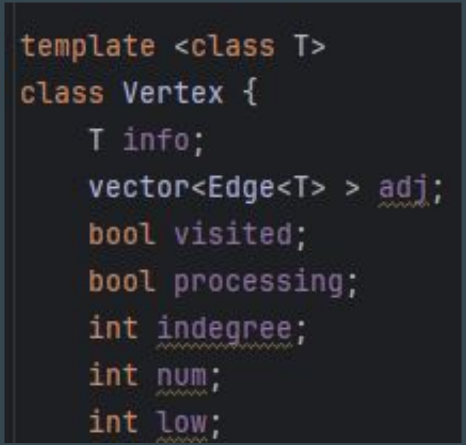
```
void WorldGraphManager::makeFlights() {
    vector<Flight> res;
    auto filevector :vector<string> = _vectors.getFlightsFile();
    for(int i = 0; i<filevector.size(); i+=3){
        string source = filevector[i];
        string target = filevector[i+1];
        string airline = filevector[i+2];
        Flight newflight = Flight(source, target, airline);
        _flights.push_back(newflight);
    }
    addFlights();
}
```



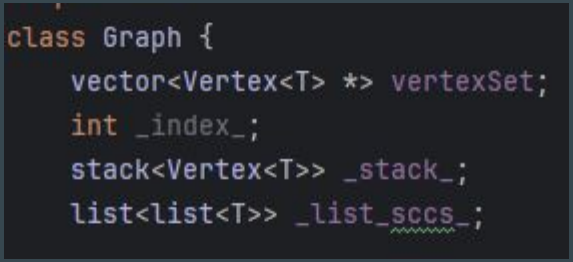
```
void WorldGraphManager::addFlights() {
    for(int i = 0; i<_flights.size(); i++){
        Vertex<Airport>* source = airportFinder( code: _flights[i].getSource());
        Vertex<Airport>* target = airportFinder( code: _flights[i].getTarget());
        float weight = sqrt(
                X: pow( x: (source->getInfo().getLongitude() - target->getInfo().getLongitude()),  y: 2) +
                pow( x: (source->getInfo().getLatitude() - target->getInfo().getLatitude()),  y: 2));
        string airl = _flights[i].getAirline();
        _world.addEdge( sourc: source->getInfo(),  dest: target->getInfo(),  w: weight, airl);
    }
}
```

The Graph

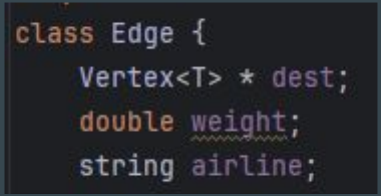
The graph structure used in the project is based on the one structures studied during classes, with few extra methods added to help with implementation.
Our Graph class represents the data that was given to us: the Vertex represent Airports and Edges represent Flights.
Some methods were added to the header file for better implementation.



```
template <class T>
class Vertex {
    T info;
    vector<Edge<T> > adj;
    bool visited;
    bool processing;
    int indegree;
    int num;
    int low;
```



```
class Graph {
    vector<Vertex<T> *> vertexSet;
    int _index_;
    stack<Vertex<T>> _stack_;
    list<list<T>> _list_sccs_;
```



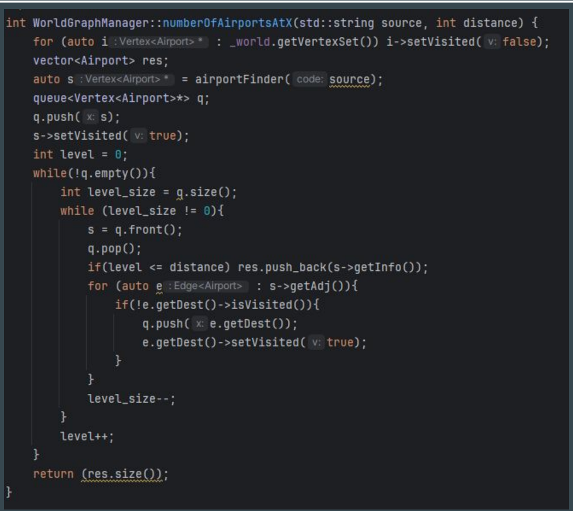
```
class Edge {
    Vertex<T> * dest;
    double weight;
    string airline;
```

Step 3: Features (1-7)

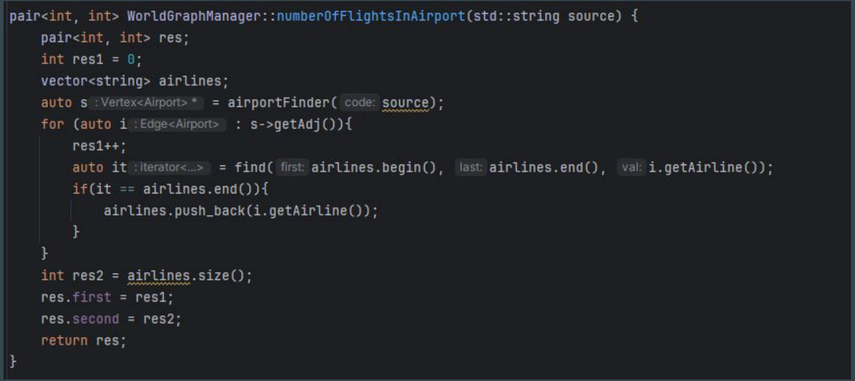
These features include basic information that isn’t related to a best flight to the user. This includes info like:

- Number of airports and number of flights;
- Number of flights out of an airport and from how many different airlines;
- Number of flights per city per airline;
- Number of different countries that a given airport/city flies to;
- Number of destinations (airports or cities) available for a given airport;
- Number of reachable airports from a given airport in a maximum of X stops;
- Airport with the most related flights.

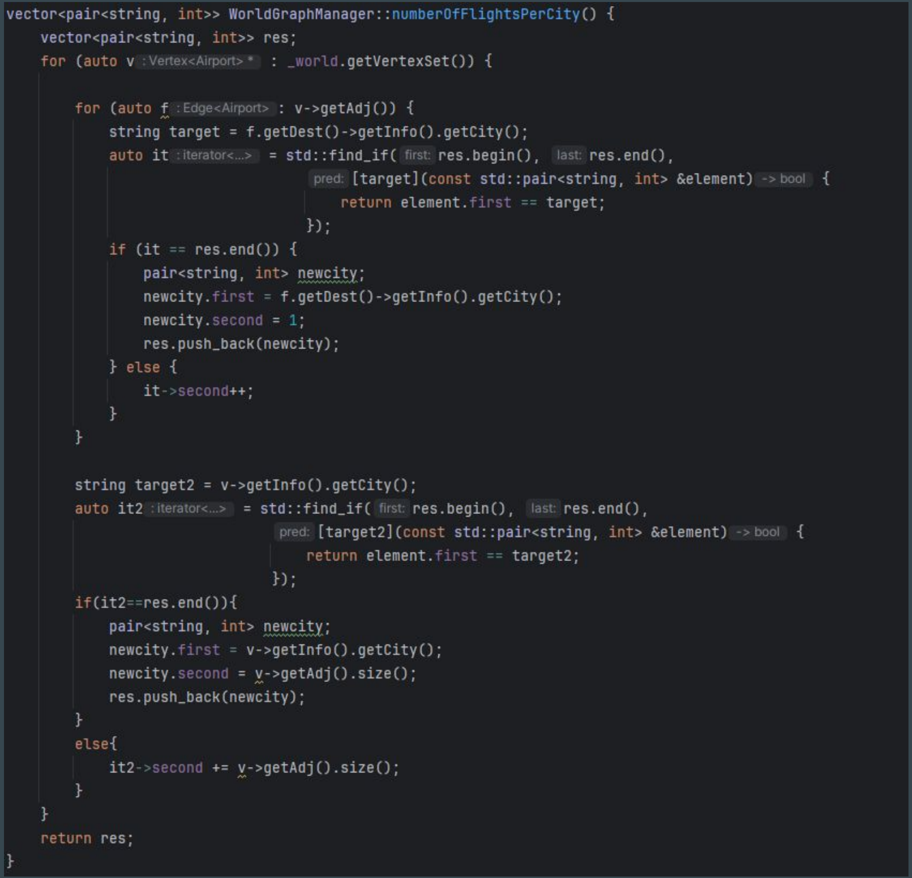
Step 3: Features (1-7) Example methods:



```cpp
int WorldGraphManager::numberOfAirportsAtX(std::string source, int distance) {
    for (auto i :Vertex<Airport> * : _world.getVertexSet()) i->setVisited( v: false);
    vector<Airport> res;
    auto s :Vertex<Airport> * = airportFinder( code: source);
    queue<Vertex<Airport>*> q;
    q.push( x: s);
    s->setVisited( v: true);
    int level = 0;
    while(!q.empty()){
        int level_size = q.size();
        while (level_size != 0){
            s = q.front();
            q.pop();
            if(level <= distance) res.push_back(s->getInfo());
            for (auto e :Edge<Airport> : s->getAdj()){
                if(!e.getDest()->isVisited()){
                    q.push( x: e.getDest());
                    e.getDest()->setVisited( v: true);
                }
            }
            level_size--;
        }
        level++;
    }
    return (res.size());
}
```



```cpp
pair<int, int> WorldGraphManager::numberOfFlightsInAirport(std::string source) {
    pair<int, int> res;
    int res1 = 0;
    vector<string> airlines;
    auto s :Vertex<Airport> * = airportFinder( code: source);
    for (auto i :Edge<Airport> : s->getAdj()){
        res1++;
        auto it :iterator<...> = find( first: airlines.begin(), last: airlines.end(), val: i.getAirline());
        if(it == airlines.end()){
            airlines.push_back(i.getAirline());
        }
    }
    int res2 = airlines.size();
    res.first = res1;
    res.second = res2;
    return res;
}
```



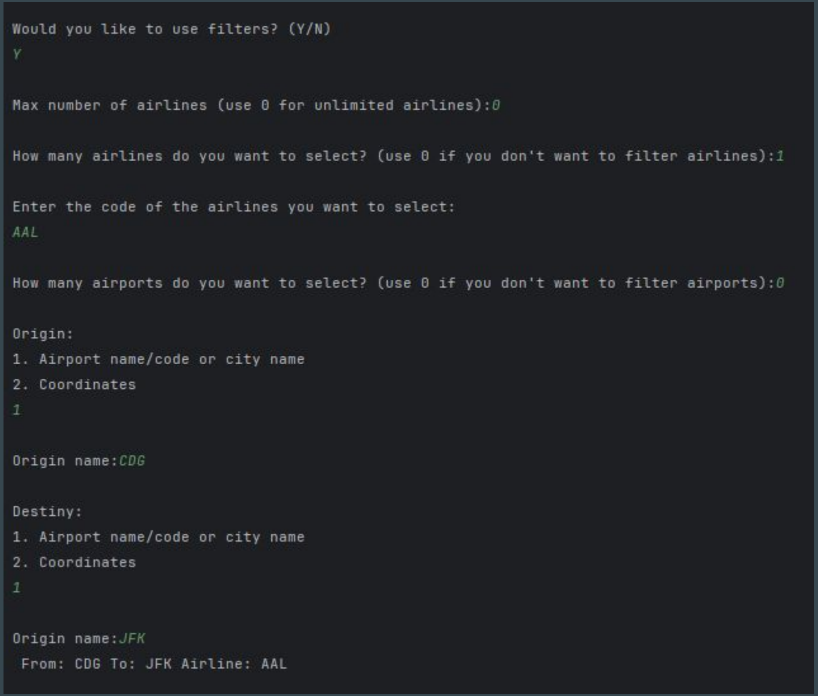
```cpp
vector<pair<string, int>> WorldGraphManager::numberOfFlightsPerCity() {
    vector<pair<string, int>> res;
    for (auto v :Vertex<Airport> * : _world.getVertexSet()) {

        for (auto f :Edge<Airport> : v->getAdj()) {
            string target = f.getDest()->getInfo().getCity();
            auto it :iterator<...> = std::find_if( first: res.begin(), last: res.end(),
                                                  pred: [target](const std::pair<string, int> &element) -> bool {
                                                      return element.first == target;
                                                  });
            if (it == res.end()) {
                pair<string, int> newcity;
                newcity.first = f.getDest()->getInfo().getCity();
                newcity.second = 1;
                res.push_back(newcity);
            } else {
                it->second++;
            }
        }

        string target2 = v->getInfo().getCity();
        auto it2 :iterator<...> = std::find_if( first: res.begin(), last: res.end(),
                                               pred: [target2](const std::pair<string, int> &element) -> bool {
                                                   return element.first == target2;
                                               });
        if(it2==res.end()){
            pair<string, int> newcity;
            newcity.first = v->getInfo().getCity();
            newcity.second = v->getAdj().size();
            res.push_back(newcity);
        }
        else{
            it2->second += v->getAdj().size();
        }
    }
    return res;
}
```

Step 4 and 5: Find the best flight(s) and add filters

For this part of the project we decided to use a modified version of the BFS algorithm. The time complexity of the main function is: O(V+E · (L+A)), where V is the number of vertices in the graph, E is the number of edges in the graph, L represents the average length of the paths and A represents the maxAirlines. If we add filters the processing is a bit faster because we only enable the Airports and Airlines that the user selects.

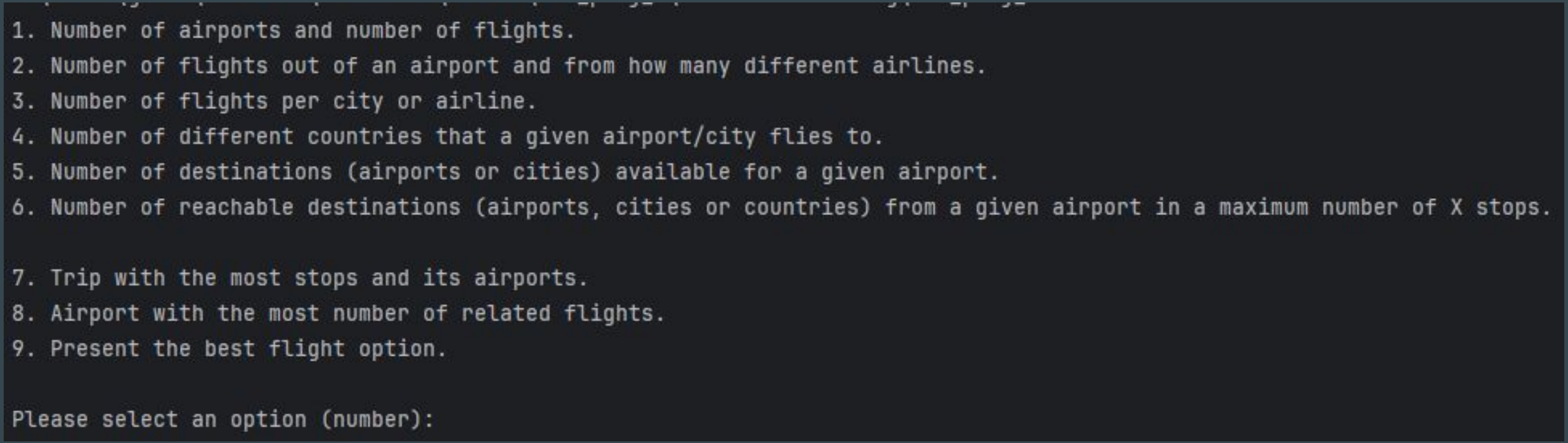
The user can select the location they want to travel from and to by airport code, airport name, city or coordenades. Then we check every single possible source and destiny point and keep the shortest ones taking into account the user's preferences. If there are no flights available with said preferences the system doesn't display anything.

This can be made faster with hashtables as the findVertex() function takes a long time and could be constant if we used a hash table instead of a vector to store our vertices and edges.



```
Would you like to use filters? (Y/N)
Y

Max number of airlines (use 0 for unlimited airlines):0

How many airlines do you want to select? (use 0 if you don't want to filter airlines):1

Enter the code of the airlines you want to select:
AAL

How many airports do you want to select? (use 0 if you don't want to filter airports):0

Origin:
1. Airport name/code or city name
2. Coordinates
1

Origin name:CDG

Destiny:
1. Airport name/code or city name
2. Coordinates
1

Origin name:JFK
 From: CDG To: JFK Airline: AAL
```

The Interface



```
1. Number of airports and number of flights.
2. Number of flights out of an airport and from how many different airlines.
3. Number of flights per city or airline.
4. Number of different countries that a given airport/city flies to.
5. Number of destinations (airports or cities) available for a given airport.
6. Number of reachable destinations (airports, cities or countries) from a given airport in a maximum number of X stops.

7. Trip with the most stops and its airports.
8. Airport with the most number of related flights.
9. Present the best flight option.

Please select an option (number):
```

The methods offered to the user are all displayed when the code is ran and can be selected through a number. Methods that ask for extra information from the user, output more questions.

Project Highlights

- The search for the best possible flights from one point to another, this included searching by airport, city or coordenades.
- The methods used for filtering output results when searching for best flight options are well implemented, quick and user-friendly.

Project Difficulties

- The main problem we had when developing the project was implementing the hash map functionality into both the vertices and edges because we started with vectors which made it impossible to add this functionality without restructuring the entire project.
- The period of time that the project was available for was enough but working through the holidays was difficult.

Members’ effort:

- Gonçalo Barroso 33%
- Joaquin Aguirre 33%
- Diogo Neves 33%