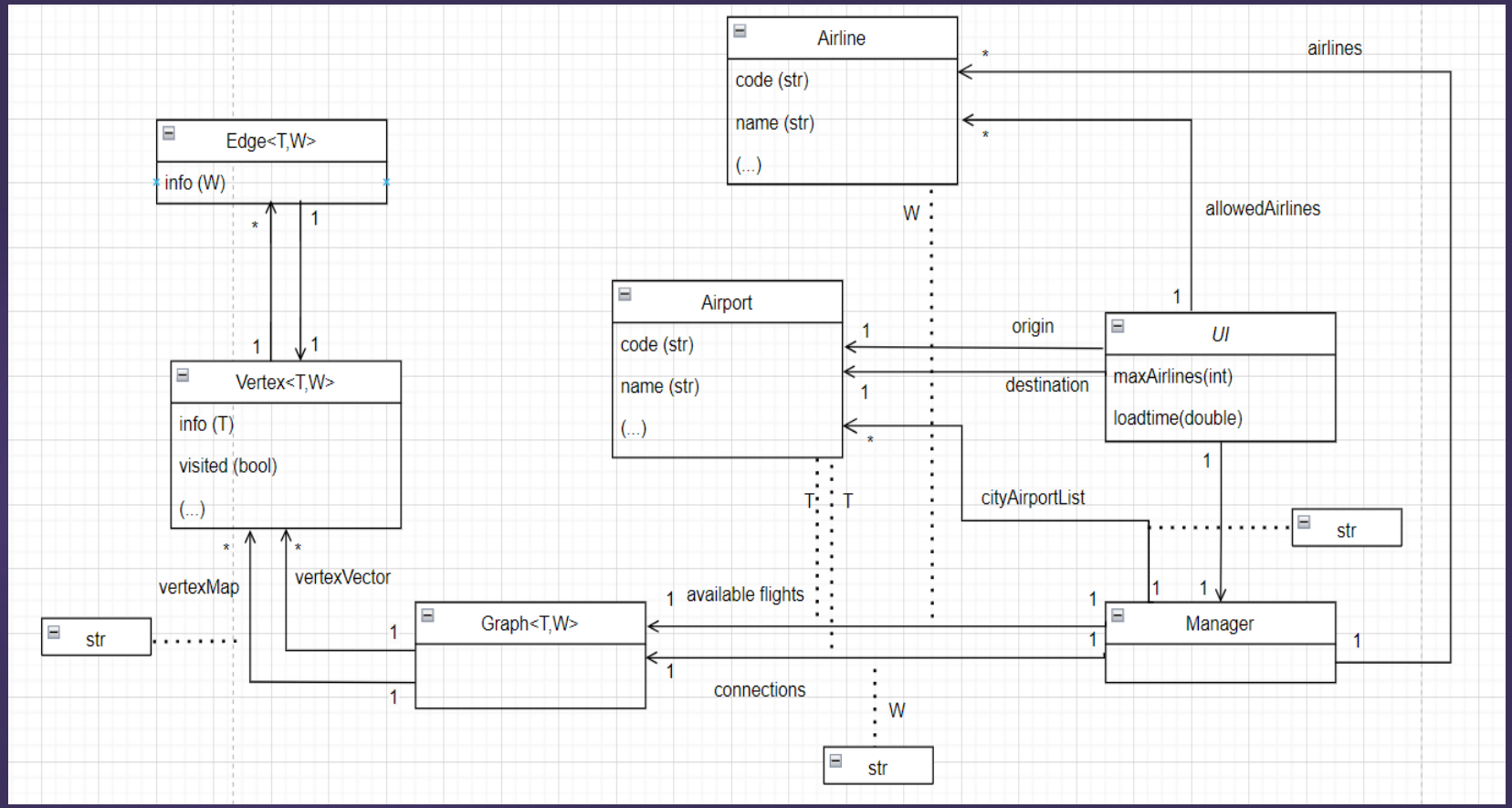
Programming project - AED

GROUP 3-8

Members:
Diogo Pinto 202205225
Gabriel Lima 202206693
Pedro Moreira 202108844

Project classes



Classes are defined in the files with the same name and implemented in the files that contain the class name in the beginning of the file name.

Graph, Vertex and Edge are the only exceptions, as they are defined and implemented in the Graph.h file.

Data loading

The data contained in the .csv files is loaded onto the program with the load functions of the Manager class, as shown in the image:

```
/**
 * Constructs the manager and loads the information from the data folder.
 * Additionally, calculates the time it took to parse and store the data.
*/
UI::UI() : manager(Manager()) {
    auto start = std::chrono::high_resolution_clock::now();
    manager.loadAirports();
    manager.loadAirlines();
    manager.loadFlights();
    auto end = std::chrono::high_resolution_clock::now();
    loadtime = std::chrono::duration<double>(end - start).count();
}
```

Data loading

Each of the .cvs files is read and stored inside a manager instance.

The data contained in airlines.cvs is stored in the airlines map, using its code as the key to access it.

The data in airports.cvs as a vertex in the connections and available_flights graphs. These vertexes from available_flights are also stored in cityAirportList, a map of with a list of airports grouped by city.

Finally, the data in flights.csv is stored as the edges in the connections and available_flights graphs.

Graphs

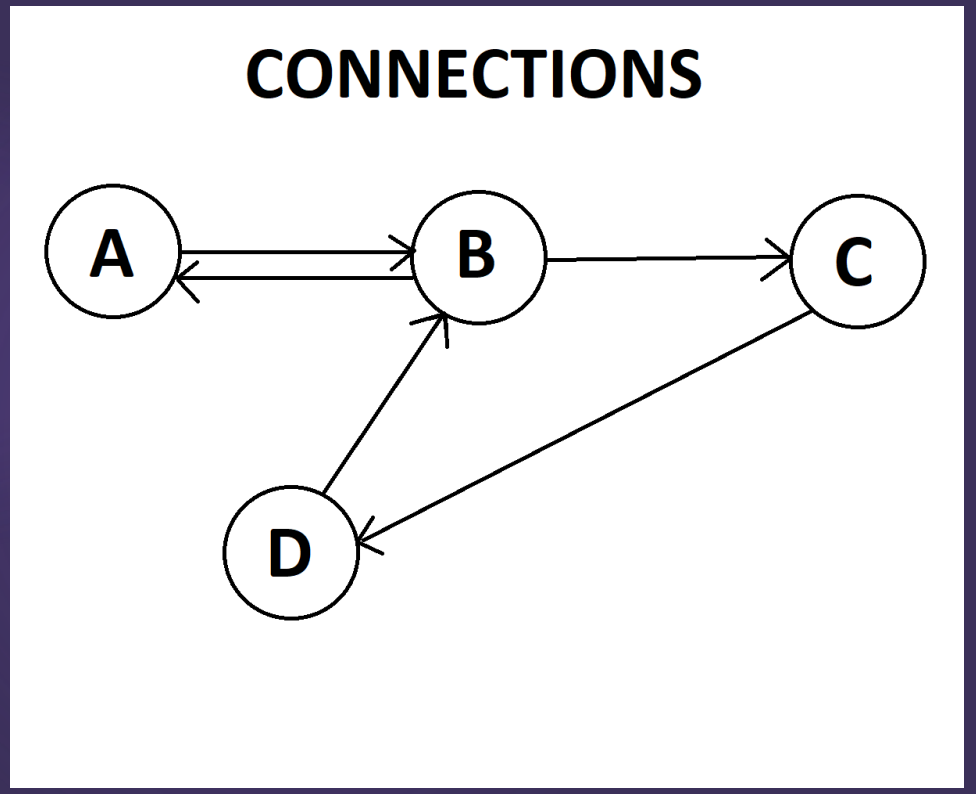
We employed unidirectional graphs as a way to store the airports and all the flights between them.

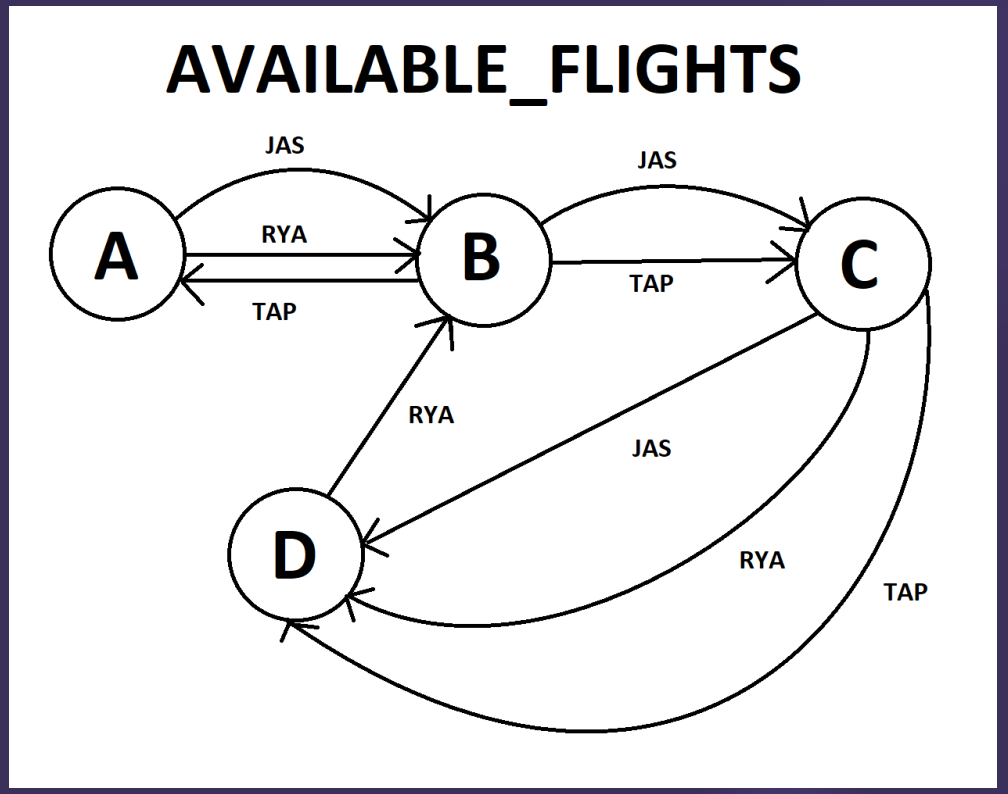
Although we used two of them, "connections" and "available_flights" generally store the same information, with the distinction that "connections" does not differentiate between different airlines, meaning that it contains less edges but less information. We used this to speed up some of the necessary functions that did not require the airline information.

It is important to note that "available_flights" can have multiple edges pointing to the same vertex.

Graphs

The following illustrative images exemplify what the graphs look like where A, B, C and D are airports and TAP, RYA and JAS are airlines.





Functionalities:

Statistics Display

```
void UI::showAirport(std::string str) {
    bool numberInputed = false;
    int stops = -20;
    std::vector<size_t> destinationDataVector;

    std::vector<size_t> dataVector1 = manager.airportStats(str);
    if (dataVector1[0] == __INT64_MAX__) {
        helpMsg("Invalid airport name! Airport doesn't belong to the network!"
        return;
    }
    std::vector<size_t> dataVector2 = manager.destinationsFromAirport(str);

    size_t totalFlights = dataVector1[0];
    size_t totalAirlines = dataVector1[1];
    size_t totalAirports = dataVector2[0];
    size_t totalCities = dataVector2[1];
    size_t totalCountries = dataVector2[2];

    size_t destinationAirports = 0;
    size_t destinationCities = 0;
    size_t destinationCountries = 0;

    Airport *airport = manager.getConnections().findVertex(str)->getInfo();
```

Each statistics screen displays all the information pertaining to the selected element. Most of these statistics are stored locally or can be obtained using the functions:

"manager.[class name]Stats"

(varying complexities)

Functionalities:

Reachable destinations

We utilized a breadth first search to find all the possible destinations in a specified number of steps.

We find the number of reachable cities and countries by adding them to a set and then returning the size of that set.

```
for (auto i : connections.getVertexSet()) {
    i->setVisited(false);
    i->setNum(__INT32_MAX__);
}

std::set<std::string> cities, countries;
list<Vertex<Airport *, int>*> queue;
size_t count = 0;
vtx->setVisited(true);
vtx->setNum(0);
queue.push_back(vtx);

while (!queue.empty()) {
    auto u = queue.front();
    queue.pop_front();
    for (auto i : u->getAdj()) {
        auto w = i.getDest();
        if (!w->isVisited()) {
            auto lvl = u->getNum() + 1;
            w->setVisited(true);
            w->setNum(lvl);
            queue.push_back(w);
            if (lvl <= x) {
                auto wa = w->getInfo();
                cities.insert(wa->getCity() + ", " + wa->getCountry());
                countries.insert(wa->getCountry());
                count++;
            }
        }
    }
}
```

Functionalities:

Maximum trip

To find the maximum possible trips between two airports we did a breadth first search for every airport trying to find all the trips with maximum depth.

```
std::pair<MaxTripVector, int> Manager::maximumTrip() {
    MaxTripVector res;
    int maxTrip = 0;

    for (auto i : connections.getVertexSet()) {
        bfsFind(i, maxTrip, res);
    }

    return std::make_pair(res, maxTrip);
}
```

```
void Manager::bfsFind(Vertex<Airport *, int> *vtx, int &maxTrip, MaxTripVector &res)
    for (auto i : connections.getVertexSet()) {
        i->setVisited(false);
        i->setNum(__INT32_MAX__);
    }

    list<Vertex<Airport *, int> *> queue;
    vtx->setVisited(true);
    vtx->setNum(0);
    queue.push_back(vtx);

    while (!queue.empty()) {
        auto u = queue.front();
        queue.pop_front();
        for (auto i : u->getAdj()) {
            auto w = i.getDest();
            if (!w->isVisited()) {
                w->setVisited(true);
                w->setNum(u->getNum() + 1);
                queue.push_back(w);
                if (w->getNum() > maxTrip) {
                    maxTrip = w->getNum();
                    res.clear();
                }
                if (w->getNum() == maxTrip)
                    res.push_back(std::make_pair(vtx->getInfo(), w->getInfo()));
            }
        }
    }
}
```

Functionalities:

Searching for a trip

A user might want to know if he can travel between two locations (airports, cities or coordinates), and what connections and flights he is required to take.

To do this, we used a Breadth first algorithm to calculate the best path in a fast way. To output more than one path, the last vector saves the various vertexes that reach there with the same number of stops.

```
vector<Trip> res;
list<Vertex<Airport *, Airline *>*> queue;
for (auto i : start) {
    i->setVisited(true);
    i->setNum(0);
    queue.push_back(i);
}
int minStops = __INT32_MAX__;
while (!queue.empty()) {
    auto u = queue.front();
    queue.pop_front();
    if (u->getNum() + 1 > minStops)
        continue;
    for (auto i : u->getAdj()) {
        auto w = i.getDest();
        if (!w->isVisited()) {
            auto found = std::find(end.begin(), end.end(), w);
            if (found != end.end() && u->getNum() + 1 <= minStops) {
                minStops = u->getNum() + 1;
                (*found)->addLast(u, i.getInfo());
                continue;
            }
            w->setVisited(true);
            w->clearLast();
            w->addLast(u, i.getInfo());
            w->setNum(u->getNum() + 1);
            queue.push_back(w);
        }
    }
}
```

Functionalities:

Searching for a trip with filters

Even though, the other way of searching might display the best path between locations, it does not allow customization/filtering nor shows all the available options.

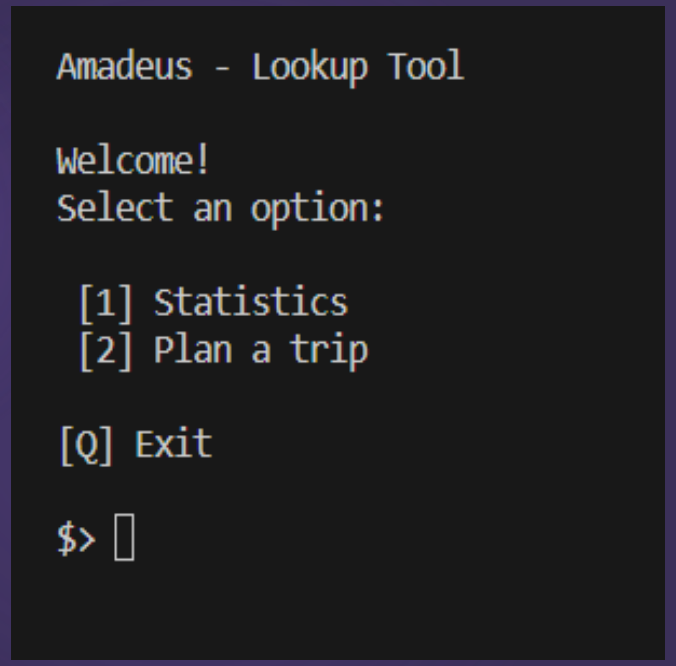
By using a Breadth first algorithm, and saving all the valid paths, it is possible to determine all the possible paths. As such task can take some time without filters, they are imperative to do such search. Additionally, they allow more customization.

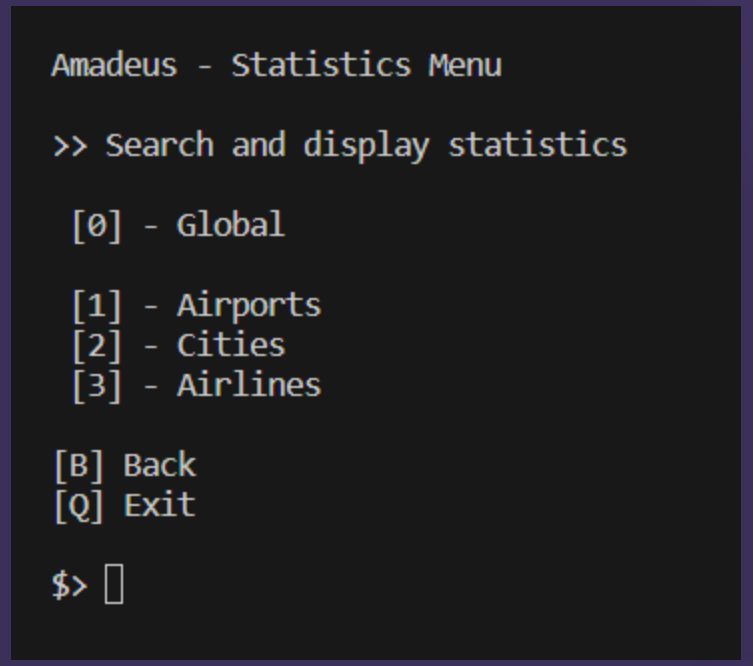
```
int minStops = __INT32_MAX__;
while (!queue.empty()) {
    auto u = queue.front();
    queue.pop_front();
    if (u->getNum() + 1 > minStops)
        continue;
    for (auto i : u->getAdj()) {
        auto w = i.getDest();
        if (w->getNum() == __INT32_MAX__ || u->getNum() + 1 <= w->getNum()) {
            if (!allowedAirlines.empty()) {
                if (std::find(allowedAirlines.begin(), allowedAirlines.end(), i.getInfo()) == allowedAirlines.end())
                    continue;
            }
            w->addLast(u, i.getInfo());
            if (std::find(end.begin(), end.end(), w) != end.end() && u->getNum() + 1 <= minStops) {
                minStops = u->getNum() + 1;
                continue;
            }
            w->setNum(u->getNum() + 1);
            queue.push_back(w);
        }
    }
}
std::cout << "bop\n";
```

User Interface

The User Interface is displayed and navigated using the terminal.

The menus are traversed by inputting the correct commands as prompted by the program, according to what the user wants to do.

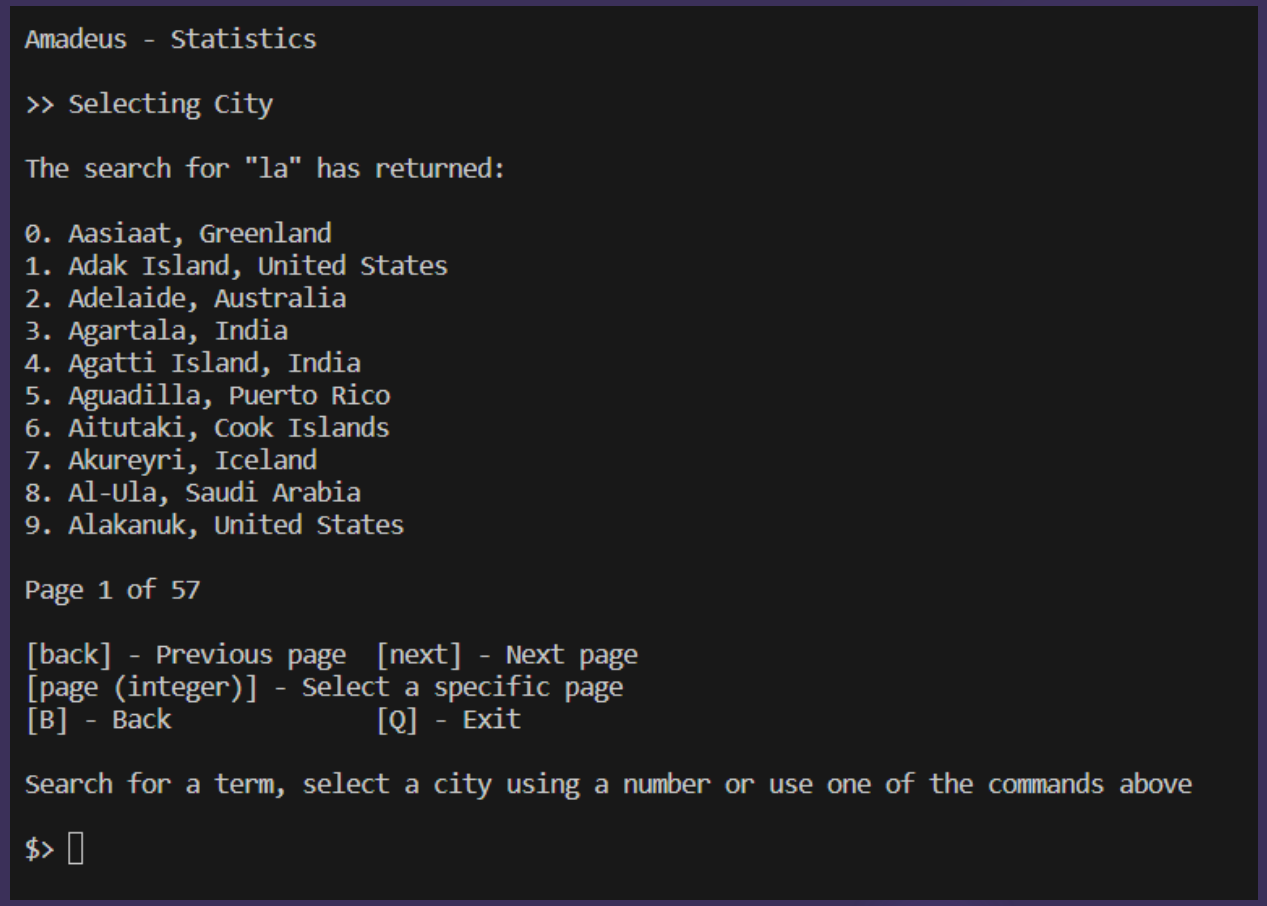




Highlighted features

We would like to highlight the search menu used to search for specific elements both in the statistics and planner sections.

We believe it is particularly intuitive and useful in sparing the user from having to know the specific name of the element they are searching for.



Main difficulties and individual effort

The most difficult parts of this project were certainly the optimization of the loading functions and the path-finding algorithm for the trip planner.

The closeness of the deadline to other tests and projects also made us put our time management skills to the test.

Effort distribution:

- Gabriel Lima 60%
- Diogo Pinto 40%
- Pedro Moreira 0%