FEUP - L.EIC

ROUTE PLANNING TOOL

DA Course 24/25
1st Project

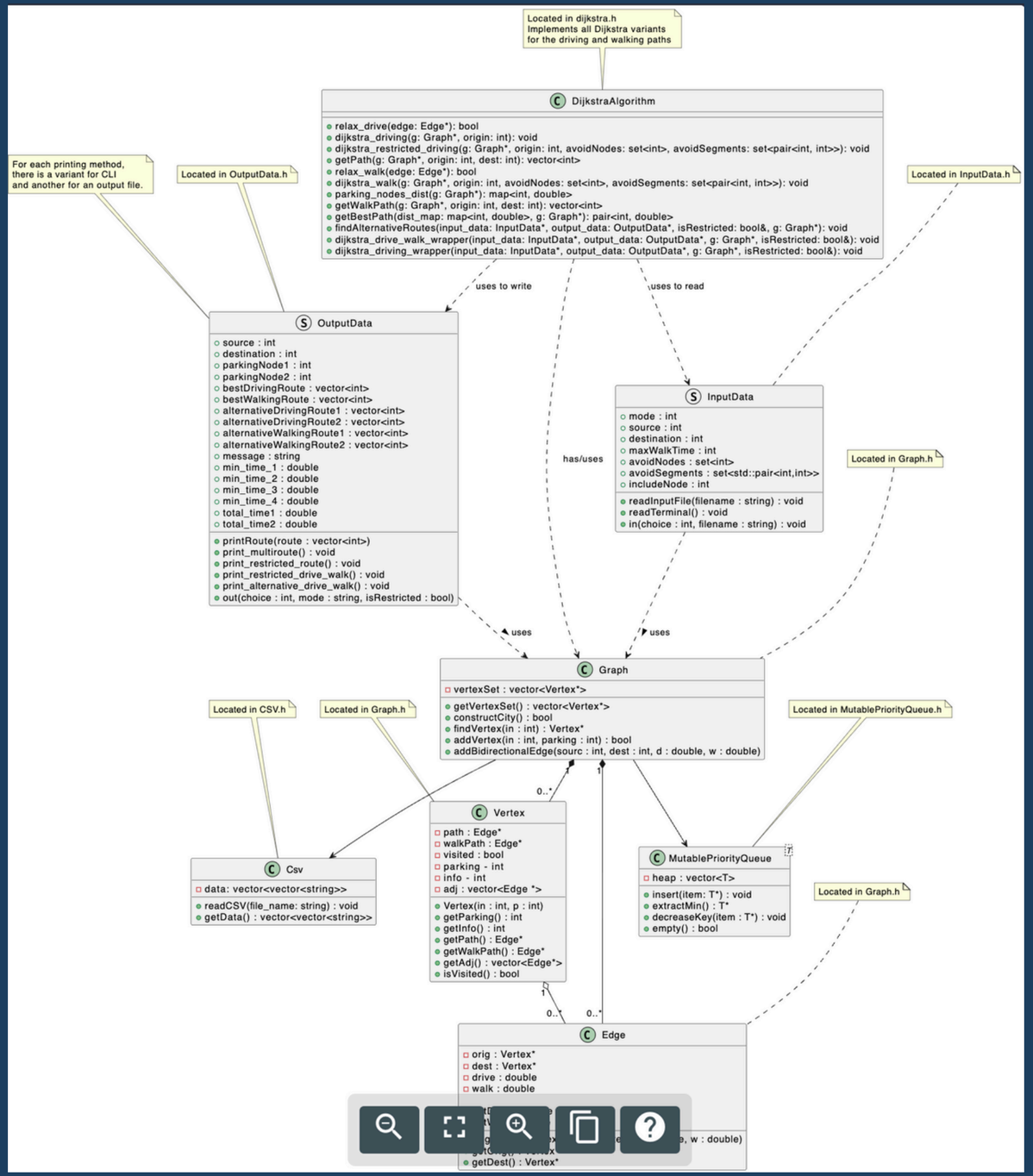
Casemiro de Medeiros up202301897
José Sebastião Vizcaíno up202305009
Rafael Rodrigues up202303855

CONTENT

CLASS DIAGRAM

The Graph, Edge and Vertex datas structures were designed with the data structures used in the TP classes as a base, with some slight modifications.

In addition to these classes, we also used **structs** to represent the input and output data , defined in the InputData.h and OutputData.h header files.



READING THE DATASET

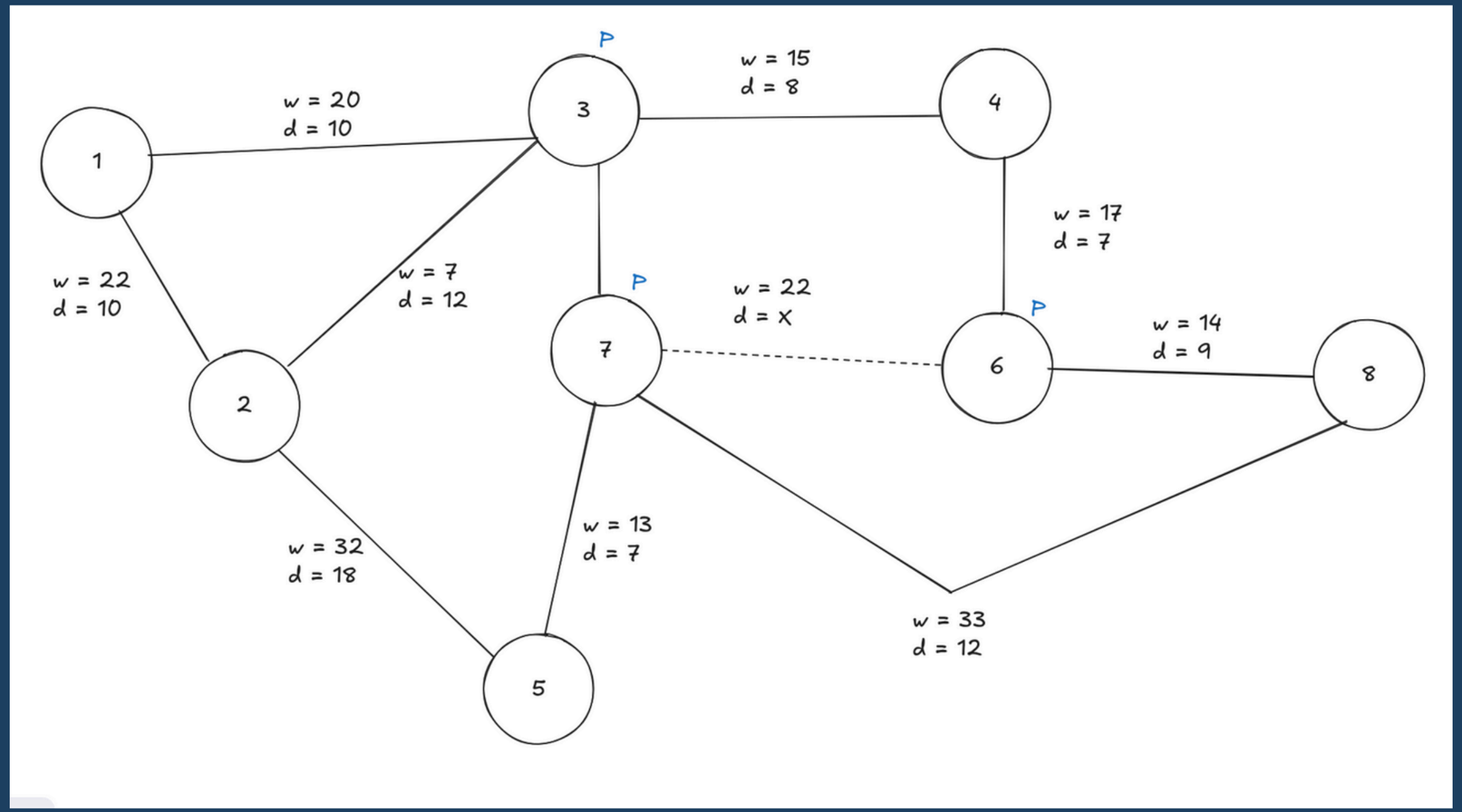
- When starting up the program, the main function creates an empty Graph, and then calls the method *constructCity*.

- The *constructCity* method creates a CSV object, which calls its *readCSV* method with the file Locations.csv.

- In the *readCSV* method, with the use of file and string streams, as well as the getline function, the data is parsed and then stored in a 2D vector. The *constructCity* then creates the vertexes and a map “loc_Idcode”.

- Using the loc_Idcode map, each location code is mapped to its Id. Then the *readCSV* method is called a second time with the file Distances.csv containing information about the edges.

EXAMPLE GRAPH



EXAMPLE GRAPH

```
mode:driving
source:2
destination:8
q
Source:2
Destination:8
BestDrivingRoute:2,3,4,6,8
(36)
AlternativeDrivingRoute:2,5,7,8
(37)
```

```
mode:driving
source:2
destination:8
includenode:7
q
Source:2
Destination:8
RestrictedDrivingRoute:2,5,7,8
(37)
```

```
mode:driving
source:2
destination:8
avoidsegments:(2,5),(3,4)
q
Source:2
Destination:8
RestrictedDrivingRoute:2,3,7,8
(40)
```

```
mode:driving-walking
source:2
destination:8
maxwalktime:8
q
Source:2
Destination:8
DrivingRoute:2,3,4,6
(27)
ParkingNode:6
WalkingRoute:6,8
(14)
TotalTime:68
```

```
mode:driving-walking
source:2
destination:8
maxwalktime:20
q
Source:2
Destination:8
DrivingRoute:2,3,4,6
(27)
ParkingNode:6
WalkingRoute:6,8
(14)
TotalTime:41
```

```
mode:driving-walking
source:2
destination:8
maxwalktime:8
q
No path with max. walk time of 8 minutes was found.

Alternative routes found:

Source:2
Destination:8
DrivingRoute1:2,3,4,6
(27)
ParkingNode1:6
WalkingRoute1:6,8
(14)
TotalTime1:41
DrivingRoute2:2,5,7
(25)
ParkingNode2:7
WalkingRoute2:7,8
(33)
TotalTime2:58
```

IMPLEMENTED FUNCTIONALITIES

- Simple command-line menu;
- Reading and parsing input data, either through command-line inputs or through an input file;
- Route planning and analysis for driving only;
- Route planning and analysis for driving with node and segment restrictions;
- Environmentally friendly route planning and analysis for driving and walking;
- Aproximate solution display if no suitable route is found;

USER INTERFACE

The menu is the main part of the User Interface. There you can see what is implemented and read inputs.

To pass inputs via a .txt file, place the file inside the input_output folder in the project directory. After that, you only need to specify the file name. The output will be generated in the same way. If the input requires alternative routes, those will also be generated automatically.

```
Test City constructed successfully!
========================================
    ROUTE PLANNING SYSTEM - MAIN MENU
developed by Casemiro, Rafael and Sebastião
========================================
To choose an option, enter the symbol inside the brackets:

[0] What we have implemented!
[1] Read an input via .txt file
[2] Read an input via CLI
[x] Close the program
```

```
[0] What we have implemented!
[1] Read an input via .txt file
[2] Read an input via CLI
[x] Close the program
1
Please enter the filename: input.txt
output.txt successfully created in ./input_output/

alternatives_routes.txt successfully created in ./input_output/
```

USER INTERFACE

In the CLI version, you can enter all your input lines. When finished, type "q" to exit input mode. The routes will then be displayed.

To close the program, simply type "x".

```
[0] What we have implemented!
[1] Read an input via .txt file
[2] Read an input via CLI
[x] Close the program
2
When you have finished passing inputs, enter 'q'
mode:driving
source:2
destination:8
q
Source:2
Destination:8
BestDrivingRoute:2,3,4,6,8
(36)
AlternativeDrivingRoute:2,5,7,8
(37)
```

ROUTE PLANNING - DRIVING ONLY

Using Dijkstra's algorithm taught in class as a foundation, this implementation computes the shortest driving distances from the origin node to the other nodes in Graph g.
It uses the *relax_drive* function for edge relaxation and a Mutable Priority Queue to always process the closest unvisited node.

For finding the alternative route, the nodes used in the first route are marked as visited by the *getPath* function. Then, when dijkstra_driving is called a second time, it ignores visited nodes.

Since this implementation uses a Priority Queue, the time complexity for this algorithm is O((|V|+|E|)log(|V|)), with V vertexes and E edges.

```
void dijkstra_driving(Graph * g, const int &origin) {
    if (g->getVertexSet().empty()) return;
    for (auto v : g->getVertexSet()) {
        v->setDist(INF);
        v->setPath(nullptr);
    }

    Vertex* s = g->findVertex(origin);
    s->setDist(0);

    MutablePriorityQueue<Vertex> pq;
    pq.insert(s);

    while (!pq.empty()) {
        auto v = pq.extractMin();

        for (auto e : v->getAdj()) {
            if (e->getDest()->isVisited()) continue;

            double oldDist = e->getDest()->getDist();
            if (relax_drive(e)) {
                if (oldDist == INF) {
                    pq.insert(e->getDest());
                }
                else {
                    pq.decreaseKey(e->getDest());
                }
            }
        }
    }
```

ROUTE PLANNING - RESTRICTED DRIVING

This implementation is a more complex version of the previous algorithm. Aside from the Graph and origin node, the function dijkstra_restricted_driving takes as parameters a set<int> of avoidNodes and a set<pair<int, int>> of avoidSegments which cannot be used to plan a route.

Similarly to the visited nodes in the previous implementation, the function checks if the node or edge/segment to be processed is restricted, and if so ignores it.

Time complexity: $O((|V|+|E|)\log(|V|))$

```
void dijkstra_restricted_driving(Graph * g, const int &origin,
    const set<int> &avoidNodes, const set<pair<int, int>> &avoidSegments) {

    if (g->getVertexSet().empty()) return;

    for (auto v : g->getVertexSet()) {
        v->setDist(INF);
        v->setPath(nullptr);
    }

    Vertex *s = g->findVertex(origin);
    s->setDist(0);

    MutablePriorityQueue<Vertex> pq;
    pq.insert(s);

    while (!pq.empty()) {
        auto v = pq.extractMin();
        if (avoidNodes.contains(v->getInfo())) continue;

        for (auto e : v->getAdj()) {

            if (avoidSegments.contains({v->getInfo(), e->getDest()->getInfo()})) continue;

            double oldDist = e->getDest()->getDist();

            if (relax_drive(e)) {
                if (oldDist == INF) pq.insert(e->getDest());
                else pq.decreaseKey(e->getDest());
            }
        }
```

ROUTE PLANNING - DRIVING AND WALKING

For this implementation, both the driving and walking segments had to be taken into account. Thus, we created functions that, similarly to the ones related to the restricted driving task, processed the walking distances/weights accordingly (*dijkstra_walk, getWalkPath, relax_walk).*

```
bool relax_walk(Edge *edge)
```

```
void dijkstra_walk(Graph * g, const int &origin,
    const  set<int> &avoidNodes, const set<pair<int, int>> &avoidSegments)
```

```
vector<int> getWalkPath(Graph* g, const int &origin, const int &dest)
```

Another important aspect of this implementation is that *dijkstra_restricted_driving* is used with the source node as the origin, to compute the distances from the source to the parking nodes, whilst *dijkstra_walk* is used with the dest node as the origin, computing the distances from it to the parking nodes as well.

The Graph structure was altered to include methods and attributes related to walking(for example, walkPath, walk weight, etc.), and in the wrapper function for this implementation, **map** data structures and auxiliary functions were used to maintain and store data about both the driving and walking distances(*parking_nodes_dist)*, as well as the best combinations of both driving + walking while respecting the restrictions given (*getBestPath)*.
Time complexity: $O((|V|+|E|)\log(|V|))$

ROUTE PLANNING - APPROX. SOLUTION

This task was the most challenging and complex to implement. The function iterates through a loop, relaxing the given restrictions (maxWalkTime restriction is relaxed through 'Walk Increments') until it finds two alternative routes . It also makes use of maps, similarly to the previous implementation.

Time complexity: $O(Wn^2 + |E| + |V| \log |V|)$, where:

- W = number of walk time increments needed
- n = number of parking nodes
- E = number of edges in graph
- V = number of vertices in graph

```
void findAlternativeRoutes(const InputData* input_data, OutputData* output_data,
    bool& isRestricted, Graph* g) {
```

HIGHLIGHTS

- Being able to manage all the route restrictions and implementing all of the tasks given.
- Simple, but clean command-line interface for the menu.
- Being able to find an approximate optimal route that is humanly possible (avoiding absurdly long walking distances).

MAIN DIFFICULTIES AND PARTICIPATION

The most challenging aspects of this project were the implementation and debugging of the code regarding the parsing and handling of the input data, but also correctly implementing the last task of aproximate routes.

Also, trying to maintain the code base clean and the most modular as possible.

- Casemiro (33%)
- Rafael (33%)
- Sebastião (33%)