

Abstract geometric lines forming various polygons and shapes, primarily in the upper left quadrant of the page.

# A PATH FINDING SYSTEM DONE BY

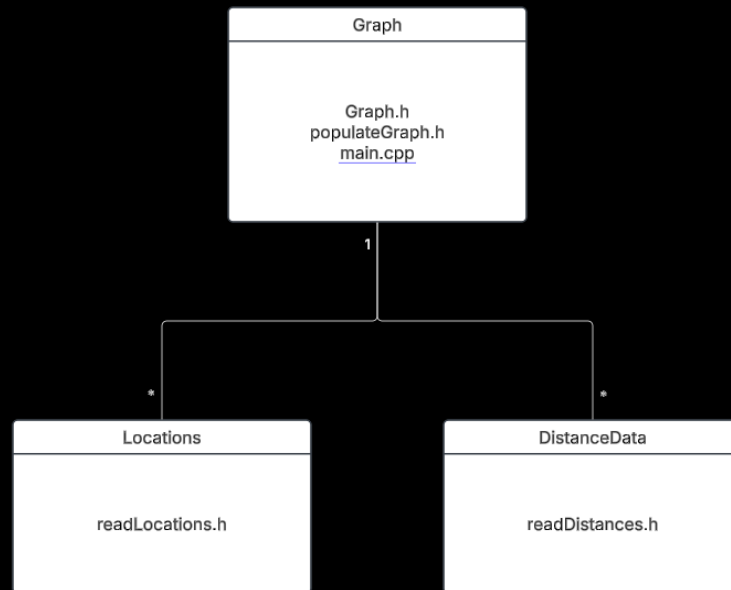
André Gomes - up202304252

André Pinho - up202307008

Carlos Coutinho - up202303946

# CLASS DIAGRAM

Our program has only 1 class, Graph and 2 structs location and DistanceData, making it's class diagram very simple



# READING THE DATASET

We parse the CSV files line by line

And stream that line to get the desired locations and distances

And save variables of these structs into vectors

```
struct DistanceData {  
    std::string CODE1;  
    std::string CODE2;  
    int Driving;  
    int Walking;  
};
```

```
struct location {  
    std::string location;  
    int Id;  
    std::string CODE;  
    bool parking;  
};
```

## Still on Parsing...

```
while(std::getline(file, line)) {  
    std::stringstream stream(line);  
    location row;  
  
    {  
        std::getline(stream, row.location, ',');  
        stream >> row.Id;  
        stream.ignore();  
        std::getline(stream, row.CODE, ',');  
        stream >> row.parking;  
    }  
  
    data.push_back(row);  
}
```

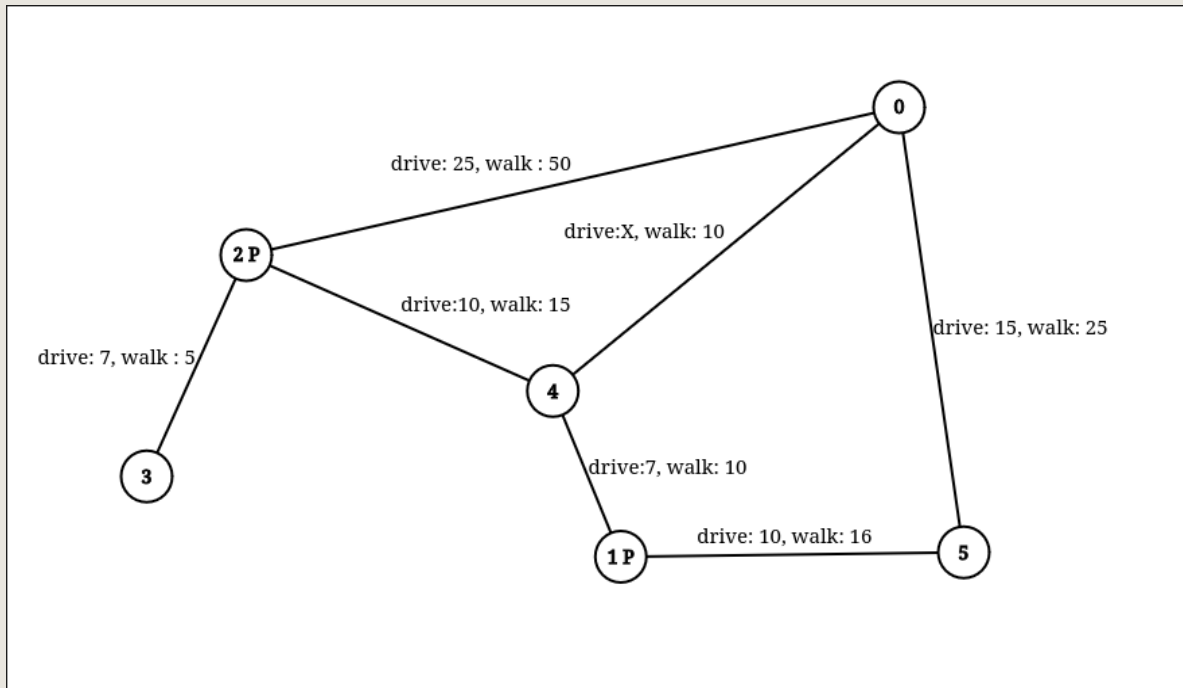
Diagram illustrating the parsing of a CSV line: `MOREIRA SÁ,1223,MRS2,0`

- `location`: MOREIRA SÁ
- `Id`: 1223
- `CODE`: MRS2
- `parking`: 0

# GRAPH

Our nodes are the location Id's, and have a parking attribute

Our edges are double weighted, having both driving and walking weights



# Changes made to Graph.h

On the Vertex class:

- Changed addEdge to create a double weighted edge
- Added parking attribute
- Created getParking() and setParking() methods

On the Edge class:

- Changed constructor to handle 2 weights
- Added second weight attribute
- Changed getWeight() method, to return a pair of weights

On Graph class:

- Changed addEdge function and addBidirectionalEdge to construct double weighted edges

# Implemented Functionalities

## Independent Route Planning

Determines the best route between a source and destination

## Restricted Route Planning

Determines the best route between a source and destination whilst excluding specific nodes or edges from the graph

## Environmentally-Friendly Route Planning

Determines the best route between a source and destination, combining driving and walking

Algorithms used : Dijkstra,  
We simply apply it 1 or more times



# APPROACH

## Independent Route Planning

We just do a normal dijkstra get the best path, remove the edges of this path from the graph and get the best path again for the alternative route (if available)

## Restricted Route Planning

We simply don't initialize the nodes and/or edges that the user wants to avoid, when creating the graph. From there we just do a normal dijkstra and get the best path

## Environmentally-Friendly Route Planning

We dijkstra (driving) from source to every node where there is parking, and from those nodes to the destination (walking) we store the total time (driving + walking) and choose the best paths



# Algorithms

Dijkstra driving and Dijkstra walking

$$O((V + E) \log V)$$

Driving and walking route

$$O(P * (V + E) \log V)$$

where P is number of Parking nodes



# USER INTERFACE

Users are offered a simple menu that allows them to input the data, manually or via a file, the result is printed in the cli and a text file is also generated with the result.

```
How would you like to get there?
We recommend choosing the most Environmentally-Friendly mode!
Please pick one of the options bellow:
```

- 1 - Driving;
- 2 - Driving and Walking.

```
Option: 1
```

```
-----
Where are you starting your trip?
```

```
Location: 8
```

```
Where would you like to go?
```

```
Location: 10
```

```
-----
Is there any places you would like to avoid?
Input them one by one and press enter to finish:
```

```
Place: 4
Place:
```

```
-----
Is there any routes you would like to avoid?
Submit the pair of locations, comma separated, one by one.
Press enter on a new pair to finish.
```

```
Pair: 7,8
Pair:
```

```
-----
Is there any stop you need to make?
Press enter to skip.
```

```
Place:
```

```
-----
CLI Demo
```

The user is prompted, step by step about the route.

1. The mobility mode
2. The start and the destination
3. Locations to avoid
4. Segments to avoid
5. Locations to include

And receives the result printed and on a text file (output.txt)

```
Source:1
```

```
Destination:485
```

```
BestDrivingRoute:1,485(6)
```

```
AlternativeDrivingRoute:1,1045,485(7)
```



## FUNCTIONALITIES TO HIGHLIGHT, WHAT WE ARE MOST PROUD OF

The overall simplicity of the program, and our solution, using almost only 1 algorithm for every route option possible

# MAIN DIFFICULTIES AND PARTICIPATION FROM EACH ONE

Our main difficulties where:

- The final algorithm for the enviromental friendly route, and the approximate solution
- Some of the communication between the CLI and logic

We all ended up doing a little bit of everything and helping each other, but the main division of tasks was :

The CLI and Doxygen - André Pinho

Main Algorithms - André Gomes

The Graph, final Algorithm and PowerPoint – Carlos Coutinho

A series of white, thin, overlapping geometric lines on a black background, forming a complex, abstract shape on the left side of the slide.

THANK YOU