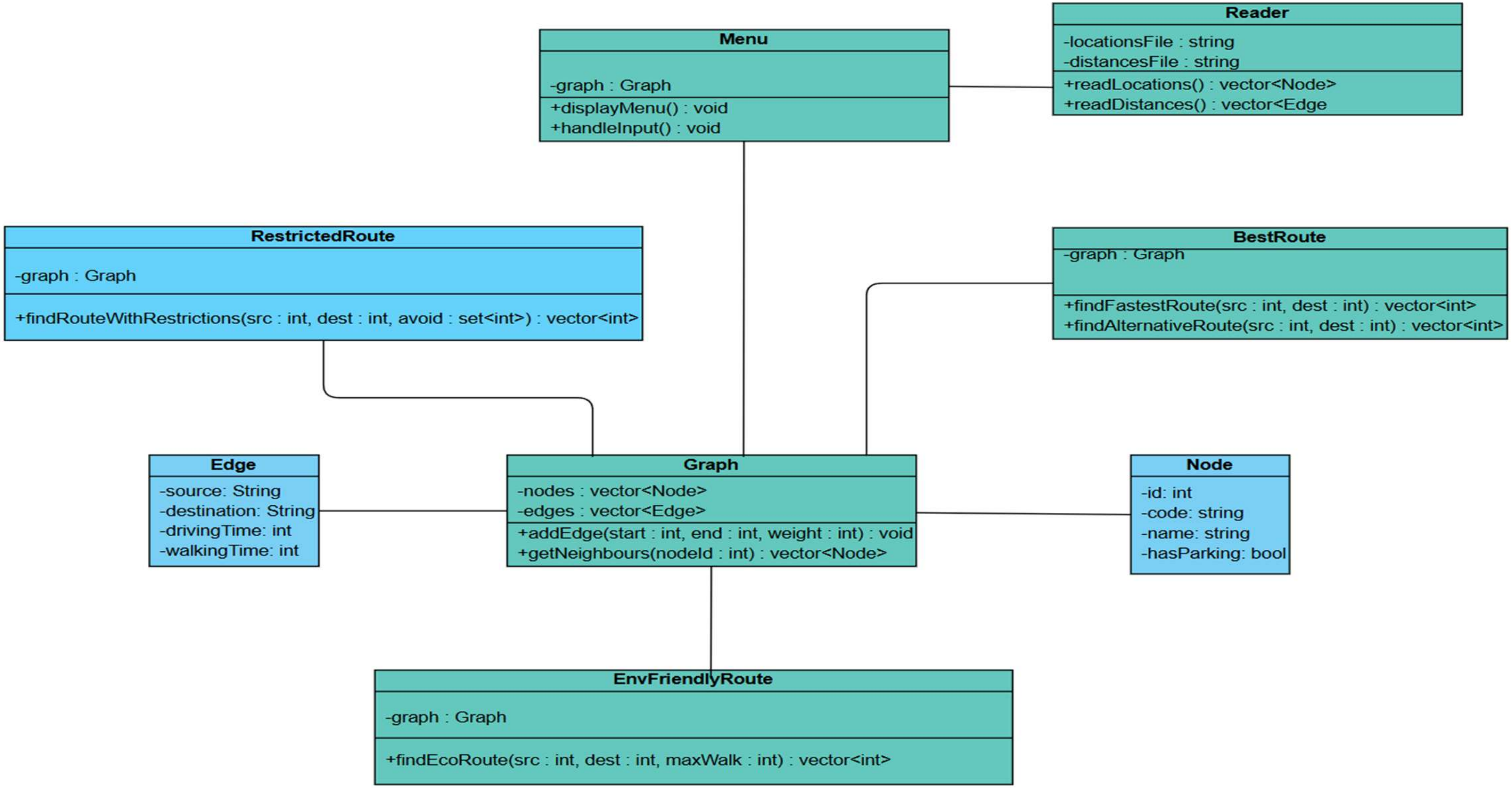
Individual Route Planning Tool

Design of Algorithms (DA) – spring 2025

Group members:
- Rui Teixeira (202103345)
- Domingos Neto (202108728)
- Emina Bašić (202412345)

Class diagram & File structure



Dataset Description

We are provided with two CSV files representing the structure of an urban environment:

- Locations.csv

  Contains information about each location (graph node), including:
  - Location (name)
  - Id (unique identifier)
  - Code (used for referencing)
  - Parking (1 if parking is available, 0 otherwise)
- Distances.csv

  Describes connections between locations (graph edges), including:
  - Location1, Location2 (node IDs)
  - Driving (time in minutes)
  - Walking (time in minutes)

Special case: If a segment cannot be driven, the Driving field contains an "X", which is handled by internally setting a large constant or marking the segment as non-drivable.

Location	Id	Code	Parking
LIDADOR /HOSPITAL	1	LD3372	1
SRA.CAMPANHÃ	2	SR2852	0

We use the Reader class to load and parse the dataset into appropriate data structures.

- readLocation()
  - Parses Locations.csv and creates a list of Node objects, each containing:
  - id, name, code, hasParking
- readDistances()
  - Parses Distances.csv and creates Edge objects with:
  - source, destination, drivingTime, walkingTimeThese edges are then added to the graph's adjacency list.

The parsed data is stored in the Graph class:

- vector<Node> nodes stores all locations
- unordered_map<string, vector<Edge>> adjacencyList stores connection

Location1	Location2	Driving	Walking
LD3372	QTI	3	17
LD3372	PR7649	4	24

Graph Representation of the Dataset

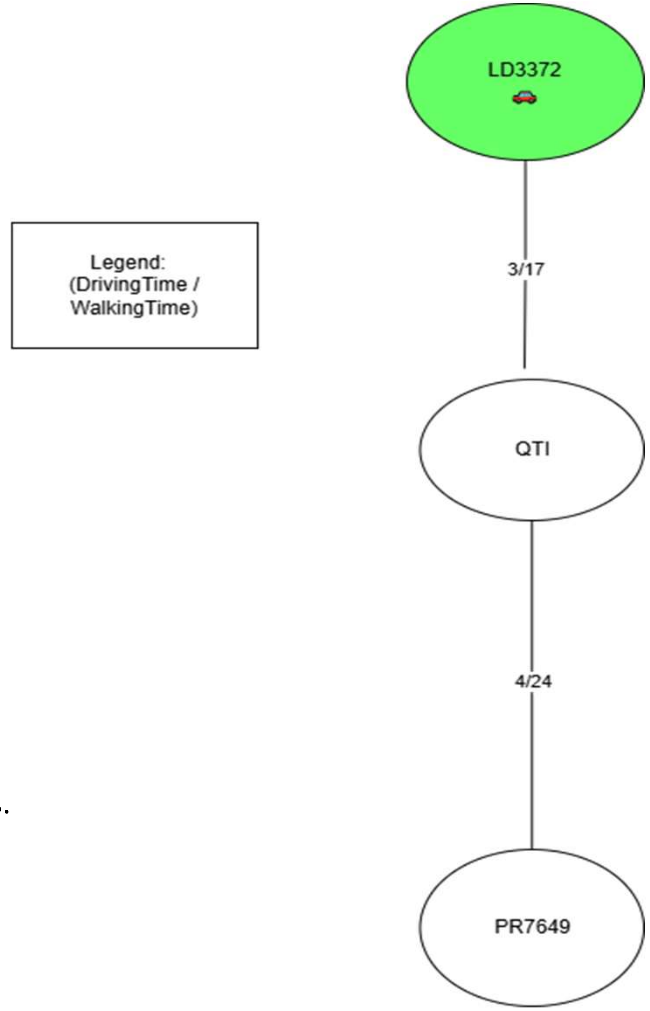
We used a weighted undirected graph to represent the city map.

- Each node is a location from the Locations.csv file.
  It contains:
  - Id, Code, Name, and Parking info.
- Each edge connects two nodes and comes from Distances.csv.
  It includes:
  - Driving time and Walking time (both in minutes).
    If driving is not allowed, the value is "X".

The graph is built in the Graph class using:

- vector<Node> nodes
- unordered_map<string, vector<Edge>> adjacencyList

We used this graph in all parts of the project. To find the shortest paths, we applied Dijkstra's algorithm, adapted for different scenarios: best route, restricted routes, and eco-friendly ones.



Overview of Functionalities

Functionality	Description
Best Route	Finds the fastest driving route between two locations.
Restricted Route	Avoids specific nodes/edges or forces a path through a chosen node.
Eco-Friendly Route	Combines driving and walking, respecting a max walking distance after parking.
Alternative Route	Finds a second-best route avoiding the previously used eco-friendly path.

All functionalities are implemented using variations of Dijkstra’s algorithm, adapted to different constraints.

Driving Only Route (Best Route)

This feature includes:

•the fastest driving route from a source to a destination

•an alternative independent route that shares no intermediate nodes or segments with the first one.

This provides a reliable “Plan B” in case the main path becomes unavailable.

Algorithm:

•First run: standard Dijkstra’s algorithm to get the best route

•Second run:

    •Remove or ignore all nodes/segments from the first route

    •Run Dijkstra again to find the independent alternative

Time complexity:

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

Driving Only Route (Restricted Route)

This functionality allows the user to plan a route while avoiding certain nodes or segments, or forcing the route through a specific node.

We handled three types of constraints:

•Avoid nodes

•Avoid segments

•Include a specific node

Algorithm:

We use Dijkstra's algorithm, but skip any forbidden nodes or edges during the path relaxation step.

•If a node or segment is marked as forbidden, it is removed from the graph or ignored during traversal.

•If a specific node must be included, we split the route into two Dijkstra runs:

- Source → Mandatory Node
- Mandatory Node → Destination

Time complexity:

1 or 2 × $O((V + E) \log V)$

Eco-Friendly Route

This feature finds a route that combines driving and walking, while respecting a maximum walking distance after parking.

How it works:

Run Dijkstra's Algorithm for Driving and Walking

- Compute the shortest driving routes from source to all reachable nodes (driveMap).
- Compute the shortest walking routes from destination to all nodes (walkMap), but in reverse.

Find Valid Parking Nodes

- Iterate through all reachable nodes in driveMap.
- Check if the node is a **valid parking spot**.
- Check if the node has a **walking path** to the destination within maxWalk.

Filter and Sort Routes

- If no valid routes exist, return a failure message.
- Otherwise, sort routes by **total travel time**, with a preference for routes requiring less walking.

Return the Best Route and Up to Two Alternatives

- The best route is the one with the shortest total travel time.
- If alternative routes exist, store up to **two additional options**.

Algorithm:

- Multiple runs of Dijsktra's algorithm
- One for each parking candidate

Time complexity: O(ElogV)

User interface

The application provides a text-based menu through the terminal (also a batch mode one).

The Menu class handles all interaction by displaying available options and reading user input.

Available options:

1 Driving Only

- **Best Route** → The fastest possible route based on driving time.
- **Restricted Route** → A route that avoids certain roads or areas due to restrictions.
- **Exit option**

2 Driving + Walking

- **Eco-Friendly Route** → Park at the best available location and walk the remaining distance.
- **Alternative Eco-Friendly Route** → A secondary walking + driving route with different parking options.
- **Exit option**

Each option prompts the user to enter relevant data:

- Source and destination
- Optional inputs: restricted nodes, max walking distance...

All input is read from the console, and the output (route and time) is displayed in the same interface.

Example of Use

Input/output (batch mode)

1. Best Route and Alternative Independent Route

Input:

```
Mode:driving
Source:3
Destination:8
```

Output:

```
Source:3
Destination:8
BestDrivingRoute:3,2,4,8(19)
AlternativeDrivingRoute:3,7,8(34)
```

7. Environmentally-Friendly Route Planning (driving and walking)

Input:

```
Mode:driving-walking
Source:8
Destination:5
MaxWalkTime:18
AvoidNodes:
AvoidSegments:
```

Output:

```
Source:8
Destination:5
DrivingRoute:8,4,2,3(19)
ParkingNode:3
WalkingRoute:3,5(10)
TotalTime:29
```

3. Restricted Route Planning - Excluding Nodes

Input:

```
Mode:driving
Source:5
Destination:4
AvoidNodes:2
AvoidSegments:
IncludeNode:
```

Output:

```
Source:5
Destination:4
RestrictedDrivingRoute:5,3,7,8,4(52)
```

9. Approximate Solution

Input:

```
Mode:driving-walking
Source:8
Destination:5
MaxWalkTime:5
AvoidNodes:
AvoidSegments:
```

Output:

```
Source:8
Destination:5
DrivingRoute1:8,4,2,3(19)
ParkingNode1:3
WalkingRoute1:3,5(10)
TotalTime1:29
DrivingRoute2:8,4,6(21)
ParkingNode2:6
WalkingRoute2:6,5(10)
TotalTime2:31
```

Highlighted Functionalities

- We’re proud of how the eco-friendly route balances driving and walking and then give us good alternatives.
- It respects a user-defined walking distance limit.
- We had to run Dijkstra multiple times and analyse several parking nodes.
- It simulates real-world behaviour and constraints.
- The restricted route allowed flexible planning with avoid/include options.
- Adapting Dijkstra for custom constraints made us think creatively.
- These features show a practical use of graph algorithms.

Main difficulties and participation of each group member

Difficulties

•Designing reusable classes that work across different functionalities.

•Handling "X" values in input and deciding how to treat non-drivable edges.

•Running Two Dijkstra Searches Efficiently.

•Handling Cases with No Valid Routes.

Participation

•Rui – 35 %

•Domingos – 40 %

•Emina - 25 %

THANK YOU!