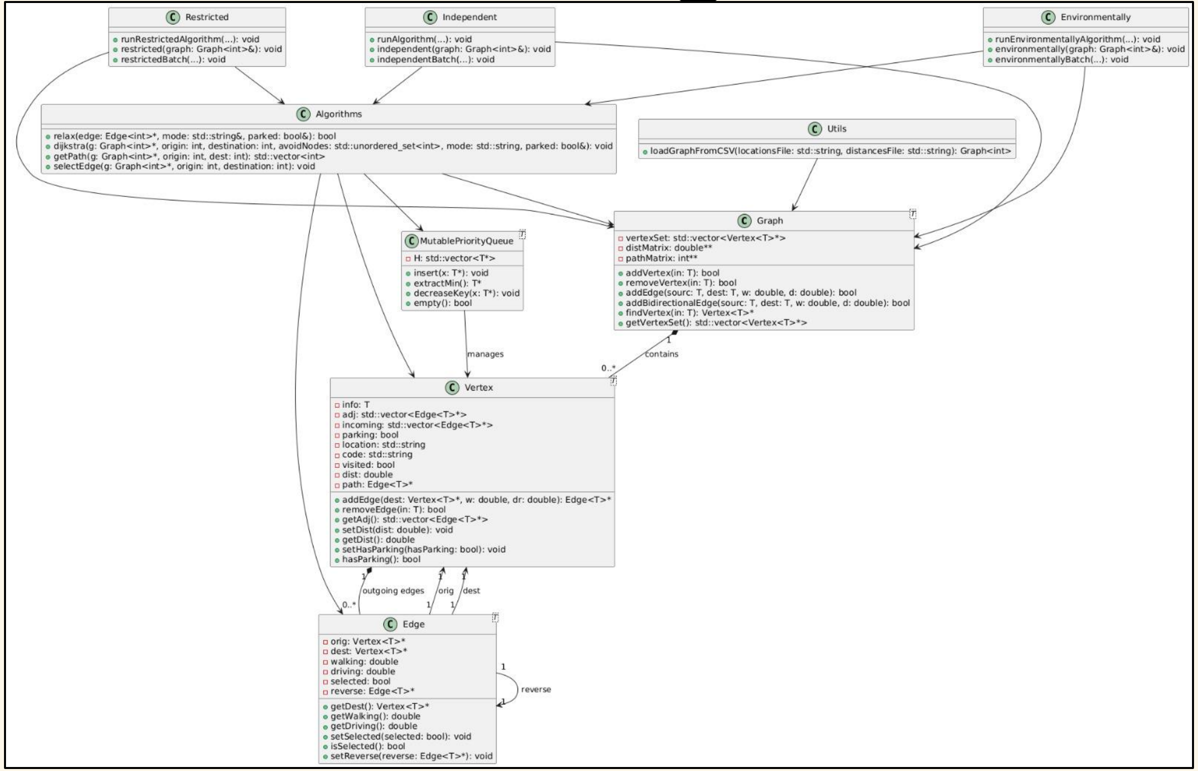
Individual Route Planning tool

Work Group

- Carlos André Gomes Cerqueira (up202305021)
- João Pedro Magalhães Marques (up202307612)
- João Pedro Nunes Ferreira (up202305204)

Class Diagram



Restricted

- runRestrictedAlgorithm(...): void
- restricted(graph: Graph<int>&): void
- restrictedBatch(...): void

Independent

- runAlgorithm(...): void
- independent(graph: Graph<int>&): void
- independentBatch(...): void

Environmentally

- runEnvironmentallyAlgorithm(...): void
- environmentally(graph: Graph<int>&): void
- environmentallyBatch(...): void

Algorithms

- relax(edge: Edge<int>*, mode: std::string&, parked: bool&): bool
- dijkstra(g: Graph<int>*, origin: int, destination: int, avoidNodes: std::unordered_set<int>, mode: std::string, parked: bool&): void
- getPath(g: Graph<int>*, origin: int, dest: int): std::vector<int>
- selectEdge(g: Graph<int>*, origin: int, destination: int): void

Utils

- loadGraphFromCSV(locationsFile: std::string, distancesFile: std::string): Graph<int>

Graph T

- vertexSet: std::vector<Vertex<T>*>
- distMatrix: double**
- pathMatrix: int**

- addVertex(in: T): bool
- removeVertex(in: T): bool
- addEdge(sourc: T, dest: T, w: double, d: double): bool
- addBidirectionalEdge(sourc: T, dest: T, w: double, d: double): bool
- findVertex(in: T): Vertex<T>*
- getVertexSet(): std::vector<Vertex<T>*>

MutablePriorityQueue T

- H: std::vector<T*>

- insert(x: T*): void
- extractMin(): T*
- decreaseKey(x: T*): void
- empty(): bool

Vertex T

- info: T
- adj: std::vector<Edge<T>*>
- incoming: std::vector<Edge<T>*>
- parking: bool
- location: std::string
- code: std::string
- visited: bool
- dist: double
- path: Edge<T>*

- addEdge(dest: Vertex<T>*, w: double, dr: double): Edge<T>*
- removeEdge(in: T): bool
- getAdj(): std::vector<Edge<T>*>
- setDist(dist: double): void
- getDist(): double
- setHasParking(hasParking: bool): void
- hasParking(): bool

Edge T

- orig: Vertex<T>*
- dest: Vertex<T>*
- walking: double
- driving: double
- selected: bool
- reverse: Edge<T>*

- getDest(): Vertex<T>*
- getWalking(): double
- getDriving(): double
- setSelected(selected: bool): void
- isSelected(): bool
- setReverse(reverse: Edge<T>*): void

Reading the dataset

Loading Locations from CSV

Opens and reads the Locations.csv file with the help of the header <fstream> and loads it onto a string object.

Extracts **location name, ID, code, and parking availability**. (One location per line info separated by “,”)

Stores the **code-to-ID mapping** in an unordered map for fast lookup.

Adds **vertices (nodes) to the graph**, setting their properties.

Reading the dataset

Loading Distances from CSV

Opens and reads the Distances.csv file with the help of the header <fstream> and loads it onto a string object.

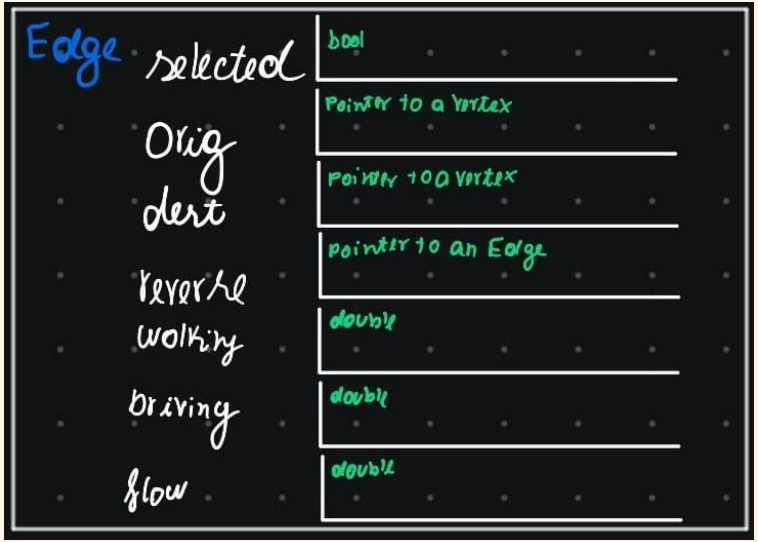
Extracts 2 **location codes** (start and finish), **driving time** (or -1 if unavailable), and **walking time** (always present).

Uses **code-to-ID mapping** to find the nodes IDs

Adds **edges** between locations, making the graph bidirectional.

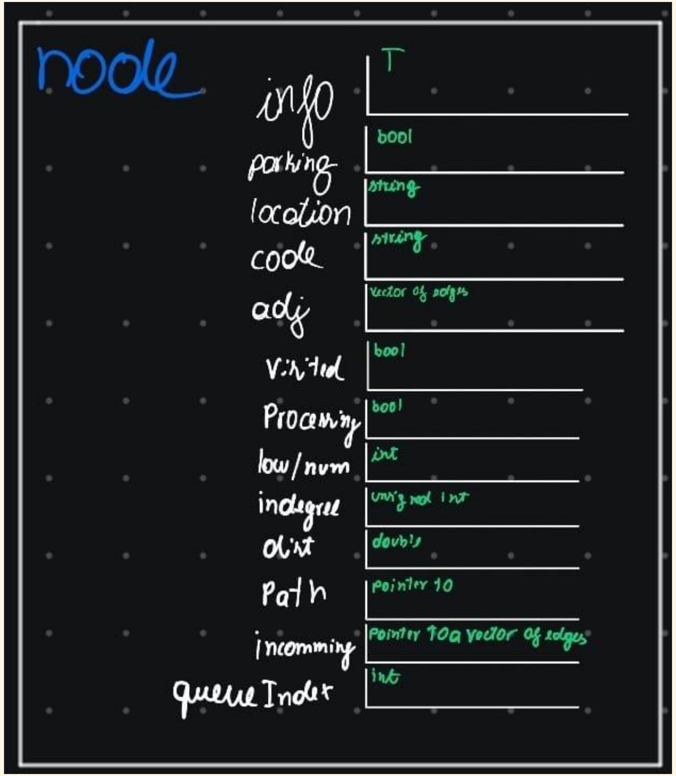
Nodes and Edges

Made some alterations to edge like eliminating getWeight() and adding instead getDriving() and getWalking()



Edge

selected	bool
Orig	Pointer to a Vertex
dest	Pointer to a vertex
reverse	pointer to an Edge
walking	double
Driving	double
flow	double

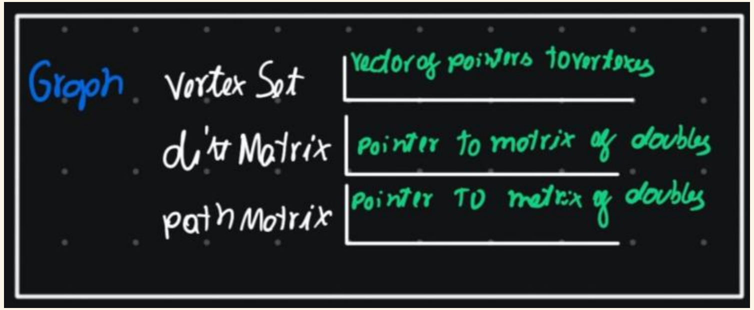


node

info	T
parking	bool
location	string
code	string
adj	vector of edges
visited	bool
Processing	bool
low/num	int
indegree	unsigned int
dist	double
Path	pointer to
incomming	Pointer to a vector of edges
queueIndex	int

Made some small alterations to node like adding:
setHasParking(),
hasParking(),
getCode(),
setCode(),
And others

Graphs

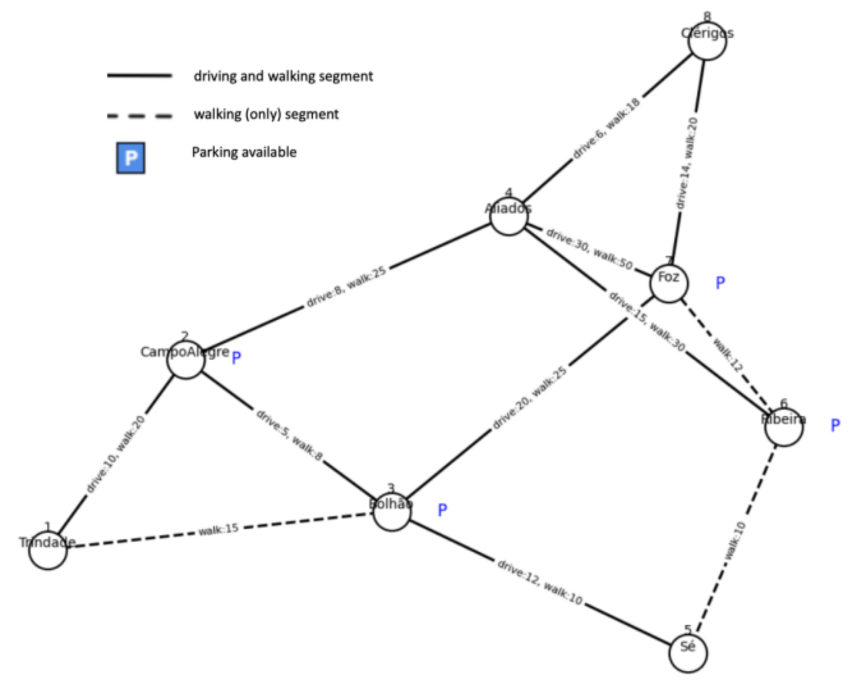


Graph

Vertex Set | Vectorof pointers tovertexes

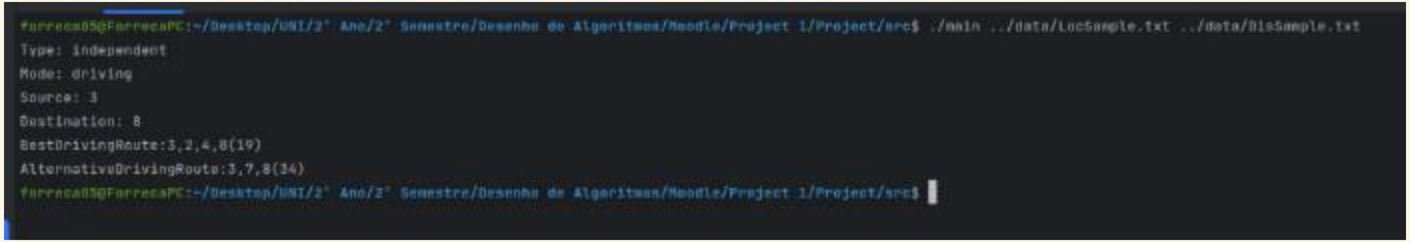
dist Matrix | Pointer to matrix of doubles

path Matrix | Pointer TO matrix of doubles



User interface

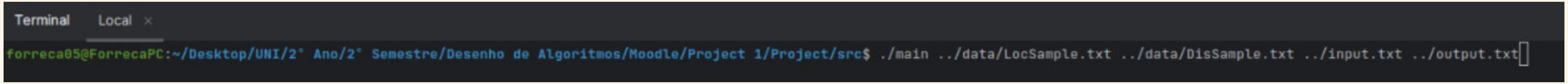
Terminal Mode



```
forreca05@ForrecaPC:~/Desktop/UNI/2° Ano/2° Semestre/Desenho de Algoritmos/Moodle/Project 1/Project/src$ ./main ../data/LocSample.txt ../data/DisSample.txt
Type: independent
Mode: driving
Source: 3
Destination: 8
BestDrivingRoute:3,2,4,8(19)
AlternativeDrivingRoute:3,7,8(34)
forreca05@ForrecaPC:~/Desktop/UNI/2° Ano/2° Semestre/Desenho de Algoritmos/Moodle/Project 1/Project/src$
```

User manually writes the data

Batch Mode



```
forreca05@ForrecaPC:~/Desktop/UNI/2° Ano/2° Semestre/Desenho de Algoritmos/Moodle/Project 1/Project/src$ ./main ../data/LocSample.txt ../data/DisSample.txt ../input.txt ../output.txt
```

Uses input and output files to process information in batch mode.

Participation

Our biggest challenge was implementing the environmentally friendly route, which had to include a parking spot along the way. However, the path to the parking spot was passing through the destination first, which was not making sense.

—André Cerqueira

Designed and optimized main pathfinding algorithms, structured and managed graph data, and incorporated auxiliary structures for efficient computation.

—João Ferreira

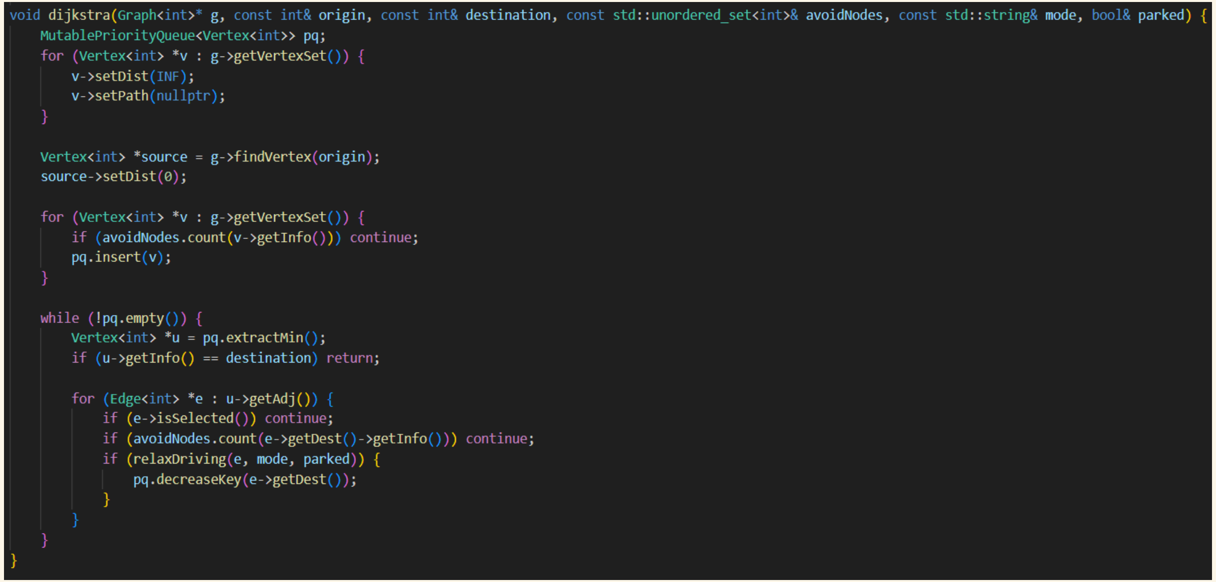
Developed core routing and pathfinding algorithms, implemented graph representation and manipulation, and integrated auxiliary data structures to optimize performance.

—João Marques

Handled data parsing, structured documentation, and presentation design, while assisting in the refinement of routing algorithms and graph architecture.

Highlight

The most rewarding part of our project was implementing a customized version of **Dijkstra's Algorithm** for efficient pathfinding. Our approach extended the classic algorithm by incorporating constraints such as avoiding specific nodes and segments, as well as accounting for different modes of transportation (driving and walking).



```cpp
void dijkstra(Graph<int>* g, const int& origin, const int& destination, const std::unordered_set<int>& avoidNodes, const std::string& mode, bool& parked) {
    MutablePriorityQueue<Vertex<int>> pq;
    for (Vertex<int> *v : g->getVertexSet()) {
        v->setDist(INF);
        v->setPath(nullptr);
    }

    Vertex<int> *source = g->findVertex(origin);
    source->setDist(0);

    for (Vertex<int> *v : g->getVertexSet()) {
        if (avoidNodes.count(v->getInfo())) continue;
        pq.insert(v);
    }

    while (!pq.empty()) {
        Vertex<int> *u = pq.extractMin();
        if (u->getInfo() == destination) return;

        for (Edge<int> *e : u->getAdj()) {
            if (e->isSelected()) continue;
            if (avoidNodes.count(e->getDest()->getInfo())) continue;
            if (relaxDriving(e, mode, parked)) {
                pq.decreaseKey(e->getDest());
            }
        }
    }
}
```

This allowed us to create flexible and robust routing system that adapts to real-world scenarios. It was especially interesting to work with algorithms we learned in class, applying them in a practical context to solve complex challenges. The experience of enhancing Dijkstra’s Algorithm and tailoring it to our needs was both challenging and rewarding.

Algorithms & Other Functionalities

Dijkstra

```
void dijkstra(Graph<int>*
g, const int& origin,
const int&
destination, const
std::unordered_set<in
t>& avoidNodes, const
std::string& mode,
bool& parked)
```

relax

```
bool relax(Edge<int>* edge,
const std::string&
mode, bool& parked)
```

getPath

```
std::vector<int>
getPath(Graph<int>* g, const
int& origin, const int& dest)
```

selectEdge

```
void selectEdge(Graph<int>*
g, const int& origin,
const int& dest)
```

Dijkstra

Objective.....

Find the shortest path between two nodes in a weighted graph

Complexity

O((V+E)log V)

Where v is the # of vertexes and e the # of edges

Steps

.

- •Initialize distances for all vertices as infinity.
- •Insert vertices into a priority queue.
- •Relax the neighbors of the vertex with the smallest distance.
- •Continue until the destination is found.

```
void dijkstra(Graph<int>*
g, const int& origin, const
int& destination, const
std::unordered_set<int>&
avoidNodes, const
std::string& mode, bool&
parked)
```

Key Feature: Can avoid specific nodes using `avoidNodes`.

Relax

Objective »»»

Update the shortest distance to reach a vertex.

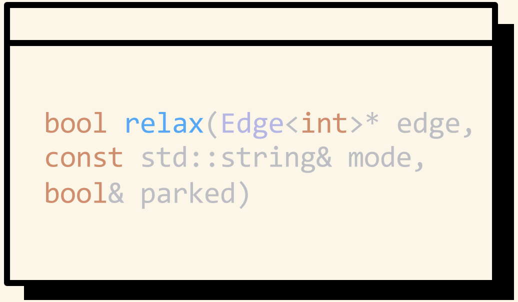
Complexity

O(1)

Constant time

How it Works:

- If a shorter path is found, the distance is updated.
- Different weights are used for walking and driving.
- If parking is unavailable, driving is ignored.



```
bool relax(Edge<int>* edge,
const std::string& mode,
bool& parked)
```

SelectEdge

Objective

Mark an edge as "selected" in the graph.

Complexity

O(V)

Where v is the # of vertexes

How it Works:

- Checks if the source and destination nodes exist.
- Iterates through the edges of the source node.
- Marks the corresponding edge as selected.

```
void selectEdge(Graph<int>*
g, const int& origin, const
int& dest)
```

Application: Can be used to highlight the optimal path found.

GetPath

Objective

Retrieve the sequence of nodes forming the shortest path.

Complexity

O(V)

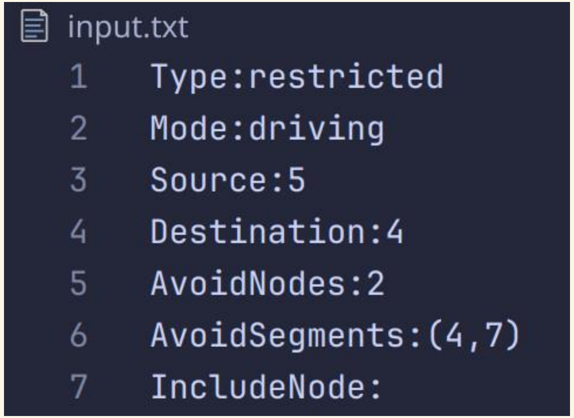
Where v is the # of vertexes

How it Works:

- Traverses nodes from destination to origin.
- If no path exists, returns an empty vector.
- Reverses the order of nodes to display the correct path.

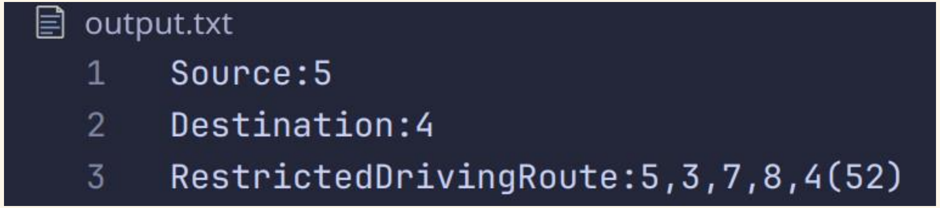
```
std::vector<int>
getPath(Graph<int>* g, const
int& origin, const int& dest)
```

Usage Examples



input.txt

```
Type:restricted
Mode:driving
Source:5
Destination:4
AvoidNodes:2
AvoidSegments:(4,7)
IncludeNode:
```

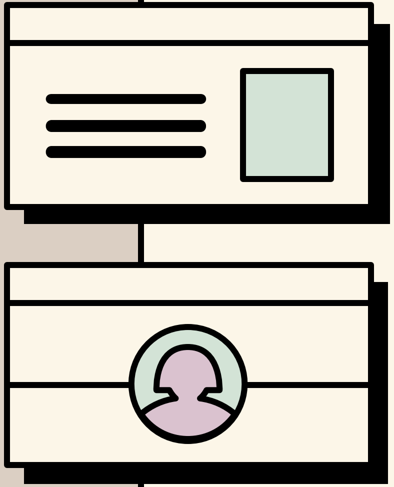


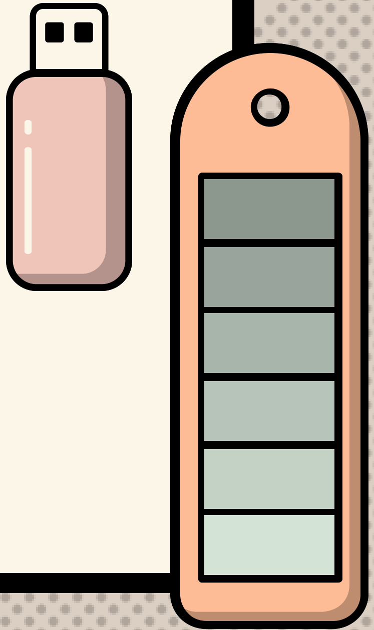
output.txt

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

Thanks!

Does anyone have any questions?





CREDITS: This presentation template was created by **Slidesgo**, including icons by **Flaticon**, and infographics & images by **Freepik**