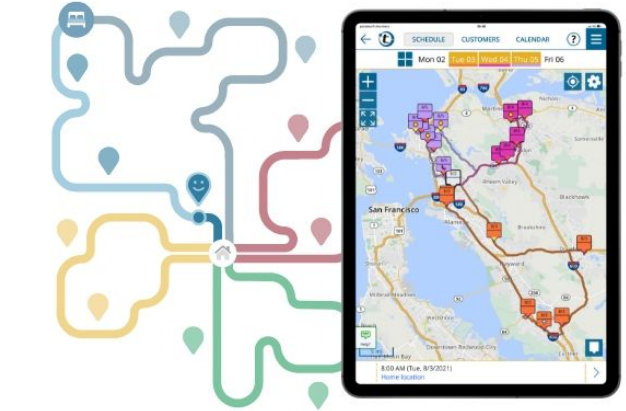
Ferramenta de Planeamento de Rotas Individuais - Desenho de Algoritmos

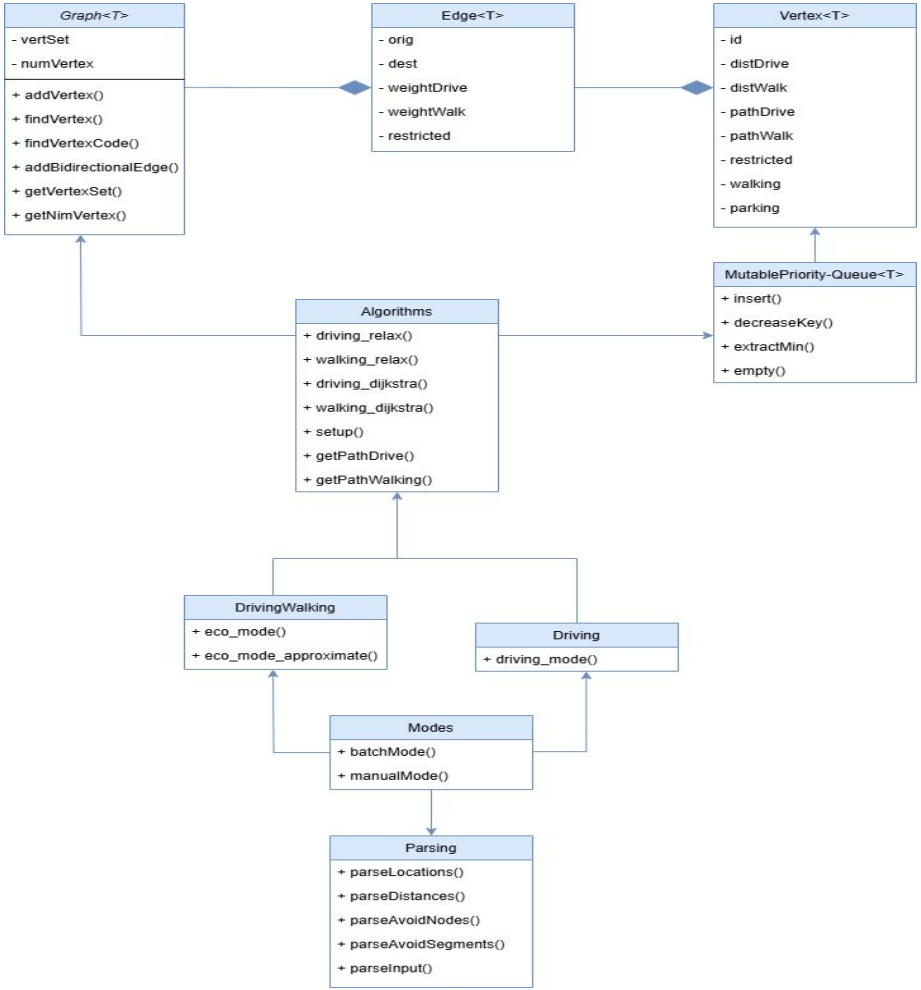


Grupo 7, Turma 1:
Raquel Fernandes - 202207134
Nicolas Ramos - 202304442
Hugo Alves - 202305395

Regência: Pedro Diniz

Docente: Iohan Soares

Diagrama de Classes



Descrição da Leitura do Conjunto de Dados Fornecido

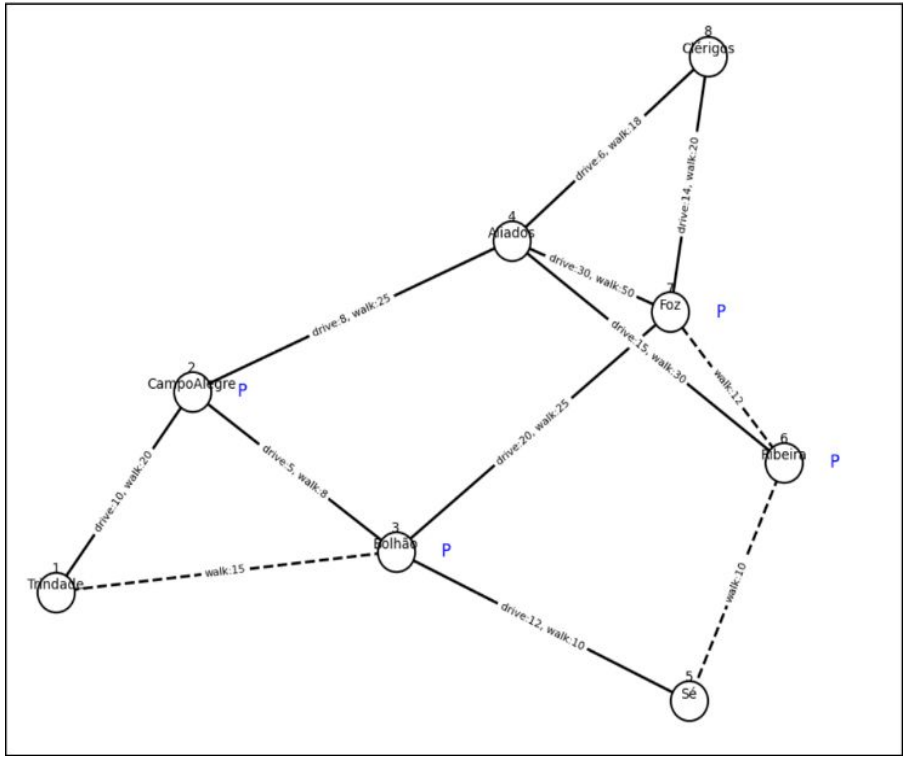
A Leitura dos dados procede da seguinte forma:

1. *Parsing* do ficheiro das localizações e a inserção dos nomes, ids, códigos e existência de parques no grafo;

2. *Parsing* do ficheiro das distâncias e a construção das arestas com os valores a pé e de condução associados aos respetivos vértices no grafo.

3. *Parsing* do ficheiro de *input*, deduzir o tipo de problema e devolver os dados.

Descrição do grafo utilizado para representar os conjuntos de dados



Funcionalidades Implementadas: Complexidade Temporal

```
void walking_dijkstra(Graph<int> *g, const int &origin) {

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

    MutablePriorityQueue<Vertex<int>> pq;

    for (auto s :Vertex<int> * :g->getVertexSet()) {
        s->setDistWalk( dist_walk: INF);
        s->setPathWalk( path: nullptr);
        pq.insert( x: s);
    }

    auto temp :Vertex<int> * = g->findVertex( id: origin);
    temp->setDistWalk( dist_walk: 0);
    pq.decreaseKey( x: temp);

    while (!pq.empty()) {
        auto v :Vertex<int> * = pq.extractMin();

        if (v->isRestricted()) continue;

        for (auto e :Edge<int> * :v->getAdj()) {

            if (e->isRestricted()) continue;

            if (walking_relax( edge: e)) pq.decreaseKey( x: e->getDest());
```

```
void driving_dijkstra(Graph<int> *g, const int &origin) {

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

    MutablePriorityQueue<Vertex<int>> pq;

    for (auto s :Vertex<int> * :g->getVertexSet()) {
        s->setDistDrive( dist_drive: INF);
        s->setPathDrive( path: nullptr);
        pq.insert( x: s);
    }

    auto temp :Vertex<int> * = g->findVertex( id: origin);
    temp->setDistDrive( dist_drive: 0);
    pq.decreaseKey( x: temp);

    while (!pq.empty()) {
        auto v :Vertex<int> * = pq.extractMin();

        if (v->isRestricted()) continue;

        for (auto e :Edge<int> * :v->getAdj()) {

            if (e->isRestricted()) continue;

            if (driving_relax( edge: e)) pq.decreaseKey( x: e->getDest());
```

O((V + E) log V)

A complexidade combinada é dominada por:

Inserções na fila: O(V log V)
Extrações do mínimo: O(V log V)
Atualizações (*decreaseKey*): O(E log V)

Funcionalidades Implementadas: Complexidade Temporal

```
void driving_mode(Graph<int> *g, const int &origin, const int &dest, const bool batch) {

    std::ofstream fout;
    std::ostream& out = batch ? (fout.open("../I-O/output.txt"), fout) : std::cout;

    out << "Source:" << origin << '\n';
    out << "Destination:" << dest << '\n';

    setup(g);

    driving_dijkstra(g, origin);
    auto path = getPathDrive(g, origin, dest);

    if (path.empty() || path[0]->getID() == dest) {
        out << "BestDrivingRoute:none\n";
        return;
    }

    out << "BestDrivingRoute:" << path[0]->getID();

    for (int i = 1; i < path.size(); i++) {
        out << ',' << path[i]->getID();
        path[i]->setRestricted(true);
    }

    path.back()->setRestricted(false);
    out << '(' << path.back()->getDistDrive() << ')' << '\n';

    driving_dijkstra(g, origin);
    path = getPathDrive(g, origin, dest);

    if (path.empty() || path[0]->getID() == dest) {
        out << "AlternativeDrivingRoute:none\n";
        return;
    }

    out << "AlternativeDrivingRoute:" << path[0]->getID();

    for (int i = 1; i < path.size(); i++) {
        out << ',' << path[i]->getID();
        path[i]->setRestricted(true);
    }

    path.back()->setRestricted(false);
    out << '(' << path.back()->getDistDrive() << ')' << '\n';
}
```

```
void driving_mode(Graph<int> *g, const int &origin, const int &dest, const vector<int> &avoid_nodes, const vector<pair<int,int>> &avoid_edges, const int &include_node, const bool batch) {

    std::ofstream fout;
    std::ostream& out = batch ? (fout.open("../I-O/output.txt"), fout) : std::cout;

    out << "Source:" << origin << '\n';
    out << "Destination:" << dest << '\n';

    setup(g);

    for (int an:avoid_nodes) {
        auto v = g->findVertex(an);
        v->setRestricted(true);
    }

    for (auto p:avoid_edges) {
        auto v = g->findVertex(p.first);
        for (auto e :v->getAdj()) {
            if (e->getDest()->getID() == p.second) {
                e->setRestricted(true);
                e->getReverse()->setRestricted(true);
            }
        }
    }

    if (include_node != -1) {
        vector<int> nodes = {origin, include_node, dest};
        string res = "RestrictedDrivingRoute:";
        double dist = 0;

        for (int i = 0; i < 2; i++) {
            driving_dijkstra(g, nodes[i]);
            auto path = getPathDrive(g, origin: nodes[i], dest: nodes[i+1]);

            if (path.empty() || path[0]->getID() == dest) {
                out << "RestrictedDrivingRoute:none\n";
                return;
            }

            if (path[0]->getID() != include_node)
                res += to_string(val: path[0]->getID());

            for (int j = 1; j < path.size(); j++)
                res += "," + to_string(val: path[j]->getID());


            dist += path.back()->getDistDrive();
        }

        out << res << '(' << dist << ')' << '\n';
    }

    else {

        driving_dijkstra(g, origin);
        auto path = getPathDrive(g, origin, dest);

        if (path.empty() || path[0]->getID() == dest) {
            out << "RestrictedDrivingRoute:none\n";
            return;
        }

        out << "RestrictedDrivingRoute:" << path[0]->getID();

        for (int i = 1; i < path.size(); i++) {
            out << ',' << path[i]->getID();
        }

        out << '(' << path.back()->getDistDrive() << ')' << '\n';
    }
}
```

O(VE)

A complexidade combinada é dominada por:

Aplicação de restrições: O(VE)
Dijkstras: O((V + E) log V)
Obter os caminhos: O(V)

Funcionalidades Implementadas: Complexidade Temporal

eco_mode

⬇

O(VE)

A complexidade combinada é:

Restrições: O(VE)
Dijkstras: O((V + E) log V)
Junção das soluções: O(V)
Obter os caminhos: O(V)

eco_mode_approximate

⬇

O(E(V + E) log V)

A complexidade combinada é:

Restrições: O(VE)
Dijkstras: O((V + E) log V)
Junção das soluções: O(V)
Obter os caminhos: O(V)
Obter caminhos alternativos: O(E(V + E) log V)

Descrição da Interface do Utilizador

```
[hribalves@hugo-ms7d98 cmake-build-debug]$ ./DA_PROJ1
Location of the locations file: ../data/Locations.csv
Location of the distances file: ../data/Distances.csv
1. Driving Mode
2. Eco Mode
1
Source: 2
Destination: 800
Any nodes to include? (Write -1 if not; Write node id if yes)
4
Avoid nodes? (Y/n)
y
How many?
2
Which ones?
5 900
Avoid segments? (Y/n)
y
How many?
1
Which ones? (Format: node1 node2)
1 2
Source:2
Destination:800
RestrictedDrivingRoute:2,945,216,16,10,1060,1240,17,4,1232,933,327,550,463,358,142,684,800(101)
```

```
[hribalves@hugo-ms7d98 cmake-build-debug]$ ./DA_PROJ1
Location of the locations file: ../data/Locations.csv
Location of the distances file: ../data/Distances.csv
1. Driving Mode
2. Eco Mode
2
Source: 2
Destination: 1000
What's the maximum time you want to spend walking?
200
Avoid nodes? (Y/n)
y
How many?
3
Which one(s)?
1
2
Invalid node! Try another one.
5
6
Avoid segments? (Y/n)
y
How many?
2
Which one(s)? (Format: node1 node2)
900 702
800 701
Source:2
Destination:1000
DrivingRoute:2,420,428,448,1142(12)
ParkingNode:1142
WalkingRoute:1142,688,334,331,514,511,459,1000(90)
TotalTime:102
```

Principais Dificuldades

- [T3.2] Solução Aproximada (*eco_mode_approximate*): Pesquisa de caminhos alternativos, em particular, quando são caminhos similares.