STCPlan

Criação de uma aplicação para planeamento de viagens na linha STCP

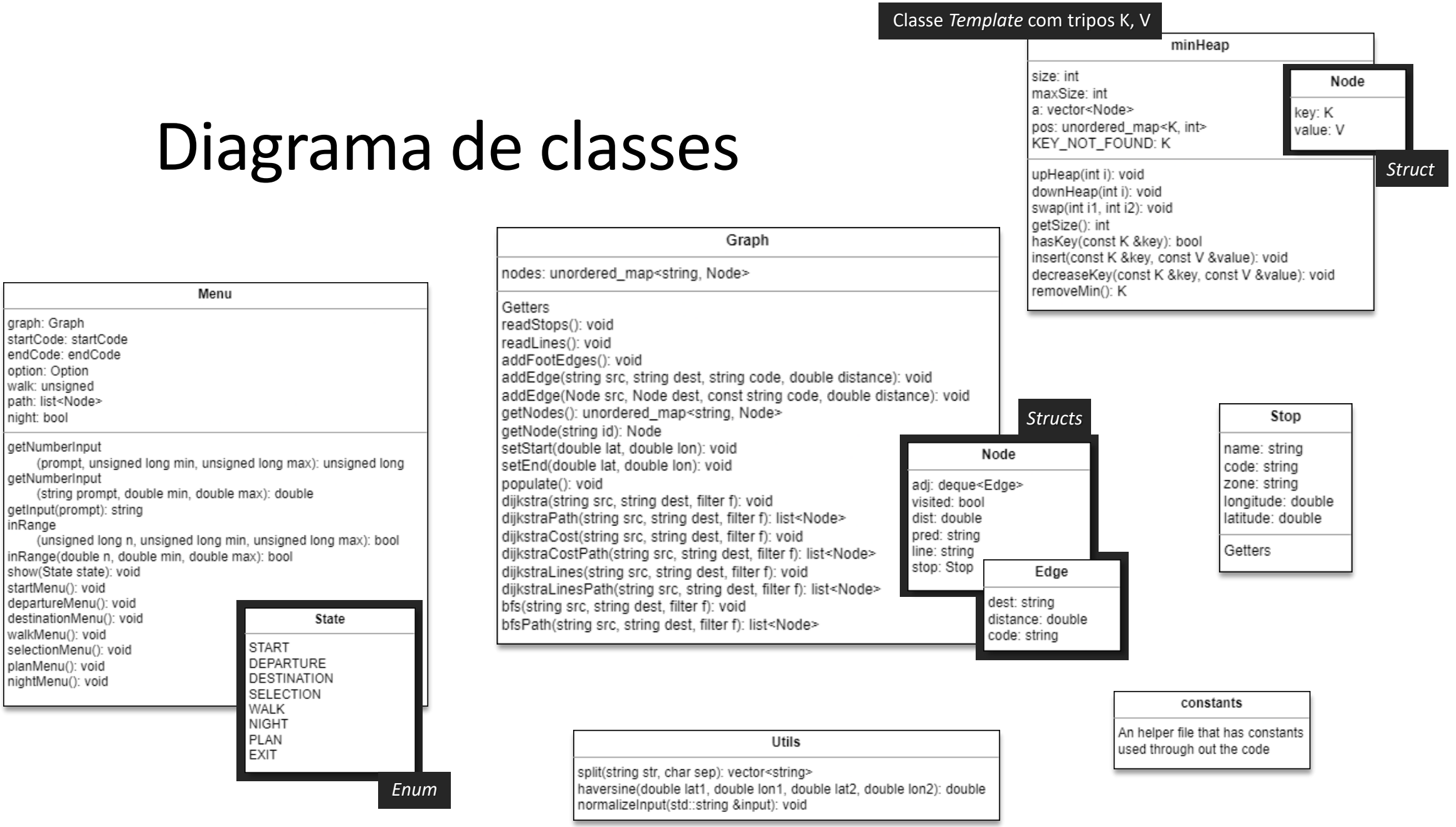
António Oliveira Santos – up202008004

João António Semedo Pereira – up202007145

Pedro Alexandre Ferreira e Silva – up202004985

Índice

Diagrama de classes



Leitura

A leitura dos ficheiros é feita na inicialização da aplicação, deste modo, é evitado abrir e fechar ficheiros "*on the fly*", o que melhora questões de complexidade temporal.

São lidos os ficheiros .csv:

stops.csv

lines_[LINECODE]_[DIR].csv

E a sua informação guardada nas estruturas de dados adequadas (grafo)

O ficheiro *lines.csv* não é lido, uma vez que, a informação que carrega pode ser adquirida de outros modos, poupando o tempo da abertura de mais um ficheiro.

É de realçar que foram adicionadas entradas no ficheiro stops.csv com informação relativa às estações do Metro do Porto.

De modo semelhante foram adicionados ficheiros com o formato *lines_[LINECODE]_[DIR].csv* com informação das rotas das linhas constituintes do Metro.

Isto foi feito para evitar criar um novo ficheiro que também teria que ser lido individualmente, aumentando o tempo de arranque da aplicação.

Grafos

Para este projeto decidimos utilizar um único grafo no qual toda a informação se encontra armazenada, quer de rotas e paragens da STCP como do Metro do Porto.

O ponto mais fraco desta implementação é a sua complexidade espacial, não é de todo eficiente em termos de espaço armazenar tanta informação (mais de 6M de *edges!*).

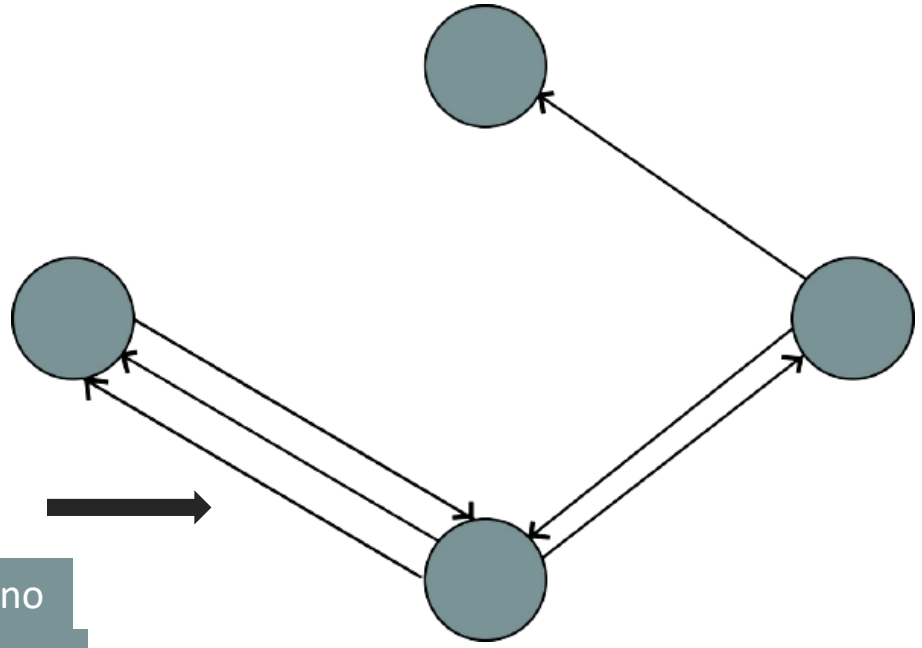
Para além disto, o grafo também armazena todas as ligações a pé possíveis entre *nodes*.

No entanto, o ponto mais forte deste grafo revela-se na sua utilização.

O facto de toda esta informação estar perenemente guardada no grafo garante que não são necessários cálculos complexos durante a execução do programa!

Por sua vez, isto permite reduzir o tempo de apresentar a rota ao utilizador consideravelmente, sendo apenas relevantes as complexidades temporais dos algoritmos utilizados.

Grafos



Cada *node* tem:

- Um *deque* com as linhas que partem dele
- Uma *Stop*

Cada *edge* tem:

- Código do *node* de destino
- Distância ao *node* de destino
- Código da linha

Ao correr um algoritmo:

- O nó que lhe precede
- A distância ao nó que lhe precede
- A linha que está a ser percorrida
- Um estado de “visita”

É de realçar que as *Stops* têm informação pertinente a coordenadas, código, nome e zona daquilo que seria uma paragem na realidade

Funcionalidades

Origem/Destino

Existem 2 modos de indicar o local de início e fim da viagem:

Coordenadas

Neste caso é calculada a paragem mais próxima das coordenadas dadas, tendo em conta a distância que o utilizador pretende andar.

Código da paragem

Esta situação não necessita de cálculos auxiliares, uma vez que já temos toda a informação necessária para traçar o percurso.

Estas opções podem ser escolhidas independentes uma da outra, ou seja, é possível utilizar o código de uma paragem para o ponto de partida e coordenadas para o destino ponto final.

Funcionalidades

"Melhor caminho"

Existem 4 definições possíveis de "melhor caminho" que podem ser selecionadas pelo utilizador:

Menor número de paragens

```
void Graph::bfs(const std::string &src, const std::string &dest,
                const filter &f) {
    for (auto i{nodes.begin()}, end{nodes.end()}; i != end; ++i)
        i->second.visited = false;

    std::queue<std::string> q; // queue of unvisited nodes
    q.push(src);
    nodes[src].visited = true;
    while (!q.empty()) { // while there are still unvisited nodes
        std::string u = q.front();
        q.pop();

        if (u == dest)
            return;

        for (auto e : nodes[u].adj) {
            std::string w = e.dest;

            if (!f(nodes[u], nodes[w], e))
                continue;

            if (!nodes[w].visited) {
                q.push(w);
                nodes[w].line = e.code;
                nodes[w].visited = true;
                nodes[w].pred = u;
            }
        }
    }
}
```

O(V + E)

BFS

```
auto Graph::bfsPath(const std::string &src, const std::string &dest,
                    const filter &f) -> std::list<Node> {
    bfs(src, dest, f);

    std::list<Node> path{};

    path.push_front(getNode(dest));
    std::string v = dest;
    while (v != src) {
        v = nodes[v].pred;
        path.push_front(getNode(v));
    }

    return path;
}
```

Menor distância total

```
void Graph::dijkstra(const std::string &src, const std::string &dest,
                     const filter &f) {
    MinHeap<std::string, double> q(nodes.size(), "");

    for (auto i{nodes.begin()}, end{nodes.end()}; i != end; ++i) {
        i->second.dist = INF;
        q.insert(i->first, INF);
        i->second.visited = false;
    }

    nodes[src].dist = 0;
    q.decreaseKey(src, 0);
    nodes[src].pred = src;

    while (q.getSize() > 0) {
        std::string uc = q.removeMin();
        Node &u = getNode(uc);
        u.visited = true;

        if (uc == dest)
            return;

        for (auto e : u.adj) {
            std::string vc = e.dest;
            Node &v = getNode(vc);
            double w = u.dist + e.distance;

            if (!f(u, v, e))
                continue;

            if (!v.visited && w < v.dist) {
                v.line = e.code;
                v.dist = w;
                q.decreaseKey(vc, v.dist);
                v.pred = uc;
            }
        }
    }
}
```

O(V log(E))

Dijkstra

```
std::list<Node> Graph::dijkstraPath(const std::string &src,
                                    const std::string &dest, const filter &f) {
    dijkstra(src, dest, f);

    std::list<Node> path{};
    if (nodes[dest].dist == INF)
        return path;

    path.push_back(getNode(dest));
    std::string v = dest;
    while (v != src) {
        v = nodes[v].pred;
        path.push_front(getNode(v));
    }

    return path;
}
```

Funcionalidades

"Melhor caminho"

Menor custo (menos zonas)

O(V log(E))

Dijkstra

```
void Graph::dijkstraCost(const std::string &src, const std::string &dest,
                         const filter &f) {
    MinHeap<std::string, double> q(nodes.size(), "");

    for (auto i{nodes.begin()}, end{nodes.end()}; i != end; ++i) {
        i->second.dist = INF;
        q.insert(i->first, INF);
        i->second.visited = false;
    }

    nodes[src].dist = 0;
    q.decreaseKey(src, 0);
    nodes[src].pred = src;

    while (q.getSize() > 0) {
        std::string uc = q.removeMin();
        Node &u = getNode(uc);
        u.visited = true;
        for (auto e : u.adj) {
            std::string vc = e.dest;
            Node &v = getNode(vc);

            if (!f(u, v, e)) // descriptive variables :D
                continue;

            double w = u.dist + e.distance +
                       (v.stop.getZone() != u.stop.getZone()) * 10;

            if (!v.visited && w < v.dist) {
                v.line = e.code;
                v.dist = w;
                q.decreaseKey(vc, v.dist);
                v.pred = uc;
            }
        }
    }
}
```

```
std::list<Node> Graph::dijkstraCostPath(const std::string &src,
                                        const std::string &dest,
                                        const filter &f) {
    dijkstraCost(src, dest, f);

    std::list<Node> path{};
    if (nodes[dest].dist == INF)
        return path;

    path.push_back(getNode(dest));
    std::string v = dest;
    while (v != src) {
        v = nodes[v].pred;
        path.push_front(getNode(v));
    }

    return path;
}
```

Menos mudanças de linha

O(V log(E))

Dijkstra

```
void Graph::dijkstraLines(const std::string &src, const std::string &dest,
                          const filter &f) {
    MinHeap<std::string, double> q(nodes.size(), "");

    for (auto i{nodes.begin()}, end{nodes.end()}; i != end; ++i) {
        i->second.dist = INF;
        q.insert(i->first, INF);
        i->second.visited = false;
    }

    nodes[src].dist = 0;
    q.decreaseKey(src, 0);
    nodes[src].pred = src;

    while (q.getSize() > 0) {
        std::string uc = q.removeMin();
        Node &u = getNode(uc);
        u.visited = true;

        for (auto e : u.adj) {

            std::string vc = e.dest;
            Node &v = getNode(vc);

            if (!f(u, v, e))
                continue;

            double w = u.dist + e.distance + (v.line != u.line) * 1000;

            if (!v.visited && w < v.dist) {
                v.line = e.code;
                v.dist = w;
                q.decreaseKey(vc, v.dist);
                v.pred = uc;
            }
        }
    }
}
```

```
std::list<Node> Graph::dijkstraLinesPath(const std::string &src,
                                         const std::string &dest,
                                         const filter &f) {
    dijkstraLines(src, dest, f);

    std::list<Node> path{};
    if (nodes[dest].dist == INF)
        return path;

    path.push_back(getNode(dest));
    std::string v = dest;
    while (v != src) {
        v = nodes[v].pred;
        path.push_front(getNode(v));
    }

    return path;
}
```

Funcionalidades

Mudanças de autocarro

As mudanças de autocarro/metro podem ser feitas dos seguintes modos:

Mesma paragem -> Linha diferente

A pé

Como foi mencionado anteriormente, as mudanças de transporte a pé são todas calculadas para todos os nós possíveis.

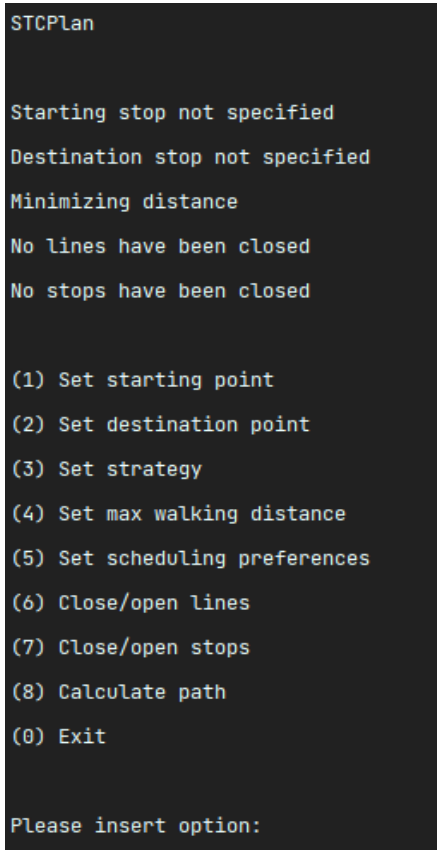
Na prática isto permite-nos manter o código da pesquisa simples e extremamente rápido.

Interface

A interface consiste num menu principal a partir de onde o utilizador iniciar o planeamento da sua viagem.

O planeamento consiste em:

- Escolher local de partida
- Escolher destino
- Escolher a estratégia a utilizar
- Distância máxima para andar a pé
- Escolher preferências de horário
- Fechar paragens
- Abrir paragens
- Calcular o caminho



Funcionalidades a realçar

Filtragem

Esta funcionalidade encontra-se implementada de uma forma elegante e modular de tal modo que filtros completamente novos podem ser adicionados eficientemente.

Esta funcionalidade possui, atualmente as seguintes iterações:

- Filtro de distância para andar a pé
- Filtro para ignorar linhas
- Filtro para ignorar paragens
- Filtro de horário noturno/diurno

Metro

O Metro do Porto encontra-se integrado no nosso grafo o que permite uma maior aproximação à realidade do que seria delinear um percurso utilizando os transportes públicos da cidade do Porto.

Dificuldades

Não foi fácil desenvolver os algoritmos para certas estratégias de criação de caminhos, nomeadamente o de menor número de mudança de linhas.

Ao longo do projeto, reformulámos algumas vezes a nossa solução para diversas situações, por exemplo, o armazenamento e utilização das escolhas do utilizador quanto à viagem: acabámos por criar uma classe Trip que serve de intermediário entre o input do utilizador e a execução do algoritmo para gerar o caminho.

António Santos	João Pereira	Pedro Silva
32/100	40/100	28/100

Observações

Para operações como limpar o ecrã estamos a utilizar escape sequences pelo que certos terminais poderão não mostrar a output da forma correta

Existe uma versão sem *ANSI escape sequences* para terminais que não suportam

Para os melhores resultados utilizar Windows PowerShell