# 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 Grafos Funcionalidades Interface Funcionalidades a realçar Dificuldades

#### Classe Template com tripos K, V

size: int

maxSize: int a: vector<Node>

pos: unordered\_map<K, int>

KEY\_NOT\_FOUND: K

upHeap(int i): void downHeap(int i): void

minHeap

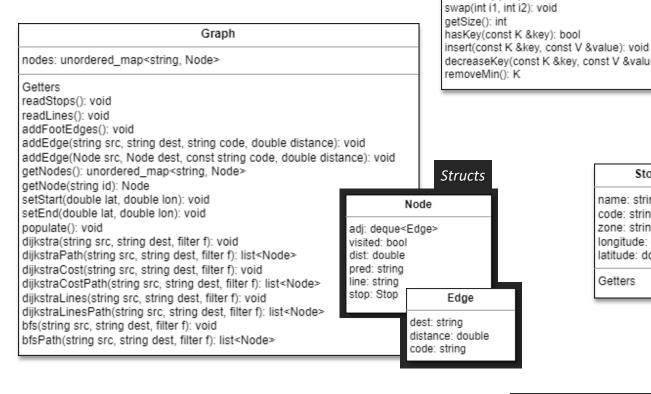
constants

An helper file that has constants

used through out the code

## Diagrama de classes

#### Menu graph: Graph startCode: startCode endCode: endCode option: Option walk: unsigned path: list<Node> night: bool aetNumberInput (prompt, unsigned long min, unsigned long max): unsigned long getNumberInput (string prompt, double min, double max): double getInput(prompt): string (unsigned long n, unsigned long min, unsigned long max); bool inRange(double n, double min, double max); bool show(State state): void startMenu(): void departureMenu(): void destinationMenu(): void State walkMenu(): void START selectionMenu(): void DEPARTURE planMenu(): void DESTINATION nightMenu(): void SELECTION WALK NIGHT PLAN EXIT Enum



decreaseKey(const K &key, const V &value): void Stop name: string code: string zone: string Ionaitude: double latitude: double Getters

Node

Struct

kev: K

value: V

Utils

split(string str. char sep): vector<string> haversine(double lat1, double lon1, double lat2, double lon2); double normalizeInput(std::string &input): void

### 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!*).

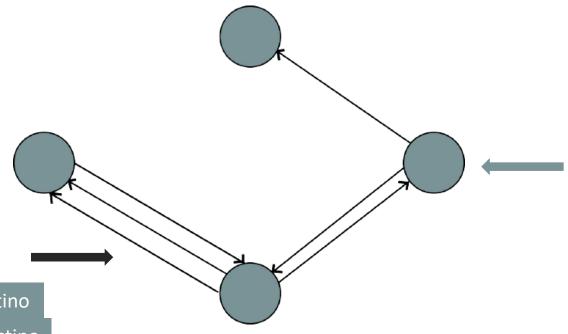
Para além disto, o grafo também armazena todas as ligações a pé possíveis entre *nodes*.

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 *edge* tem:

Código do *node* de destino

Distância ao *node* de destino

Código da linha

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

Cada *node* tem:

Um *deque* com as linhas que partem dele

Uma *Stop* 

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"

Origem/Destino

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

Coordenadas

Código da paragem

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

"Melhor caminho"

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

Graph::dijkstra(const std::string &src, const std::string &dest,

#### 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();
                                                                  O(V + E)
                                                                                                                  BFS
      if (u == dest)
                                                     auto Graph::bfsPath(const std::string &src, const std::string &dest,
                                                                        const filter &f) -> std::list<Node> {
      for (auto e : nodes[u].adj) {
                                                         bfs(src, dest, f);
          std::string w = e.dest;
          if (!f(nodes[u], nodes[w], e))
                                                         std::list<Node> path{};
                                                         path.push_front(getNode(dest));
          if (!nodes[w].visited) {
                                                         std::string v = dest;
              q.push(w);
                                                         while (v != src) {
              nodes[w].line = e.code;
                                                             v = nodes[v].pred;
              nodes[w].visited = true;
                                                             path.push_front(getNode(v));
              nodes[w].pred = u;
                                                         return path;
```

#### Menor distância total

```
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;
nodes[src].pred = src;
                                                                    O(V log(E))
                                                                                                                                      Dijkstra
while (q.getSize() > 0) {
   std::string uc = q.removeMin();
   Node &u = getNode(uc);
   u.visited = true;
                                                                  std::list<Node> Graph::dijkstraPath(const std::string &src,
                                                                                                         const std::string &dest, const filter &f)
   if (uc == dest)
                                                                      dijkstra(src, dest, f);
   for (auto e : u.adj) {
                                                                     std::list<Node> path{};
                                                                     if (nodes[dest].dist == INF)
       Node &v = getNode(vc);
       double w = u.dist + e.distance:
                                                                          return path;
                                                                      path.push_back(getNode(dest));
                                                                      std::string v = dest;
                                                                      while (v != src) {
       if (!v.visited && w < v.dist) {
          v.line = e.code;
                                                                          v = nodes[v].pred;
                                                                          path.push_front(getNode(v));
          q.decreaseKey(vc, v.dist);
          v.pred = uc:
```

"Melhor caminho"

Menor custo (menos zonas)

```
id 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;
                                                                    O(V log(E))
                                                                                                                     Dijkstra
while (q.getSize() > 0) {
    std::string uc = q.removeMin();
    Node &u = getNode(uc);
                                                              std::list<Node> Graph::dijkstraCostPath(const std::string &src,
    u.visited = true;
                                                                                                         const std::string &dest,
    for (auto e : u.adi) {
                                                                                                         const filter &f) {
        std::string vc = e.dest;
                                                                   dijkstraCost(src, dest, f);
        Node &v = getNode(vc);
        if (!f(u, v, e)) // descriptive variables :D
                                                                   std::list<Node> path{};
                                                                   if (nodes[dest].dist == INF)
                                                                       return path;
        double w = u.dist + e.distance +
                 (v.stop.getZone() != u.stop.getZone()) * 10;
                                                                   path.push_back(getNode(dest));
        if (!v.visited && w < v.dist) {
                                                                   std::string v = dest;
                                                                   while (v != src) {
           v.dist = w;
                                                                       v = nodes[v].pred;
                                                                       path.push_front(getNode(v));
                                                                   return path;
```

#### Menos mudanças de linha

```
oid 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;
  q.decreaseKey(src, 0);
                                                                     O(V \log(E))
  nodes[src].pred = src;
                                                                                                                            Dijkstra
  while (q.getSize() > 0) {
     std::string uc = q.removeMin();
    Node &u = getNode(uc);
                                                                     std::list<Node> Graph::dijkstraLinesPath(const std::string &src,
                                                                                                                  const std::string &dest,
                                                                                                                  const filter &f) {
                                                                         dijkstraLines(src, dest, f);
        std::string vc = e.dest;
        Node &v = getNode(vc);
                                                                         std::list<Node> path{};
                                                                         if (nodes[dest].dist == INF)
                                                                             return path;
        double w = u.dist + e.distance + (v.line != u.line) * 1000;
                                                                         path.push_back(getNode(dest));
                                                                         std::string v = dest;
        if (!v.visited && w < v.dist) {
                                                                         while (v != src) {
           v.line = e.code:
                                                                             v = nodes[v].pred;
                                                                              path.push_front(getNode(v));
           q.decreaseKey(vc, v.dist);
           v.pred = uc;
```

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

#### STCPlan

Starting stop not specified

Destination stop not specified

Minimizing distance

No lines have been closed

No stops have been closed

- (1) Set starting point
- (2) Set destination point
- (3) Set strategy
- (4) Set max walking distance
- (5) Set scheduling preferences
- (6) Close/open lines
- (7) Close/open stops
- (8) Calculate path
- (0) Exit

Please insert option:

## 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