U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Trip Planner

Trabalho realizador por:

- Miguel Alexandre Brandão Teixeira (up201605150@fe.up.pt)
- Pedro Xavier T. M. C. Pinho (up201605166@fe.up.pt)

10 de abril

1. Descrição do tema:

Uma das preocupações mais urgentes para as chamadas "smart cities" é o desenvolvimento de uma política de utilização de transportes públicos e coletivos em detrimento do veículo particular. Entretanto, muitas vezes isto implicará a utilização de vários serviços, por exemplo, para uma pessoa se deslocar de casa até ao seu local de trabalho. Apesar dos operadores de transportes públicos coletivos muitas vezes disponibilizarem informação relativa aos seus serviços, frequências e respetivas rotas, é deixado ao utente a tarefa de identificar as melhores combinações de transportes e linhas que o poderão levar desde a sua origem até ao seu destino, pelo itinerário de menor custo.

Com este trabalho pretendeu-se elaborar um planeador de itinerário multimodal, capaz de identificar a melhor combinação de modos de transporte para se realizar uma viagem desde um ponto de origem até um ponto de destino.

O melhor caminho deve ser avaliado de forma diferente, consoante o critério escolhido pelo utilizador:

1.1 Distância da viagem:

O melhor caminho será aquele que percorre a menor distância entre o ponto de inicio e o ponto de fim, independentemente to tipo de transporte utilizado e o número de zonas que este atravessa.

1.2 Número mínimo de paragens:

Para este critério, o caminho deverá passar pelo número mínimo de paragens.

1.3 Tempo da viagem:

Escolhendo este critério, o utilizador procura o caminho mais rápido entre dois pontos, independentemente da distância percorrida pelo mesmo. O tipo de transporte entre cada paragem é tido em conta, dando prioridade ao metro em relação ao autocarro (devido à diferença de velocidade entre ambos).

1.4 Preço da viagem:

Neste critério, o caminho deverá passar pelo número mínimo de zonas.

1.5 Aproximação à realidade:

Este critério oferece um caminho mais aproximado ao que um utilizador pretende na vida real. Ao contrário dos critérios anteriores, que apresentam soluções bastante extremas, este critério balança cada um dos critérios e apresenta uma solução mais moderada.

2. Formalização do problema:

2.1 Dados de entrada:

option – critério escolhido pelo utilizador para gerar o peso das arestas.
Gi = (Vi, Ei) – grafo dirigido pesado, composto por:

- V – vértices, que representam estações de metro/paragem de autocarro, com:
  - info – informação guardada no vértice (um apontador para estação/paragem).
  - adj ⊆ Ei – conjunto de arestas que ligam os vértices adjacentes.
- E – arestas, que representam as ligações entre estações/paragens, com:
  - owner – linha a que pertence a aresta ("walk" se corresponder à ligação a pé).
  - dest ∈ Vi – vértice de destino.
  - weight – peso da aresta, podendo representar quilómetros, horas, euros, entre outras unidades.

start – Estação/paragem inicial.
end – Estação/paragem final.

2.2 Dados de saída:

Gf = (Vf, Ef) - grafo dirigido pesado, tendo Vf e Ef os mesmos atributos que Vi e Ei. Para além desses atributos, Vf têm ainda:

- path – apontador para próximo vértice do caminho, definido pela execução de um dos algoritmos.
- edge_info – informação sobre a aresta (nome da linha) utilizada para ir deste vértice para o vértice apontado por path.

2.3 Restrições:

$\forall\, i \in [1; |E|],\ weight(E[i]) > 0$, pois não se está a considerar pesos negativos.

$\forall\, i \in [1; |V|],\ adjacents(E[i]) > 0$, pois cada estação/paragem tem de fazer parte da rota de, pelo menos, uma linha.

2.4 Função objetivo:

A melhor solução para este problema consiste em minimizar o critério escolhido pelo utilizador, podendo este ser a distância, o tempo, o número de paragens ou o preço.

Deste modo, sendo C o critério escolhido, a função objetivo é:

$$f = C(start, end)$$

3. Descrição da solução:

Como referido em cima, a solução do problema passa por minimizar o critério escolhido pelo utilizador.

De modo a resolver o problema, são adicionadas paragens/estações como vértices e as rotas de cada linha como arestas. Na adição das arestas ao grafo, o peso de cada uma é definido tendo em conta o critério atual.

Desta forma, transformamos o nosso problema inicial num problema do caminho mais curto. O problema pode, agora, ser facilmente resolvido recorrendo a algoritmos já existentes (que serão abordados mais à frente).

3.1 Obtenção de dados:

De modo a obter uma quantidade elevada de dados e reduzir a necessidade de input manual, são realizados *HTTP requests* à API da STCP. Assim, é possível trabalhar com estações, paragens e linhas existentes, em vez de exemplos fictícios.

Para resolver um problema do caminho mais curto existem diversos algoritmos já desenvolvidos. De modo a comparar a eficiência, decidimos implementar vários algoritmos e permitir ao utilizador escolher qual prefere usar.

Seguem-se os algoritmos implementados:

3.2 Algoritmo de Dijkstra:

Este algoritmo começa num vértice inicial e computa o caminho mais curto entre todos os vértices do grafo e esse vértice.

Utilizando um *array* desordenado, o algoritmo apresenta uma complexidade temporal $O(|E| + |V|^2)$, que pode ser simplificada para $O(|V|^2)$, considerando $|E| < |V|^2$.

Utilizando uma *binary heap* é possível reduzir a complexidade temporal para $O(|E|\log(|V|) + |V|\log(|V|))$, que pode ser simplificada para $O(|E|\log(|V|))$, considerando $|E| > |V|$.

Pode-se ainda implementar uma *Fibonacci heap*, reduzindo ainda mais a complexidade temporal para $O(|E| + |V|\log(|V|))$.

A complexidade espacial é $O(|V|)$.

3.2 Algoritmo de Bellman-Ford:

Este algoritmo, tal como o de Dijkstra, começa num vértice inicial e computa o caminho mais curto entre todos os vértices do grafo e esse vértice. A grande diferença é que este algoritmo pode ser aplicado a grafos com arestas com pesos negativos.

A sua complexidade temporal é $O(|V||E|)$ e a sua complexidade espacial é $O(|V|)$.

Como neste trabalho não existem pesos negativos, é possível afirmar *a priori* que este algoritmo não será o mais eficiente.

3.3 Algoritmo de Johnson:

Este algoritmo junta dois algoritmos já referidos. Este adiciona um vértice com arestas de peso nulo a todos os outros vértices. De seguida, corre o algoritmo de Bellman-Ford. Finalmente, remove o vértice adicionado, faz *update* ao peso das arestas já iniciais e corre o algoritmo de Dijkstra.

A complexidade temporal deste algoritmo, utilizando *Fibonacci heaps*, é $O(|V|^2 \log(|V|) + |V||E|)$.

3.4 Algoritmo A*:

Este algoritmo começa com uma estimativa do peso do caminho mais curto entre um vértice inicial e um vértice final, sendo esta estimativa sempre menor que peso real.

A partir deste ponto, o algoritmo escolhe o caminho que minimiza a função:

$$f(n) = g(n) + h(n)$$

em que n é o último vértice do caminho, $g(n)$ é o peso do caminho desde o vértice inicial ao vértice n e $h(n)$ é uma estimativa to peso do caminho desde o vértice n ao vértice final.

O algoritmo apresenta uma complexidade temporal $O(|E|)$ (E são as arestas do grafo) e uma complexidade espacial $O(|V|)$ (V são os vértices do grafo).

3.4 Pesquisa exata em *strings:*

De modo a escolher o melhor algoritmo para a pesquisa exata em *strings* foram implementados 5 algoritmos (considera-se uma *string* texto, de comprimento n, e uma *string* padrão, de comprimento m):

3.4.1 Algoritmo *naïve*:

Este algoritmo de *bruteforce* itera pelo texto (de comprimento n), comparando, caráter a caráter, o número de caracteres do padrão (de comprimento m) a pesquisar.

Apresenta uma complexidade temporal $O(n)$ para o melhor caso e $O(nm)$ para o pior caso.

3.4.2 Algoritmo de Knuth Morris Pratt:

Este algoritmo é um melhoramento do algoritmo anterior. Utiliza uma tabela pré-processada do padrão que fornece ao algoritmo a próxima posição (do padrão) a ser analisada, caso a comparação entre o caráter do texto e o caráter do padrão falhe. Deste modo, o algoritmo evita analisar caracteres já analisados anteriormente, o que diminui, consideravelmente, o número de comparações necessárias.

Apresenta, por isso, uma complexidade temporal $O(n)$ para fazer a pesquisa e $O(m)$ para o pré-processamento da tabela e uma complexidade espacial $O(m)$.

3.4.3 Algoritmo Rabin-Karp:

Este algoritmo, ao contrário dos anteriores, utiliza *hashes* para efetuar comparações. Começa por fazer o pré-processamento da *hash* do padrão a pesquisar. Depois, é calculada a *hash* para cada *substring* do texto (de comprimento m – 1), sendo esta comparada com a *hash* do padrão. No entanto, esta comparação não chega, sendo necessário comparar o padrão com a *substring* de modo a garantir que não há colisões nas *hashes*. Assim, a eficiência deste algoritmo depende da eficiência da função de *hash* utilizada: quanto melhor for a computação das *hashes*, menor é o número de colisões entre as mesmas.

Apresenta uma complexidade temporal $O(n + m)$ para o melhor caso e $O(nm)$ para o pior caso e uma complexidade espacial de $O(1)$.

3.4.4 Algoritmo de Boyer-Moore e derivações:

Este algoritmo começa por analisar o carácter do texto na posição m. Se este carácter não pertencer ao padrão, então o padrão não pode estar na posição [1..m]. Deste modo, o algoritmo "saltar" aqueles m caracteres do texto. Caso pertença, o algoritmo volta ao inicio e começa a comparar. O algoritmo recorre a tabelas pré-processadas para calcular o próximo "salto", podendo ignorar grandes quantidades de caracteres do texto.

Apresenta uma complexidade temporal $O(n/m)$ para o melhor caso e $O(nm)$ para o pior caso.

No contexto do nosso programa, o tamanho do texto onde é necessário pesquisar padrão é relativamente pequeno (entre 3 a 15 caracteres). Decidimos, por isso, optar pelo algoritmo de Knuth Morris Pratt, devido à consistência da complexidade temporal deste algoritmo e à simplicidade dos casos de utilização (não justificando a utilização do Boyer-Moore).

3.5 Pesquisa aproximada de *strings* (distância de *Levenshtein*):

De modo a escolher o melhor algoritmo para a pesquisa aproximada em *strings* foram implementados 4 algoritmos que calculam a distância de *Levenshtein*.

A distância de *Levenshtein* consiste em calcular o mínimo número de inserções, eliminações e substituições de caracteres para tornar uma *string* na outra.

3.5.1 Algoritmo recursivo:

Esta versão calcula a distância de forma recursiva. Comparada com as outras implementações, esta versão é extremamente ineficiente, pois calcula a distância das mesmas *substrings* diversas vezes.

3.5.2 Algoritmo com programação dinâmica:

Esta versão utiliza a programação dinâmica de modo a não ter de recalcular as distâncias das *substrings* mais que uma vez.

Apresenta uma complexidade temporal e espacial $O(mn)$.

3.5.3 Algoritmo com programação dinâmica melhorado:

Esta versão é semelhante à anterior, no entanto, é melhor em termos de complexidade espacial. Em vez de guardar a matriz 2D, é possível utilizar apenas uma coluna/linha.

Deste modo, apresenta a mesma complexidade temporal que a versão anterior e uma complexidade espacial de $O(n)$.

3.5.4 Algoritmo com programação dinâmica adaptado para *substrings:*

Utilizando o algoritmo anterior, é possível adaptá-lo para trabalhar com *substrings* em vez de *strings*. Apresenta uma complexidade temporal semelhante à anterior, utilizando, no entanto, 2 colunas/linhas. Logo, a sua complexidade temporal é $O(2n)$.

No contexto do nosso programa, a pesquisa aproximada de *substrings* é extremamente importante. Considerando, por exemplo, a pesquisa do padrão "ALIADSO" no texto "AV.ALIADOS": utilizando o algoritmo para *strings*, obteríamos a distância 5; por outro lado, com o algoritmo para *substrings*, obteríamos a distância 2.

Isto permite restringir a distância de edição máxima aceitável (por exemplo, 2), sem comprometer a eficácia da pesquisa. Ou seja, com a restrição referida em cima, um utilizador poderia pesquisar por "ALIADSO" (sem o "AV.") e ser encontrada a paragem pretendida.

Deste modo, para efetuar a pesquisa aproximada, decidimos utilizar o algoritmo de pesquisa adaptado para *substrings.*

3.6 Melhoramento do tempo de pesquisa:

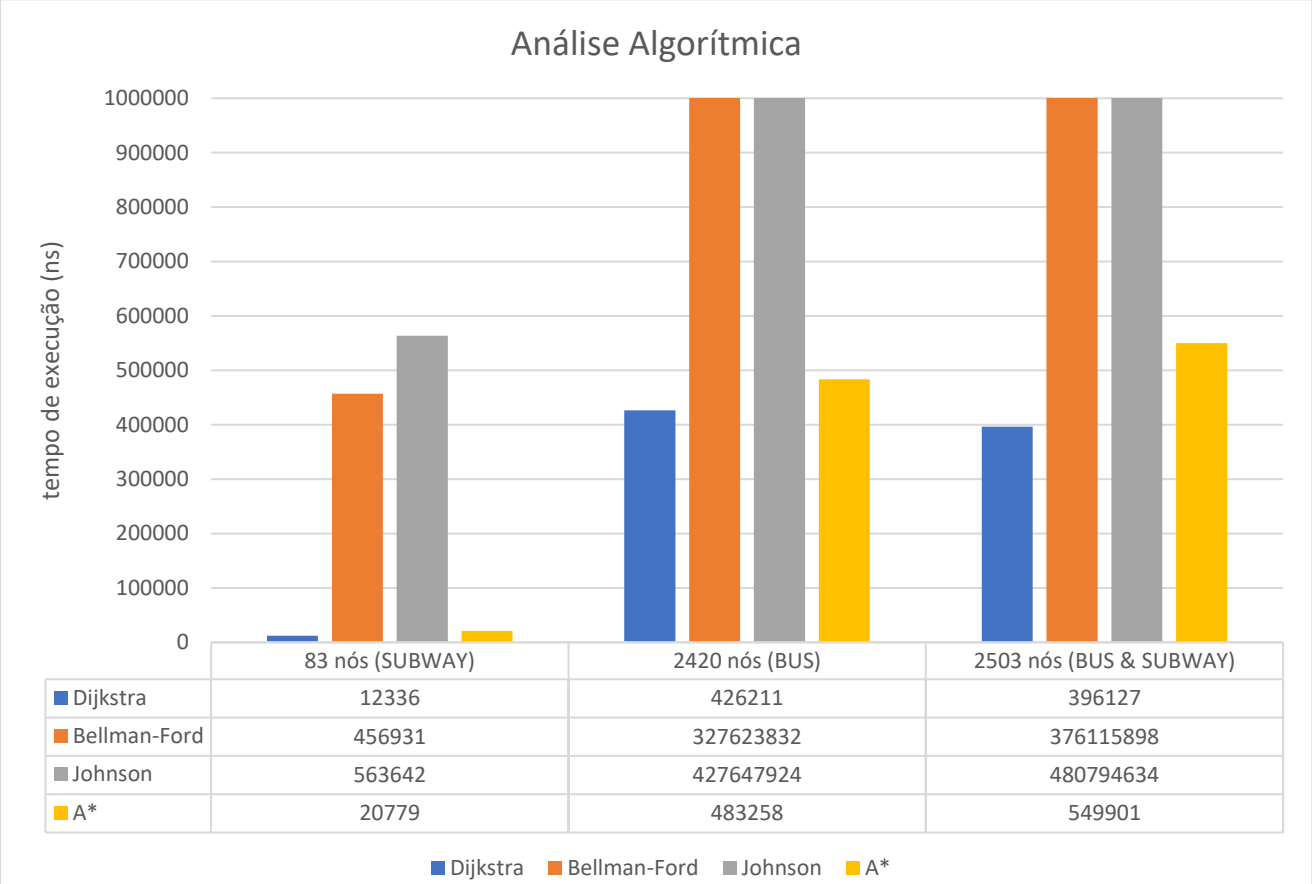
Como no nosso programa trabalhamos com milhares de paragens (cerca de 2500 paragens), decidimos impor algumas restrições na pesquisa (quer exata, quer aproximada).

Deste modo, ao iterar pelas paragens existentes, é realizado uma primeira verificação que calcula a diferença entre o comprimento do nome que o utilizador introduziu e o comprimento do nome da paragem. Se esta diferença for maior que um valor, então a paragem é ignorada (não passando pelos algoritmos de pesquisa). Isto permite filtrar uma grande quantidade de paragens, melhorando o tempo de pesquisa.

Outra restrição, já referida anteriormente, é a máxima distância de edição aceitável, ignorando uma grande quantidade de paragens que não interessam.

Apesar destas duas restrições, há casos em que existe mais que uma potencial paragem, quando isto acontece, é dado ao utilizador uma lista com as paragens cujos nomes se assemelham com o que o utilizador inicialmente inseriu para dar mais opções de uso.

3.7 Análise Empírica:



	83 nós (SUBWAY)	2420 nós (BUS)	2503 nós (BUS & SUBWAY)
Dijkstra	12336	426211	396127
Bellman-Ford	456931	327623832	376115898
Johnson	563642	427647924	480794634
A*	20779	483258	549901

Figura 1- Média de tempos de execução (ns) em 100 iterações

Como podemos observar pelo gráfico, Dijkstra e A* são os que produzem melhores resultados, obtendo os menores tempos de execução em nanossegundos, estando sempre na ordem de 10^4 enquanto os outros se aproximam da ordem de 10^8 quando se considera um número elevado de vértices.

Isto era de esperar pois o algoritmo de Bellman-Ford não faz *scale* eficientemente.

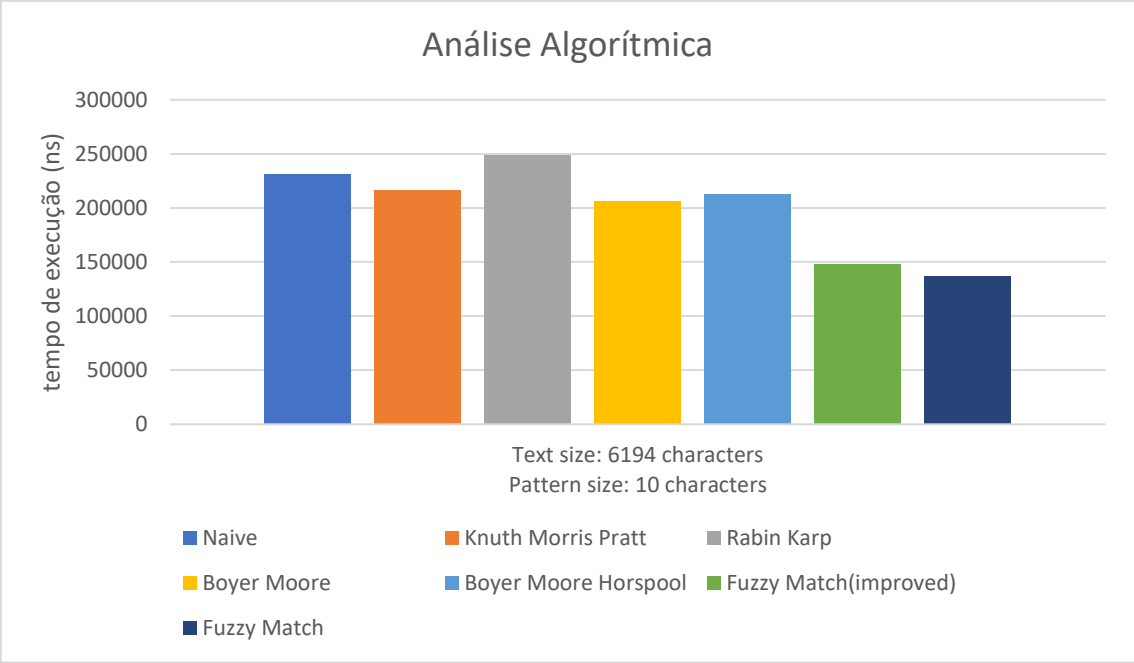


Figura 2- Média de tempos de execução (ns)

É de salientar que os tempos de execução aqui mostrados são da ordem dos nanossegundos e que as diferenças existentes entre tempos de execução são essencialmente devido a erros de medição existentes durante o seu *runtime* e não devido a melhor *performance* por parte do algoritmo, dito isto, chegamos à conclusão que todos os algoritmos são aproximadamente iguais para os dados de entrada usados.

4. Estrutura de Dados:

4.1 Representação do grafo (classes *graph* / *vertex* / *edge*):

No ficheiro "graph.h" encontra-se a estrutura de dados do grafo usada neste trabalho, a qual faz uso de um *template* T (informação genérica), que neste caso representa apontadores para objetos da classe *Stop*.

Um vértice genérico é definido no ficheiro "vertex.h", também fazendo uso do *template* T para representar a informação pretendida. Esta classe contém:

- _info - variável do tipo genérico T, única e distinta para cada vértice criado, neste caso representa um apontador para uma paragem (*stop**).
- _adj - vetor de objetos da classe *edge* representando arestas que saem do vértice associado.
- _dist - valor *double* que armazena informação do *weight* percorrido até esse vértice.
- _path - apontador para um vértice, simboliza o caminho criado pelo algoritmo.
- _fscore - usado pelo algoritmo A* para armazenar a distância ótima aproximada em cada iteração do algoritmo.
- _path_info - informação relativa à aresta que liga este vértice ao vértice adjacente representado na variável path.
- _queue_index - variável requerida pela *mutable_priority_queue*.

Uma aresta genérica é definida no ficheiro "edge.h", outra vez fazendo uso de *template* T, que contém:

- _owner - representa a linha que liga duas estações/paragens (vértices)
- _dest - apontador para um vértice de destino, pois usamos arestas unidirecionais.
- _weight - valor *double*, neste caso representando a distância entre paragens (vértices).

A classe *graph* contém apenas uma variável denominada _vertex_set, um vetor de apontadores para todos os vértices pertencentes ao grafo.

4.2 Classe d*atabase*:

Classe *singleton* que contém toda a informação a ser utilizado pelo programa. Possui funções que funcionam como *wrappers* e que interagem com as classes *loader*, *builder*, *viewer* e *graph*. Ao executar secções críticas do programa (fazer *load* dos dados, construir o grafo, correr algoritmos, etc.), a *database* mostra ao utilizador os resultados obtidos, bem como informação relativa aos tempos de execução de cada secção.

Esta classe contém:

- _lines – std::map com o código de cada linha como *key* e um objeto da classe *line* como *value*.
- _stops – std::map com o código de cada paragem como *key* e um objeto da classe *stop* como *value*.
- _graph – grafo criado com os dados.

4.3 Classe *loader*:

Classe *singleton* que obtém informações relativas às paragens e linhas da zona do Porto e atualiza as estruturas de dados na *database*. Esta classe obtém os dados através de *HTTP requests* à *API* da STCP e guarda-os num ficheiro *json* para uso posterior.

4.4 Classe *builder:*

Classe *singleton* que constrói o grafo consoante o critério escolhido pelo utilizador, adicionando cada *stop* como vértice e as rotas de cada *line* como arestas (o peso destas depende do critério). Antes de construir o grafo da *database*, esta classe efetua sempre uma "limpeza" ao mesmo, de modo a garantir que o grafo fique vazio e pronto para ser construído com novos dados.

4.5 Classe *viewer:*

Classe que inicia a aplicação "GraphViewer", permitindo ao utilizador visualizar o grafo e o *path* gerado por um algoritmo.

Esta classe contém:

- _g – grafo que se vai fazer *display*.
- _gv – objeto que faz *wrap* da aplicação GraphViewer.
- _stops_index e _lines_index – std::map's para facilitar o acesso a cada vértice/aresta da aplicação GraphViewer.

4.6 *Display* do menu (classes *menu / ibehaviour):*

A classe *menu* é uma classe *singleton* com a função de fazer *display* do menu para a consola. O comportamento desta classe (o que ela faz *display*) depende da variável _behaviour. Esta variável é um *smart pointer* que aponta para uma classe que implementa a interface *ibehaviour* (cujo único método definido é um método virtual *display()*). A transição entre *behaviours* é controlada por máquina de estados simplificada.

Esta classe contém:

- _behaviour – *smart pointer* para uma classe que implementa a interface *ibehaviour*.
- _criteria – critério escolhido pelo utilizador.
- _search_mode – modo de pesquisa escolhido pelo utilizador.
- _algorithm – algoritmo escolhido pelo utilizador.
- _src – código da *stop* inicial.
- _ dst – código da *stop* final.

4.7 Classe *line:*

Classe que representa uma linha de transportes públicos.

Esta classe contém:

- _code – código da linha.
- _pubcode – código alternativo da linha.
- _mode – modo de pesquisa escolhido pelo utilizador.
- _routeA – rota da linha, sentido ascendente.
- _routeD – rota da linha, sentido descendente.

4.8 Classe *stop:*

Classe que representa uma paragem.

Esta classe contém:

- _code – código da paragem.
- _name – nome da paragem.
- _zone – zona da paragem.
- _coords – coordenadas da paragem.

5. Manual de Utilização

O programa desenvolvido foi concebido, sucintamente, para calcular o melhor percurso usando os transportes disponibilizados pela STCP, de acordo com os parâmetros dados pelo utilizador: distância, tempo, número de paragens, zonas ou simulação real.

5.1 Funcionamento do programa:

O programa, ao ser executado, efetua as seguintes operações:

- Fazer *load* da informação. Caso exista, faz *load* a partir dos ficheiros, senão faz *HTTP requests* à API da STCP e armazena a informação para usar mais tarde.
- Gera o grafo com as estações/paragens e as rotas das linhas, de acordo com o critério escolhido pelo utilizador.
- Corre o algoritmo escolhido pelo utilizador.
- Mostra a informação sobre o caminho mais curto.

É permitido ao utilizador escolher:

- Um critério de pesquisa, que influencia os pesos das arestas:
  - Menor distância;
  - Menor tempo de viagem;
  - Menor número de paragens;
  - Menor número de zonas / preço;
  - Estimativa real (junção de tempo, distância e preço).
- Um modo de pesquisa:
  - Pesquisa de paragens pelo nome;
  - Pesquisa de paragens por linha.
- Um algoritmo, para encontrar o caminho mais curto:
  - Algoritmo de Dijkstra;
  - Algoritmo de Bellman-Ford;
  - Algoritmo de Johnson;
  - Algoritmo A* (A star).

5.2 Compilação:

Este projeto precisa está dependente de duas bibliotecas para compilar:

- nlohmann/json (https://github.com/nlohmann/json), para fazer *parse* de JSON strings;
- curl (https://curl.haxx.se), para efetuar GET *requests*.

As instruções para compilar o programa, em Visual Studio, encontram-se no ficheiro "READ_ME.txt" (fornecido junto do código fonte).

6. Principais Dificuldades

A principal dificuldade foi a obtenção de dados para trabalhar e preencher o grafo- Optámos por utilizar a API da STCP e fazer *HTTP requests* para obter tais dados. Esta decisão atrasou muito o processo, especialmente devido a problemas com as bibliotecas requeridas para fazer tais *requests*.

Outra dificuldade encontrada foi como estabelecer uma forma de obter um caminho ótimo para alguns dos critérios, mais especificamente para o mínimo número de zonas.

8. GraphViewer

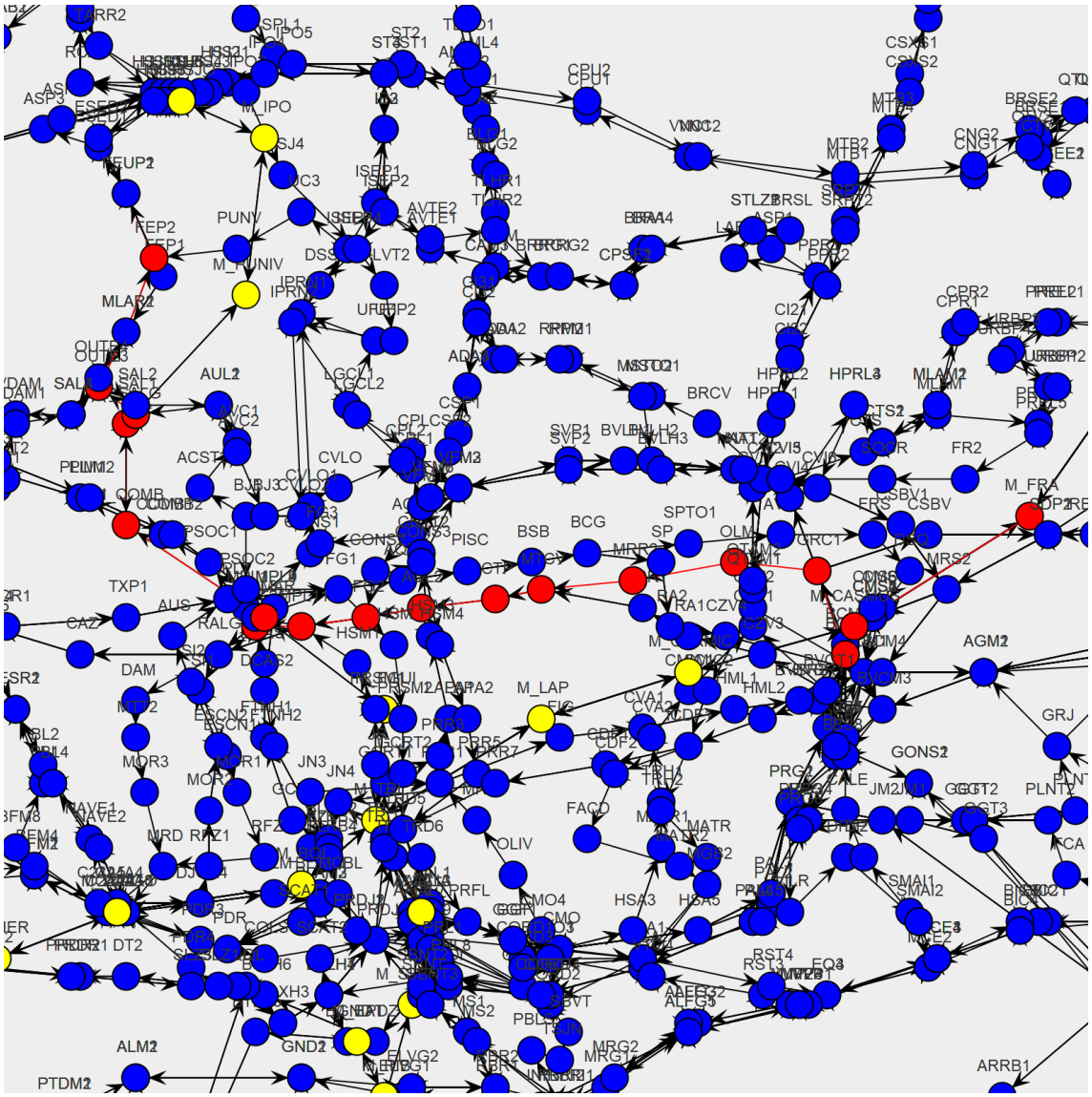


Figura 2 - Exemplo do GraphViewer para uma rota inserida

9. Bibliografia

https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm
https://en.wikipedia.org/wiki/Johnson%27s_algorithm
https://rosettacode.org/wiki/A*_search_algorithm
https://stackoverflow.com/questions/38772780/complexity-of-dijkstras-algorithm
https://ginstrom.com/scribbles/2007/12/01/fuzzy-substring-matching-with-levenshtein-distance-in-python