U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

# Concepção e Análise de Algoritmos

## TripPlanner: Itinerários para transportes públicos

TEMA 11

**Professor Regente:** João Carlos Pascoal Faria

**Professor das aulas Teórico-Práticas:** Rosaldo José Fernandes Rossetti


**Turma 3 Grupo C**

Fábio Daniel Reis Gaspar | up201503823@fe.up.pt

Filipa Manita Santos Durão | up201606640@fe.up.pt

Pedro Miguel Sousa da Costa | up201605339@fe.up.pt


28 de Março de 2018

# Índice

# Introdução

O tema do trabalho é o tema proposto número 11 - TripPlanner: Itinerários para transportes públicos, que irá ser mais especificado mais adiante neste relatório.

O presente relatório está dividido em 5 capítulos principais, começando por **descrever** o tema, **formalizar a descrição** utilizando especificações mais técnicas para a implementação, a **solução** implementada para a resolução do problema com as respetivas **análises** de funcionamento e complexidade, a **estrutura** do programa ao nível da implementação e um breve **manual de utilização** para auxiliar o utilizador no uso do programa.

# Glossário

- **w(i, j, k):** peso da aresta entre dois nós, i e j, em função de um parâmetro fornecido pelo utilizador, k. O peso da aresta tem um significado diferente dependendo do contexto em que está inserida.
- **k:** especifica o parâmetro a ser usado, neste caso tempo ou preço.
- **G(V,E):** grafo constituído por vértices e arestas
- **E:** número de arestas do grafo
- **V:** número de vértices do grafo
- **S:** vértice inicial do caminho
- **Q:** vértice final do caminho
- **$P_{ij}$:** conjunto de arestas que liga o nó i ao nó j.
- **O(x):** complexidade do algoritmo dependendo do parâmetro x

# Descrição do tema

O tema proposto para o projeto foi um planeador de viagens em transportes públicos que englobasse vários meios de transportes, ou seja, um programa que planeasse a melhor viagem, tendo em conta vários parâmetros, numa certa rede intermodal de transportes de uma cidade.

Foi-nos pedido que tomássemos em consideração parâmetros como o tempo de viagem, o preço do total da viagem, o número de transbordos efetuados, e qualquer outra opção que um possível utilizador da "aplicação" pudesse querer escolher para realizar a viagem que necessita.

# Formalização do Problema

Quando se fala de redes de transportes, estradas, paragens e percursos otimizados, a estrutura evidente de utilização é o Grafo. Então, o problema torna-se mais simples. Um enunciado que pede o melhor percurso numa rede de transportes intermodais, transforma-se numa utilização básica do algoritmo Dijkstra e do algoritmo A-star aplicados a um grafo. Com as variantes exigidas, como poder acolher certas exigências do utilizador quanto ao número de transbordos ou menor tempo de viagem, o problema metamorfoseia-se ligeiramente, passando a ser necessário modificar ou acrescentar informação às ligações entre nós do grafo para se perceber que tipo de transporte é e informações relativas a cada meio disponível.

## Dados de Entrada

Todos os dados sobre os quais o programa corre são retirados de dois ficheiros, o ficheiro de nós e o ficheiro de arestas, que foram devidamente preenchidos com os dados formatados da maneira correta.

Existem diversos nós e estes nós estão ligados entre si por diferentes meios de transporte e linhas diversas. Para distinguir o peso entre as arestas de diferentes meios de transporte há um fator multiplicativo para esse dado meio e transporte.

Informação relevante de um nó:

- **ID -** número identificador (que coincide com a sua posição no vetor total de edges no grafo para maior eficiência de acesso)
- **Info -** a informação do nó (neste caso o nome da paragem)
- **Edges -** todas as arestas que saem do nó
- **x e y -** posição geográfica do nó

Informação relevante de uma aresta:

- **NóInício -** ID do nó do qual a aresta parte
- **NóFim -** ID do nó destino da aresta
- **Meio -** Meio de transporte representado pela aresta
- **Linha -** Linha do meio previamente definido

Para efeitos de teste, foi criado um mapa que é um modelo aproximado da cidade do Porto. Nesse mapa estão representadas 6 linhas diferentes, 3 de metro e 3 de autocarro. Existem vários pontos em que se interligam diversos meios de transporte, para existirem várias maneiras de se chegar de um local a outro. As constantes utilizadas para o cálculo do preço ou tempo da viagem encontra-se definidas dentro do próprio código. A informação pré-processada no início do programa e é gerado o grafo representante do gráfico.

## Dados de Saída

O grafo original criado é modificado e é criado um caminho entre os nós. Todos os dados devolvidos pelo programa são apresentados no ecrã de uma forma percetível para o utilizador corrente. São devolvidas, por ordem, as paragens que constituem o percurso a percorrer e seguidamente uma descrição mais detalhada do que fazer em cada paragem, se seguir na mesma linha ou mudar.

No final pode ser, à escolha do utilizador, consultada a distância total do percurso e o preço total da viagem a efetuar.

## Restrições

Não podem existir arestas com pesos negativos, pois isso implicaria que com ciclos seria impossível encontrar o caminho ótimo entre dois pontos.

## Função Objetivo

O objetivo final do programa é minimizar o peso total das arestas do percurso que ligue S a Q de acordo com um critério definido pelo utilizador. É possível escolher um dos seguintes:

- Número máximo de transbordos à escolha do utilizador;
- Apenas trajetos sem troços a ser percorridos a pé;
- O percurso com o menor preço**;**
- O percurso com o menor tempo de viagem;

Dado que é de interesse minimizar o preço da viagem, foi necessário desenvolver um método de pagamento. Para a simplificação do problema, foram estabelecidos os seguintes princípios:

- os preços de cada viagem são constantes em qualquer parte do mapa;
- enquanto a viagem for efetuada na mesma linha do mesmo meio de transporte, o preço não é afetado, independentemente do tempo ou distância;
- quando muda de linha, é necessária a compra de um novo bilhete;
- o preço de andar a pé é nulo;

A função objetivo passa então por minimizar $\sum_{v,v+1\in P_{S,Q}} w(v, v+1, k)$ de acordo com uma das opções anteriores.

# Solução

## Algoritmos implementados

Para a realização do cálculo do percurso mínimo entre dois quaisquer pontos do grafo criado, foram utilizados os algoritmos de Dijkstra e A Estrela (*A Star*).

## Algoritmo de Dijkstra

Num primeiro estágio do desenvolvimento do programa, foram criadas duas versões em tudo semelhantes, que apenas diferiam na estrutura de dados onde era armazenada a informação que estavam a ser manipulados durante o algoritmo. Uma usava uma *min-heap*, onde cada elemento não guardava a sua posição na *heap*, e outra uma *mutable priority queue*, onde era guardada a posição de cada elemento (cujo código foi fornecido em aula, autoria do Professor João Pascoal Faria). Depois de feita uma análise empírica à complexidade temporal da execução dos dois programas, representada na figura 1, chegamos à conclusão que o Dijkstra que utilizava uma *mutable priority queue* era muito mais eficiente. Assim sendo, todas as adaptações que precisamos de realizar no Dijkstra foram efetuadas sobre essa versão.

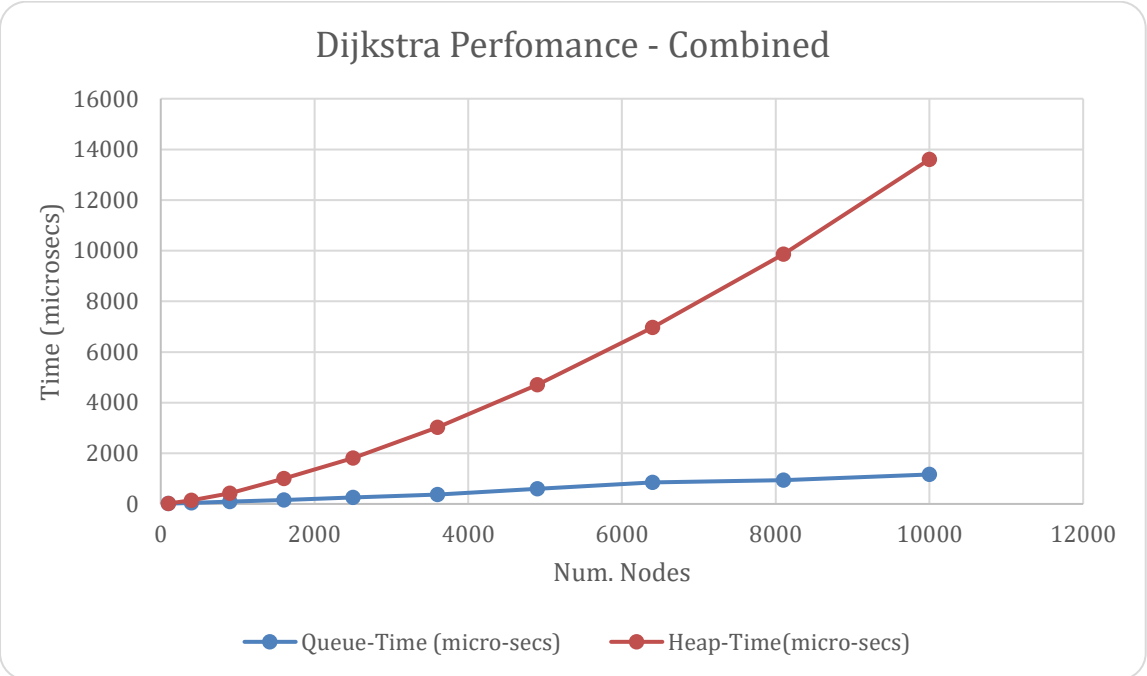


**Figure 1 - Dijkstra Comparison - Data Structures**

Dado que a nossa função objetivo possui um elemento variante, de acordo com o que o utilizador pretende, foram criadas versões específicas a cada elemento de modo a evitar a criação de um algoritmo bastante condicional:

- Uma versão onde são evitados caminhos a pé:
- Uma versão onde se limita o número máximo de transbordos;
- Uma versão onde se minimiza o preço da viagem;
- Uma versão onde se minimiza o tempo de viagem;

A seguir apresenta-se a versão mais simples, base de todas as previamente enunciadas.

```
1   DIJKSTRA(G,S,E):  //s - start node, e - end node
2       for each V in G do
3           distance(V)= INFINITE;
4           V(path)= null;
5           visted(V) = false;
6       end for
7       distance(S) = 0;
8       Q = null;  //minimum priority queue
9       insert(Q, S);
10      while(!empty(Q)) do
11          V = extractMin(Q);
12          if V = E then break;
13          end if
14          for each W in adj(V) do
15
16              if distance(W) > (distance(V) + weight(V,W)) then
17                  distance(W) = distance(V) + weight(V,W);
18
19                  if(!visited(W)) then
20                      insert(Q,W);
21                  else
22                      decreaseKey(Q,W);
23                  end if
24                  visited(W) = true;
25              end if
26          end for
27      end while
28  end function
```

**Figure 2 - Dijkstra pseudocode**

# Análise Temporal e Espacial

## Análise Analítica

O algoritmo mantém guardado em espaço auxiliar todos os nós a serem processados, que é no máximo |V|. Assim, a sua complexidade espacial é O(|V|).

O algoritmo implementa uma fila de prioridade com o mínimo à cabeça. A **operação de inserção** de um nó pode ser efetuada em tempo logarítmico, em função do tamanho da fila, N, $\boldsymbol{O(\log|N|)}$**.** A **operação de extração** pode também ser efetuada em tempo logarítmico, i.e., a seleção é feita em tempo constante (está à cabeça da fila), $\boldsymbol{O(1)}$, e a extração, como a inserção, ocorre em $O(\log|N|)$, no total, $\boldsymbol{O(\log|N|)}$. A parte fulcral da eficiência do algoritmo encontra-se agora na operação de ***decreaseKey***, que explica a diferença substancial exposta na Figura 1. A primeira abordagem passou por se usar uma *min-heap* simples onde nenhum elemento guardava a sua posição e, portanto, teria de se procurar sequencialmente o elemento no *array*, para se alterar o seu valor, com complexidade temporal $O(N)$, e organizar a árvore com o valor alterado, $O(\log|N|)$; no total $\boldsymbol{O(N)}$. No entanto, se cada elemento guardar em si a sua posição no *array*, não é necessária a pesquisa e a operação pode-se efetuar em $\boldsymbol{O(\log|N|)}$ total.

O ciclo presente nas linhas 2 a 6 é executado $|V|$ vezes, percorrendo todos os nós do grafo, executando operações de tempo constante, tendo a complexidade total de $O(|V|)$. A primeira inserção na linha 9 ocorre em tempo constante dado que a fila estava vazia. O ciclo contido nas linhas 14 a 26 é executado $|E|$ vezes, uma por cada aresta.

Nas linhas 10 a 27 os elementos são extraídos, linha 11, e inseridos, linha 20, na fila de prioridade, que no máximo terá $|V|$ elementos. O número de inserções e extrações é, no máximo, $|V|$, pois nenhum nó é introduzido mais que uma vez na fila, à primeira é marcado como visitado e ao momento da extração o nó encontra-se processado e é garantido que se encontrou o caminho mais curto. Como cada operação pode ser efetuada em $O(\log|N|)$, a complexidade é $O(V * \log|V|)$. A operação de *decreaseKey* presente no ciclo é efetuada, no máximo, $|E|$ vezes, uma por cada aresta; no total $O(\mathrm{E} * \log|V|)$.

No total, o tempo de execução é $O(|V| * \log|V| + |E| * \log|V| + |V| + |E|)$, ou simplesmente $\boldsymbol{O\big((|V| + |E|) * \log|V|\big)}$.

Note-se que a condição presente na linha 12 vai reduzir o tempo real de execução. O algoritmo foi desenhado originalmente para encontrar a distância de um nó a todos os outros. No entanto, dado o contexto deste trabalho, faz todo o sentido parar quando se encontra o destino. A comprovação pode ser vista na seção seguinte.

## Análise Empírica

Para testar a performance real do algoritmo foram gerados grafos aleatórios matriciais de duas formas, com um número variável de nós e *edges*, figura 3 e figura 4;

### Em relação aos nós:

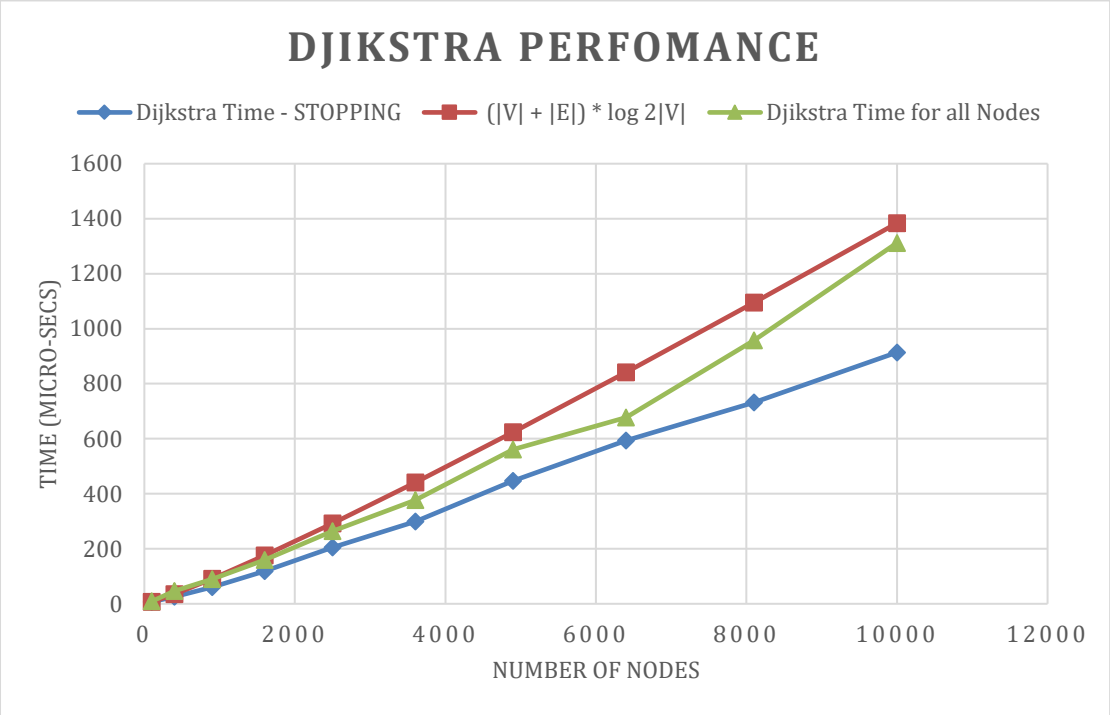


**Figure 4 - Dijkstra Performance wrt Nodes**

Como se pode verificar, a análise analítica encontra-se correta. Dado que a expressão analítica da complexidade temporal resultava em números muito elevados, face aos reais, foi calculada uma constante de proporcionalidade entre os valores para tornar o gráfico mais legível, que neste caso foi aproximadamente 0.0021.

Em relação às edges (mantendo o número de nós constantes):

$|E| \sim 4 * |V| - sqrt(|V|)$ :

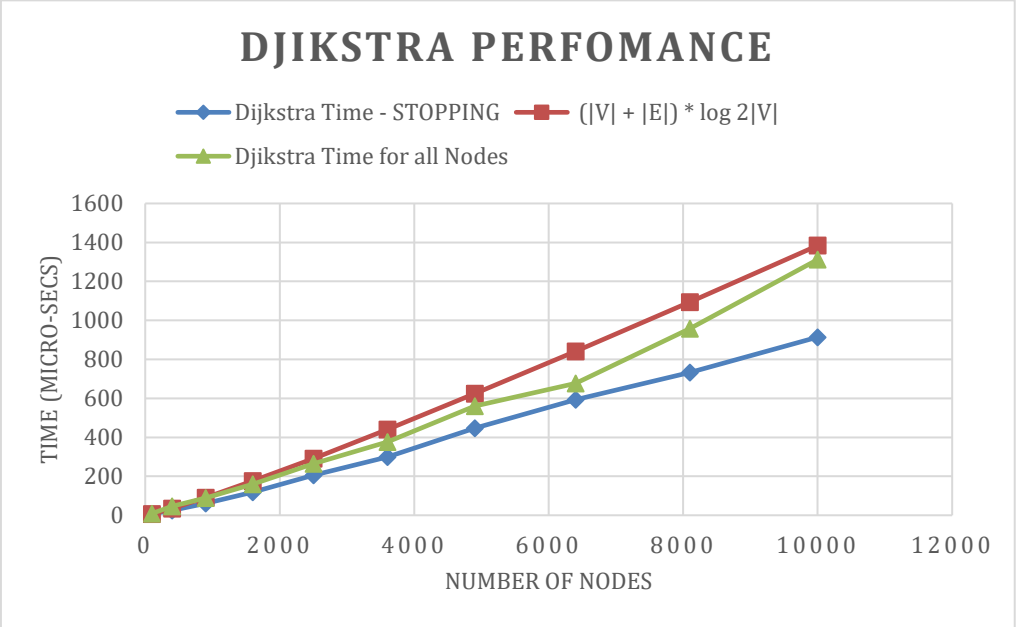


**Figure 5 - Dijkstra Performance wrt Edges, Sparse**

$|E| \sim 2 * |V| * sqrt(|V|)$ :

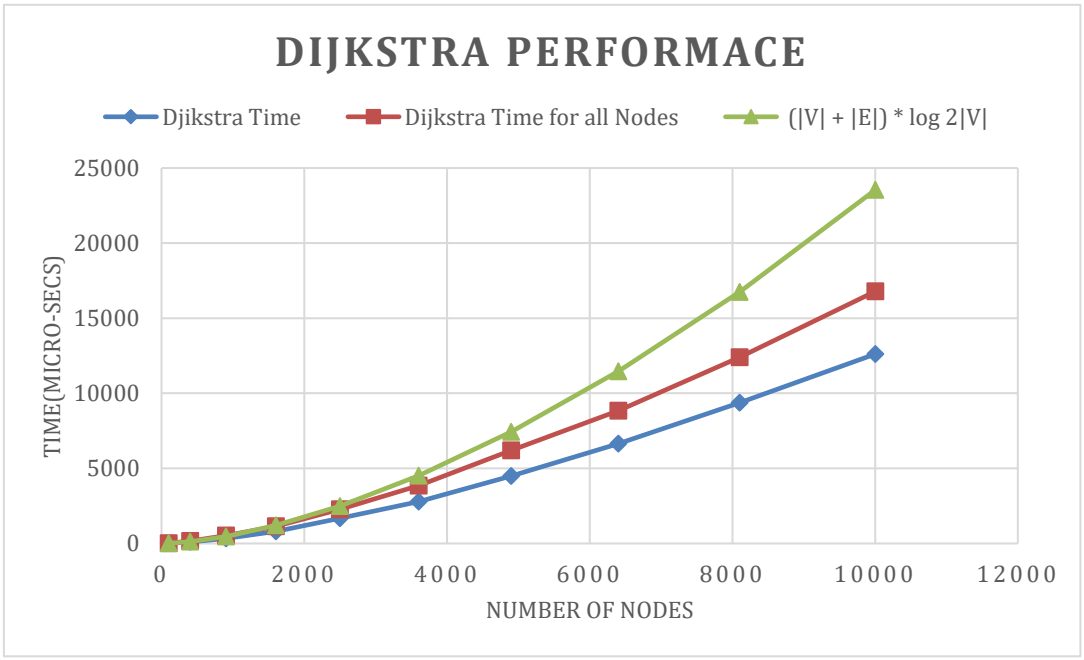


**Figure 6 - Dijkstra Performance wrt Edges, "Dense"**

Pela análise dos dois gráficos anteriores podemos verificar que conforme o número de *edges* se aproxima do quadrado do número de nós, i.e., grafos densos, figura 5, as curvas tomam um comportamento quadrático, o que seria expectável. À medida que $E \sim |V^2|$, a complexidade temporal inicial, $\boldsymbol{O}\big((|\boldsymbol{V}| + |\boldsymbol{E}|) * \mathbf{log}|\boldsymbol{V}|\big)$, vai passar a $\boldsymbol{O}\big(\big|\boldsymbol{V^2}\big| * \boldsymbol{log}\,|\boldsymbol{V}|\big)$.

## Desenvolvimento das variantes

Para a resolução de todas as funções objetivo, o foco passa por alterar o significado e valor atribuído à distância de um nó, linha 3.

Tendo isto em mente, a transformação para minimizar o tempo de viagem torna-se evidente. Sabendo as distâncias entre os nós, sabendo a velocidade do meio de transporte associado à ligação entre os nós e a informação contida nos dados de entrada, obtém-se o tempo de viagem e resolve-se o problema. Transforma-se então a "distância" de um nó a outro em tempo.

Em relação a evitar caminhos a pé a transformação é também bastante trivial. Apenas será necessário acrescentar uma nova condicionante, para além da apresentada na linha 20, que verifica se o tipo de ligação é a pé; se tal for o caso, ela é ignorada e o algoritmo procede normalmente.

No caso do preço, a adaptação já requer outras componentes, sobretudo a implementação do método de pagamento já referido. Neste caso, a distância passa a ser o preço. Se fossemos a aplicar o algoritmo sem nenhuma outra restrição, o resultado ótimo, em relação ao preço, seria uma viagem só a pé, visto que é possível andar de qualquer nó para qualquer nó. Para evitar esta situação, é também necessário definir uma distância máxima permitida a pé; quando este limite for violado, o caminho até àquele momento será ignorado e o algoritmo procederá normalmente. O que falta então é saber quando, durante uma viagem, é efetuada uma alteração de linhas. Para tal, cada *edge* tem a si associada uma linha e um meio. Durante o ciclo interior, linhas 18 a 30, sempre que um novo caminho mais barato é encontrado, é colocado no nó a informação sobre o tipo de transporte e a linha que chegou até lá. Com esta informação presente no nó, é possível verificar se houve alterações durante a construção do caminho e assim ir alterando o preço.

Finalmente, a limitação do número de transbordos. A solução desta versão passa pelo que foi descrito na anterior. Um transbordo acontece quando um utilizador troca de meio ou linha, que corresponde à informação que foi criada e guardada no anterior. Assim sendo, em vez de aumentar o preço sempre que é necessário mudar, incrementa-se o número de transbordo efetuados até àquele momento.

Pode-se então verificar que em termos de complexidade temporal, as variantes terão a mesma do descrito originalmente na secção anterior, com um acréscimo constante de um tempo de *overhead* para fazer todas as comparações e cálculos adicionais.

## Algoritmo A Estrela (A* ou A Star)

Ao contrário do algoritmo de Dijkstra, o A Estrela foi desenvolvido para obter o caminho mais curto entre dois pontos, podendo ser interpretado como uma versão "mais informada" do Dijkstra, como se poderá confirmar no pseudocódigo (figura 6). Recorrendo ao uso de uma função de avaliação, escolhe o próximo nó a processar de uma forma gananciosa; escolhe o nó que o aproxima na direção geral do seu destino (por exemplo, a distância física). Ao primeiro olhar parece resolver muito melhor o problema do que o Dijkstra (embora neste também tenha sido implementada a condição de paragem), no entanto é preciso uma boa análise da função a ser usada, o que ditará se o algoritmo é de facto admissível e garante o ótimo

```
1   A_STAR(G,S,E):  //s - start node, e - end node
2       //heuristic(X,Y) - lower bound estimate of the minimum time between X and the end
        node
3       for each V in G do
4           distance(V)= INFINITE;
5           V(path)= null;
6           visted(V) = false;
7       end for
8       distance(S) = 0;
9       Q = null;  //minimum priority queue
10      insert(Q, S);
11      while(!empty(Q)) do
12          V = extractMin(Q);
13          if V = E then
14              distance(E) = distance(E) + heuristic(S);
15          end if
16          for each W in adj(V) do
17
18              new_distance = distance(V) + weight(V,W) - heuristic(V) + heuristic(W);
19
20              if distance(W) > new_distance then
21                  distance(W) = new_distance;
22
23                  if(!visited(W)) then
24                      insert(Q,W);
25                  else
26                      decreaseKey(Q,W);
27                  end if
28                  visited(W) = true;
29              end if
30          end for
31      end while
32  end function
```

**Figure 7 - A Star Pseudocode**

## Função de avaliação, admissibilidade e garantia do ótimo

Seja V um nó intermédio do caminho entre S e Q a ser processado, $g(V)$ o custo mínimo conhecido entre S e V e $h(V)$ a estimativa do custo entre V e Q. A função de avaliação é $f(V) = g(V) + h(V)$ . A estimativa, $h(V)$, é calculada a partir de uma heurística. Para esta heurística ser considerada admissível, e por consequência o algoritmo admissível, o custo estimado, entre V e Q, terá de ser sempre inferior ou igual ao custo real, uma estimativa por baixo.

No contexto do problema, o custo pode ser ou o tempo de viagem ou o preço da viagem.

Em virtude do método de pagamento implementado, não existe nenhuma forma lógica de estimar o preço de viagem entre um nó intermédio e o final. Uma das "soluções" passaria por apenas atribuir o custo de 0, o preço mínimo de andar a pé, à função $h(V)$, no entanto isso corresponde ao funcionamento geral do Dijkstra e, no mínimo, trará algum *overhead*, em termos de execução à solução já implementada.

Relativamente ao tempo de viagem, seja $t(V)$ a estimativa do tempo. Dado que o peso de uma aresta, obtido também por $g(V)$, é a distância euclidiana entre os dois pontos, o tempo pode ser calculado através da fórmula $t(V) = (\frac{d(V)}{v(V)})$, onde $d(V)$ é a distância euclidiana entre V e Q e $v(V)$ a velocidade do transporte que liga V a Q. Se para a velocidade for usado o valor mais alto de entre os transportes, no nosso caso, o metro, o valor de $t(V)$ será sempre inferior ao tempo real. Estando verificada a condição de admissibilidade da função heurística para o tempo, o algoritmo pode ser considerado admissível. Estando provada a admissibilidade falta provar a garantia do ótimo, que a heurística é consistente.

Para ser consistente $t(V) \leq c(V,W) + t(W)$, onde W é um qualquer nó adjacente a V e $c(V,W)$ representa o custo mínimo conhecido entre V e W, neste caso o tempo. Seja $d(i,j)$ a distância euclidiana entre os nós i e j. Pela desigualdade triangular e tomando como referência a figura 7, a $d(V,Q)$ terá de ser sempre inferior a $d(V,W) + d(W,Q)$, ou, no máximo igual, quando W está na reta que une V e W. Ou seja, $d(V,Q) \leq d(V,W) + d(W,Q)$. Se dividirmos a última pela velocidade máxima na rede de transportes, obtemos $t'(V,Q) \leq t'(V,W) + t'(W,Q)$, ou seja $t(V) \leq c(V,W) + t(W)$.

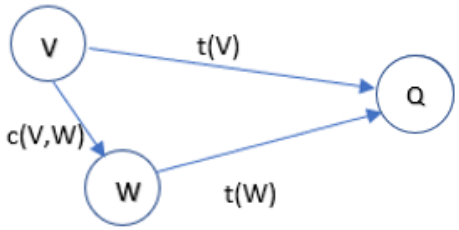


**Figure 8 - Consistency**

Está então provada a admissibilidade e garantia do ótimo para a aplicação do algoritmo A Estrela no que trata do tempo de viagem. Com a heurística consistente é então garantido que nenhum nó será processado mais do que uma vez. O algoritmo torna-se então equivalente a aplicar o algoritmo de Dijkstra com o peso das arestas modificados, como se verifica na linha 18 da figura 6, com a precaução de no final adicionar $h(S)$ ao custo total encontrado para o destino.

Estando a função de avaliação analisada podemos então passar à análise da complexidade.

## Análise Temporal e Espacial

Como podemos verificar pelo pseudocódigo do algoritmo de Dijkstra, figura 2, e pelo do A Estrela, figura 6, eles são quase idênticos, à exceção do cálculo do novo valor do custo entre o nó corrente e o próximo. Assim sendo, a complexidade espacial vai ser exatamente a mesma, $O(|V|)$, e a complexidade temporal do A Estrela vai depender essencialmente da eficiência, complexidade de cálculo e resultado, da heurística, linha 18. Analiticamente, não se pode tirar nenhuma conclusão sobre a complexidade temporal, visto que é ela também $\boldsymbol{O}\big((|\boldsymbol{V}| + |\boldsymbol{E}|) * \mathbf{log}|\boldsymbol{V}|\big)$, o overhead introduzido pelo cálculo da heurística não entra para a notação. A análise empírica do algoritmo será discutida na seção seguinte.

## Comparação dos Algoritmos

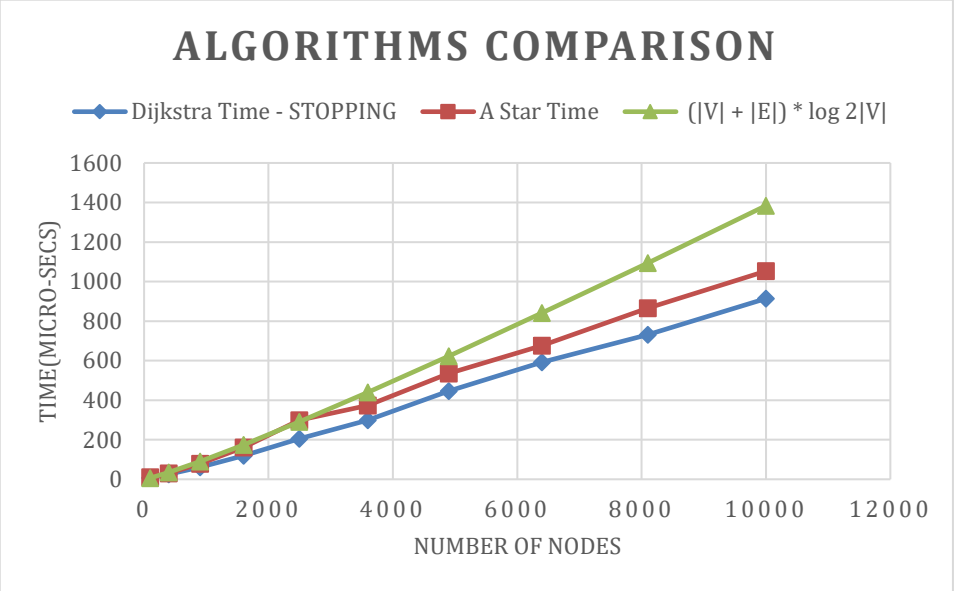


Figure 9 - Dijkstra and A Star - Sparse

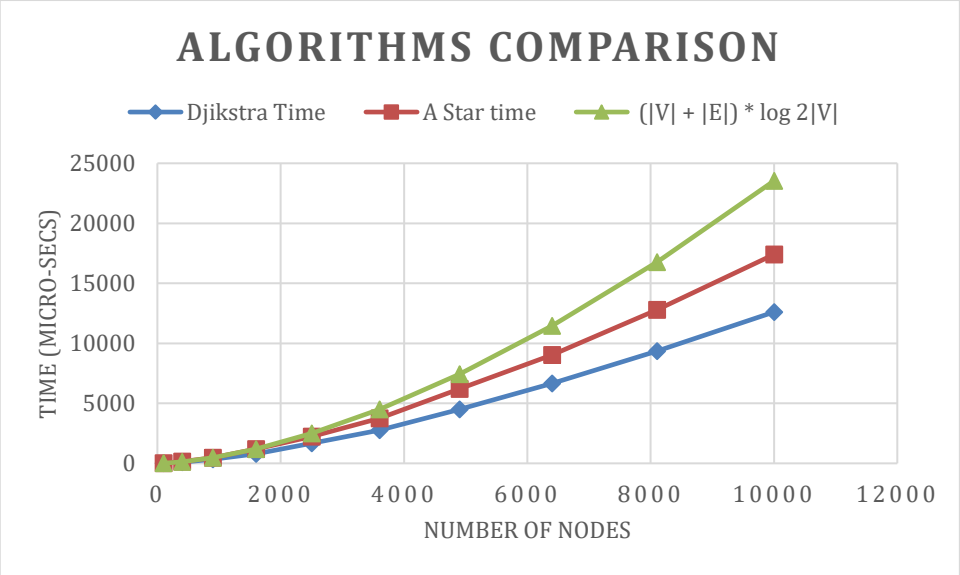


Figure 10 - Dijkstra and A Star - Dense

Como efetuado na análise empírica do algoritmo Dijkstra, aqui foram também gerados dois grafos aleatórios com um número variável de edges. Na figura 8 com $|E| \sim 4 * |V| - sqrt(|V|)$ e na figura 9 com $|E| \sim 2 * |V| * sqrt(|V|)$.

Podemos, então, verificar que o algoritmo Dijkstra teve uma performance ligeiramente superior e ambos os casos e ambos se encontram muito próximos na análise temporal feita. Note-se que a diferença entre os tempos de execução não é significativa e nenhuma conclusão empírica pode ser tirada, o que era esperado da análise temporal analítica.

Face à maior manipulação do algoritmo de Dijkstra e a sua generalização no contexto do problema (e.g., nenhuma restrição quando o valor a minimizar é o preço), optámos por usar este para responder aos pedidos do utilizador.

# Mapa Criado

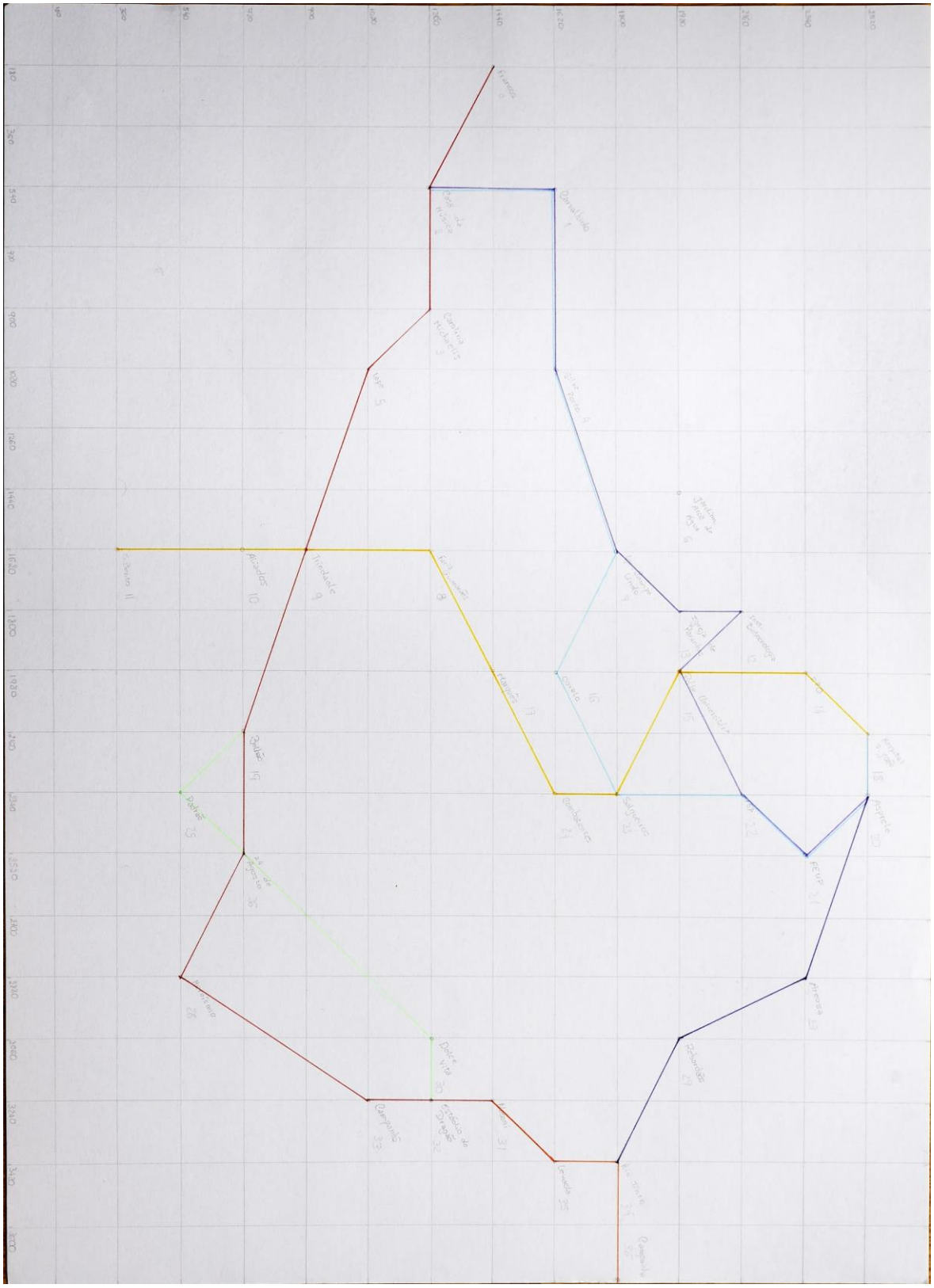

**Figure 11 – Mapa criado para a utilização da aplicação**

# Estrutura de Classes

Para a implementação do programa foram criadas três classes principais:

- Graph
- Node
- Edge

Todas estas classes são *template classes* pois isso torna-as mais genéricas e mais mutáveis para diferentes utilizações, modificações que possam ser necessárias no desenvolvimento do programa ou mesmo para reaproveitamento futuro do código.

A Classe ***Node*** representa um nó de um grafo. Cada Node contém a sua informação base e um conjunto de Edges. Também guarda em si, quando necessário, a o último transporte utilizado para chegar até ele.

A Classe ***Edge*** representa uma ligação entre dois nós, uma aresta. Cada Edge contém um certo peso e um apontador para um nó, que está no fim da Edge. É de salientar que cada edge tem também a informação que descreve o transporte por ela efetuado. Esta informação é essencial no cálculo do preço da viagem ou de uma viagem com um número limite de transbordos.

A Classe ***Graph*** representa um grafo. Um grafo tem um vetor de Nodes. Como cada Node tem um vetor de Edges (e cada Edge aponta por sua vez para o Node que tem no seu fim) um grafo da Classe Graph vai ter uma rede interligada de nós e arestas. Como seria de esperar, nesta classe encontram-se implementados todos os algoritmos.

Para facilitar a análise do código foi criada documentação com a ferramenta Doxygen para todas as funções existentes.

Além destas classes, foram criadas funções para lidar com todo o input, output e apresentação da informação. Também foi desenvolvido um menu que permite uma navegação fácil e intuitiva.

# Manual de compilação e requisitos

## Requisitos

### Java

Uma vez que o programa desenvolvido integra o GraphViewer, fornecido pelos docentes, é necessário que o Java JRE ou JDK esteja instalado no sistema, e/ou que haja uma referência na variável de ambiente PATH para o executável *java*.

### Compilador

Importa também referir que o projeto foi testado em Linux e em Windows, e compilado com o g++. Tivemos de mudar uma das *macros* de compilação no *source code* do *GraphViewer* fornecido. A macro em questão é *linux* que foi alterada para *__linux__* a fim de conseguir compilar o projeto. Uma vez que estas *macros* costumam variar de compilador para compilador, é possível que o projeto não seja compilável com outros compiladores, como *clang*.

### Standard C++

Por fim, importa referir que durante o desenvolvimento do projeto foi usado a *standard* C++14, apesar de não usarmos funcionalidades desta *standard*, era o *default* do IDE usado. Por isso, o projeto também pode ser compilado com a *standard* C++11.

### Ficheiros de input

No código do projeto, assume-se que todos os ficheiros de entrada devem estar no mesmo diretório que o executável. Esses ficheiros são:

- GraphViewerController.jar
- nos.txt
- arestas.txt

## Makefile

A fim de compilar o projeto, juntamente com o *source code* é fornecido um *GNU Makefile*. O executável resultante terá o nome *TripPlanner*.

Também foi definida a regra *clean* se necessário limpar os ficheiros objeto resultantes da compilação.

# Manual de Utilização

O programa tem um ecrã inicial onde é apresentado o nome da aplicação, "Trip Planner". De seguida, são apresentadas duas opções, uma para planear uma viagem, e outra para visualizar o mapa.

A visualização do mapa consiste em usar o *GraphViewer*[2] fornecido pelos docentes, que foi integrado no projeto, e que ilustra todas as estações existentes no nosso mapa e como se ligam entre si. Os nós, que representam as estações, são identificados pelo nome da mesma. As arestas têm cores diferentes, sendo que cada linha, de metro ou autocarro, terá uma cor única. As arestas estão devidamente identificadas pelo nome da linha. As ligações a pé entre as várias paragens foram omitidas, uma vez que o seu número é muito elevado tornando o mapa ilegível. A janela deverá ser fechada dentro do programa.

A opção de planear a viagem é onde o utilizador indica o local de partida e chegada, e define um critério. Neste menu, listam-se todas as paragens da rede intermodal criada, estando associadas a um ID que as representa. É então pedida a primeira paragem, o local de partida da viagem. O utilizador deve introduzir o número correspondente à paragem pretendida. O mesmo acontece quando é pedida a paragem final, o destino da viagem.

Seguidamente são listados os critérios que o utilizador pode escolher para planear a sua viagem:

- Número máximo de transbordos à escolha do utilizador
- Apenas trajetos sem troços a ser percorridos a pé
- O percurso com o menor preço [1]
- O percurso com o menor tempo de viagem

Finalmente o percurso final é apresentado no ecrã de duas maneiras diferentes, um **modo detalhado** em que é apresentado passo a passo o percurso (em cada paragem que transporte e linha apanhar) e o **modo resumido** em que é apresentada a sequência das paragens percorridas. É também apresentado o **preço** e a **duração total** da viagem.

Além disso, o programa irá abrir uma janela com o GraphViewer[2], ilustrando toda a rede intermodal, tal como explicado acima, mas marcando as paragens do percurso a vermelho.

**Notas:**

**1 -** Quando é selecionada a opção do trajeto com o menor preço é pedida a distância máxima que o utilizador pretende fazer a pé, pois, como fazer qualquer percurso a pé é gratuito, o caminho com o menor preço seria sempre o percurso completo a pé. Assim, o programa irá complementar a distância que o utilizador pretende caminhar com o percurso com o menor preço entre as linhas existentes.

**2 -** O Graph Viewer não deve ser encerrado na própria janela no botão de encerrar, mas antes pressionar qualquer tecla no terminal onde o programa está a ser executado.

# Dificuldades Encontradas no Decorrer do Projeto

Uma das grandes dificuldades do projeto passou pela criação de um mapa que exemplificasse bem o funcionamento do nosso projeto. Embora tenha sido fornecido um *parser* automático de informação sobre mapas, o formato dessa informação não ia de acordo com as nossas necessidades. Por outro lado, o desenvolvimento de um *parser* para obter a informação da forma desejada também nos pareceu fora do âmbito da cadeira e iria requerer tempo extra que podia afetar a conclusão do projeto. Para resolver tal problema, criámos um mapa nosso, que, a nosso ver, se enquadra bem no contexto do problema e permite testar diferentes casos.

Como já foi referido na seção do manual, o programa fornecido para a visualização de grafos criou bastantes problemas, em termos de compilação, e foram gastas várias horas a tentar perceber o motivo do erro. Uma vez descoberto, foi bastante simples de resolver.

Para além das acima referidas, não foram encontradas outras dificuldades durante o desenvolvimento do projeto.

# Conclusão

O objetivo deste trabalho era, sendo dado um problema, sermos capazes de fazer uma análise critica e concluir que se trata de um problema conhecido, para o qual existem algoritmos para o resolver. Neste caso, trata-se de encontrar o melhor percurso entre dois locais com base num critério a definir pelo utilizador, explorando as diferentes alternativas com os diferentes meios de transporte. Após alguma discussão, concluiu-se que o algoritmo do caminho mais curto de Djikstra e o A* poderiam ser uma solução. Tendo pelo menos três possíveis soluções, foi feita uma análise, descrita acima, de forma a concluir qual o melhor. Apesar de terem sido implementados ambos os algoritmos, o algoritmo que realmente ficou na versão final do projeto foi o Dijkstra, pois adaptava-se melhor às necessidades e especificidades do nosso tema. Não foi implementado o Dijkstra com *Fibonnaci Heaps,* pois após alguma pesquisa verificámos que não se adequava ao nosso tema. Foi também utilizado o *GraphViewer* para a visualização dos percursos.

Quanto à contribuição dos membros durante a realização do projeto, pode dizer-se que foi equivalente. Todos os membros contribuíram no código, relatório e documentação de forma igualitária sem existir grandes discrepâncias no esforço dedicado.

# Bibliografia

Thomas H. Cormen et al. Introduction to Algorithms, Third Edition. 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.