U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Projecto de Concepção e Análise de Algoritmos

TripPlanner: itinerários para transportes públicos (tema 11) - Parte 1

10 de Abril de 2018

Grupo F - tema 11, Turma 6:

Diogo Filipe da Silva Yaguas	up201606165@fe.up.pt
Joana Maria Cerqueira da Silva	up201208979@fe.up.pt
Ricardo Manuel Ferreira Teixeira	up201604911@fe.up.pt

Índice

1. Introdução

1.1. 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 respectivas 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.

Neste trabalho, pretende-se elaborar um planeador de itinerário multimodal, capaz de identificar a melhor combinação de modos de transportes para se realizar uma viagem desde um ponto de origem até ao ponto pretendido de destino. Os critérios a serem tidos em consideração para avaliar a qualidade do itinerário gerado poderão ser diversos, como, por exemplo, tempo total de viagem, preço dos bilhetes, distância percorrida, número de transbordos necessários, entre outros. Assim, o “caminho ideal” de uma estação de partida para outra pode assumir diversas formas, dependendo do critério escolhido.

1.2. Identificação e formalização do problema

Procuramos desde cedo compreender o enunciado proposto, e formalizar o problema, e o repartir em diversos outros problemas de menor complexidade.

Assim, identificamos as seguintes tarefas:

- A escolha do método de obtenção dos ficheiros com informações do mapa de testes a usar
- A extração da informação destes ficheiros com a informação
- A criação das classes que contêm a informação extraída
- A criação de um menu que serve como interface com o utilizador
- A criação dos grafos correspondentes aos diversos algoritmos de pesquisa com as informações recolhidas.
- A determinação do caminho ideal para um grafo usando o algoritmo adequado

1.2.1. Dados de entrada

Va - velocidade de um autocarro;
Vb - velocidade de um metro;
Vc - velocidade de um comboio;

Pa - preço por paragem de um autocarro;
Pb - preço por paragem de um metro;
Pc - preço por paragem de um comboio;

S - conjunto das estações sendo S(i) a estação com ID i. Cada estação é composta por:

- SID - ID da estação;
- XY - coordenada da estação (x, y);
- SN - nome da estação;

Src - Estação de partida;
Dst - Estação destino;

C - critério escolhido para o cálculo do percurso ótimo (escolha entre menor tempo de viagem, menor distância percorrida, menor preço da viagem ou menor número de transbordos);

L - conjunto das linhas de transportes, sendo L(i) a linha com ID i. Cada linha é composta por:

- LID - ID de uma linha, composto por:

    - N - número da linha;
    - type - tipo de transporte público (autocarro (a), metro (b) ou comboio (c));
- LS - sequência de IDs das estações por onde a linha passa, sendo LS(i) o ID da i-ésima estação.

A partir destes valores já é possível popular todas as estruturas definidas.

1.2.2. Dados de saída

G(V,E) - grafo não dirigido pesado, com pesos sempre positivos, composto por:

- V - vértices que representam estações e possivelmente paragens, com:
    - VID - ID do vértice, igual ao da estação/paragem que o criou;
    - Adj ⊆ E - conjunto de arestas que partem do vértice;
    - XY - coordenada da estação/paragem que o criou;
- E - arestas que representam ligações entre estações e paragens e entre paragens de estações diferentes. São compostas por:
    - w - peso da aresta, valor diferente dependendo do C (critério) escolhido;
    - ED - vértice de destino da aresta;

P - sequência ordenada de arestas que representam o caminho ideal para o critério escolhido, sendo P(i) a sua i-ésima aresta.

1.2.3. Restrições

1.2.3.1. Sobre os dados de entrada

LS = {sID | S(sID) ∈ S} É necessário que uma linha passe por estações que existem;

Src ∈ S;

Dst ∈ S;

Va, Vb e Vc têm de ser todas > 0;

Pa, Pb, Pc têm de ser todos ≥ 0. Se for igual a 0 todos os caminhos terão peso 0, tornando a pesquisa por preço irrelevante.

1.2.3.2. Sobre os dados de saída

Todos os v ∈ V têm um VID que corresponde a uma estação ou paragem;

Todos os e ∈ E também são pertencem a um Adj de um e um só v ∈ V;

Sendo i1 o primeiro elemento de P, É preciso que P(i1) ∈ Adj(V(Src));

Sendo ip o penúltimo elemento de P, É preciso que ED(P(ip)) seja o vértice v ∈ V que foi criado a partir de Dst, isto é, V(Dst).

Se |P| > 0, então é porque existe pelo menos um percurso possível de Src até Dst, se |P| = 0, então não há nenhum percurso entre Src e Dst, e o grafo G é não conexo.

1.2.4. Função objectivo

O percurso ótimo do problema requer minimizar o valor de:

- Minutos de viagem, caso se escolha o critério do menor tempo de viagem;
- Distância percorrida no mapa, caso se escolha o critério da menor distância percorrida;
- Preço a pagar, caso se escolha o critério do menor preço da viagem;
- número de transbordos, caso se escolha o critério do menor número de transbordos.

Logo, o percurso ótimo passa pela minimização da seguinte função:

$$f = \sum_{e \in P} w(e)$$

2. Solução Implementada

O primeiro passo foi então definir como seriam obtidas as informações para um mapa de testes. Optamos por não utilizar o *parser* disponibilizado, pois achamos que seria mais prático e elucidativo utilizarmos um mapa criado por nós próprios, e fazermos os nossos próprios ficheiros com o mapa de testes.

Com a forma de obtenção de dados definida, fomos definir qual seria a estrutura dos dados para poder criar os ficheiros com as informações necessárias.

Após muita discussão sobre a estrutura a ser implementada, optamos pela seguinte estrutura:

- Todos os nós geográficos do grafo são estações, onde autocarros, linhas de metro e comboios podem parar. Estas estações têm um ID, nome da estação, e um par de coordenadas geométricas (x, y).
- Cada linha de qualquer tipo de transporte que passar por uma estação, cria a sua paragem, que está a um certo tempo de viagem a pé do centro da estação, e para efetuar um transbordo, tem de se sair numa paragem da estação, andar até ao centro da estação e depois ir em direção à paragem desejada. Como a distância das paragens ao centro das estações é extremamente pequena comparando com as distâncias entre estações, consideramos que a distância geográfica entre a estação

e as suas respectivas paragens é zero (isto é, têm as mesmas coordenadas geográficas), no entanto, como se anda devagar de um lado para o outro, ainda existe algum peso a considerar quando o tempo de viagem for relevante.

- Cada paragem contém também o seu ID, estendido da estação, e o ID da linha que passa por ele, para acesso mais fácil a esta informação.
- As linhas são bidirecionais, e não têm diferenças de percurso de um lado para o outro. Assim, chegamos à conclusão de que o grafo será não dirigido.
- Cada estação tem um nó associado a ela, e cada paragem também. Estes nós são ligados entre si. São também ligados os nós das paragens que fazem parte da mesma linha, tal como indicado na imagem do enunciado do tema.
- O cálculo do caminho ótimo, uma vez que se trata de um grafo pesado com apenas números inteiros positivos, é calculado usando o algoritmo de ***Dijkstra***, um algoritmo ganancioso bastante conhecido para *path-finding* neste tipo de grafos.

Efectivamente, tendo sido definidos o formato dos ficheiros e a estrutura implementada, foram criados os grafos com os nós (objectos **Vertex**) que contêm todas as informações necessárias para o funcionamento dos algoritmos, como um ID que refere a estação ou paragem de onde vem e um vetor com todas as ligações a outros nós, ligações estas (objectos **Edge**) que contém um apontador para o nó de destino, e o peso a ser considerado pelo algoritmo implementado.

3. Estrutura e Casos de Utilização

3.1. Classes

As classes criadas que achamos relevantes à compreensão do código são as seguintes:

Manager - *Singleton class* (todos os membros são estáticos e construtor privado de forma a não permitir a sua instanciação) que contém a informação do programa em execução, como os grafos, as estações, paragens e linhas, bem como funções para inicialização do programa.

Station - classe que representa o conceito de Estação, contendo um ID, o nome da estação, um par de coordenadas, um vetor com as suas paragens e um vetor com as ligações que as suas paragens têm com outras estações.

Stop - classe que representa o conceito de Paragem, contendo um ID, uma struct com o número e o tipo de linha que passa por ela, e o tempo que demora uma pessoa a andar da paragem ao centro da sua estação.

Link - classe que representa o conceito de ligação, semelhante aos objectos **Edge** do grafo, mas em vez de ser entre paragens, é entre as estações em si. Contém um apontador para a estação de destino, e o ID da linha a qual esta ligação se refere. Existe para facilitar procuras de ligações entre stations sem ter de pesquisar o grafo.

Line - classe que representa o conceito de uma linha. Contém o seu ID e o vetor com os IDs das paragens por onde passa.

3.2. Ficheiros

stations.txt - ficheiro que contém a informação relativa às estações. Cada linha é uma estação, e tem formato como formato: ID;X;Y;Nome. Para adicionar uma estação ao programa, basta adicionar uma linha com este formato ao ficheiro.

lines.txt - ficheiro que contém as linhas dos diversos transportes públicos. Cada linha do ficheiro corresponde a uma linha a adicionar ao iniciar o programa. O formato de uma linha deve ser: NL;Tipo;E#1;T#1;E#2;T#2; etc… sendo:

NL - número da linha;

Tipo - caracter, sendo que ‘a’ representa linha de autocarro, ‘b’ linha de metro, e ‘c’ linha de comboio;

E#x - ID da estação número x da linha;

T#x - Tempo que demora andar da paragem desta linha até ao centro da estação referida anteriormente.

3.3. Algoritmos

O nosso programa tem 4 tipos de pesquisa do caminho ótimo:

- Tempo total de viagem
- Preço dos Bilhetes
- Distância Percorrida
- Número de transbordos

No entanto, todos os tipos são para a minimização das suas respectivas características principais.

Para o tempo total de viagem, o grafo inteiro tem de ser usado, e escolhemos usar o algoritmo de ***Dijkstra*** para fazer a pesquisa.

Para a distância percorrida, o grafo pode ser reduzido, pois linhas diferentes que passam pelas mesmas estações não precisam de ser duplicadas, já que a distância entre dois pontos é sempre a mesma, independentemente da linha que a percorre. Assim, o grafo apenas irá conter como nós o número de estações e como ligações no máximo uma por cada destino de ligação. Isto reduz muito o número de nós que os algoritmos têm de processar, pois já não existem os nós que representavam as paragens, logo esta otimização afecta mais os grafos muito densos em linhas (com muitas linhas a passar por cada nó) do que grafos menos densos.

Escolhemos usar o algoritmo de ***Dijkstra*** para fazer a pesquisa. Uma opção melhor teria sido a de se usar o algoritmo A* devido à menor quantidade de nós que costuma processar para o mesmo resultado, levando a tempos de execução médios mais baixos, no entanto não tivemos a possibilidade de o implementar por falta de tempo.

Para o preço dos bilhetes, nós assumimos que o usuário dos transportes públicos paga uma certa quantidade por paragem.

Inicialmente pensamos que o tipo de transporte não teria diferença no preço, e se fosse esse o caso então o grafo a usar seria o mesmo da distância percorrida, com a diferença de os pesos das ligações serem todos iguais a 1, e seria feita uma pesquisa em largura para chegar ao número mínimo de paragens percorridas entre a origem e o destino.

No entanto, escolhemos ser mais flexíveis, e permitir preços diferentes entre tipos diferentes de transportes. Assim, tivemos de abandonar o grafo anterior para este caso e tivemos de usar o grafo inteiro, sendo que os pesos das arestas dependem apenas do tipo de transporte. Para fazer a pesquisa, usamos o algoritmo de ***Dijkstra***.

Para o número de transbordos, tivemos em mente utilizar uma pesquisa em profundidade, de forma a obter o nó de destino com o mínimo de mudanças de linha, no entanto, dada a nossa estrutura, em que entre duas estações consecutivas podem existir diversas linhas a passar entre elas, não conseguimos ver um método de obter um grafo capaz de correr uma pesquisa em profundidade.

Acabamos por optar por utilizar o grafo inteiro, sendo que as ligações entre nós relativos a uma estação e nós relativos às suas paragens teriam peso 0,5 (como para

mudar de linha é necessário passar por 2 ligações entre a estação e uma paragem, 2 x 0,5 = 1, que é o peso de mudar de linha) e as ligações entre nós de paragens diferentes teriam peso 0. Assim, aplicamos o algoritmo de ***Dijkstra*** para obter um caminho ótimo, e subtraímos 1 à distância do nó de destino, para obtermos o número de transbordos ocorridos (pois existem 2 ligações entre o a estação de origem/destino e a paragem de onde se sai ou entra da estação, cada uma com peso 0,5).

3.3.1. Algoritmo de Dijkstra

Para o cálculo do percurso ótimo, devido às razões expostas acima, acabamos por usar o algoritmo de ***Dijkstra*** para estes cálculos, pois tratam-se de grafos pesados com apenas pesos positivos.

Em termos de análise do complexidade temporal e espacial este algoritmo apresenta:

- Complexidade Temporal: O((E + V) * log(V)) onde E representa o número de ligações entre nós e V o número de nós do grafo.
- Complexidade Espacial: O(V) onde V representa o número de nós.

Este algoritmo é um dos mais eficientes dos leccionados para este tipo de grafos, isto é, grafos com pesos das arestas sempre positivos.

O pseudocódigo do algoritmo é o seguinte:

```
function dijkstra(Graph, source):  // Graph=(Vertex,Edge),
                                   // source is a Vertex
for each vertex v in Graph do
     dist(v) = infinity
     path(v) = undefined
dist(source) = 0     // Distance to source
Q = { }  // empty min-priority queue
INSERT(Q, (source, 0)) // inserts source with key 0
while Q is not empty do
     v  EXTRACT-MIN(Q) // greedy, node in Q with the smallest
dist(v)
for each neighbour w of v do
     if dist(w) > dist(v) + weight(v,w) then
          dist(w) =  dist(v)+ weight(v,w)
          path(w) = v
          if w is not in Q then  // old dist(w) was infinite
               INSERT(Q, (w, dist(w)))
          else
               DECREASE-KEY(Q, (w, dist(w)))
```

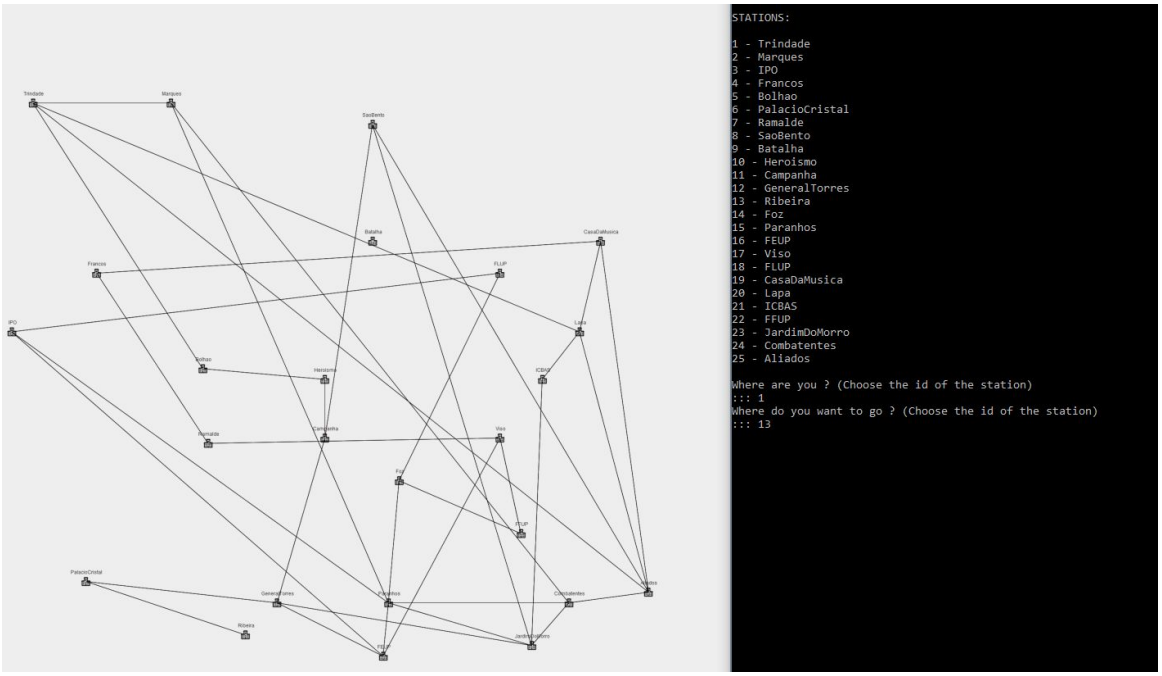
Gostaríamos de ter implementado otimizações, como a de fazer a pesquisa começar do nó de origem e do nó de destino, pesquisando o grafo de forma intercalada entre estes dois pontos, o que leva a uma melhoria moderada do tempo de execução, ou o uso de Fibonacci Heaps, que reduziria a complexidade temporal para O(V * log(V)), em que V é o número do nós do grafo. No entanto, não tivemos a possibilidade de implementar estas melhorias.

3.4. Programa

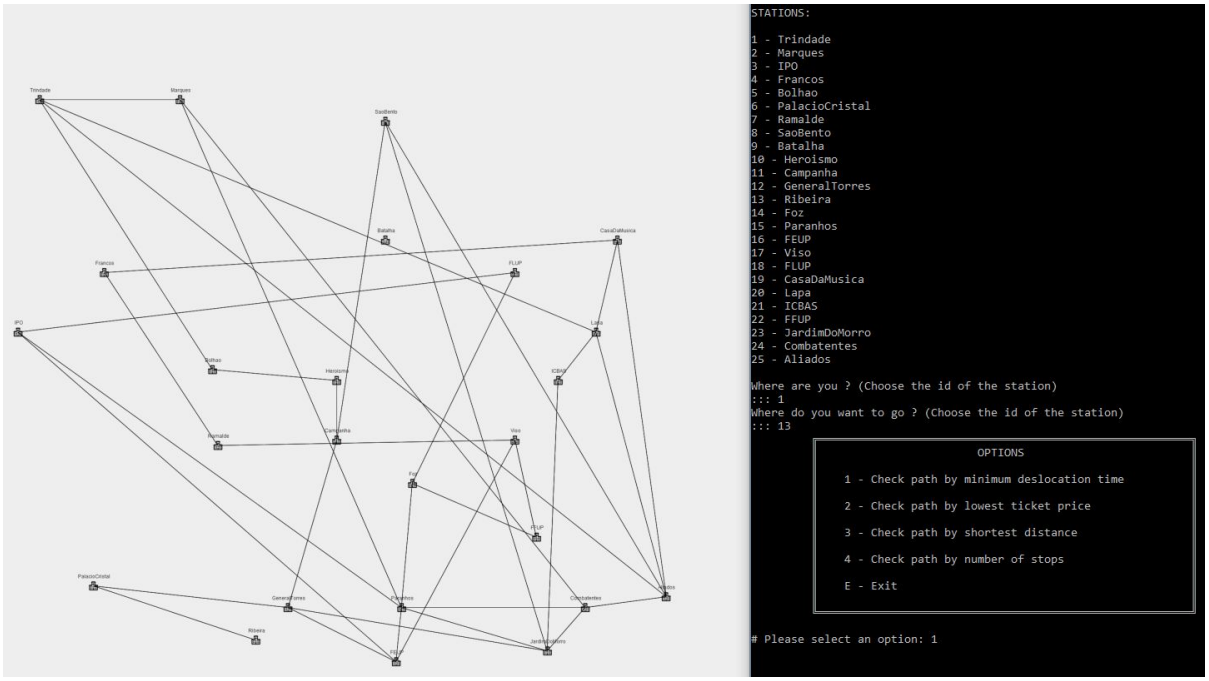
O programa começa por processar a informação do mapa e das linha que está dentro dos ficheiros de texto, cria os grafos necessários e a visualização do grafo de estações com a informação necessária para a identificação de cada aresta e nó.

Após esta fase estar concluída, são listadas as estações carregadas, acompanhadas dos seus respectivos IDs, sendo de seguida pedido ao utilizador o ID da sua estação de partida e da estação de chegada através da consola.

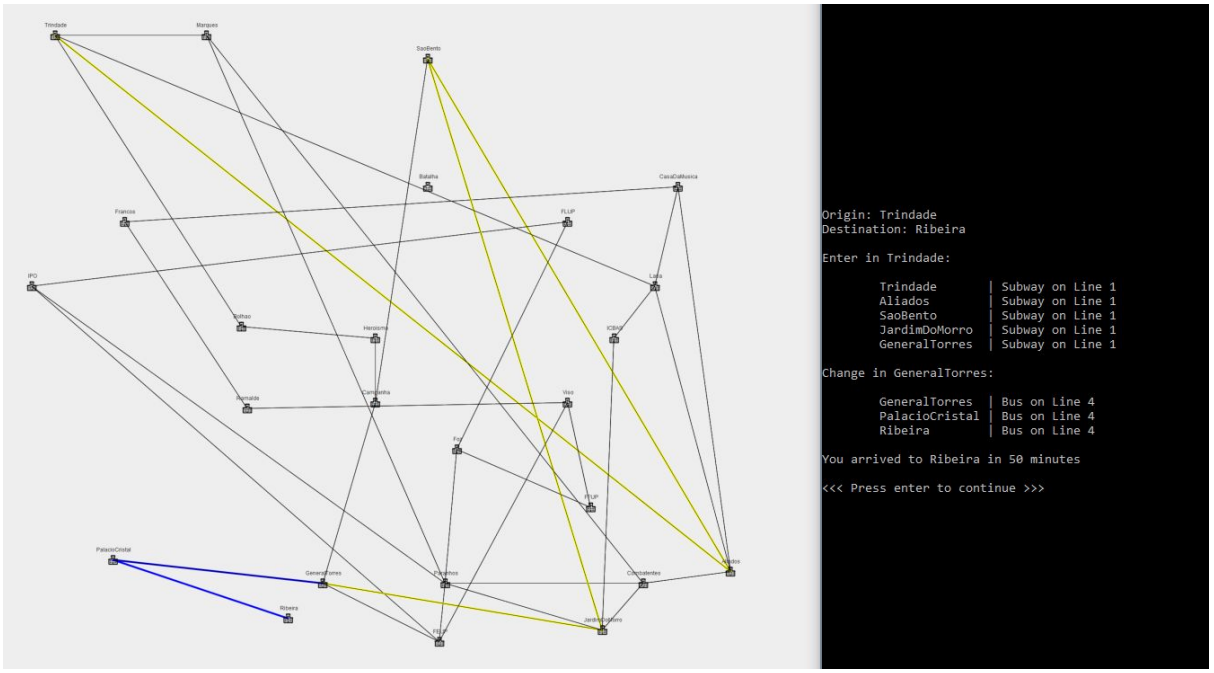
O seguinte ecrã é um exemplo do que pode aparecer:



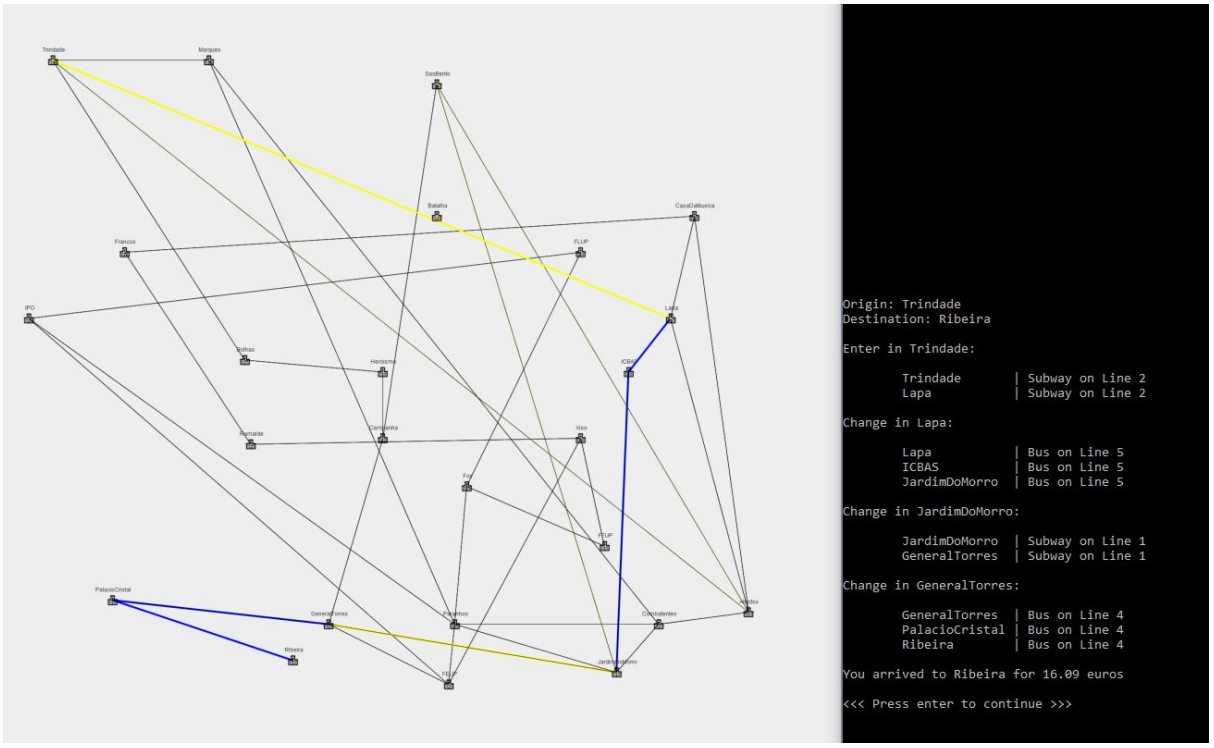
Após terem sido identificadas as estações de partida e destino, aparece o menu principal que permite a escolha de qual o método de pesquisa pretendido, tal como demonstrado na seguinte imagem:



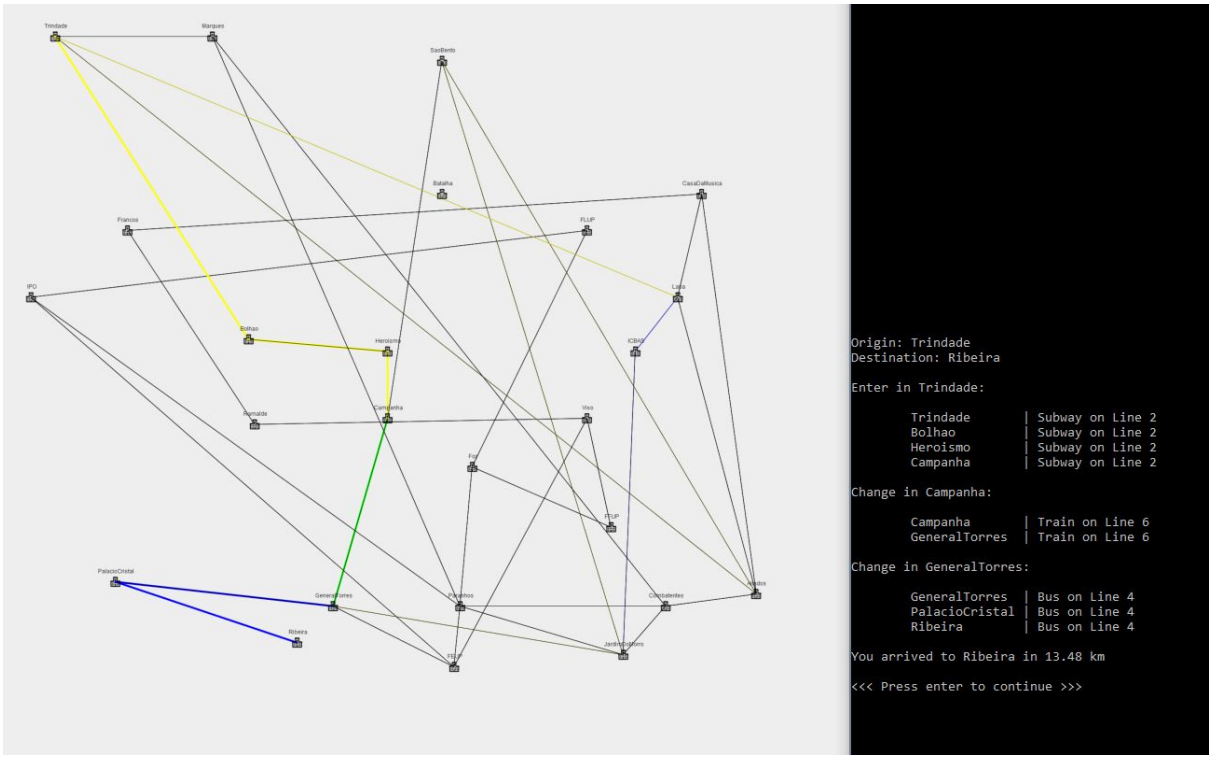
Após a pesquisa de um percurso ótimo para a opção escolhida no menu ter sido feita, é mostrado o resultado, e qual o caminho obtido, como mostrado na seguinte imagem:



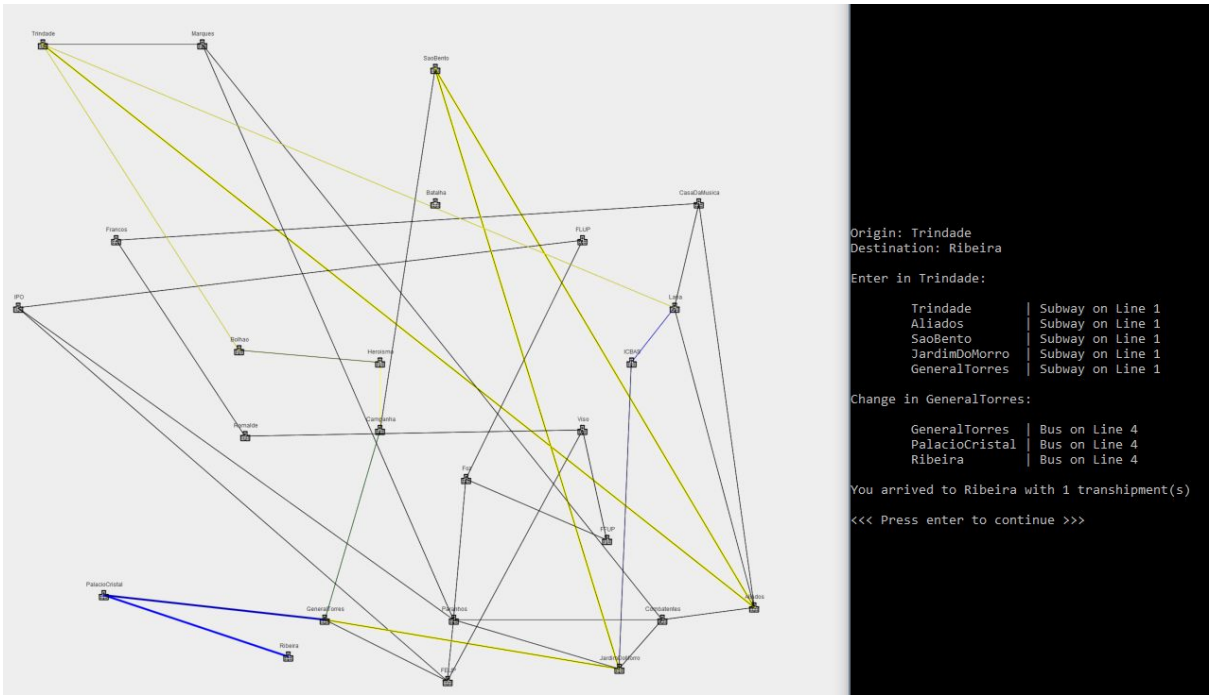
Este foi o resultado de uma pesquisa pelo tempo menor de viagem. O resultado para uma pesquisa entre as mesmas estações mas para o preço menor é:



Para uma distância percorrida menor:

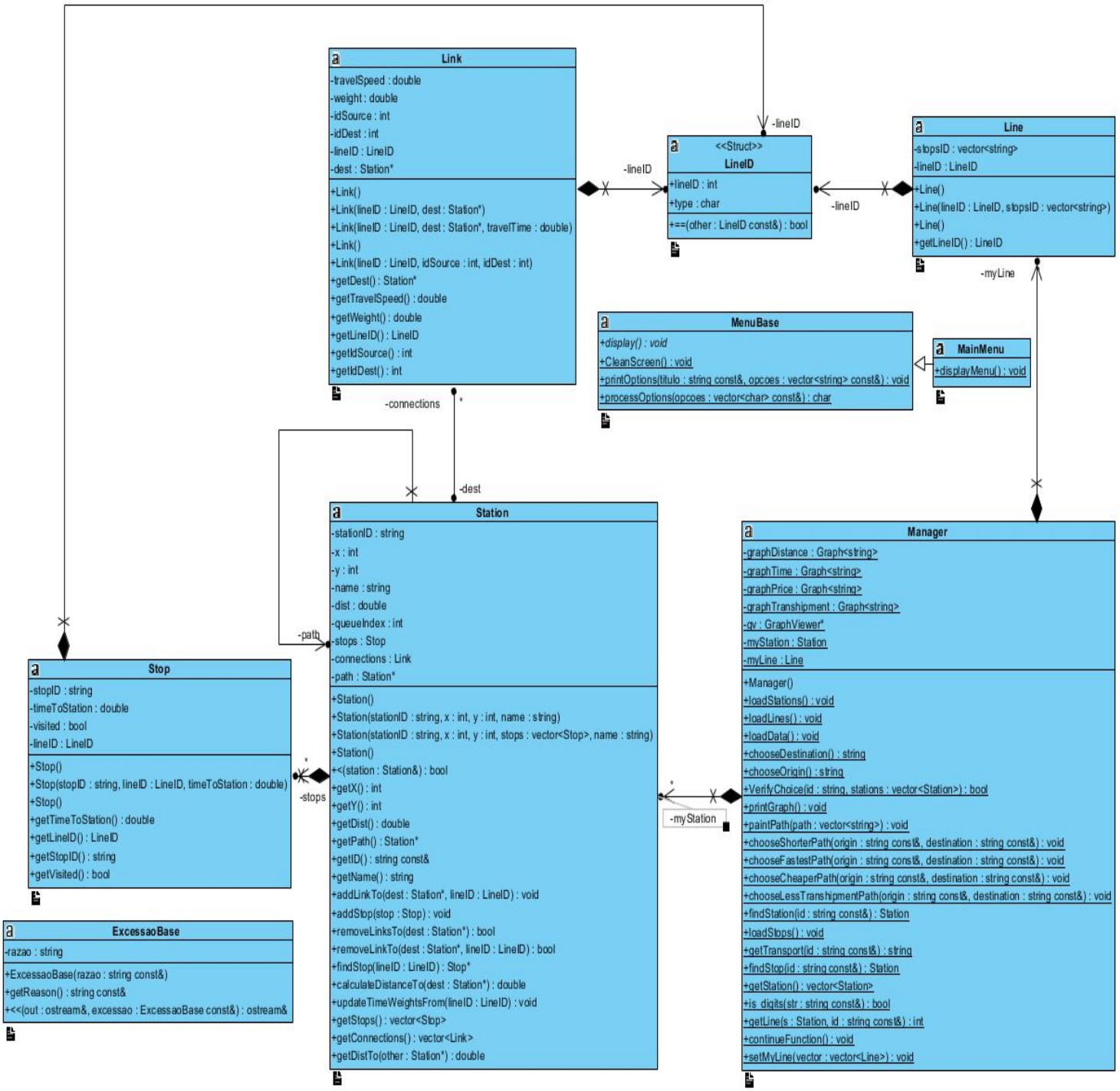


Finalmente, para o menor número de transbordos:



É importante notar que as linhas têm cores diferentes, amarelo para metro, azul para autocarro e verde para comboio.

3.5 Diagrama de Classes



4. Dificuldades

Durante a realização deste projeto as principais dificuldades foram no processo de desenho da estrutura do trabalho e na compreensão do enunciado, assim como a compreensão do funcionamento dos software apresentado (*GraphViewer*).
Houveram também alguns mal entendidos devido a pequenas diferenças nos pressupostos do funcionamento do nosso programa, mas que foram sendo discutidos e resolvidos ao longo do trabalho.

Quanto ao nível da programação não houveram grandes dificuldades, excepto quando eram problemas derivados dos mal entendidos acima mencionados.

Ao usarmos o *GraphViewer* tivemos diversos problemas de compilação e de setup devido ao uso de diferentes IDE's ao longo do projecto, sendo que o projecto foi criado em *Eclipse* e mais tarde importado para *Visual Studio* e *CLion*, onde o projeto continuou a ser desenvolvido. Isto levou a que foste gasto muito mais tempo do que esperado para poder ter o projecto a funcionar corretamente.

5. Distribuição do Trabalho

Todos os membros do grupo se esforçaram igualmente na compreensão e estruturação inicial do problema e trabalharam de uma forma empenhada ao longo da realização do projeto.

Diogo Yaguas

- Extração de informação dos ficheiros
- Implementação da classe Manager
- Implementação das funções que procuram os caminhos ótimos
- **Percentagem: 33.3%**

Joana Silva

- Programação para o software GraphViewer
- Implementação do menu de escolhas e das funções de verificação do input do utilizador
- Implementação da documentação
- **Percentagem: 33.3%**

Ricardo Teixeira

- Implementação das classes da estrutura do programa
- Implementação do algoritmo de *Dijkstra*.
- Redação do relatório
- **Percentagem: 33.3%**

6. Conclusão

Este projeto foi bastante educativo, e exigido da nossa parte uma boa compreensão das estruturas dos grafos e dos algoritmos de pesquisa nas mesmas, mas também um conhecimento de como os implementar.

Mas para além disto, também melhoramos também as nossas habilidades de trabalho em grupo.

Concluímos assim que os objetivos pretendidos com este projeto de grupo foram atingidos, uma vez que cada elemento do grupo tem um domínio sobre as matérias lecionadas pela unidade curricular, e é capaz de os aplicar numa componente prática.