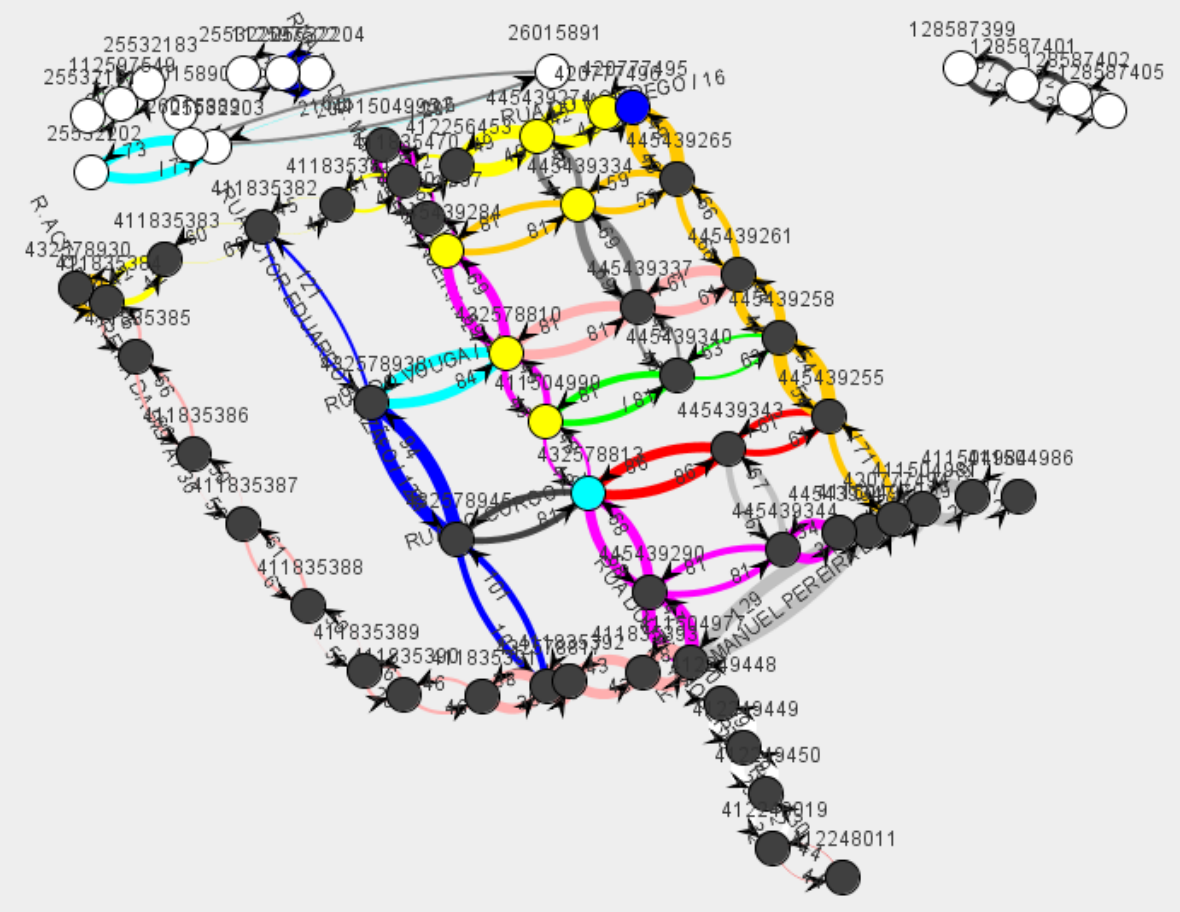
Trabalho de grupo 1 – grupo 8 – tema 9
Determinação do melhor caminho entre dois pontos numa rede viária



Trabalho realizado por:

Carlos Miguel Correia da Costa

ei09097 – FEUP

ei09097@fe.up.pt / carlosmccosta@live.com.pt

e entregue a 14/05/2010

Introdução

O presente trabalho tem por objectivo criar um programa capaz de determinar o melhor caminho entre dois nós de um dado grafo orientado pesado (mapa), sendo que esse caminho pode ser determinado tendo em conta a distância ou tempo que demora a percorrer.

Como tal, foi implementado o algoritmo de Dijkstra, visto que foi aquele que foi abordado nas aulas e aquele que fornece sempre o melhor caminho (caso ele exista).

Foi ponderada a implementação do algoritmo A*, ou o de Dijkstra com heurística para acelerar a resolução (por exemplo usando as sugestões que são dadas nos slides das aulas teóricas da semana 6), mas após discussão com o professor foi decido implementar apenas o algoritmo de Dijkastra, porque a implementação destes "extras" iria dar muito mais trabalho e tempo a fazer debug, e seria pouco valorizada (por não ter sido dada nas aulas, e por não ser o pedido).

Por outro lado, apesar de ser uma subtileza, o algoritmo de Dijkstra faz o que é pedido no enunciado, enquanto os outros mais eficientes não o fazem. Ou seja, determina o melhor caminho entre dois nós, enquanto o A* e o de Dijkastra com heurística, poderiam não fazer, porque por exemplo se os dois nós estiverem muito longe e o caminho envolver uma rota que sai muito do caminho que seria de esperar (por exemplo em zonas rurais com poucas estradas, que obrigam a ir muito longe para um sitio relativamente perto), ou outros algoritmos poderiam não encontrar um caminho quando na realidade ele existe (ou seja a heurística tinha excluído o caminho "pouco provável", mas que era dos únicos existentes.

Descrição da implementação

Para a construções desta aplicação foram implementadas as seguintes classes:

Map, Node, Road, SubRoad, Exceptions, GraphViewerColors bem como um namespace com uma biblioteca de funções para a interacção com o utilizador (utils).

De seguida passa-se à descrição das principais estruturas de dados de cada classe, e da funcionalidade de cada uma dessas classes.

1) Map

Classe que faz a gestão de um mapa, fornecendo toda a implementação necessária para carregamento, análise e visualização do grafo que representa o mapa, bem como o algoritmo de Dijkstra para a determinação do melhor caminho entre dois nodes.

O mapa usa como estruturas de dados o container set da STL, para acelerar a pesquisa dos elemento do mapa, visto que a pesquisa no set é logarítmica, (e como tal muito mais eficiente do que a pesquisa em vectores, que seria linear).

Foi ponderada a utilização de hashTables, que neste caso poderiam ter tempo de pesquisa quase sempre constante, porque a hashkey seria o ID, que supostamente é único.

Mas como a hashtable não faz parte da STL, ocupa mais espaço que o set e é mais dispendiosa quando precisa de fazer o resize por já não ter mais espaço (visto que faz o rehash a todos os elementos) e ainda porque tendo em conta que o programa depende de fontes externas, (ficheiros com os nodes, roads e subroads que podem ter nodes ou subroads com o mesmo ID quando não deviam), foi optado por usar o set, visto que garante que não existem dois elementos com o mesmo ID.

Para a ordenação dos sets são usados functors de forma a controlar como o set é indexado. Ou seja uma struct com o operator() lá dentro.

Portanto, o map é constituído por:

```
/// set que armazena os nós do mapa por ID.
set <Node*, Node::functorToOrderNodeSet> mapNodesSet;

/// set que armazena as ruas do mapa por ID (usado para acelerar a pesquisa interna aquando do carregamento do mapa).
set <Road*, Road::functorToSortRoadsByID> mapRoadsSet;
```

```
    /// set que armazena as ruas do mapa por nome da rua (usado para acelerar a pesquisa do nome da rua que o utilizador fornece).
    set <Road*, Road::functorToSortRoadsByName> mapRoadsSetSortedByName;

    /// set que armazena as subroads do mapa por ID.
    set <SubRoad*, SubRoad::functorToSortSubRoadsByID> mapSubRoadsSet;


    /// string que descreve o Map em questão.
    string description;

    /// Variável estática usada para permitir usar várias janelas do GraphViewer (ou seja, é a porta que o GraphViewer irá usar na comunicação entre a implementação em C++ e Java).
    static int graphViewPort;

    /// Vector com os nós que fazem parte do melhor caminho determinado pelo algoritmo de Dijkstra segundo o nodeID / roadName dado pelo user.
    Este vector tem os nodes do fim do caminho para o inicio.
    vector <Node*> currentPath;
```

2) Node

Classe que representa um nó no grafo, com as ligações que esse nó tem com os restantes nós do mapa, bem como informações para possibilitar a sua correcta visualização no GraphViewer.

É constituída por:

```
    /// Contador dos números de nós criados até ao momento
    static unsigned int nodeIDCounter;

    unsigned int nodeID;
    int latitude;
    int longitude;
    int longitudeProjection; //Posição no x na janela do GraphViewer
    int latitudeProjection;  //Posição no y na janela do GraphViewer

    /// distância deste Node ao Node inicial
    unsigned int cumulativeDistanceToSourceNode;
    //tempo que demora a chegar deste Node ao Node inicial
    unsigned int cumulativeTimeToSourceNode;

    /// Nó que está num caminho que chega ao nó de inicio da pesquisa
    Node* pathToSourceNode;

    /// Nó que indica se já foi visitado (usado no algiritmo de Dijkstra)
    bool visited;

    string nodeColor;
    string nodeLabel;
```

```
    /// vector que contém as ligações (SubRoads), que este Node tem com os restantes Nodes do mapa
    vector <SubRoad*> conectedSubRoads;
```

3) Road

Classe que representa uma rua no mapa.

Cada rua é constituída por subruas, que correspondem a subsegmentos da rua no mapa.

É constituída por:

```
/// Contador dos números de Roads criados até ao momento
static unsigned int roadIDCounter;

unsigned int roadID;
string roadName;
bool oneWayRoad;        //Flag que indica se esta rua é de um sentido

/// set com as subruas que constituem esta rua
set <SubRoad*, SubRoad::functorToSortSubRoadsByID> StreetSubRoads;
```

4) SubRoad

Classe que representa uma parte de uma rua, ou seja, uma ligação entre dois Nodes.

É constituída por:

```
/// Contador do numero de SubRoads criadas até ao momento
static unsigned int SubRoadIDCounter;

unsigned int subRoadID;

unsigned int RoadID;
unsigned int sourceNodeID;
unsigned int destinationNodeID;

int lenght;
int timeToTravel;

// apontadores para os nós de inicio e fim da SubRoad para melhorar a performance
Node* sourceNode;
Node* destinationNode;

string subRoadColor;
string subRoadLabel;

/// flag que indica se nesta SubRoad foi colocada o nome da rua no GraphViewer
bool labelIsStreatName;
```

5) GrapViewerColors

Struct usada para manter uma base de dados das cores que o GraphViewer permite.

É constituida por:

```
/// enumeração com as cores que o GraphViewer suporta
enum GrapViewerColorsEnum { BLACK, BLUE, CYAN, DARKGREY, GRAY, GREEN,
LIGHTGREY, MAGENTA, ORANGE, PINK, RED, WHITE, YELLOW };

/// vector com as strings respectivas de cada de cada uma das cores que
estão na enumeração
static vector<string> grapViewerColorsStrings;
```

6) Exceptions

Classe que contem as excepções que são usadas no programa para tratar de anomalias na execução do código.

É usado sobretudo para tratar de inputs inválidos do utilizador.

Por exemplo quando o utilizador introduz o nome de um ficheiro para ler no programa que não existe no disco, ou quando introduz o nome a dar a um ficheiro onde serão guardados dados do programa, mas cujo nome já está em uso por outro ficheiro naquela pasta…

Notas sobre a implementação do algoritmo de Dijkstra

O algoritmo de Dijkstra é um "dos mais rápidos" algoritmos conhecidos para resolver o problema de encontrar o caminho mais "curto" num grafo orientado pesado (cujos pesos das arestas sejam positivos), dando um nó inicial. No entanto a sua complexidade é de O(|E|*log|V|), visto que está a usar uma binary heap para a fila de prioridades que usa.

Foi ponderada a implementação da fila de prioridades usando uma Fibonacci heap, mas como a sua implementação é relativamente difícil e visto que ainda não está na STL, foi decidido usar binary heaps, que já estão incluídas na STL e amplamente testadas.

Foi ainda analisada a possibilidade de usar a libraria de bibliotecas Boost, que tem uma implementação de Fibonacci heaps em \boost\pending\fibonacci_heap.hpp, mas como não foi dito que era possível usar librarias externas e visto que parece ainda haver alguns bugs na implementação presente na libraria do Boost, foi decido manter o uso de binary heaps.

Notas sobre a implementação

No namespace utils estão as funções genéricas de comunicação da classe Map com a interface CLI com o utilizador.

No ficheiro defs.h estão os defines do programa.

No main.cpp é onde está implementado o menu para fazer a interface CLI com a API da classe Map.

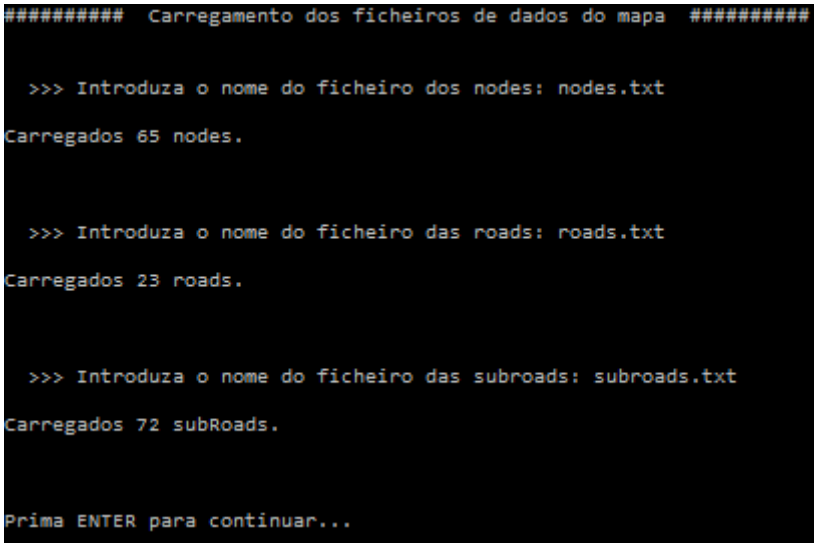
Em relação à apresentação feita no dia 13/5, foi acrescentada a possibilidade de guardar e carregar o caminho calculado pelo algoritmo de Dijkastra, e foi corrigido a transformação das projecções no x e y, que o OpenStreatMapsParser dava, para coordenadas na janela do GraphViewer de forma a aparecerem sempre os nós todos na janela, sem ser necessário estar a fazer zooms e deslocações do grafo.

Para além disso foi acrescentada a possibilidade de mostrar o antigo caminho determinado (1 – Mostrar mapa) e de limpar o mapa, pondo-o com as cores iniciais e reinicializando os nós com os valores por default (8 – Limpar mapa).

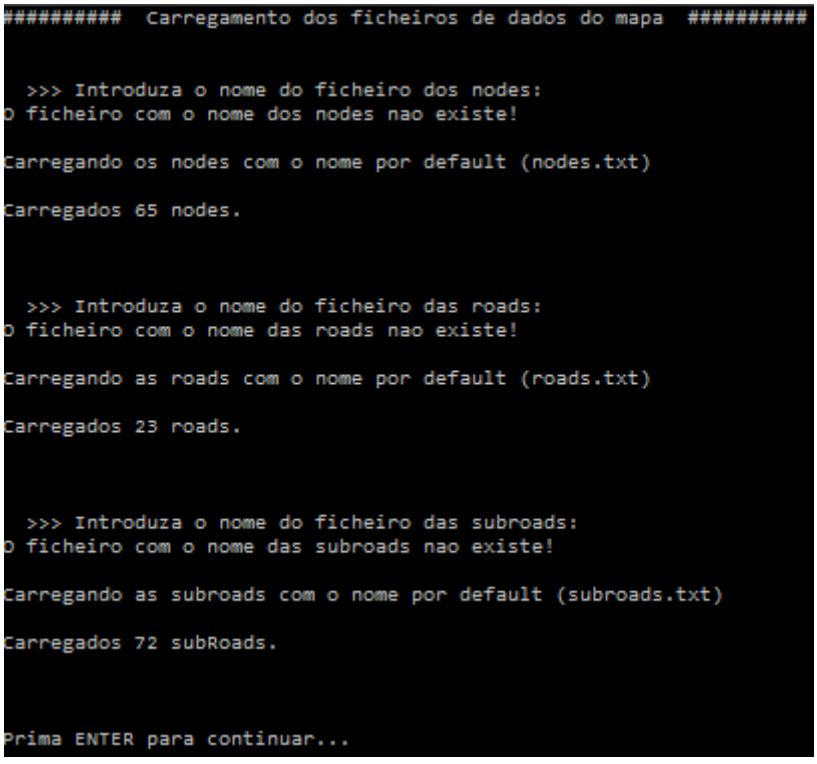
Descrição da interface com o utilizador

Nota: não são usados acentos porque na consola fica desformatado…

O programa começa com o carregamento dos ficheiros onde estão guardados os nodes, roads, e subroads, tal como é mostrado em baixo.



Caso o utilizador introduza um nome de um ficheiro que não exista o programa tenta carregar o ficheiro com o nome por default (que são os que estão mostrados na imagem em baixo).



Caso um dos ficheiros que são pedidos não exista (nem com o nome dado nem com o de default), o programa encerra, visto que não possui uma componente fundamental para funcionar.

```
##########  Carregamento dos ficheiros de dados do mapa  ##########


  >>> Introduza o nome do ficheiro dos nodes: naoexiste.txt
O ficheiro com o nome dos nodes nao existe!

Carregando os nodes com o nome por default (nodes.txt)
O ficheiro com o nome por default dos nodes tambÚm nao existe!


Encerrando o programa...

Prima ENTER para continuar...
```

De seguida é perguntado se o utilizador pretende usar a pesquisa dos nós de início e fim pelo nome da rua (inserindo N), ou se pretende introduzir os IDs dos nós que pretende seleccionar (inserindo S).

```
Pretende usar a pesquisa usando o ID do no? (N para usar a pesquisa por nome da rua)
(S/N):
```

Em termos de visualização da lógica do programa é indiferente qual delas escolher.

Se escolher pesquisar por ID tem a certeza que será o nó que escolheu.

Se escolher por nome de rua o algoritmo pega no sourceNode da primeira SubRoad que estiver na Road com o nome introduzido.

Em ambos os casos se não for encontrado um dos nós (início ao fim), retorna para o menu.

Visto que o enunciado dá a entender que o objectivo é usar o nome da rua, vou usar a pesquisa por rua, nas restantes visualizações.

Apenas vou mostrar como é a interface com IDs.

Por exemplo para a opção 2 do menu:

```
Introduza o ID do no de inicio: 666
O ID do no introduzido nao existe!

Prima ENTER para continuar...
```

```
Introduza o ID do no de inicio: 432578813

Introduza o ID do no de fim: 420777495
```

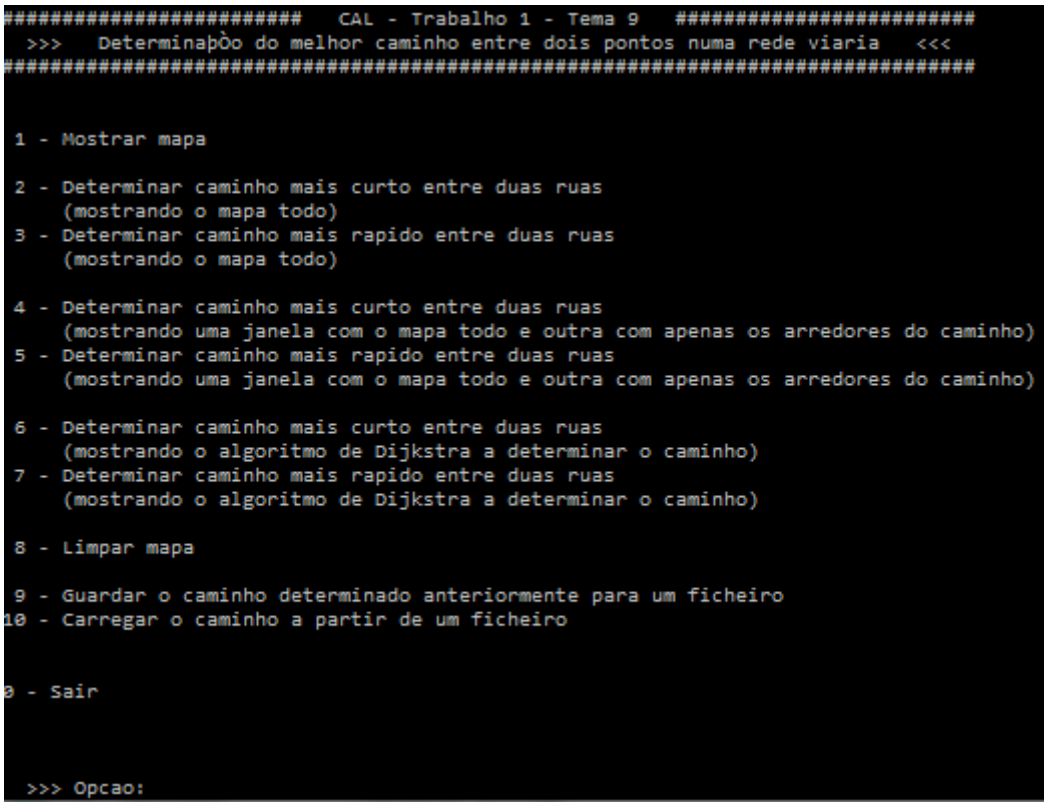
Após determinar o caminho entre os nodes dados é mostrado quantos nós esse caminho tem:

```
Caminho com 8 nodes.

Prima ENTER para continuar...
```

Na restante parte do relatório serão usados os ficheiros do programa com o nome por default (nodes.txt, roads.txt, subroads.txt), que correspondem aos mapas no ficheiro "Mapa tamanho XS" disponibilizado na página da disciplina.

http://paginas.fe.up.pt/~rossetti/rrwiki/doku.php?id=teaching:1011:cal:osmparser

Menu do programa, em que o user selecciona o que quer fazer introduzindo um dos números.



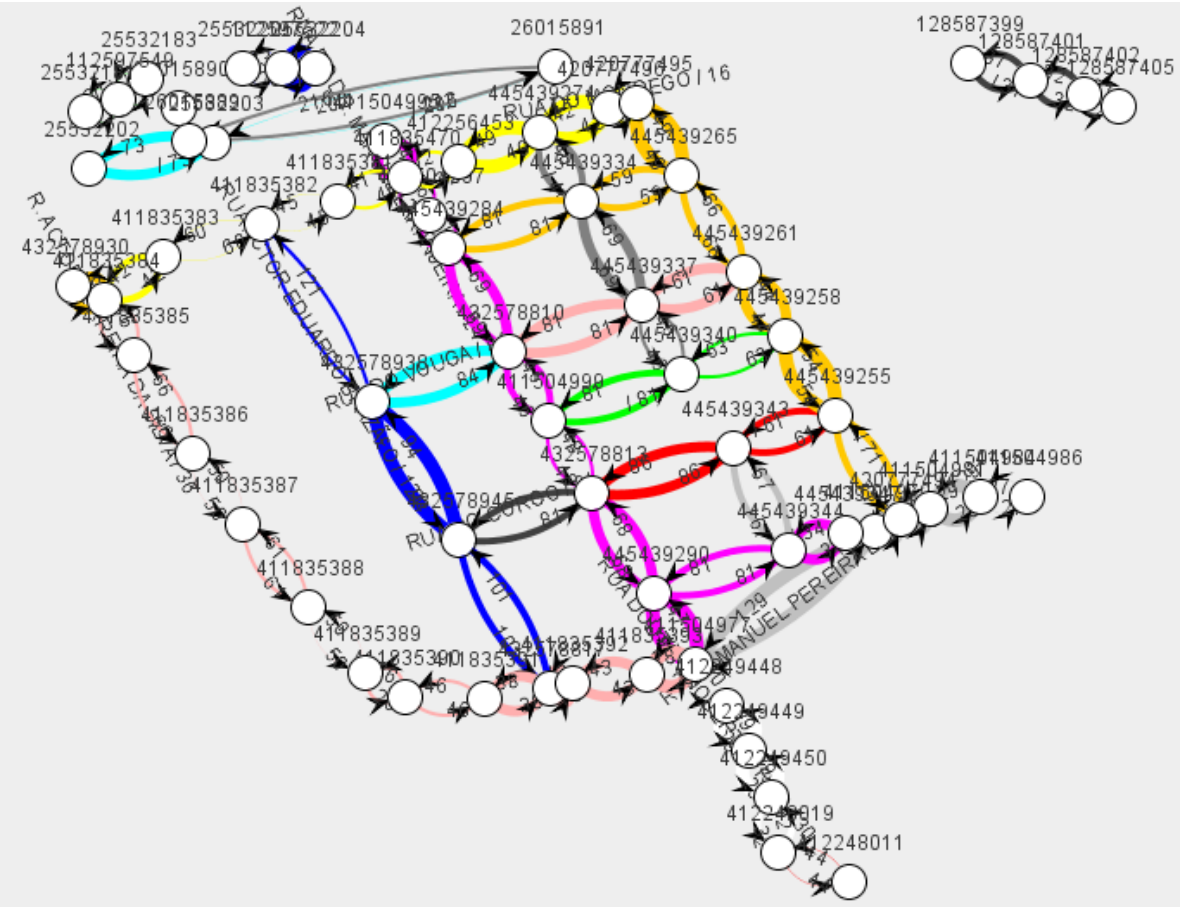
Na opção 1, é mostrado o mapa actual no GraphViewer.

No mapa mostrado as SubRoads que pertencem à mesma rua têm cor igual (ligações entre os nós da mesma rua com a mesma cor).

O nome de cada rua aparece apenas em uma das suas subroads.

No início do programa ou após escolher a opção "8 – Limpar mapa", o mapa que será apresentado ao escolher a opção "1 – Mostrar mapa", será a visualização do mapa com os nodes inicializados por default e com as cores por default.

Por exemplo como é mostrado a seguir:

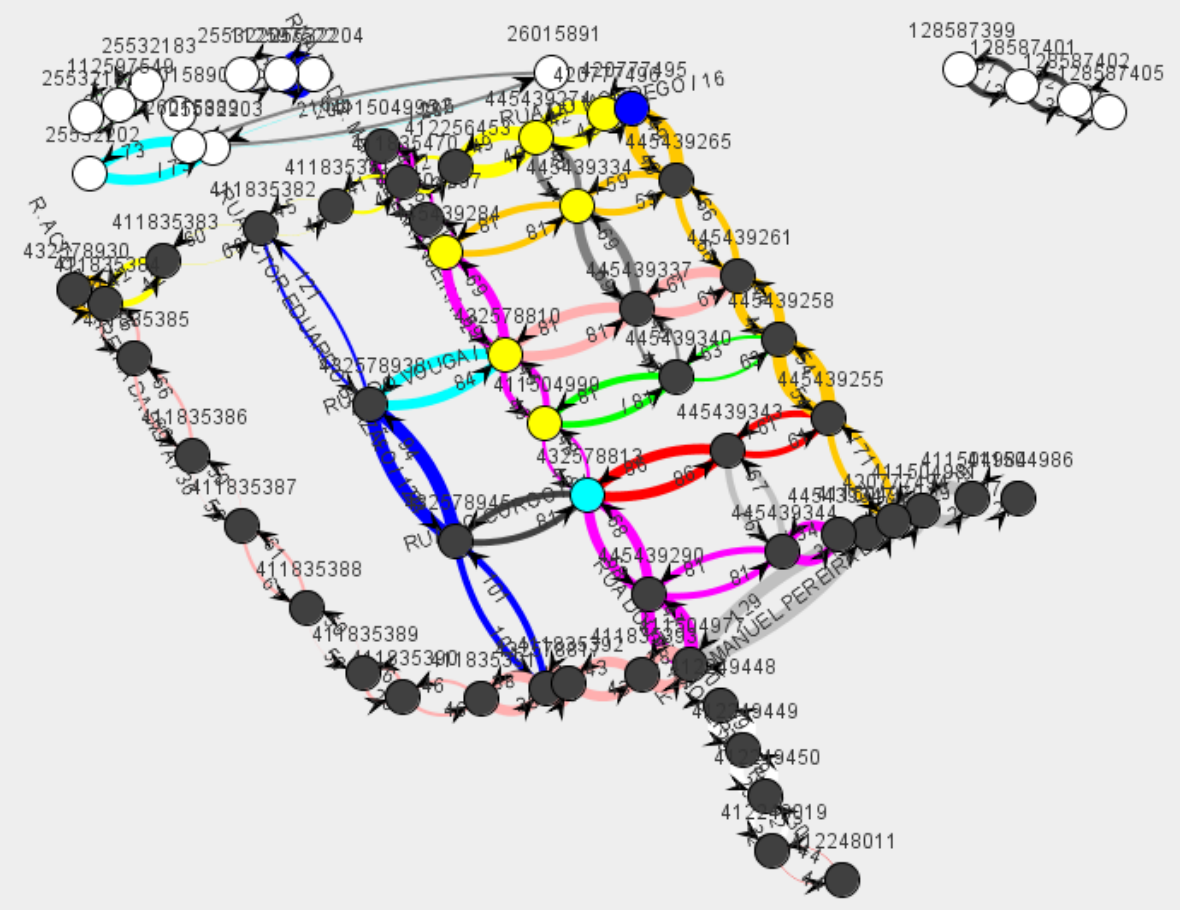


As restantes opções têm uma interface semelhante.

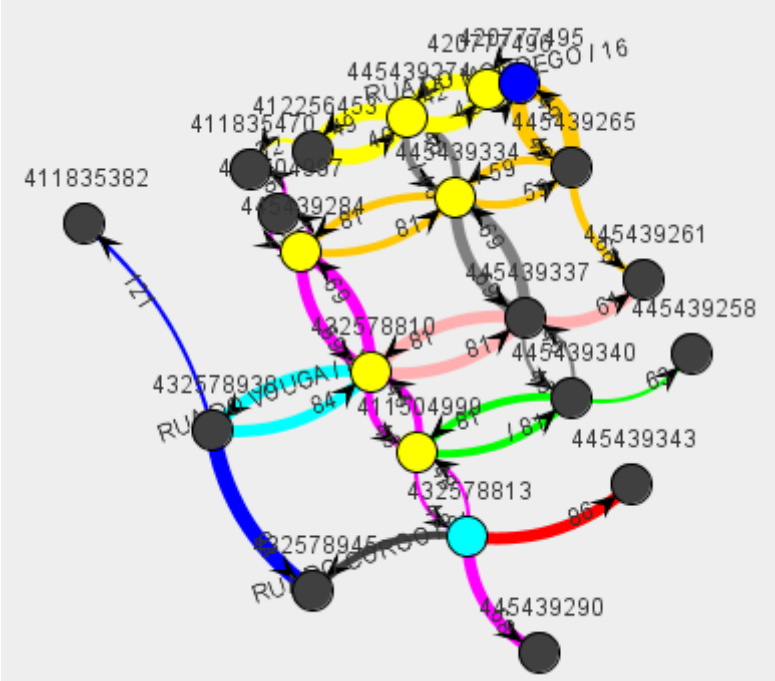
O que muda é que mostra o caminho encontrado e se um dado nó foi ou não visitado pelo algoritmo de Dijkstra.

Por exemplo nas opções 2 e 4 encontra-se o menor caminho entre 2 nodes, só que na 4, para além de fazer o que a opção 2 faz (que é mostrar uma janela com o mapa todo e o caminho encontrado), também mostra uma outra janela com o caminho encontrado e os nós e subroads que estão conectados ao caminho até 2 níveis de distância.

Ou seja se a opção 2 mostrasse a imagem que vem a seguir:



A opção 4, para além dessa janela mostraria após inserir enter:



A opção 3 e 5 é semelhante à opção 2 e 4, só que em vez de pesquisar tendo em conta o cominho mais curto, pesquisa tendo em conta o caminho que demora menos tempo a percorrer.

Como a API de OpenStreetMapsParser não disponibilizou nenhum dado acerca do possível tempo que demoraria a percorrer cada SubRoad, foi usado uma heurística simples, para determinar um possível numero, que apenas serviria para distinguir entre possíveis tempos que demoraria a percorrer cada SubRoad. Por isso é um número que não tem unidades.

A heurística simples a usar para calcular o tempo de cada SubRoad foi multiplicar o comprimento da SubRoad pelo número de subroads que estavam conectadas ao destino da subroad em questão e por fim multiplicar por uma constante.

Foi usada esta estratégia porque consegue distinguir ruas urbanas de auto-estradas, ou ruas rurais. Isto porque em meios urbanos cada subroad tem muito mais subroads a ela conectadas do que em auto-estradas ou meios rurais. Sendo assim consegue-se diferenciar razoavelmente qual o tempo que uma dada subroad demoraria a percorrer em relação às outras subroads do mapa.

A opção 6 e 7 fazem o mesmo que a 2 e 3, mas mostram o algoritmo de Dijkstra a correr.

A cada segundo é mostrada o status de tratamento do nó actual no algoritmo.

Ou seja:

se um node estiver a branco, ainda não foi visitado pelo algoritmo;

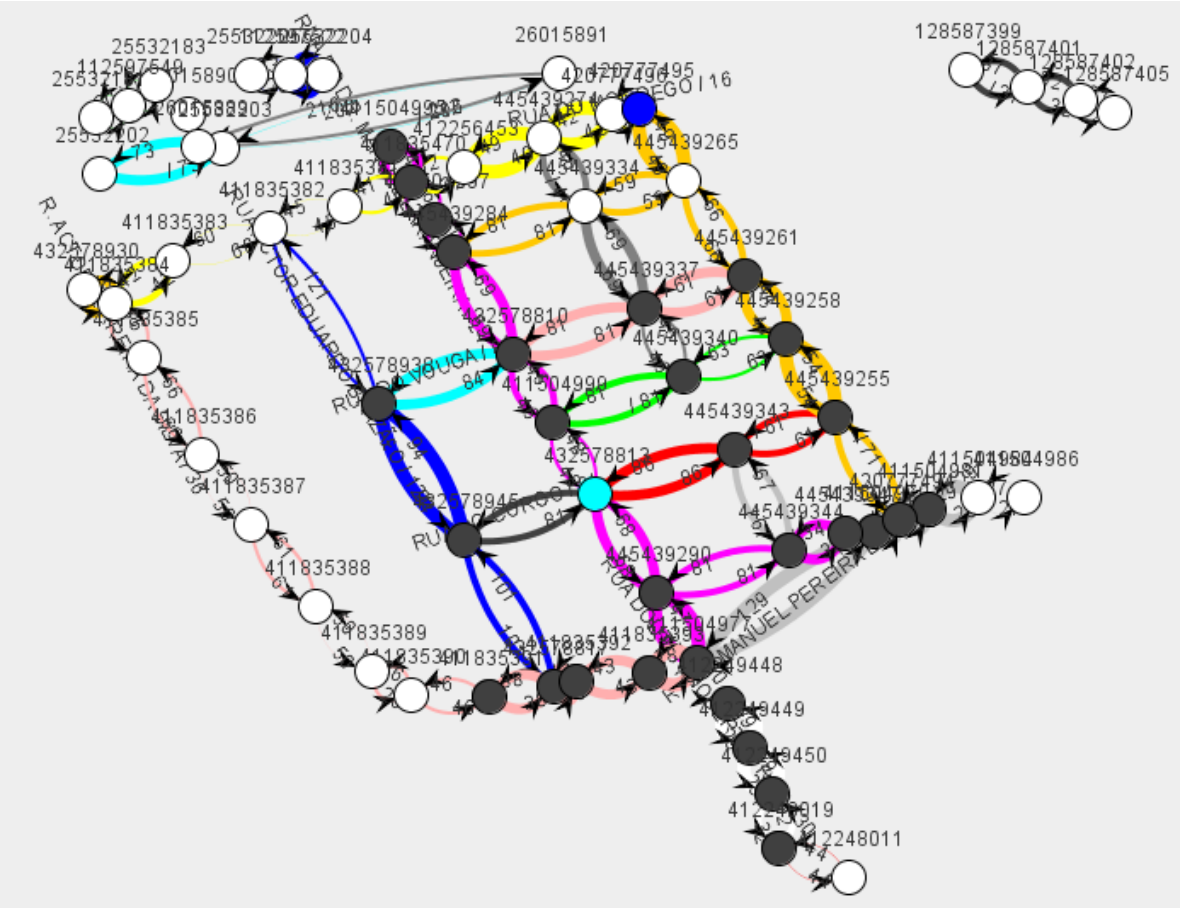
se estiver a cinzento, foi visitado pelo algoritmo;

se estiver a azul claro é o nó de inicio do caminho;

se estiver a azul escuro é o nó de fim do caminho

e por fim se estiver a amarelo o nó pertence ao melhor caminho determinado.

Por exemplo, na opção 6, após algum tempo seria mostrado:



A interface de pesquisa de ruas no mapa funciona de duas maneiras.

Se o utilizador introduzir o nome da rua exactamente como aparece no mapa (não toma em atenção se as letras são maiúsculas ou não, porque faz um toUpper da string introduzida), então a pesquisa no set é feita em tempo logarítmico, é encontrada o apontador para a Road e é seleccionado como nó de inicio o Node que é o sourceNode da primeira subroad do que esta no set<SubRoad*> da Road seleccionada.

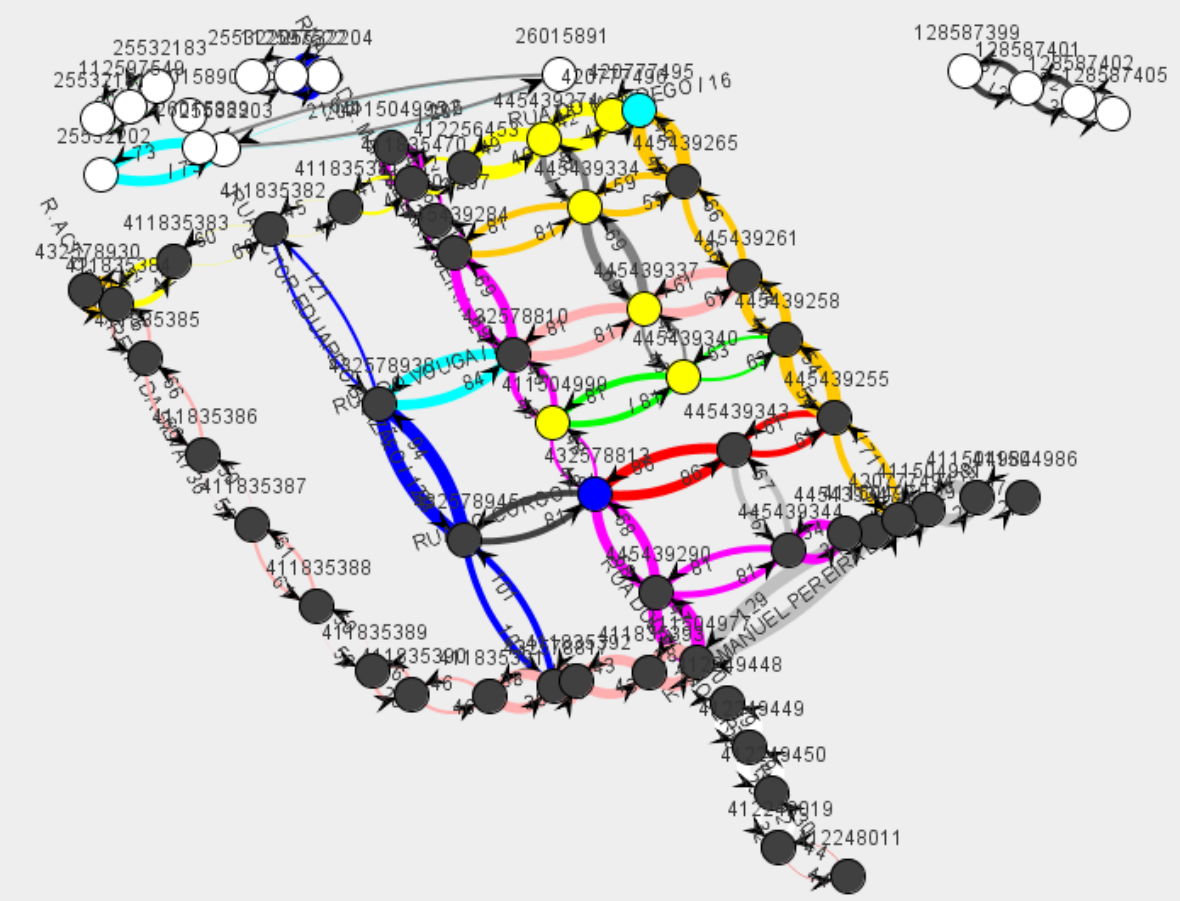
Caso não exista um exact match, é feita uma pesquisa sequencial ao set<Road*> do Map, e é apresentada um tabela com as ruas que contêm no seu nome a string que o utilizador introduziu. De seguida, para seleccionar uma, o user apenas tem de introduzir o índice da tabela.

Caso não exista nenhuma rua com um nome que contem a string introduzida retorna ao menu principal.

Exemplo de um "exact match" do nome da rua para o nó de início, e de seguida usando a funcionalidade para a pesquisa de ruas com o nome similar ao introduzido:

```
Introduza o nome da rua de origem: rua do mondego

Introduza o nome da rua de destino: rua
Escolha a rua que pretende selecionar introduzindo o respectivo indice da tabela:



  >>>>>  Ruas com nome similar ao introduzido  <<<<<

+---+------------------------------+
| # |                     Nome da rua |
+---+------------------------------+
| 0 |     RUA ACTOR EDUARDO BRAZÃO |
| 1 |                   RUA DO CORGO |
| 2 | RUA DO DR. MANUEL LARANJEIRA |
| 3 |                 RUA DO MONDEGO |
| 4 |                   RUA DO VOUGA |
| 5 |      RUA DOUTOR ROBERTO FRIAS |
+---+------------------------------+

Introduza o indice da rua: 1
```

Cujo resultado foi:



Por fim, é disponibilizada a opção de guardar o último caminho determinado pelo algoritmo de Dijkstra para um ficheiro (que está armazenado no `vector <Node*> currentPath;` do Map).

```
>>> Introduza o nome do ficheiro onde quer guardar o ·ltimo caminho determinado
>>> Nome:
```

E de o carregar de um ficheiro.

Ao carregar de um ficheiro os nós do grafo são reinicializados para o valor por default, e depois é carregado o caminho que tinha sido armazenado no ficheiro.

```
>>> Introduza o nome do ficheiro de onde quer carregar o caminho
>>> Nome:
```

Lista de casos de utilização

Esta aplicação foi desenvolvida de forma a determinar o menor caminho entre 2 nós de um mapa. Mas visto que o algoritmo de Dijkstra é muito flexível, facilmente poderia ser estendida para outros objectivos.

Principais dificuldades encontradas

As principais dificuldades não foram relacionadas com o projecto em si, porque era relativamente simples, mas antes com as APIs que foram disponibilizadas.

Ou seja, por exemplo na realização da translação de coordenadas dos valores que o openStreetMapsParser dava de forma a colocar sempre todos os nós no ecrã independentemente dos valores das projecções da latitude e longitude que eram dadas no ficheiro de nodes.txt.

Isto era algo que o próprio parser já deveria fazer.

Por último, o facto de apesar de a API do GraphViewer dizer que era possível atribuir labels ao nós, quando na realidade para além de o não fazer colocava sempre como label o ID do nó. Isto faz com que mesmo em mapas com poucos nós e ruas fiquem um bocado difíceis de perceber porque em cada nó é mostrado o seu ID e esse ID é quase sempre de 9 algarismos.

Indicação do esforço de cada elemento do grupo

Não se aplica neste trabalho porque decidi faze-lo sozinho, visto que era um trabalho pequeno e assim pude controlar melhor quando é que estava a trabalhar para este projecto no meu calendário de testes e outros trabalhos para entregar.

Bibliografia

http://en.wikipedia.org/wiki/Dijkstra's_algorithm

http://www.dgp.toronto.edu/~jstewart/270/9798s/Laffra/DijkstraApplet.html

http://en.wikipedia.org/wiki/Fibonacci_heap

http://stackoverflow.com/questions/504823/has-anyone-actually-implemented-a-fibonacci-heap-efficiently

http://www.inf.ufsc.br/grafos/temas/custo-minimo/dijkstra.html

http://www-b2.is.tokushima-u.ac.jp/~ikeda/suuri/dijkstra/Dijkstra.shtml