Conceção e Análise de Algoritmos

Relatório do 1º Trabalho de Grupo

“Planeamento de Itinerários Multimodais”

MIEIC - Turma 2 – Grupo D

André Sousa Lago – up201303313@fe.up.pt

Gustavo Rocha da Silva – up201304143@fe.up.pt

Ricardo Dantas Cerqueira – up201304000@fe.up.pt

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Ano Letivo 2014/2015

Conteúdo

Descrição do problema e da implementação

O objetivo do trabalho é, através da informação proveniente do sistema de transportes públicos da cidade do Porto, elaborar um sistema que permita ao seu utilizador descobrir o caminho mais favorável entre dois pontos, segundo critérios por ele definidos.

O sistema possui informações reais da rede de autocarros da STCP e das várias linhas do Metro do Porto. A partir disso e dos dados introduzidos pelo utilizador (locais de partida e chegada e prioridades de trajeto), o sistema calcula o melhor caminho tendo em conta as preferências do utilizador.

Os resultados são apresentados numa interface gráfica implementada com a biblioteca SDL (Simple Directmedia Layer). A informação das redes foi processada com recurso a bibliotecas de parsing (RapidJson principalmente).

A SDL é uma biblioteca cross-platform que permite a simplificação de interfaces gráficas em C++, não possuindo restrições de licença para a utilização neste trabalho. Neste caso, a biblioteca SDL é usada para implementar a interface gráfica da nossa solução, ou seja, permite desenhar o grafo e o caminho calculado no ecrã, bem como obter o input do utilizador (clique e deslocamento do rato). Para realizar o input inicial do utilizador (preferências a nível de utilização do sistema e do algoritmo) utilizaremos a interface da consola.

A implementação do sistema consistirá na representação do mapa das redes em causa através de um grafo, em que as várias paragens são representadas por vértices e os trajetos por arestas. Para além disso, cada paragem estará ligada por arestas às paragens mais próximas de si de forma a considerar possíveis trajetos realizados a pé (em algumas situações, ir a pé de uma paragem para outra constitui vantagens no trajeto, a nível, principalmente, do tempo de espera).

Instruções de utilização da aplicação

Num primeiro momento, o utilizador deve escolher se pretende aceder a funcionalidades internas ao cálculo do caminho mais curto, ou seja, se quer escolher qual o algoritmo e qual a estrutura de dados a ser utilizados.

Se o utilizador apenas pretender saber o melhor caminho entre dois pontos, deve escolher a opção “Não” (N) e avançará para a indicação dos pesos relativos dos critérios de cálculo, como é descrito mais à frente.

Em caso contrário, o utilizador passa para uma nova pergunta, onde deve decidir se pretende escolher um algoritmo e estrutura de dados em específico ou se quer utilizar todos os algoritmos para comparar as suas performances. Nesta última opção, o utilizador avança para a

seleção dos pesos relativos (descrito em baixo). Pela outra opção, o utilizador passa por uma série de perguntas que lhe permitem escolher o algoritmo a ser utilizado (Dijkstra ou A*), a estrutura de dados auxiliar ao algoritmo (listas ou Fibonacci Heaps) e se pretende ver a performance do algoritmo (em milissegundos).

Independentemente das opções prévias do utilizador, é necessário que passe pela fase de seleção dos pesos relativos para o cálculo do caminho. Os pesos relativos são valores introduzidos pelo utilizador que especificam qual a sua preferência de caminho, ou seja, quais são os fatores que considera mais críticos para que o caminho obtido seja aquele que prefere. Esses fatores são o tempo de viagem, o custo monetário, a distância percorrida e o número de transbordos efetuados. Esses valores são arbitrários, isto é, não possuem nenhum limite máximo, de forma a permitir maximizar a prioridade de um dos fatores.

Assim, se, por exemplo, o utilizador pretender obter o caminho mais rápido, deve dar um valor alto (como 1000) aos tempo, e valores baixos (como 0) aos restantes. Contudo, se também interessa o preço da viagem numa importância igual à do tempo, em vez de 0 esse fator deve ter um valor mais alto, como 800 ou 1000.

Por fim, é necessário introduzir a data de partida, uma vez que é fundamental para o cálculo do caminho sempre que o peso do tempo de viagem for maior do que zero.

Uma vez indicada a hora de partida, surge o painel gráfico com o mapa dos transportes. Nele estão representadas as linhas de autocarros (STCP) e de metro do distrito do Porto. As ruas por onde passam trajetos de autocarro estão representadas a vermelho, e as linhas do metro são visíveis a azul. A interação com esta interface é feita com o rato e o teclado, sendo possível arrastar o mapa premindo o botão direito do rato e arrastando-o, bem como fazer zoom utilizando a roda do rato.

Para selecionar o ponto de partida do trajeto o utilizador deve clicar com o botão esquerdo do rato no ponto de partida, que não tem de ser nenhuma paragem ou rua. Quando fizer isto, surge nesse ponto um quadrado indicativo da seleção feita. O processo para selecionar o ponto de destino é igual. Quando ambos os pontos forem escolhidos, o caminho ideal é calculado e apresentado com linhas verdes que unem os dois pontos. Para além disso, surge na consola a lista das paragens que fazem parte do percurso, bem como a hora em que se passa nessa paragem.

Se o utilizador pretender escolher novos pontos de partida e chegada, basta premir a tecla “ESC” e pode refazer a sua seleção.

Para terminar o programa ou mudar as definições que introduziu inicialmente, basta premir a tecla “Control” do lado esquerdo, e será perguntado ao utilizador se pretende sair do programa ou alterar as ditas definições.

Notas relativas à implementação das bibliotecas

Para compilar/executar o nosso programa é necessário fazer alguns preparativos indispensáveis. Em primeiro lugar, é necessário colocar o ficheiro "SDL2.dll", presente na pasta "lib" do projeto, na pasta onde estiver o executável para que o código relativo à biblioteca SDL execute. Para além disso, nas definições do projeto é indicado ao compilador que o código da bilbioteca boost se encontra na pasta "C:\boost", logo, é necessário descarregar a biblioteca do site indicado nas referências bibliográficas e colocar a pasta descarregada no diretório "C:\" com o nome de "boost". Para o desenvolvimento da nossa solução utilizamos a versão 1.58.0 desta biblioteca, mas outras versões poderão funcionar também.

Formalização do problema

Os dados de entrada serão a informação dos pontos do grafo, os pontos de partida e de chegada pretendidos, e uma indicação da prioridade do utilizador em relação aos diferentes critérios disponíveis (tempo, distância, custo e número de transbordos efetuados), associando a cada critério um custo monetário correspondente, de forma a homogeneizá-los (por exemplo, fazer com que percorrer 10 *km* corresponda a gastar 50 cêntimos).

As linhas dos autocarros serão obtidas diretamente a partir do site dos STCP, assim como os horários a que chegam os autocarros em cada paragem. A informação relativa ao Metro do Porto, no entanto, terá que ser introduzida manualmente, visto que, após conversa por e-mail com o respetivo departamento de comunicação, se concluiu que não existe nenhuma API que permita obter essa informação.

Os dados estão limitados às redes acima referidas, que estão limitadas à cidade do Porto.

Dados de entrada:

G – vértice de destino (paragem de autocarro ou estação de metro)

O – vértice de origem (paragem de autocarro ou estação de metro)

C_t – peso dado a uma unidade de tempo;

C_d – peso dado a uma unidade de distância;

C_n – peso dado a um transbordo;

c(A) – função correspondente ao custo monetário de percorrer a aresta A;

t(A) - função correspondente ao tempo necessário para percorrer a aresta A;

d(A) – função correspondente à distância percorrida na aresta A;

n(A) – função correspondente ao número de transbordos efetuados ao percorrer a aresta A;

g(A) – função que, partindo dos critérios acima definidos, calcula o peso de uma aresta A;

Gr – grafo que contém os vértices e arestas necessários à construção da rede, utilizados para os calculos do caminho ideal

Dados de saída:

T – conjunto ordenado de vértices a percorrer, tal que T ϵ P, sendo P o conjunto de todos os caminhos possíveis entre O e G, ou seja, P: { $\{V_0, \ldots, V_i\}$, … }, V_0 = O, V_i = G.

Restrições:

Sendo V0 e Vf o primeiro e último elemento de T, respetivamente, V0=O ∧ Vf=G.

Para evitar que os algoritmos utilizados entrem em ciclos infintos que impeçam o cálculo, o grafo não deve possuir pontos repetidos nem pesos negativos.

Objetivo (resultado esperado):

$$T : \forall p \in P, G(T) \leq G(p)$$

O peso de uma dada aresta *A*, calculado pela função *g(A),* será:

$$g(A) = t(A) \cdot C_t + d(A) \cdot C_d + n(A) \cdot C_n + c(A)$$

Utilizando os pesos acima descritos, serão utilizados algoritmos de caminho mais curto como Dijkstra e A*.

Quanto a este último, é proposto que seja aplicada uma função heurística que, para cada vértice, devolva o custo associado ao trajeto, em linha reta, no modo de transporte mais rápido, com o menor custo disponível e sem transbordos efetuados para o destino. Este cenário, sendo o mais otimista possível, serve de limite inferior para o custo do caminho, potencializando o funcionamento correto, por ser admissível, e eficiente, por ser consistente, do algoritmo. Seja v_{max} a velocidade máxima atingida por qualquer um dos meios de transporte. A função heurística *h(V)* , para um vértice atual *V* e um vértice objetivo *G*, será portanto:

$$h(V) = \min(custo\ tempo) + \min(custo\ distância) + \min\left(\frac{custo}{transbordos}\right) + \min(custo\ monetário) =$$

$$= \frac{|pos(G) - pos(V)|}{v_{max}} \cdot C_t + |pos(G) - pos(V)| \cdot C_d + 0 * C_n + 0$$

Para V = G, $|pos(G) - pos(G)| = 0$, pelo que h(G) = 0.

Para cada caminho possível partindo de V e terminando em G, constituído por k vértices (V = V_0, G = V_k), o custo observado, h*(V), é dado por:

$$h^*(V) = \sum_{i=0}^{k-1} t(V_i, V_{i+1}) \cdot C_t + \sum_{i=0}^{k-1} d(V_i, V_{i+1}) \cdot C_d + \sum_{i=0}^{k-1} n(V_i, V_{i+1}) \cdot C_n + \sum_{i=0}^{k-1} c(V_i, V_{i+1})$$

Por palavras, h*(V) corresponde à soma dos custos de tempo, distância, transbordos e dinheiro entre todos os pontos do caminho entre G e V. Podemos então concluir que, cada um desses somatórios será sempre menor ou igual ao custo correspondente calculado pela heurística, ou seja:

$$\frac{|pos(G) - pos(V)|}{v_{max}} \leq \sum_{i=0}^{k-1} t(V_i, V_{i+1}) \qquad |pos(G) - pos(V)| \leq \sum_{i=0}^{k-1} d(V_i, V_{i+1})$$

$$0 \leq \sum_{i=0}^{k-1} n(V_i, V_{i+1}) \qquad 0 \leq \sum_{i=0}^{k-1} c(V_i, V_{i+1})$$

Assim, h(V) é um limite inferior para o custo efetivo de um trajeto, logo, é uma função heurística admissível.

De seguida demonstra-se que esta função heurística é também consistente (ou monótona) pois verifica as duas equações que se irão comprovar.

Em primeiro lugar, para que seja consistente, a heurística deve verificar a condição de que o seu valor para qualquer vértice N do grafo deve ser sempre menor ou igual do que a soma do valor da heurística para qualquer vértice L vizinho de N com o custo associado à aresta A que une N e L, ou seja:

$$h(N) \leq g(A) + h(L)$$

Desenvolvendo o lado direito da inequação:

$$\begin{aligned} g(A) + h(L) &= t(A) \cdot C_t + d(A) \cdot C_d + n(A) \cdot C_n + c(A) + \frac{|pos(G) - pos(L)|}{v_{max}} \cdot C_t \\ &\quad + |pos(G) - pos(L)| \cdot C_d \\ &= \left(t(A) + \frac{|pos(G) - pos(L)|}{v_{max}} \right) \cdot C_t + (d(A) + |pos(G) - pos(L)|) \cdot C_d \\ &\quad + n(A) \cdot C_n + c(A) \end{aligned}$$

Assim, podemos comparar as componentes de tempo, distância, número de transbordos e custo monetário da heurística em N com o desenvolvimento feito acima:

$$0 \leq n(A) \qquad 0 \leq c(A)$$

$|pos(G) - pos(N)| \leq d(A) + |pos(G) - pos(L)|$, uma vez que o caminho direto entre G e N será sempre menor ou igual que qualquer outro caminho

$\frac{|pos(G)-pos(N)|}{v_{max}} \leq t(A) + \frac{|pos(G)-pos(L)|}{v_{max}} <=> \frac{|pos(G)-pos(N)|}{v_{max}} \leq \frac{|pos(L)-pos(N)|}{v[N-L]} + \frac{|pos(G)-pos(L)|}{v_{max}}$, uma vez que o tempo mínimo gasto entre G e N corresponde a percorrer o trajeto mais curto entre eles à velocidade máxima possível.

Deste modo, como cada componente do desenvolvimento feito é menor que o componente equivalente da heurística em N, confirma-se que $h(N) \leq g(A) + h(L)$.

Para completar a prova da consistência da heurística, é necessário provar que $h(G) = 0$. Então, sabendo que $|pos(G) - pos(G)| = 0$:

$$h(G) = \frac{|pos(G) - pos(G)|}{v_{max}} \cdot C_t + |pos(G) - pos(G)| \cdot C_d + 0 * C_n + 0 = 0$$

Assim, comprova-se que a heurística definida é admissível e consistente, pelo que pode ser usada nos algoritmos de cálculo de caminho mais curto para aumentar a eficiência destes.

Para aumentar a eficiência temporal da nossa solução, utilizamos a classe Fibonacci Heap da biblioteca Boost (versão 1.58.0), que é portanto necessária para compilar o nosso projeto.

Utilizando a Fibonacci Heap, conseguimos que o algoritmo de Dijkstra tenha complexidade temporal O(|V|*log|V|) (quando |E|>|V|). Esta estrutura de dados também confere ao algoritmo de A* a mesma complexidade temporal para o pior caso, contudo, a aplicação desta técnica e a natureza do próprio algoritmo diminuem a probabilidade de se atingir o pior caso. Assim, a complexidade temporal do algoritmo A* quando aplicado com Fibonacci Heaps é melhor do que no algoritmo de Dijkstra.

Métricas de avaliação

Os dados de entrada variáveis na nossa aplicação serão os vértices de origem e destino do percurso e os custos relativos de cada componente de custo de uma aresta (distância, tempo, transbordos e custo monetário), uma vez que estes são introduzidos pelo utilizador.

Sendo E o número de arestas (edges) e V o número de vértices, espera-se que o cálculo do caminho ideal, usando o algoritmo de Dijkstra tenha complexidade temporal de O(|E|*log|V|), sendo possível melhorá-lo para O(|V|*log|V|), se |E|>|V| e forem utilizadas Fibonacci Heaps. Esta é também a complexidade do algoritmo A*, embora ele na verdade seja mais eficiente uma vez que a complexidade mede o pior caso.

Para avaliar empiricamente os algoritmos, serão efetuadas medições de tempo e serão modificadas as cores dos vértices na representação gráfica, de modo a demonstrar a diferença entre os diferentes algoritmos relativamente ao número de vértices processados.

Para além disso, poderão ser usadas técnicas de brute-force para confirmar que a solução obtida através dos outros algoritmos é a correta.

Descrição da solução

A nível de algoritmos utilizados, como já foi referido, realizamos o cálculo do caminho mais curto com recurso aos algoritmos de Dijkstra e A*. Para além disso, para efeitos de teste, utilizamos um algoritmo de brute force.

Como também já foi referido, os algoritmos de Dijkstra e A* apresentam complexidade temporal O(|V|*log|V|) quando aplicados com Fibonacci Heaps, embora na verdade o algoritmo A* seja mais eficiente uma vez que a complexidade indicada é para o pior caso.

A nossa solução foi implementada com recurso a diferentes "módulos", representados pela organização em pastas do código.

Na pasta "docs" estão os documentos da solução, como o relatório (em formato PDF e Word) e o diagrama de classes.

Na pasta/módulo dos algorimtos ("algorithms") encontra-se o código relativo à implementação dos algoritmos. Os ficheiros com os nomes "Dijkstra", "AStar" e "BruteForce" implementam os algoritmos indicados pelos seus nomes, ao mesmo tempo que fornecem uma interface que permite fornecer ao algoritmo diferentes estruturas de dados para armazenar temporáriamente os vértices. Assim, sem modificar o código, conseguimos testar a variação na sua eficiência com diferentes estruturas de dados que obedeçam às regras definidas em "GraphQueue". Os ficheiros com nomes iniciados por "GraphQueue" implementam a adaptação das estruturas de dados já existentes ao formato aceite pela interface que nós criamos para os algoritmos. De uma forma simplificada, são classes que herdam de "GraphQueue" implementando todos os métodos necessários para o funcionamento dos algoritmos, baseando-se em estruturas de dados diferentes (neste caso, lista e Fibonacci Heap).

Estes algoritmos são passíveis de ser utilizados com qualquer estrutura de grafos genérica, que implemente os métodos e atributos usuais de grafos.

Por fim, o ficheiro "PathFinder" implementa uma classe que serve de interface entre a interface com o utilizador e os algoritmos. As opções do utilizador são armazenadas num objeto do tipo "ProgramConfig" (descrito à frente), e são interpretadas pelo "PathFinder" de forma a não só chamar os algoritmos corretos, mas também medir o seu desempenho temporal e mostrá-lo ao utilizador ou não, conforme aquilo que pretender.

Na pasta relativa a grafos ("graph") estão presentes as classes relativas a grafos ("Edge", "Graph" e "Vertex"). Embora não sejam exatamente iguais às classes genéricas, são o mais próximo delas que conseguimos, diferindo apenas em pequenos detalhes. Contudo, estas classes funcionariam em qualquer algoritmo de cálculo para grafos, uma vez que as poucas alterações se verificam em aspetos não cruciais para esse tipo de cálculos. Para além

destas classes, existe também a classe “Path” feita para armazenar o conjunto de arestas di grafo que representam o caminho calculado. Esta classe poderia também, na sua maioria, ser utilizada com outros algoritmos para problemas diferentes do nosso. De uma forma geral, este módulo foi desenvolvido de forma a ser o mais genérico possível.

No módulo da interface gráfica do utilizador (“gui”) estão classes definidas por nós que são usadas em conjunto com a bilioteca SDL de forma a produzir a interface pretendida. A classe “Camera” implementa o ponto de vista do utilizador, permitindo a funcionalidade de zoom e arrastamento do mapa na interface gráfica. A classe “SDLGraphDraw” faz o desenho de grafos e caminhos calculados, permitindo salientar de formas e cores diferentes tanto grafos inteiros, como caminhos, vértices ou arestas. A classe “SDLRGB” fornece uma abstração de cores utilizando o sistema RGB. A classe “Slider” foi construída para permitir uma interface com uma barra que desliza sobre outra, de forma a implementar um slider que permitiria ao utilizador selecionar valores entre dois limites definidos. Esta classe seria utilizada para definir os pesos relativos dos diferentes fatores de cálculo do caminho ideal, contudo não foi utilizada no produto final da nossa solução. Contudo, mantivemos o seu código no projeto para uma possível futura utilização.

Na pasta “include” estão presentes ficheiros de bibliotecas de SDL e RapidJson, sendo que a primeira é utilizada, como já foi referido, para implementar a interface gráfica, e a segunda para análise e importação da informação das redes de transportes públicos. Na pasta “lib” encontram-se outros ficheiros auxiliares às bibliotecas utilizadas, entre os quais o ficheiro “SDL2.dll” que, como já foi referido, deve estar na pasta do ficheiro executável para que este possa executar.

Na pasta relativa aos métodos de transporte (“transport”) estão todas as classes relativas à adaptação e implementação das informações das redes de transportes do distrito do Porto. Em primeiro lugar, construímos classes que herdam das classes de grafos (neste caso, “Vertex” e “Edge”) para representar a própria rede de transportes como um grafo. No nosso problema, os vértices são as paragens (de metro e autocarro). Ou seja, os vértices do nosso grafo serão “MetroStop” e “BusStop”, que herdam de “TransportStop”. As “Stops” permitem também saber a que linha do respetivo meio de transporte pertencem (“BusRoute” e “MetroRoute”, que herdam de “TransportRoute”). As arestas são as ligações entre paragens, sendo representadas por “MetroEdge” e “BusEdge”, que herdam de “TransportEdge”. Contudo, as “Edges” apresentam-se como ligeiramente diferentes de arestas regulares de grafos, devido à necessidade de apresentar as ligações entre duas paragens como ruas reais em vez de linhas retas. Assim, as “Edges” definidas neste módulo possuem também um conjunto de pontos que representam os diferentes pontos a unir para se formar a rua real. Isto é, com esses pontos é possível que as arestas do grafo não sejam representadas como linhas retas, e assim podemos obter o aspeto real das ruas do Porto.

Ainda nesta pasta, existem outras classes auxiliares como “Map” (que representa o mapa da cidade, armazenando as arestas e vértices do grafo na forma de mapa; esta classe permite também obter o grafo da rede), “Hour” (representa um timestamp de horas e minutos), “Coordinates” (representa as coordenadas em latitude e longitude de pontos), “TransportSpeeds” (representa as velocidades dos diferentes meios de transporte) e “WeightInfo”. Esta última é essencial para o cálculo do caminho ideal, uma vez que é utilizada no cálculo dos pesos das arestas e do valor da heurística em cada vértice (no

algoritmo A*). Esta classe armazena os pesos indicados pelo utilizador no início do programa, bem como o custo associado às arestas específicas dos meios de transporte.

Por fim, na raíz da pasta do código do programa "source" estão presentes 3 ficheiros adicionais. Os ficheiros relativos a "ProgramConfig" implementam uma classe que armazena as opções do utilizador no que toca ao modo em que executa o programa. Esta classe indica a "PathFinder" como aplicar os algoritmos, quais a aplicar a que estruturas de dados utilizar, bem como se deve ou não apresentar o desempenho temporal dos algoritmos. No ficheiro "main" está o código que liga todos os componentes acima referidos, produzindo o produto desejado. As diversas ações invocadas por "main" são:

- Carregamento das informações das redes dos ficheiros presentes na pasta "data" para as estruturas definidas em "transport"
- "Leitura" da consola das preferências do utilizador e seu armazenamento para futura utilização
- Inicialização da interface gráfica (SDL) com o mapa obtido no primeiro ponto
- Leitura do input do utilizador e sua interpretação (ações de rato e teclado)
- Chamada ao interpretador do input do utilizador que executa os algoritmos disponíveis
- Apresentação dos resultados obtidos

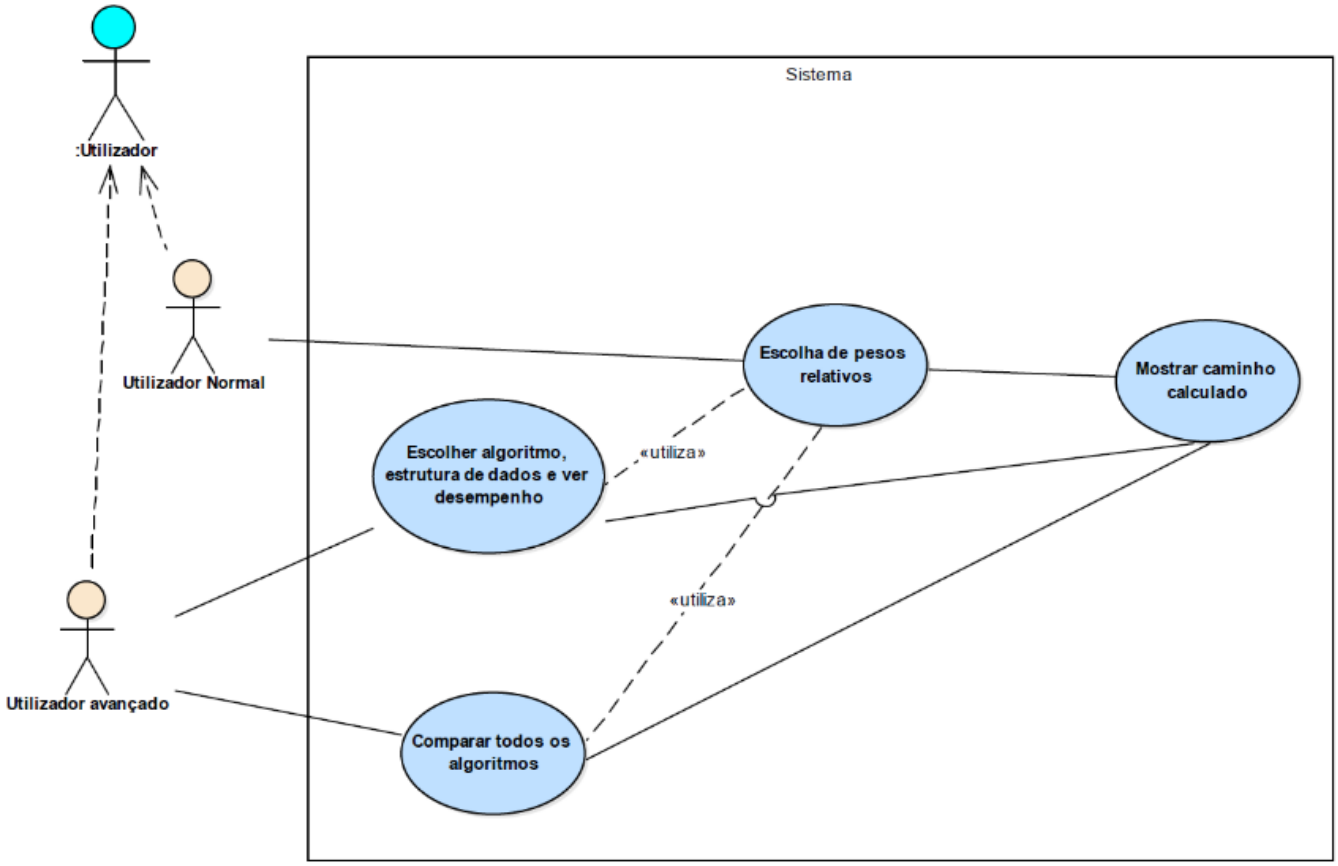
De uma forma muito resumida, os principais passos de execução da nossa solução são:

1. Leitura dos ficheiros de texto (maioritariamente obtidos com recurso a scripts que os descarregaram das páginas da STCP e metro), e sua transposição para as estruturas definidas em "transport". Ao fazer isto, cria-se uma estrutura semelhante a um grafo ("Map") que armazena o mapa propriamente dito da rede criada. Para além disto, há um préprocessamento da rede real de forma a criar troços de caminho que podem ser feitos a pé, sem utilizar autocarro ou metro.
2. Input do utilizador, via consola e interface SDL, que permite saber as preferências do utilizador relativas aos pesos dos diferentes fatores do cálculo e aos algoritmos e estruturas a ser utilizados.
3. Análise do input e chamadas aos diferentes algoritmos implementados ("algorithm"), bem como suas medições de performance e comparações.
4. Apresentação dos resultados obtidos através da interface gráfica de SDL, utilizando diferentes cores e tipos de pontos para os vértices do grafo.

Diagrama de classes

Ver ficheiro PDF em anexo.

Diagrama de casos de utilização



Principais dificuldades encontradas

As principais dificuldades encontradas no desenvolvimento da nossa solução foram a transposição das informações das redes de transportes para estruturas compatíveis com grafos. Contudo, utilizando herança de classes, elaboramos uma solução que faz esta transposição das informações reais para estruturas genéricas de grafos que poderiam ser utilizadas para outros problemas de grafos diferentes do nosso.

Esforço de cada membro

A divisão de trabalhos foi aproximadamente a seguinte:

André Lago – elaboração dos algoritmos de caminho mais curto; interface de consola; interpretação das opções do utilizador; medições de complexidade dos algoritmos e estruturas de dados utilizados; relatório; apresentação do resultado obtido sobre a forma de texto (pesos finais e pontos de passagem).

Gustavo Silva – obtenção das informações das redes de transportes públicos para ficheiros a ser lidos pelo programa; implementação da conversão das informações de redes para estrutura em grafo, incluindo os pesos ponderados para os percursos.

Ricardo Cerqueira – elaboração dos algoritmos de caminho mais curto; interface gráfica com recurso à biblioteca SDL; implementação da abstração de câmara para utilização com a interface da SDL; relatório; apresentação do resultado de forma visual.

Referências

Biblioteca SDL de interface gráfica:

- https://www.libsdl.org/

Biblioteca Boost utilizada para implementar filas de prioridade e Fibonacci Heaps:

- http://www.boost.org/

Páginas com informação sobre os algoritmos utilizados:

- http://en.wikipedia.org/wiki/Heuristic
- http://pt.wikipedia.org/wiki/Algoritmo_de_Dijkstra
- http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
- http://en.wikipedia.org/wiki/A*_search_algorithm

Slides da disciplina, disponíveis na página do Moodle