Universidade do Porto
FEUP Faculdade de Engenharia

Flight Advisor

Mestrado Integrado em Engenharia Informática e Computação

Elementos do Grupo

1. André Freitas – ei10036 – freitas.andre@fe.up.pt
2. Olivier Alves – ei10069 – ei10069@fe.up.pt
3. Tiago Tavares – ei10140 – ei10140@fe.up.pt

1 de Maio de 2012, Porto

Índice

Introdução

A gestão da rota de um determinado voo é uma tarefa de grande responsabilidade, visto que esta não pode ter qualquer tipo de lapso. Assim sendo, há uma necessidade de recorrer a programas que, a partir de um determinado algoritmo, selecionem uma ou mais rotas, consoante a origem e o destino do voo. Estas aplicações devem ser robustas, com capacidade de lidar com imensas rotas predefinidas e dar uma resposta ao utilizador eficiente e sem falhas em tempo útil.

No contexto desta problemática e a Unidade Curricular de Conceção e Análise de Algoritmos do MIEIC, este relatório visa a documentar uma implementação da solução deste problema (*Flight Advisor*), recorrendo à linguagem C++, estruturas de dados adequadas e algoritmos que se aplicam a estas mesmas estruturas.

Foi usada uma API de visualização de grafos, para representação dos caminhos, e uma biblioteca para representação de Grafos que foi disponibilizada pelos docentes da disciplina.

De modo a facilitar a compreensão da nossa implementação, iremos recorrer a diagramas UML, fluxogramas e pseudocódigo/código de algoritmos que utilizemos, sendo mais transparente a sua interpretação.

Especificação

Ilustração 1: Rotas a partir de Lisboa para o Reino Unido

É necessário implementar um programa que apresenta um plano de voo, usando rotas existentes, para um determinado aeroporto de origem e de destino.

Descrição e identificação do(s) problema(s)

É dado ao programa um ou mais ficheiro(s) que contêm a informação da lista de rotas existentes. O programa é capaz de conseguir interpretar essas rotas e representá-las num grafo, que indica os aeroportos que se ligam a partir destes mesmos caminhos. Para descobrir o melhor caminho entre um aeroporto A e B, o programa deve escolher aquele que tem menos pontos de passagem. Caso existam mais do que uma alternativa que têm o mesmo número de pontos, escolhe-se o caminho que tem a menor distância.

Interpretação das Rotas

Ora, desde já identifica-se a necessidade de implementar um algoritmo que permita ler um ficheiro que contém as rotas existentes e converter essas rotas num grafo de aeroportos, em que as arestas são definidas a partir das rotas.

Assim sendo, será a partir do grafo que todas as decisões do programa irão ser tomadas, a fim de sugerir as melhores rotas para o utilizador, utilizando-se o algoritmo de *Dijkstra* ou outro que se considerar mais adequado. Supondo que o ficheiro de texto tem o seguinte conteúdo (formato csv), em que cada aeroporto é identificado pelo seu código ICAO:

```
Rota A, LPPR, LPPT, LPFR, LPMA
Rota B, LPFR, LPMA, LPAZ
Rota C, LPAZ, LPFR, LPPT
```

Ora, a representação das rotas anterior num grafo conexo seria a seguinte:

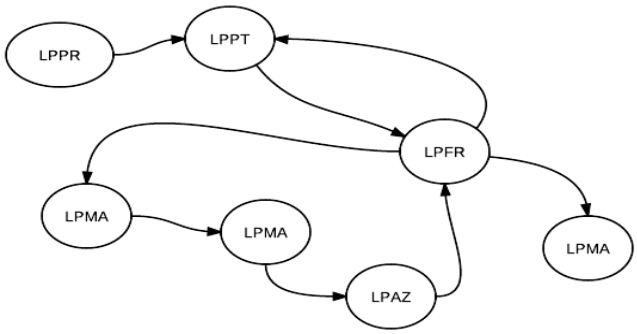

Ilustração 2: Exemplo de uma rede de rotas

É necessário desde já introduzir um conceito de *Waypoint*. Ora, entre dois aeroportos, suponhamos o Aeroporto de Lisboa e o Aeroporto de Madrid, existe pontos de passagem obrigatórios que são chamados de *Waypoints*. Estes pontos não são necessariamente aeroportos, mas apenas sítios onde as rotas de aviação devem passar. Assim sendo, é necessário colocar no grafos, nós deste tipo. Supõe-se ainda que a rede é conexa, ou seja, que se pode ir de qualquer um ponto para outro e, se é possível ir de *A* para *B*, também é possível ir de *B* para *A*.

Representação dos percursos

É necessário representar visualmente os percursos que são calculados mediante o a origem e o destino, usando um grafo que irá articular-se sobre um mapa conhecido (ex. Mapa da Europa). Assinalar os pontos com uma cor diferente seria uma boa prática para facilitar a interpretação das rotas.

Dados de Entrada

No que toca em termos de dados de entrada, o utilizador pode dar um aeroporto de origem e o outro destino e o programa calcula o melhor caminho entre os dois. Porém, o utilizador poderá também dar apenas um aeroporto de origem e o programa calcular todos os caminhos para os restantes ou mesmo não dar nenhum aeroporto, sendo calculado os melhores caminhos entre todos os aeroportos. Este último cálculo poderá implicar um maior tempo de execução, dada à sua dimensão.

Assume-se um programa interactivo que pede estes dados e imprime informações na *Standard Output*. Porém, deve haver a possibilidade de usar o programa de uma maneira mais simples, usando o argumentos de lançamento de execução do mesmo, ou seja "flightadvisor LPPT LPPR", que retorna o percurso.

Implementação da Solução

Ora, a implementação do programa que resolve as especificações anteriormente referidas, parte da seguinte estruturação lógica:

Ilustração 3: Fluxo do Programa

Ficheiros de Dados

O carregamento das Rotas será feito sobre ficheiros em formato CSV em que existe um ficheiro para Aeroportos, outro para os *Waypoints* e outro que representa a Rede. Estes devem ter o seguinte formato:

Aeroportos.csv [País;Nome;IATA;ICAO;Latitude;Longitude]

```
Spain;Valladolid;VLL;LEVD;41.706112;-4.851666
Spain;La Coruña;LCG;LECO;43.301388;-8.379167
Spain; Santiago; SCQ;LEST;42.897499;-8.416111
```

Waypoints.csv [País;Nome;Código;Latitude;Longitude]

```
Spain;Waypoint Salamanca;WYPSMC;40.963463;-5.665177
France;Waypoint Le Mans;WYPLMA;48.006110;0.199556
France;Waypoint Montélimar;WYPMTAR;44.556944;4.749496
```

Rotas.csv [Nome da Rota;(Código/ICAO);(Código/ICAO);(Código/ICAO);....;]

```
I;LFPG;WYPROE;LFCR;EGJB
J;LFCR;EGJJ;LFRB;WYPSEA;LEXJ;LEBB
K;LFRB;WYPSEA;LECO;LEST;LEVX;LPPR
```

Impõe-se a restrição de que todos os Aeroportos e Waypoints estão presentes em pelo menos uma rota no ficheiro criado para o efeito.

Modelo UML

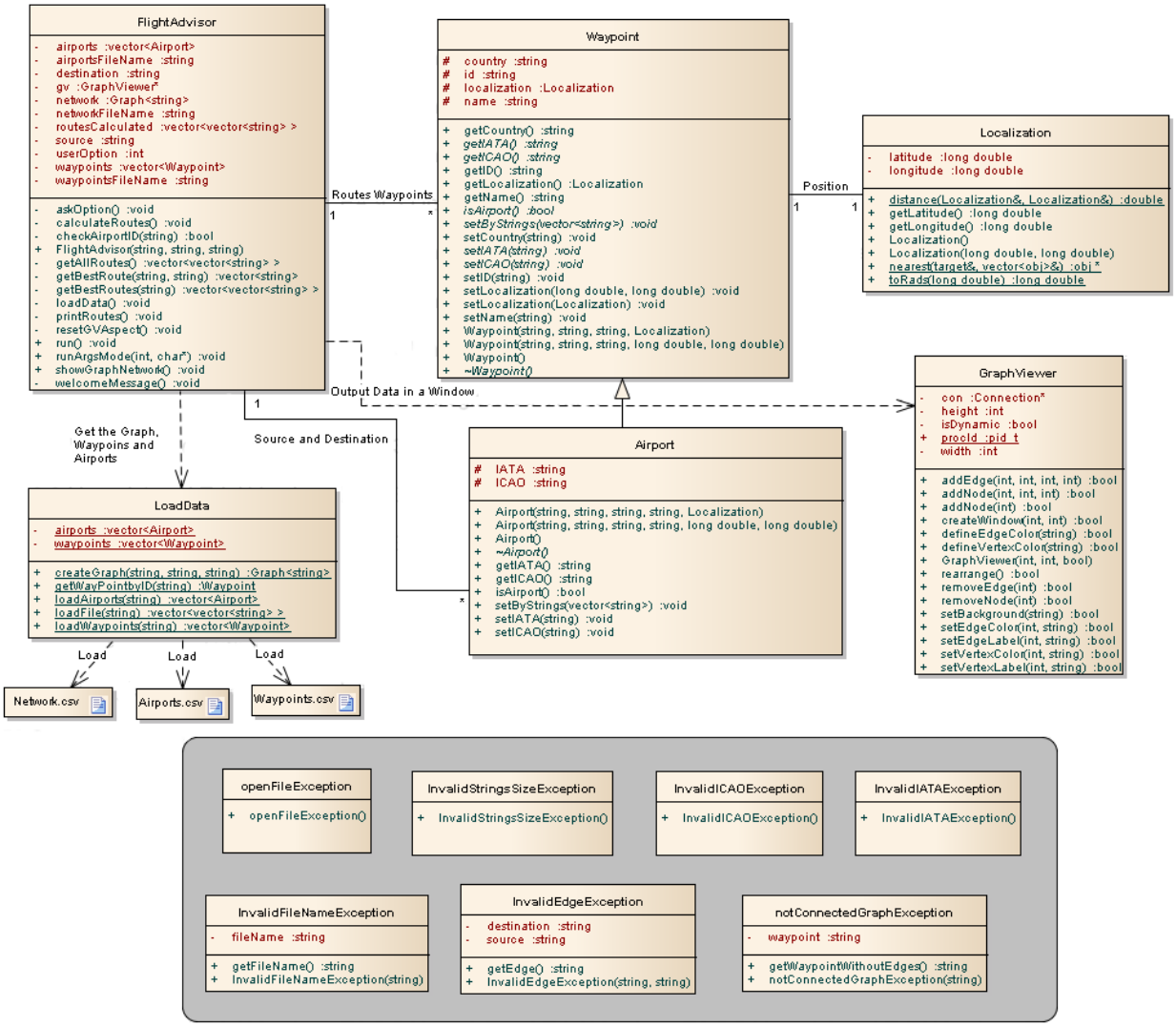

Ilustração 4: Modelo UML da Solução

Como podemos constatar, todos os Waypoints são identificados por um ID e possuem uma Localização GPS. Um Aeroporto pode ser um Waypoint, adicionando os dados do *IATA* e *ICAO*, que são identificadores internacionais. Ora a classe que descreve o programa em si possui várias Waypoints e Aeroportos e depende dos dados carregados pela classe *LoadData.* O resultado do processamento dos percursos é mostrado através da classe *GraphViewer* que permite criar numa janela um grafo, com uma imagem de fundo, que neste caso é o mapa Ibérico.

Implementação em C++

Ora a linguagem que usamos para implementar esta solução foi o C++, que, apesar de ser de médio/baixo nível, facilita a utilização de bibliotecas e permite lidar com muitas especificações. Nesta secção será descrita a maneira como o programa foi implementado e os algoritmos adjacentes ao mesmo, que são a chave do funcionamento do mesmo. Todo o código está no directório "code" que segue em anexo a este relatório.

Organização dos Ficheiros

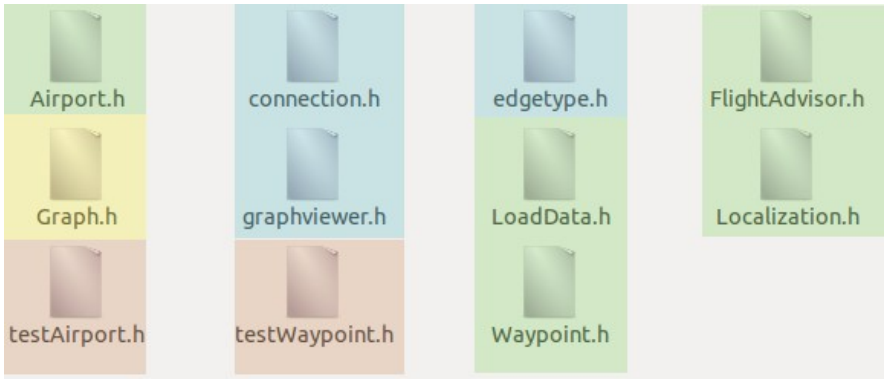

Ilustração 5: Organização dos Ficheiros

Os ficheiros a azul representam as bibliotecas de visualização de grafos numa janela. Os ficheiros a verde são as classes que implementamos para este programa que contêm a lógica de funcionamento do programa. Os ficheiros a vermelho são os de testes unitários do Cute C++ e o ficheiro a amarelo é a Biblioteca que descreve a estrutura de um grafo contendo operações que se articulam sobre o mesmo.

Testes

As classes sobre as quais o programa assenta em termos de dados a serem manipulados, nomeadamente Aeroporto e Waypoint, foram testadas usado a biblioteca de testes unitários *Cute Unit Tester.* Foram criadas as classes *testAirport* e *testWaypoint* para o efeito. São testados diversos casos de criação de instâncias destas classes. Para correr estes testes basta incluir os *headers* dos mesmos e invocar o método estático "run()".

Tratamentos de Exceções

Todos os problemas que são considerados críticos para a correcta execução do programa são tratados com exceções.

1. **InvalidEdgeException:** esta excecão é lançada quando temos uma rota no ficheiro da rede de caminhos que tem um waypoint que não existe.
2. **InvalidFileNameException:** é lançada quando damos o nome de um ficheiro que não existe;
3. **InvalidIATAException:** é lançada quando um aeroporto é criado com um nome de IATA inválido, ou seja, que tem tamanho diferente de três caracteres;
4. **InvalidICAOException:** é lançada quando um aeroporto é criado com um ICAO inválido, ou seja, que tem tamanho diferente de quatro caracteres;
5. **InvalidStringsSizeException:** podemos criar um aeroporto passando um *vector* de s*trings*, que são os dados de aeroporto. Quando o tamanho deste *vector* é inválido, é lançada esta excepção;
6. **notConnectedGraphException:** o grafo da rede do programa tem de ser conexo, por isso se for carregado um grafo que não seja conexo, é lançada esta excecão;
7. **openFileException:** quando existe qualquer problema de abertura com o ficheiro é lançada esta classe.

Algoritmos

São necessários 3 algoritmos cruciais para o funcionamento do programa que devem resolver os seguintes problemas:

1. Obter o melhor caminho num grafo não pesado entre dois nós;
2. Obter o melhor caminho num grafo pesado entre dois nós;
3. Escolher qual dos algoritmos anteriores usar, nas situações adequadas.

Vamos verificar as seguintes situações de caminhos entre nós, em que se pode verificar que o primeiro algoritmo não resolve o problema, pois existem dois bons caminhos entre A e B e, mesmo escolhendo um deles, não há a garantia se a distância entre A e B é a menor possível, por isso nesta situação temos que usar um algoritmo que se articule sobre nós pesados.

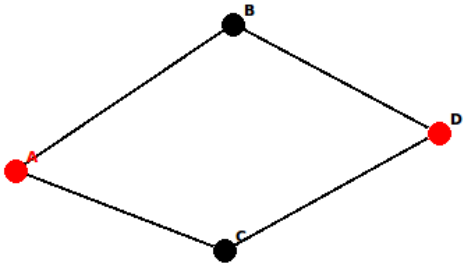

Ilustração 6: Grafo com mais de um caminho óptimo não pesado

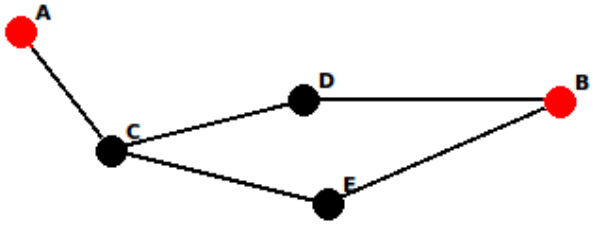

Ilustração 7: Grafo com mais de um caminho óptimo não pesado mais complexo

Obter o melhor caminho num grafo não pesado entre dois nós;

1. Colocar a distância de todos os pontos como infinito e o ponto de origem com distância a 0;
2. Meter todos os vértices numa fila e ir processando cada um;
3. Visitar cada aresta de cada vértice e, se a distância do ponto a que essa aresta leva for infinita, adicioná-lo à fila de prioridade e colocar o pai desse vértice como o actual;
4. Efectuar o passo anterior até a fila ficar vazia;

Ora, este algoritmo é simples e tem complexidade temporal $O(vertices * arestas + vertices)$, sendo a implementação deste a seguinte:

```
for (unsigned int i = 0; i < vertexSet.size(); i++) {
                vertexSet[i]->path = NULL;
                vertexSet[i]->dist = INT_INFINITY;
        }
        Vertex<T>* v = getVertex(s);
        v->dist = 0;
        queue<Vertex<T>*> q;
        q.push(v);
        while (!q.empty()) {
                v = q.front();
                q.pop();
                for (unsigned int i = 0; i < v->adj.size(); i++) {
                        Vertex<T>* w = v->adj[i].dest;
                        if (w->dist == INT_INFINITY) {
                                w->dist = v->dist + 1;
                                w->path = v;
                                q.push(w);
                        }
                }
        }
```

Obter o melhor caminho num grafo pesado entre dois nós

Duas restrições devem ser tido em conta na utilização do algoritmo do Dijkstra na implementação realizada assim como uma nota adicional deve ser exposta:

1. A rede (= grafo) utilizada deve ser conexa, o que significa que, a partir de qualquer ponto V, de um grafo G, deve ser possível atingir um ponto U, não necessariamente em linha recta.
2. O "peso" de uma aresta deve ser positivo ou nulo, neste tipo de programa o peso assemelha-se a distância entre dois pontos, mas num outro grafo poderá ser, por exemplo, a o tempo necessário para atingir uma certa etapa.

Nota importante: no problema abordado o grafo é não orientado.

Por que escolher o algoritmo do *Dijkstra*?

É importante relembrar que no primeiro esboço do trabalho foi decidido utilizar o algoritmo de *Floyd*, pois este algoritmo permitiria calcular a distância entre qualquer dois pontos do grafo, mas não se revelou ser uma boa opção por varias razões. O algoritmo de Dijkstra tem imensas vantagens em relação ao algoritmo de Floyd, considerando o exemplo seguinte: acrescentar um aeroporto a uma rede já definida que já contém uma lista de melhores caminhos.

1. No caso do *Floyd*: se assim for a situação, o *Floyd* terá de recalcular todos os pontos entre si, ou seja não aproveitará os caminhos pré-existentes o que não tem razão de ser;
2. No caso do *Dijkstra*: dado que este algoritmo parte de um ponto de origem e calcula todos os caminhos para os outros pontos, bastará lhe indicar que queremos partir do ponto novo e ele criará apenas as rotas necessárias o que resulta num ganho de tempo muito importante.

O *Dijkstra* também tem uma outra importante vantagem: ele vai dar o caminho mais curto de um ponto de origem a todos os outros em $O(a.ln(n))$. Sendo assim, para um mesmo resultado *Floyd* conseguirá uma complexidade temporal de $O(n^3)$ enquanto que o *Dijkstra* terá uma complexidade de $O(a^{(2)}.\ln(n))$, onde "a" é o numero de aresta do grafo (conexo), ou seja $O(n)$. Ou seja, é um algoritmo mais rápido na situação abordada.

Descrição do algoritmo

O algoritmo do *Dijkstra* é um algoritmo ganancioso, que em cada etapa escolhe a solução óptima. Falta definir a maneira de escolher esse vértice e o tratamento a efectuar, em suma, é o algoritmo em si.

Ao longo do cálculo é necessário manter dois conjuntos:

1. C, conjunto de vértice que falta visitar. No inicio C = V - {fonte}, V representando os vértices presentes no grafo;
2. D, conjunto de vértices para os quais já se conhece o caminho mais curto desde a fonte. No inicio D = {fonte}.

O algoritmo acaba quando o conjunto C for vazio.

Para cada vértice *v* no conjunto D, conserva-se numa "tabela distância" o peso do caminho mais curto até a fonte, e, numa "tabela caminho" o vértice que o antecede num caminho mais curto da fonte a *v*. Sendo assim para descobrir o caminho mais curto bastará "seguir o caminho" em sentido contrário de "Pai" em "Pai" por assim dizer.

Inicialização do Algoritmo

No inicio do algoritmo, o caminho mais curto entre a fonte e cada um dos vértices é o

caminho directo, com uma aresta de peso infinito se não tiver ligação entre os dois vértices. Inicia-se a "tabela distância" com os pesos das arestas ligando a fonte a cada vértice, e a "tabela caminho" por fonte para cada vértice.

Etapa i

Supondo que depois de tratar vértice i, a tabela "caminho" e a tabela "distância", contêm respectivamente o peso e o antecessor dos mais curtos caminhos para cada um dos vértices já tratados.

Sendo *v* o vértice do conjunto "C", realizando o mínimo da tabela "distância", suprime-se *v* do conjunto "C" e o adicionamos ao conjunto "D". Basta fazer o *update* das tabelas "distância" e "caminho" para os vértices *u* ligados directamente a *v* por uma aresta, como explicado a seguir: se distância[v] + F(v,u) < distância[u], então troca-se a distância[u] por distância[v] + F(v,u) e caminho[u] por *v*. (sendo F o "peso" entre dois vértices, neste caso representando um distância).

(n-2)-ésima etapa

No início existem (n-1) vértices por visitar, mas como explicado mais a frente, a última etapa é inútil porque não trás nada de importante, Assim desde a (n-2)-ésima etapa, as tabelas "distância" e "caminho", contêm toda a informação necessária para encontrar os caminhos mais curtos a cada um dos vértices (pois D = V).

- Distância[v] é o peso do caminho mais curto da fonte a *v*;
- Caminho[u] é o Pai de *v* num caminho mais curto da origem a *v*.

Finalmente bastará correr o algoritmo para cada ponto do grafo evitando obviamente calcular caminhos que já foram previamente calculados num sentido, por exemplo se já se calculou um caminho de A para B não se calcula o caminho de B para A dado que o grafo é conexo e não orientado (como referido anteriormente).

Pseudo-código

```
function Dijkstra(Graph, source):
   for each vertex v in Graph:            // Initializations
      dist[v] := infinity ;              // Unknown distance function from source to v
       previous[v] := undefined ;         // Previous node in optimal path from source
    end for ;
    dist[source] := 0 ;                   // Distance from source to source
    Q := the set of all nodes in Graph ;  // All nodes in the graph are unoptimized - thus are in Q
    while Q is not empty:                  // The main loop
```

```
         u := vertex in Q with smallest distance in dist[] ;
         if dist[u] = infinity:
          break ;    // all remaining vertices are inaccessible from source
        end if ;
       remove u from Q ;
        for each neighbor v of u:  // where v has not yet been removed from Q.
            alt := dist[u] + dist_between(u, v) ;
            if alt < dist[v]:              // Relax (u,v,a)
                 dist[v] := alt ;
                 previous[v] := u ;
                 decrease-key v in Q;       // Reorder v in the Queue
            end if ;
         end for ;
    end while ;
     return dist[] ;
  end Dijkstra.
```

Escolher qual dos algoritmos anteriores usar nas situações adequadas

Este algoritmo tem de ser capaz de seleccionar entre os dois algoritmos de caminho mais curto, resolvendo o problema dos grafos ilustrados anteriormente. Este foi implementado da seguinte maneira:

```
template<class T>
// Brute force solution
bool Graph<T>::havePathsWithSameSize(const T &o, const T &d) {
        // (1) Calculate the best path between o and d
        vector<T> bestPath = bestPathUnweighted(this, o, d);
        // (2) Run that path and by removing each time a node, check if is possible another path
        for (int unsigned i = 1; i < (bestPath.size() - 1); i++) {
                Graph<T> tempGraph;
                this->clone(tempGraph);
                tempGraph.removeVertex(bestPath[i]);
                vector<T> tempBestPath = bestPathUnweighted(&tempGraph, o, d);
                if (tempBestPath.size() == bestPath.size())
                        return true;
        }
        return false;
}
```

Ou seja, calculamos o melhor caminho sem contabilizar pesos e vamos no grafo tirar os pontos intermédios entre A e B e verificar se existe outro caminho óptimo possível. Se existir, então temos de usar o *Dijkstra*.

Casos de Utilização

A utilização do programa assenta básica em receber os dados dos aeroportos de origem e de destino e imprimir as rotas. O utilizador pode ainda omitir o aeroporto de destino ou ambos, dando o programa as rotas adequadas.

Caso de utilização número 1

- Nome: selecção do caminho entre 2 aeroportos (origem e destino)
- Descrição: o utilizador liga o programa e preenche, no sítio reservado a esse efeito, dois aeroportos;
- Actor: o utilizador do programa;
- Referência: nenhuma por enquanto;
- Pré-requisitos: programa operacional;
- Consequência: O utilizador receberá o caminho mais curto a seguir para chegar ao ponto de destino.

Sequência de eventos

1. O utilizador abre o programa;
2. O sistema pede ao utilizador os dados da viagem ;
3. O utilizador preenche os dados com aeroporto de origem e o de destino;
4. O sistema procura na base de dados o melhor caminho com origem e destino pedidos pelo utilizador ;
5. O sistema apresenta ao utilizador o caminho.

Excepções

1. O utilizador introduz um aeroporto que não existe;
2. O sistema encontra o erro e pede novamente outro aeroporto;
3. O utilizador introduz outros dados.

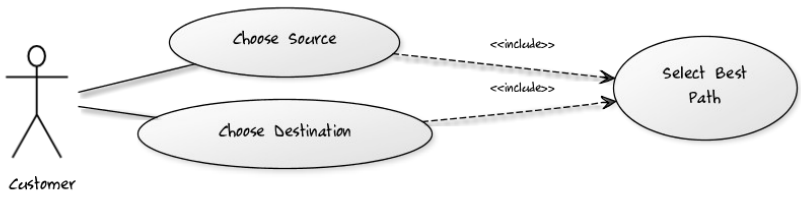

Ilustração 8: Diagrama de Utilização 1

Caso de utilização número 2

- Nome: selecção de caminho entre aeroporto de origem e qualquer outro aeroporto
- Descrição: o utilizador liga o programa e preenche, no sitio reservado a esse efeito um aeroporto;
- Actor: o utilizador do programa;
- Referência: nenhuma por enquanto;
- Pré-requisitos: programa operacional;
- Consequência: O utilizador receberá o caminho mais curto para todos os aeroportos pertencentes a rede a partir do ponto de origem seleccionado.

Sequência de eventos

1. O utilizador abre o programa;
2. O sistema pede ao utilizador os dados da viagem;
3. O utilizador preenche os dados com aeroporto de origem;
4. O sistema procura na base de dados o melhor caminho com a origem introduzida pelo utilizador e qualquer outro aeroporto;
5. O sistema apresenta ao utilizador todos os caminhos encontrados.

Excepções

1. O utilizador introduz um aeroporto que não existe;
2. O sistema encontra o erro e pede novamente outro aeroporto;
3. O utilizador introduz outros dados.

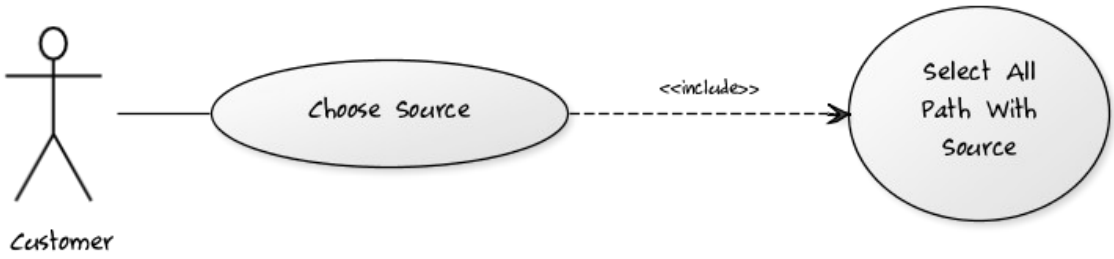

Ilustração 9: Diagrama de caso de utilização 2

Caso de utilização número 3

- Nome: selecção de caminho quaisquer par de aeroportos
- Descrição: o utilizador liga o programa não preenche nem destino, nem origem;
- Actor: o utilizador do programa;
- Referência: nenhuma por enquanto;
- Pré-requisitos: programa operacional;
- Consequência: o utilizador recebera o caminho mais curto a seguir para chegar de qualquer aeroporto a qualquer outro.

Sequência de eventos

1. O utilizador abre o programa;
2. O sistema perde ao utilizador os dados da viagem;
3. O utilizador não preenche nem destino, nem origem;
4. O sistema apresenta ao utilizador todos os caminhos mais curtos contido na base de dados;

Excepções

1. O utilizador introduz um aeroporto que não existe;
2. O sistema encontra o erro e pede novamente outro aeroporto;
3. O utilizador introduz outros dados.

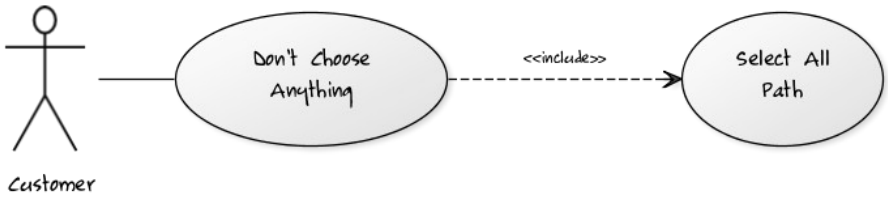

Ilustração 10: Diagrama de caso de utilização 3

Desenvolvimento do projeto

O desenvolvimento do projeto foi faseado. Numa primeira fase delineamos o modo como as rotas entre os aeroportos seriam representadas, chegando assim a um consenso. De seguida desenvolvemos os algoritmos para fazer o carregamento dos dados para um grafo.

Estando a parte dos dados serem carregados para o grafo feita, seguiu-se a etapa mais importante, que foi o desenvolvimento de soluções que permitem calcular as rotas entre determinados aeroportos, recorrendo-se assim aos algoritmos anteriormente enumerados.

Finalmente, após a parte de carregamento e processamento de dados feitas, desenvolvemos a representação gráfica das rotas, usando o Graphviewer que foi fornecido

Contribuição de cada elemento

1. **André Freitas:** Desenvolvimento da solução em C++, Criação da Documentação Doxygen, Modelo UML e Relatório;
2. **Olivier Alves:** Criação da Rede de Rotas; Desenvolvimento do carregamento de dados em C++; Relatório;
3. **Tiago Tavares:** Definição dos cabeçalhos das classes inicialmente definidas.

Conclusão

O programa criado permite responder a um pedido principal a saber dar a conhecer ao utilizador o melhor caminho entre dois pontos de uma forma amigável, usando um algoritmo de selecção de caminho baseando-se num grafo conexo, pesado e não dirigido.

Este foi optimizado para ser o mais versátil possível de facto é possível modificar a rede e continuar a utilizar o programa sem perdas de dado e sem precisar de modificar a rede inteira acrescentando unicamente dados suplementares, assim como ser o mais rápido possível usando o *Dijkstra*, apesar de existirem outros algoritmos, que embora, parecem a partida melhor não se revelaram ser a melhor opção pelas sua complexidades temporais elevadas. Foram usados Aeroportos com localizações reais, descrevendo o mapa da Europa. As rotas que foram utilizadas permitem testar de uma maneira eficaz as especificações fornecidas.

Alcançamos todos os nossos objectivos em termos de eficiência do programa, ergonomia e acessibilidade, cumprindo todas as especificações fornecidas. Consideramos assim o "Flight Advisor" um "produto final".

Bibliografia

1. Wikipedia. “Dijkstra's algorithm”. Wikipedia. http://en.wikipedia.org/wiki/Dijkstra's_algorithm. (acedido dia 28 de Abril)