EasyPilot

Relatório 1

Mestrado Integrado em Engenharia Informática e Computação

CONCEPÇÃO E ANÁLISE DE ALGORITMOS

2MIEIC05, E, 1
Luís Costa - 200806068
Alexandre Ribeiro - up201205024
José Mendes - up201200647
*

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

April 2016

*Github - `https://github.com/01alex/FEUP-CAL`

Conteúdo

INTRODUÇÃO

No âmbito da unidade curricular de Concepção e Algoritmos de Dados foi proposto a realização de um projecto no qual apresentariamos um sistema GPS, desenvolvido em C++, recorrendo a algoritmos que foram desenvolvidos ao longo do semestre.

Easy-Pilot, é um navegador que identifica o caminho a seguir, numa dada rede, a partir de uma origem até ao destino desejado. O itinerário poderá ser simples, ou ainda incluir vários pontos de interesse (POIs), como bombas de combustível para reabastecimento, monumentos, ou outros cuja posição sejam indicadas pelo utilizador; outros critérios a utilizar poderão incluir menor distância, menor tempo de viagem, e ainda a existência ou não de pontos de portagem.

Mediante o problema proposto o grupo desenvolveu uma solução que entendeu ser eficiente e adequada ao problema apresentado. Sabendo da existência de vários algoritmos associados ao *route-finding system*, o grupo tentou associar a eficiência ao que era exequível. Há a certeza de que existiram algoritmos mais correctos e que trariam uma solução substancial, mas privelgiámos encontrar uma solução óptima que tivesse resultados concretos e avaliáveis.

Ao longo deste projecto vamos demonstrar as várias etapas de realização do projecto bem como desenvolver uma crítica ao trabalho feito. A ideia principal deste relatório é elaborar uma crítica fundamentada aos algoritmos de *pathfinding* e enaltecer os conceitos primordiais desta ideia.

PATH FINDING ALGORITHMS

Path finding ou *pathing* é nome dado a uma aplicação informática que tenha como função determinar o caminho mais curto entre dois pontos. Este campo de pesquisa está fortemente ligado ao algortimo desenvolvido por um famoso cientista de computação holandês, **Edsger Dijkstra**. Pode assumir-se que Djikstra é o pai do *pathing* computacional tendo uma forte contribuição em toda a ciência computacional. Até então os algoritmos existentes para a solução possuíam complexidade cúbica em relação à quantidade de nós envolvidos, enquanto o *algoritmo de Dijkstra* possuía complexidade quadrada, potencialmente menor que os anteriores.

Figura 1: *Edgar Djikstra*

Várias modificações foram sugeridas ao algortimo de Djikstra usando heurísticas para reduzir o tempo de computação exigido na procura do caminho mais curto. Uma das "heurísticas"mais usadas é o **A*** (aStar) *search*

algorithm cujo objectivo é reduzir o tempo de execução reduzindo o espaço de procura. O algortimo A* foi desenvolvido por *Peter Hart* e é uma extensão do algortimo de Djikstra, 1959.

A algortimo de Djikstra e A* são conceitos muito importantes que queremos reter ao longo do relatório visto que são dos dois algortimos óptimos e que trariam os melhores resultados para o nosso problema.

Porém, ao longo dos anos novas soluções foram apresentados e em 2009 a google usou um método designado *Contraction hierarchies* que consiste numa técnica que acelera o *shortest path routing* criando um pre processamento do grafo original originando multi camadas de nós organizados hierarquicamente. Desde então o número de algoritmos têm proliferado e hoje em dia somos capazes de encontrar os mais variados algoritmos com tempos de computação espantosos. *Contracted Hierarchies* pode ser usado para encontrar *shortest path routes* de uma maneira muito mais eficiente que o Djikstra e é usado em várias técnicas de *routing* avançadas. Apesar da Google não desvendar que algoritmos usa para estruturar os seus mapas é consensual que será alguma variância de Contracted Hierarchies.

Hub Labeling Algorithms

Hub labeling é uma estutura de dados usada para encontrar a distância entre dois pontos num grafo. Providencia um excelente *query time* para *real-world graphs* como redes de estrada e redes sociais. Hub labelling tem dois estados - **préprocessamento** e ***querying***. Enquanto que a parte de *query* é simples o préprocessamento pode demorar muito tempo. Existe um algoritmo para encontrar $O(\log n)$-aproximadamente *Hub Labels* óptimos. No entanto, é lento. A ideia é que os algoritmos de *Hub Labeling* criem uma estratificação hierárquica. Sendo assim os **HHL** (Hirarchichal Hub Labeling) ordenam os vértices e encontrarm os hub labels que respeitem a ordem. Concluindo, os algoritmos de *path finding* têm ganho um grande relevo nos últimos anos e é cada vez mais um assunto a ser investigado e explorado. Apesar do algortimo conseguir resolver os problemas em quase tempo linear, aplicações que abrangem continentes necessitam de algoritmos mais rápidos. O pré processamento torna possivel a exitência de algoritmos sub lineares.

EASY PILOT

Após uma introdução ao tema que se vai tratar ao longo do relatório, damos agora uma maior especificação do que vai consistir o nosso projecto. Neste trabalho, pretende-se implementar um navegador que identifique o caminho a seguir, numa dada rede, a partir de uma origem até ao destino desejado. O itinerário poderá ser simples, ou ainda incluir vários pontos de interesse (POIs), como bombas de combustível para reabastecimento, monumentos, ou outros cuja posição sejam indicadas pelo utilizador; outros critérios a utilizar poderão incluir menor distância, menor tempo de viagem, e ainda a existência ou não de pontos de portagem.

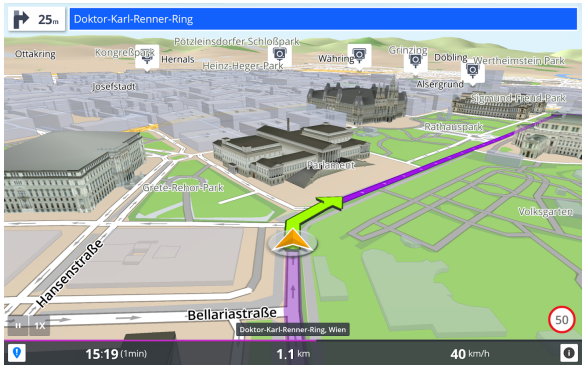

Figura 2: Exemplo de um sistema navagação GPS

Para resolver este problema o grupo começou por fazer uma engenharia de requisitos, funcionais e não funcionais e estabelecer metas que acharia pertinentes e exequíveis.

Começamos por decidir qual seria o tamanho mais adequado ao nosso mapa. O mapa teria de ser suficientemente grande para testar várias computações com tempos de execução diferentes mas não poderia ser demasiado grande para que a validação e verificação do software fosse feita de uma maneira mais pragmática. Posto isto, decidimos abranger grande parte da zona do Porto, incluindo a Faculdade de Engenharia da Universidade Do Porto e a Baixa.

Escolhida a nossa base de dados era necessário ter um *parser* que nos permitisse extrair todas as informações necessárias para formar o nosso **grafo**. Ao longo das aulas práticas foi-nos informado que essa informação seria facultada pelos monitores da cadeira, mas como tal demorou a acontecer o grupo investigou e conseguiu extrair os seus dados através de outras ferramentas. Para isso utilizadmos um *routing engine* que faz o *parsing* da informação existente no *Open Street Maps* e faz com que seja possível fazer a *route* de toda essa informação. Gera o *SQL Insert scripts* para *PostGIS* compatível com *pgRouting* e *Quantum GIS*. Dada a base de dados, foi feita uma query para gerar a informação que acharíamos pertinente.

```
1     create table vertices as
2     select source as ID, st_makepoint(x1,y1) as geom from edges_table
3     union
4     select target as ID, st_makepoint(x2,y2) as geom from edges_table
```

Resposta à *query*:

```
1  220;"Rua de Sendim";8629;45558;0.0304899;60;-8.6757897;41.1871374;-8.675474;41.1872743
```

Que representam, respectivamente, o **id**, o **nome**, o **source**, **target**, a **distancia**, **a velocidade coordenada X1**, **coordenada Y1**, **coordenada X2** e a **coordenada Y2**.

Nesta tabela cada registo repressenta uma **aresta**. Cada aresta tem dois **vértices** *source* e *target* cujas coordenadas estão nas colunas *x1*, *y1*, *x2*, *y2*.

Com esta informação criamos um ficheiro de texto (.txt) que mais tarde seria utilizada para preencher o nosso grafo.

Q & A

1. Q — Como vai ser representado o mapa?

   A — Vai ser representado através de um grafo dirigido

2. Q — Quais serão os elementos do grafo?

   A —

   - Vértices (*vertexes*) que têm uma informacao, que e um comparável, neste caso uma *intersection*.
   - Arestas (*Edges*) que representam as ruas.
   - Um Estado inicial que será o sítio de onde o utilizador se quer deslocar
   - Um Estado Final que será o local para onde o utilizador quer ir.
   - Todos os vértices devem ser atingíveis.

3. Q — O que é um caminho ponta a ponta e o que é que representa?

   A — É um caminho que começa num local e acaba noutro, neste caso representa a viagem que o utilizador quer realizar.

4. Q — Como é dado o comprimento de um caminho?

   A — O comprimento do caminho é dado através do somatório total do comprimento das arestas

5. Q — Quais foram os algoritmos utilizados para resolver o problema?

   A — **Djikstra** e **A***(A star) foram os algoritmos utilizados para chegar a uma solução óptima do problema.

FORMALIZAÇÃO DO PROBLEMA

Formalizamos agora o problema de acordo com aquela que achamos ser a melhor forma para resolver aquilo a que nos propusemos.

- **Input**.
  $G < V, E >$ V: Pontos do mapa
  E: Ligação entre dois pontos - ruas.

- **Output**.
  Path $= V_i, I = 1...n$
  Value

- **Objectivo**
  $\min(valor)$
  $valor = \sum_{i=1}^{n} E_{ij}$

- **Restrição**
  $\forall i \in Path = \{V_i\}$
  $\exists j \in Path : V_i = V_j \wedge i \neq j$

ASSUMPÇÕES DO PROBLEMA

- Dado que tivemos que encurtar o grafo para que fosse mais fácil para efeitos de validação, nem todos os **vértices** têm solução. Porém, aplicada uma verificação para saber se é possível e se não é requistado que o utilizador introduza novos valores.
- A route será calculada será sempre a mais óptima e eficiente consoante o algoritmo escolhido.

SOLUÇÃO CONCEPTUAL

Djikstra

Para encontrar uma solução, um dos algoritmos que implementamos foi o **Djikstra** que já foi falado ao longo do relatório. Decidimos usar o Djikstra porque este permite encontrar o **caminho mais curto** entre dois pontos considerando apenas o seu custo real.

Tempo computacional do Djikstra:

$O(|E| + |V| \log |V|))$

Dijkstra tem uma função de um custo que é o valor real da *source* para o *target*.

$f(x) = g(x)$

Dijkstra não tem heurística e a cada passo escolhe arestas com o custo inferior, ou seja, tende sempre a cobrir uma área muito maior do nosso grafo.

Este algoritmo é um algoritmo ganancioso, tomando decisões que parecem óptimas no momento, determinando assim o conjunto de melhores caminhos intermediários.

Pseudo código

1. Inicialização de todos os nodes com distância "infinita"; inicialização da starting node a 0;
2. Marcar a distância da starting node como permanente e todas as outras como temporárias
3. Marcar a *starting node* como activa;
4. Calcular as distâncias temporárias de todos os vértices vizinhos do vértice *activo* somando a sua distância com o peso das arestas;
5. Se a distância calculada de um vértice for inferior ao que está a ser examinado, fazer *update* à distância e colocar o vértice activo como antecessor. Este é um processo de *update* e é fundamental no funcionamento do Dijkstra.
6. Colodar o vértice com a menor distância temporária como activo e marcar a sua distância como permanente.
7. Repetir os passos 4 a 7 até não existirem mais vértices com distância permanente cujos vizinhos tenham distâncias temporárias.

```
1          function Dijkstra(Graph, source):
2          for each vertex v in Graph:     // Inicializacao
3          dist[v] := infinity     // distancia inicial da source ate ao vertex v posta
               infinito
4          previous[v] := undefined        // Previous node in optimal path from source
5          dist[source] := 0       // Distancia da source a source
6          Q := the set of all nodes in Graph      // todos os nos no grafo nao foram
               processatos e sao adicionados a Q (pode ser priority queue)
7          while Q is not empty:   // main loop
8          u := node in Q with smallest dist[ ]
9          remove u from Q
10         for each neighbor v of u:       // v nao foi removido de Q
11         alt := dist[u] + dist_between(u, v)
```

```
12          if alt < dist[v]        // Relax (u,v)
13          dist[v] := alt
14          previous[v] := u
15          return previous[ ]
```

Implementação

Esta foi a implementação do grupo:

```
1
2           template<class T>
3   void Graph<T>::DijkstraShortestPath(const T &start){            //baseado teorica 06.
        grafos2_a
4
5           for(unsigned i=0; i<getNumVertex(); i++){
6                   vertexSet[i]->path = NULL;
7                   vertexSet[i]->setDistance(INF);
8                   vertexSet[i]->processing = false;
9           }
10
11          Vertex<T> *s = vertexSet[addVertex(start)];
12          s->setDistance(0);
13
14          vector<Vertex<T>*> pq;
15
16          pq.push_back(s);
17
18          make_heap(pq.begin(), pq.end());
19
20          while(!pq.empty()){
21
22                  s = pq.front(); // diferente do git daquele gaj
23                  pop_heap(pq.begin(), pq.end());
24                  pq.pop_back();
25
26                  for(unsigned i=0; i<s->adj.size(); i++){
27
28                          Vertex<T> *w = s->adj[i].dest;
29
30                          if( (s->distance + s->adj[i].length) < w->distance){
31
32                                  w->setDistance(s->distance + s->adj[i].length);
33                                  w->path = s;
34
35                                  if(!w->processing){
36                                          w->processing = true;
37                                          pq.push_back(w);
38                                  }
39
```

```
40                                  make_heap(pq.begin(), pq.end(), vertex_greater_than<T>())
                                        ;
41                              }
42                      }
43              }
44
45  }
```

Resultados

Dado que o Dijkstra tem em atenção o tamanho do grafo para realizar a sua computação, realizamos vários testes nos quais aumentamos a nossa base de dados. Como dito anteriormente, a nossa base de dados original tinha informação de todo o distrito do Porto. Com vista a avaliar os resultados e compreender se estão de acordo com com o esperado e aquilo que é o funcionamento do Dijkstra.

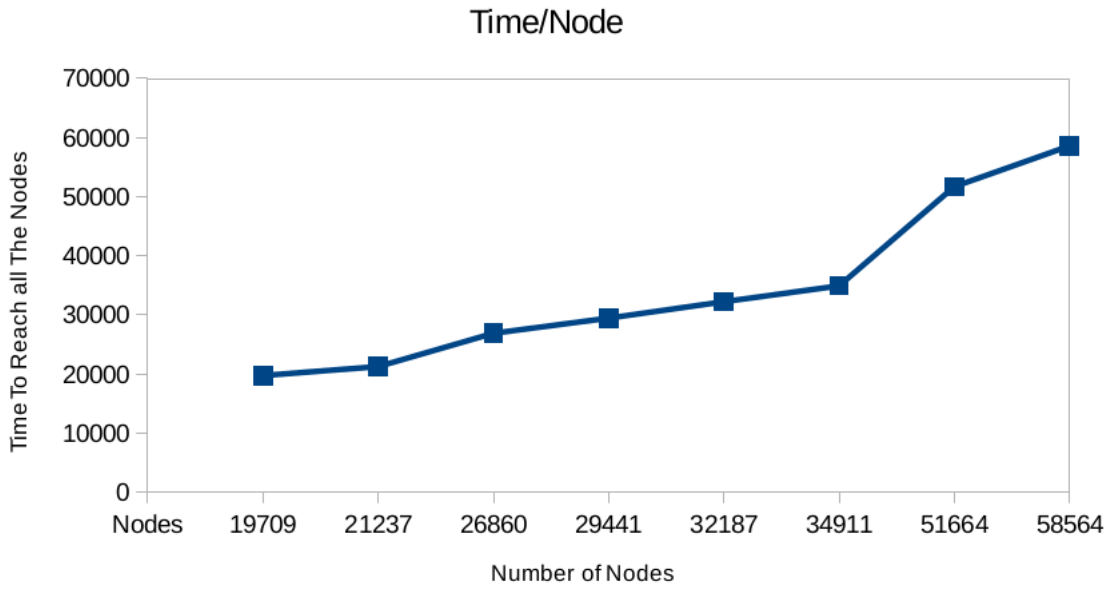

Figura 3: Grárfico tempo de computação Djikstra por número de nodes

Tal como previamos a função do djikstra ao longo do tempo tem uma evolução logaritmica. Apesar de não ser bem compreensível através da imagem, se tivessemos colocado mais pontos, ou seja, uma maior variedade do número de vértices o Djikstra iria demorar mais tempo a executar. Isto acontece porque Djikstra não possui heurísticas, ou seja, quanto maior for o grafo maior serão o número de vértices e consequentemente o tempo que o algoritmo vai demorar a chegar a todos os vértices pelo seu *real cost* também será maior.

Time(s)	Nodes
0.2076189965	1979
0.2234530002	21237
0.2633250058	26860
0.2849009931	29441
0.3500050008	32187
0.4250009954	34911
0.9803569913	51664
0.9691500068	58564

Tabela 1: Tabela de Resultados

UMA SOLUÇÃO MELHORADA

A*

Como esforço complementar e por questões de compreensão e brio, o grupo decidiu implementar um segundo algoritmo, o **A***. O A* foi um algoritmo que foi descrito pela primeira vez em 1968 por Peter Hart, Nils Nilsson e Bertram do *Instituto de pesquisa de Stanford*. **A*** é uma variação ou generalização do **Djikstra** sendo que a sua principal diferença é que o **Djikstra** não usa heurísticas para encontrar a melhor aproximação ao caminho mais curto.

Porém, muitas vezes surgem más interpretações em relação a estes dois algoritmos pelo que tentamos esclarecer alguns conceitos.

O A é mais rápido que o Djikstra e usa uma best-first-search para acelerar o processo.*

O **A*** é uma variação do Djikstra. A* é considerado um *best-first-search* porque gananciosamente escolhe qual o vértice a explorar de seguida de acordo com o valor de

$$g(x) = h(x) + g(x)$$

onde o h é a **heurística** e o g o custo até ao momento.

Notar que se se usar uma heurística não informativa, $h(x) = 0$ para cada vértice, faz-se com que o **A*** escolha o vértice a desenvolver baseado no "custo até ao momento", o mesmo que o algoritmo de Djikstra. Portanto, se $h(x) = 0$, **A*** volta a ser **Djikstra**.

*Se eu precisar de um algoritmo para correr em milissegundos, quando é que o **A*** se torna mais promissor?*

Não é algo que seja linear e depende de alguns conceitos. Se tiver uma heurística decente (*e.g Greedy best first*) é mais rápido que o **A*** mas não é, de todo, uma solução óptima.

A não retorna necessariamente os melhores resultados*

A* é completo (encontra um caminho se existir) e óptimo (encontra sempre o **caminho mais curto**) se usarmos uma função heurística admissível.

Se precisar de resultados rápidos, é melhor pré-computar os caminhos? Pode ocupar megabytes de espaço para os guardar

Em alguns problemas, pré computar é uma optimização coumum como por exemplo no problema *15-puzzle*, mas muito mais avançada. O caminho de um ponto A para um ponto B é chamado um **macro**. Alguns caminhos são muito úteis e devem ser relembrados. **Machine learning** é adicionado ao algoritmo de maneira que os Macros possam ser memorizados mais facilmente.

Resumindo, **A*** apenas expande um vértice se este parecer promissor. Apenas se foca em atingir o vértice final do vértice actual e não todos os outros vértices. É óptimo, se a heurística for admissível. Então, se a heurística é boa para aparoximar o custo futuro, vão ser precisos serem examinados muito menos vértices que com o **algoritmo de Djikstra.**

Pseudo código

```
function A*(start, goal)
    // The set of nodes already evaluated.
    closedSet := {}
    // The set of currently discovered nodes still to be evaluated.
    // Initially, only the start node is known.
    openSet := {start}
    // For each node, which node it can most efficiently be reached from.
    // If a node can be reached from many nodes, cameFrom will eventually contain the
    // most efficient previous step.
    cameFrom := the empty map

    // For each node, the cost of getting from the start node to that node.
    gScore := map with default value of Infinity
    // The cost of going from start to start is zero.
    gScore[start] := 0
    // For each node, the total cost of getting from the start node to the goal
    // by passing by that node. That value is partly known, partly heuristic.
    fScore := map with default value of Infinity
    // For the first node, that value is completely heuristic.
    fScore[start] := heuristic_cost_estimate(start, goal)

    while openSet is not empty
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)
        closedSet.Add(current)
        for each neighbor of current
            if neighbor in closedSet
                continue               // Ignore the neighbor which is already evaluated
                    .
            // The distance from start to a neighbor
            tentative_gScore := gScore[current] + dist_between(current, neighbor)
            if neighbor not in openSet  // Discover a new node
```

```
                openSet.Add(neighbor)
            else if tentative_gScore >= gScore[neighbor]
                continue                // This is not a better path.

            // This path is the best until now. Record it!
            cameFrom[neighbor] := current
            gScore[neighbor] := tentative_gScore
            fScore[neighbor] := gScore[neighbor] + heuristic_cost_estimate(neighbor, goal
                )

    return failure

function reconstruct_path(cameFrom, current)
    total_path := [current]
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.append(current)
    return total_path
```

Implementação

```

template <class T>
float heuristic_aStar(Vertex<T> *s, Vertex<T> *t, bool dt){
        return (sqrt(pow(s->getIntersection().getCoord().x - t->getIntersection().
            getCoord().x, 2) + pow(s->getIntersection().getCoord().y - t->getIntersection
            ().getCoord().y, 2)));
}


template<class T>
void Graph<T>::updateFlood(Vertex<T>* vertex, T goal) {
        if((vertex->closed != NULL) && (!(vertex->closed)))
                return;

        for(int i = 0; i < vertex->adj.size(); i++) {

                vertex->adj[i].getDest()->setDistance(vertex->adj[i].getDest()->
                    getDistance()  + vertex->adj[i].getLength());
                vertex->adj[i].getDest()->fx = vertex->adj[i].getDest()->getDistance() +
                    heuristic_aStar(vertex, vertexSet[addVertex(goal)],1);
                updateFlood(vertex->adj[i].getDest(), goal);

        }
}

template <class T>
void Graph<T>::aStar(const T &start, const T &goal, bool dist){ //MAKE THIS BOOL

        for(unsigned i = 0; i < vertexSet.size(); i++) {
```

```
26                  vertexSet[i]->path = NULL;
27                  vertexSet[i]->distance = INF;
28                  vertexSet[i]->fx = INF;
29                  vertexSet[i]->closed = NULL;
30                  vertexSet[i]->inQueue = false;
31          }
32
33
34          Vertex<T> *s = vertexSet[addVertex(start)];
35
36          s->setDistance(0);
37          s->closed = false;
38          s->fx = s->getDistance() + heuristic_aStar(s, vertexSet[addVertex(goal)], dist);
39
40          vector<Vertex<T>*> pq;          //nextVertices
41
42          pq.push_back(s);
43          s->inQueue = true;
44
45          //cout << s->getIntersection().getID() << endl;
46
47          make_heap(pq.begin(), pq.end());
48
49          while (!pq.empty()) {
50                  Vertex<T> *current = pq.front();
51                  pop_heap(pq.begin(), pq.end());
52                  pq.pop_back();
53
54
55                  current->inQueue = false; //extra(?)
56
57                  if (current == vertexSet[addVertex(goal)])
58                          break;
59
60
61                  current->closed = true;
62
63                  for(unsigned i = 0; i < current->adj.size(); i++) {
64                          Edge<T> edge = current->adj[i];
65                          Vertex<T> *neighbour = edge.dest;
66                          double weight = dist ? edge.length : neighbour->time;
67
68                          if(neighbour->closed == NULL) {
69                                  neighbour->closed = false;
70                                  neighbour->distance = current->distance + weight;
71                                  neighbour->path = current;
72                                  neighbour->fx = neighbour->distance + heuristic_aStar(
                                        neighbour, vertexSet[addVertex(goal)], dist);
73
74                                  pq.push_back(vertexSet[addVertex(neighbour->
                                        getIntersection())]);
75
```

```
76                          }
77
78                          if (!neighbour->closed) {
79                                  if (neighbour->distance > current->distance + weight) {
80                                          neighbour->distance = current->distance + weight;
81                                          neighbour->path = current;
82                                          neighbour->fx = neighbour->distance +
                                                heuristic_aStar(neighbour, vertexSet[
                                                addVertex(goal)], dist);
83                                  }
84                          }
85
86
87                          if(neighbour->closed) {
88                                  if(neighbour->distance > current->distance + weight) {
89                                          neighbour->path = current;
90                                          neighbour->distance = current->distance + weight;
91                                          neighbour->fx = neighbour->distance +
                                                heuristic_aStar(neighbour, vertexSet[
                                                addVertex(goal)], dist);
92                                          updateFlood(neighbour, goal); //com problemas
93                                  }
94                          }
95
96                          make_heap(pq.begin(), pq.end(), vertex_greater_than_fx<T>());
97
98                  }
99          }
100
101
102         return;
103 }
```

Heurística

A função heurística $h(n)$ diz a **A*** uma estimativa do custo mínimo de cada vértice(source) até outro(target). É muito importante escolher uma boa função heurística.

A heurística pode ser usada para controlar o comportamento do **A***.

- Num extremo se $h(n)$ é 0, então apenas o $g(n)$ tem preponderância e o A* transforma-se no algoritmo de Djikstra e é garantido que vai encontrar **o caminho mais curto.**
- Se o $h(n)$ é sempre **menor (ou igual)** ao custo de mover de n até ao *goal*, então A* é garantido que encontra o **caminho mais curto**. Quando menor for $h(n)$, mais o A* se expande, fazendo com que seja **mais lento.**
- Se $h(n)$ for **exactamente igual** ao custo de se mover de n até ao *goal*, então A* seguirá **o melhor caminho** e nunca vai expandir mais nada, o que o torna **muito rápido**. Apesar de não poder ser possível fazer isto acontecer na maior parte dos casos, é possível faze-lo exacto em alguns

casos especiais. É entusiasmante saber que se dermos a informação perfeita, A* comportar-se-á na perfeição também.

- Se às vezes o $h(n)$ é maior que o custo de se mover de n para *goal*, então não é garantido que A* possa encontrar o **caminho mais curto** mas vai ser mais rápido.
- No outro extremo, se $h(n)$ é muito alto em relação a $g(n)$, então apenas o $h(n)$ tem preponderância e o A* transforma-se num *Greedy Best-First-Search*.

Então temos uma situação interessante em que podemos decidir o que queremos retirar do A*. Num ponto exacto, com a informação perfeita, retiraremos o **caminho mais curto** de uma maneira rápida. Se estivermos muito abaixo do esperado vamos continuar a ter o caminho mais curto mas de uma maneira lenta. Se estivermos muito acima, não teremos o caminho mais curto, mas o A* terá uma boa *performance*.

Falamos agora de três heurísticas para calcular a distância. Notar que existem muitas mais mas que tivemos estas em contas por estarem associadas ao movimento e serem de simples implementação.

- **Manhattan distance**. Normalmente usada em *square grids* que permitem movimento de 4 direcções. É a heurística *standard*.
- **Diagonal Distance**. Usada em *square grids* que permitem movimento em 8 direcções. Inclui os 4 movimentos tradicionais (cima, baixo, esquerda, direita) mais os movimentos diagonais.
- Euclidean distance. Se nos pudermos mover em qualquer direcção, esta é a melhor heurística porque permite calcular uma linha directa de um ponto ao outro.

Para o nosso projecto decidimos usar a **distância Euclideana** visto que no nosso grafo e de um vértice para o outro podemos movimentarmos em mais que uma direção e calcular a distância de um ponto ao outro seria muito mais imediato.

DIAGRAMA DE CASOS DE USO

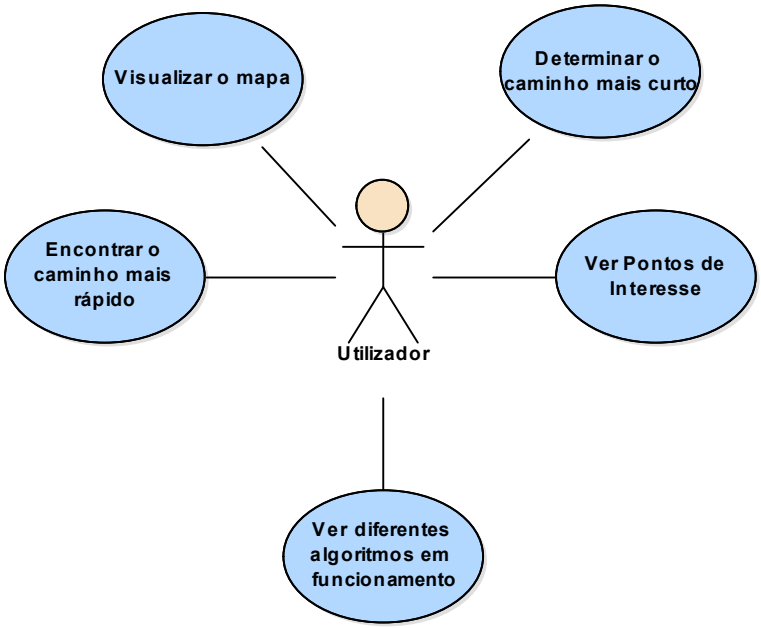

Figura 4: *Diagrama de Casos de Uso*

DIAGRAMA DE CLASSES

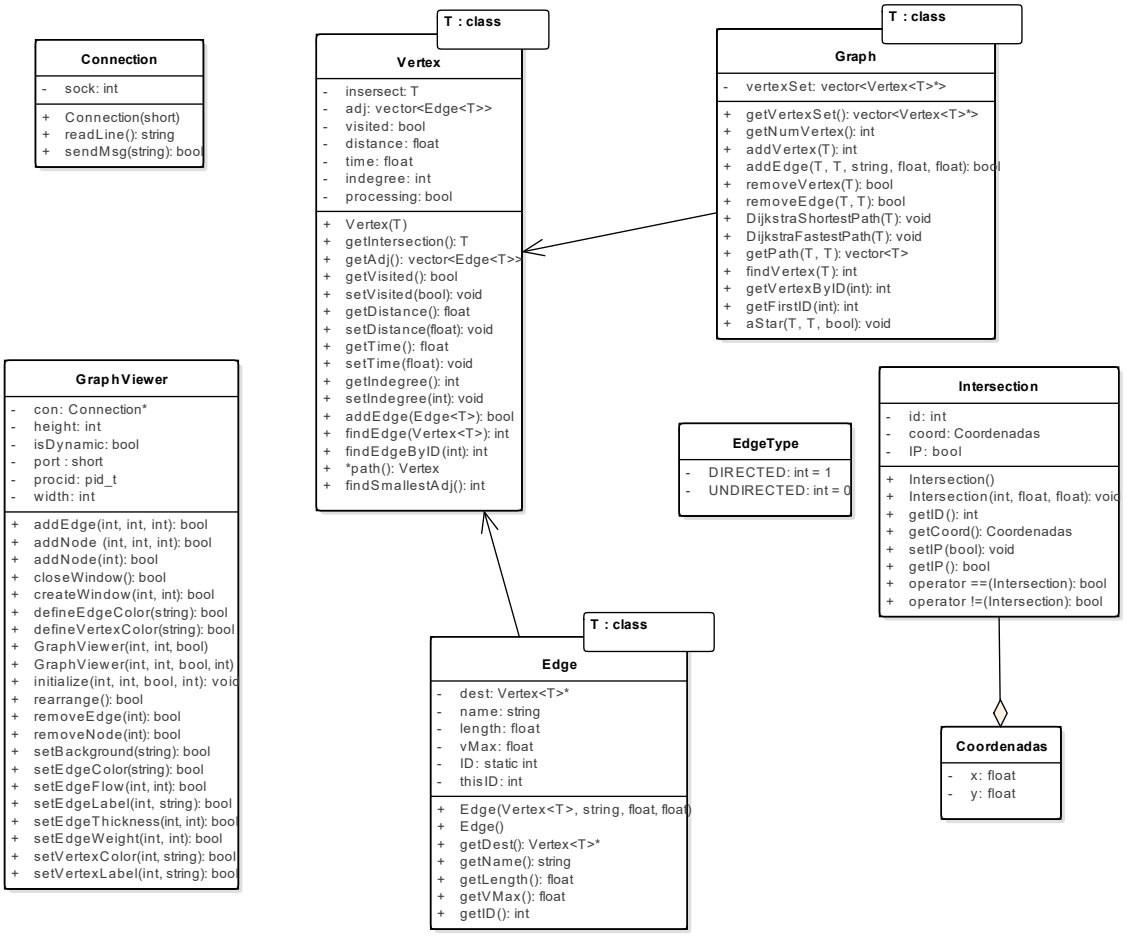

Figura 5: *Diagrama de Classes*

CONCLUSÕES E TRABALHO FUTURO

Terminado o relatório e após toda a investigação feita deixamos algumas conclusões que achamos ser pertinentes e relatam o percurso e actividades do grupo.

Começamos por enaltecer toda a dedicação do grupo na maneira como encarou este desafio e tentou esmiuçar os conceitos. Desde o ínico que o nosso foco foi tentar aprender o mais possível e tirar conclusões que realmente fosse apalpáveis e com carácter pedagógico. Começando pelo parser que foi conseguido 100% por trabalho nosso e que nos permitiu começar o trabalho e tratar a informação da maneira mais conveniente. Nesse processo entramos em contacto com ferramentas nunca usadas bem como conhecimentos que não foram adquiridos até ao mesmo

momento como o *PostGIS*. Numa primeira fase do trabalho, implementamos o algoritmo **Djikstra** mas achamos que tornaria tudo bastante linear e que os nossos conhecimentos não estavam a ser explorados pelo que partimos para uma nova implementação, de uma solução óptima que nos permitisse comparar resultados e perceber em que casos essas soluções se adequam ou são mais eficientes. Temos noção de que o algortimo **A*** foi bastante trabalhoso e que nos atrasou na entrega e, estando alerta das consequências que atraso pode acarretar, decidimos fazer um último esforço para concretizar aquilo que sempre tivemos planeado.

Tivemos dificuldades na construção da interface. Encontrar uma função que convertesse coordenadas geográficas em pixéis foi algo que nos tomou algum tempo e que não conseguimos encontrar uma solução perfeita. Mesmo usando algumas fórmulas para dimensionar o nosso *background* (mapa do Porto) ao sítio onde era colocado o nosso grafo, este nunca ficou no sítio e tivemos de o ajudar através de um processo bruteforce, alterando as coordenadas x e y até estas estarem nos limites do *background*. Tivemos também algumas dificuldades na implementação do algoritmo A* e apesar de estar ser encontrar uma solução que calcula o *shortest path*, está não está 100% optimizada e precisaria de alguns ajustes.

Posto isto, agradecemos a oportunidade que tivemos em fazer um projecto que nos cativou e nos fez ter vontade de saber mais. O grupo dinamizou-se de uma maneira agradável e proactiva sendo que todos os membros se disponibilizaram a mais e melhor e ajuda de cada um foi preciosa.

Para o próximo projecto pretendemos manter a postura e encarar como desafio que pode ser sempre optimizado.

Referências

[1] Stack Overflow, `http://stackoverflow.com/`