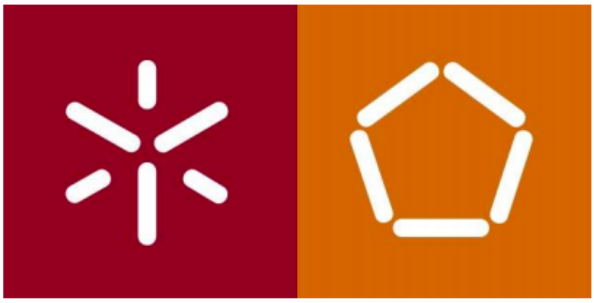

Universidade do Minho

Mestrado Integrado em Engenharia Informática

Sistemas de Representação de Conhecimento e Raciocínio

TP Recurso

Eduardo Conceição (A83870)

14/07/2020
Braga

Resumo:

Neste relatório irei expôr as decisões que tomei na conceção de uma Base de Conhecimento a ser aplicada a um *dataset* relativo a várias cidades de Portugal. Para além disto, explicarei como importei os dados para a BC e como implementei os algoritmos de procura sobre este *dataset*.

Contents

Lista de Figuras

1 Lista de Acrónimos/Siglas

- BC: Base de Conhecimento
- ID: Identificador
- CSV: *Comma Separated Values*

2 Introdução

No primeiro projeto individual para a cadeira de Sistemas de Representação de Conhecimento e Raciocínio, foi-nos proposta a criação de uma BC para a representação, em *Prolog*, de cidades ds vários distritos de Portugal continental, e a aplicação de algoritmos para podermos encontrar caminhos entre cidades.

Este processo passaria por duas fases: a primeira passaria por importar os dados para passarem a ter a semântica do *Prolog* a partir de um documento em Excel, e a segunda, e mais pesada, seria desenvolver um sistema de *queries* de procura para os dados importados, tendo em conta as várias *queries* pedidas no enunciado.

Neste documentos explicarei que decisões tomei para concretizar estas duas partes.

3 Preliminares

Ao contrário do primeiro projeto desenvolvido para esta cadeira, o sistema com que vamos trabalhar não necessita obrigatoriamente de ser compatível com lógica extendida. Com isto, eu quero dizer que não iremos trabalhar com o "desconhecido", uma vez que todos os parâmetros dos vários predicados são sempre conhecidos.

- **Pressuposto dos Nomes Únicos**: duas constantes diferentes referem-se a duas entidades diferentes.
- **Pressuposto do Mundo Fechado**: todas as afirmações são verdadeiras ou falsas, e nós sabemos se são, respetivamente, verdadeiras ou falsas.
- **Pressuposto do Domínio Fechado**: no universo do discurso apenas existem os objetos que nos são conhecidos.

Apesar da implementação do Conhecimento Desconhecido não ser impossível neste contexto, acredito que a implementação de tal não seria necessária. No universo de discurso, nós sabemos todos os elementos que caracterizam uma cidade. Como tal, não existe informação incompleta a retratar neste sistema, o que torna a implementação de tal desnecessária.

4 Predicados de Representação dos Dados

Para representar os dados, existem 3 predicados diferentes:

- **cidade**: Representa uma cidade, com todas as informações necessárias para a caracterizar;
- **distrito**: Constitui um identificador numérico de distrito, e uma lista de identificadores de cidades que fazem parte desse distrito.
- **aresta**: Representa uma aresta, ou seja, uma ligação entre duas cidades, com o correspondente valor heurístico que, para todos os casos, é a distância entre as duas cidades.

Tendo isto em conta, cada um destes predicados traduzem-se em Prolog com os seguintes componentes:

- **cidade/8**: $\#Id, Nome, Latitude, Longitude, \#IdDistrito,$ $IsCapital, IsPatrimonio, IsNomeavel \rightarrow \{V, F\}$
- **distrito/2**: $\#IdDistrito, ListaCidades \rightarrow \{V, F\}$
- **aresta/3**: $Inicio, Fim, Distancia \rightarrow \{V, F\}$

Algo que é de notar é que, apesar de definirmos e inserirmos na BC o predicado *distrito/2*, este não revelou ter utilidade para a algoritmia. No entanto, como não afetavam negativamente a *performance*, não achei haver necessidade de alterar o código de forma a remover o predicado.

Na próxima secção explicarei como converti as informações do Excel nos predicados descritos.

5 Importação do *Dataset*

5.1 Pré-preparação

Em oposição ao *dataset* fornecido para o trabalho individual, o *dataset* para este trabalho de Recurso não apresentava nenhumas inconsistências. A única mudança que foi necessária para podermos importar mais facilmente foi apenas mudar os caracteres acentuados para a sua versão não acentuada, ou seja, passar 'á' para 'a', 'ç' para 'c', etc.

De resto, não haviam mais mudanças a fazer no Excel fornecido. Como tal, para poder processar os dados, apenas converti o Excel para um ficheiro CSV, e processei-o com um *parser* feito em Java, que descreverei a seguir.

5.2 *Parsing* para Prolog

Para poder importar os dados a partir do CSV, eu criei um *parser* em Java. Este parser consiste em ler o ficheiro de texto puro, linha a linha, guardar as informações em estruturas de dados em Java, criar as arestas necessárias para criar os grafos, e, por fim, escrever no ficheiro *output*, "cidades.pl".

Os dois predicados importantes que o *parser* constrói são o *cidade/8* e o *aresta/3*. Para os poder construir, o programa **ReadCidades** lê o ficheiro de *input* com a classe *BufferedReader* e escrevo em ficheiro com *FileWriter*. A informação sobre as cidades é guardada num *HashMap* que guarda as informações de todas as cidades. Por fim, uma vez que acabamos de ler o ficheiro, e temos todas as cidades em memória, criamos as várias aretas que constituirão o grafo, guardando-as também em memória do programa e depois, no final, escrevo-as em ficheiro e termino a operação.

5.2.1 Criação das Arestas

As arestas são a parte mais importante do programa de Prolog, uma vez que são o único constituinte do grafo.

Uma vez que não foi definida uma estratégia obrigatória para a definição das arestas, eu decidi seguir um grupo de estratégias, duas para as capitais distritais e duas para as restantes cidades.

Um aspeto que é importante definir antes de prosseguir, é que, para poder implementar o algoritmo de criação de arestas que eu criei, é necessário termos em conta, para um distrito A, os distritos que fazem fronteira, pois usamos esta ideia para criar uma lista de adjacências.

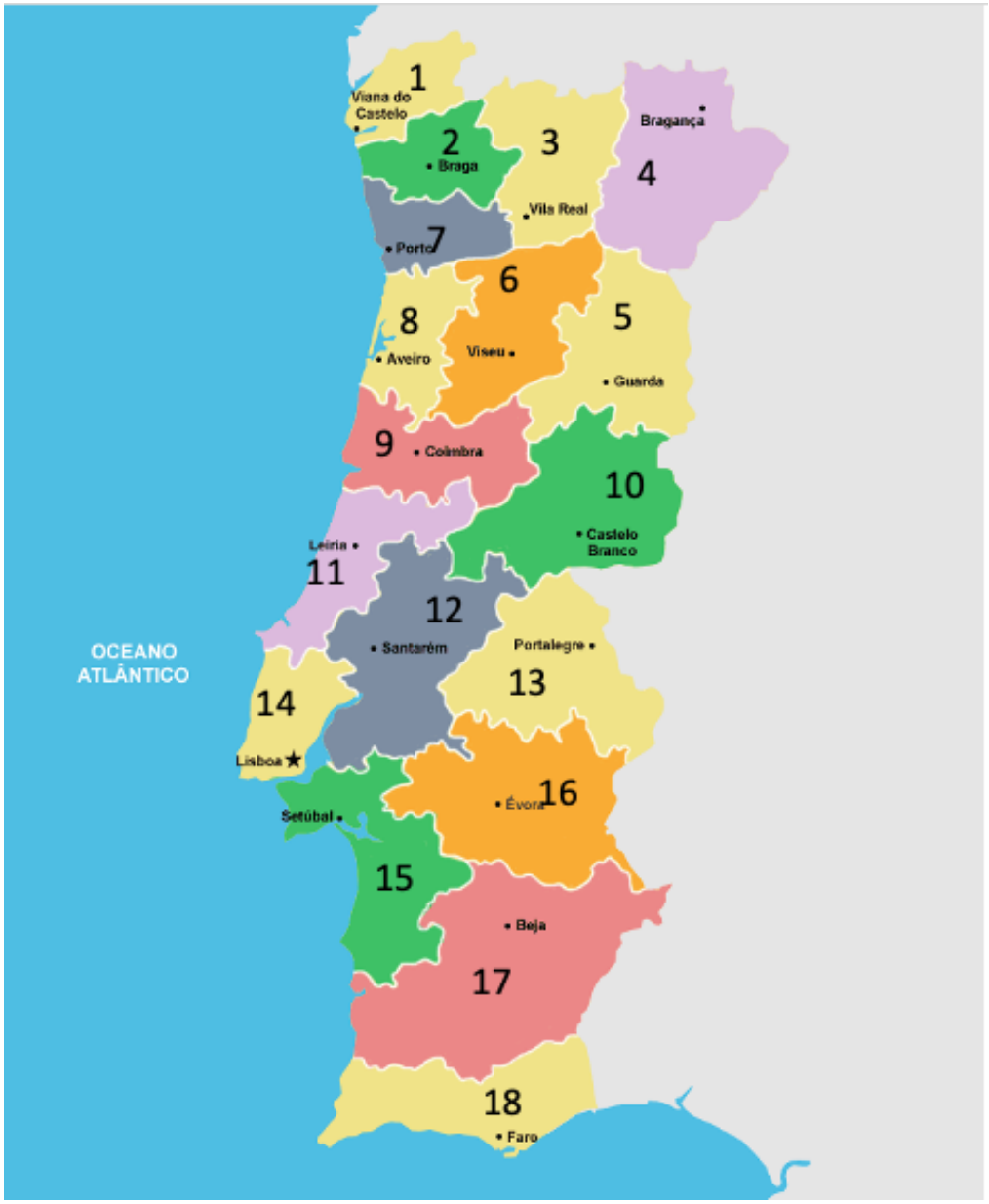

Figure 1: Distritos de Portugal associados ao seu ID

Olhando para a Figura 1, podemos ver que, por exemplo, o distrito de Braga faz fronteira com os distritos 1 (Viana do Castelo), 3 (Vila Real) e 7 (Porto).

```
private static final Integer[][] adj_dists = {
        {2},                    //Viana do Castelo
        {1,3,7},                //Braga
        {2,7,6,4},              //Vila Real
        {3,5,6},                //Bragança
        {4,6,9,10},             //Guarda
        {3,5,9,8,7},            //Viseu
        {2,3,6,8},              //Porto
        {6,7,9},                //Aveiro
        {5,6,8,10,11},          //Coimbra
        {5,9,11,12,13},         //Castelo Branco
        {9,10,12,14},           //Leiria
        {10,11,13,14,15,16},    //Santarém
        {10,12,16},             //Portalegre
        {11,12,15},             //Lisboa
        {12,14,16,17},          //Setúbal
        {12,13,15,17},          //Évora
        {15,16,18},             //Beja
        {17}                    //Faro
};
```

Figure 2: Lista das Adjacências Distritais

Tendo isto em conta, para as capitais distritais, temos as seguintes estratégias para fazer arestas:

- Cada capital distrital está ligada às capitais dos distritos que fazem fronteira com o distrito do qual é capital. Ou seja, por exemplo, Viana do Castelo está ligada a Braga;
- Cada capital distrital está ligada a todas as cidades do próprio distrito. Ou seja, por exemplo, Viana do Castelo está ligada a Paredes de Coura, Ponte de Lima, Melgaço, etc.

No entanto, para as cidades *minor*, defini as arestas do seguinte modo:

- Cada cidade está ligada à capital do seu distrito. Ou seja, Esposende, por exemplo, está ligada a Braga;
- Cada cidade *minor* está também ligada a um número aleatório de cidades do próprio distrito e de distritos adjacentes, sendo que este número aleatório varia entre 1 e 3. Ou seja, por exemplo, Ponte de Lima poderá estar ligada a 3 cidades de Viana do Castelo e 2 de Braga, sendo que estes números variam consoante a execução do programa, porque são aleatórios. No entanto, isto garante que existem pelo menos algumas ligações entre distritos que não passam pela capital distrital, aspeto que é necessário assegurar para poder fazer a pergunta 8.

A implementação destas estratégias é algo complexa, especialmente para a última. No entanto este relatório está mais dedicado à explicação dos outros aspetos do trabalho. Como tal, o *parser* em Java encontra-se anexado com o resto do código, e está devidamente documentado, portanto, para mais informações sobre a implementação destas ideias, sugiro uma leitura do ficheiro "ReadCidades.java".

A distância entre cidades é calculada com base na latitude e longitude do ponto, utilizando o seguinte método:

```
private static double distance(float x1, float y1, float x2, float y2){
    double R = 6371e3;
    double psi1 = x1 * Math.PI/180;
    double psi2 = x2 * Math.PI/180;
    double delta_psi = (x2 - x1) * Math.PI/180;
    double delta_lambda = (y2 - y1) * Math.PI/180;

    double a = Math.pow(Math.sin(delta_psi/2), 2) * Math.cos(psi1) * Math.cos(psi2) * Math.pow(Math.sin(delta_lambda/2), 2);
    double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));

    return c * R;
}
```

Na secção "Referências" podemos encontrar um *link* para um *site* onde eu retirei esta informação.

6 Resolução das Perguntas

Uma vez que a informação já se encontra escrita de uma forma que o Prolog consegue interpretar mais facilmente, passarei agora a explicar a solução para resolver cada uma das *queries* que nos pedem no enunciado fornecido, tentando o máximo de vezes possível aplicar mais do que um algoritmo, de modo a poder comparar a performance de cada um.

6.1 Pergunta 1

Calcular um trajeto possível entre duas cidades

Para resolver esta pergunta, implementei dois algoritmos: um **Depth-First** e um **Breadth-First**, mas, antes de falar deles, irei introduzir uma ideia que encontramos em todas as respostas que eu implementei. Uma vez que as arestas são definidas utilizando dois IDs, os algoritmos, para que não haja uma repetição ridícula do mesmo código, trabalham todos com ditos IDs. No entanto, de modo a tornar cada pergunta mais *user friendly*, passamos como *input* o nome das cidades de onde partimos e para onde queremos ir. Para que possa ser feita a conversão, existe o predicado auxiliar *cities_to_ids/4*, que converte os nomes das cidades nos respetivos IDs. No entanto, uma vez que a lista que cada algoritmo produz que corresponde a caminho encontrado também é constituida por IDs, para se tornar mais fácil a apresentação do resultado ao utilizador, existe o predicado inverso *ids_to_cities/2*, que recebe uma lista de IDs e converte-os nos nomes das cidades.

Tendo isto em conta, os algoritmos que implementei são baseados naqueles que fizemos nas aulas, exceto o *Breadth-First*, que fiz de uma forma ligeiramente diferente, mas com a mesma ideia em geral. Estes não sofrem nenhuma alteração interessante neste exercício, são apenas a versão mais elementar de cada um. De notar que, como se vai repetir nos outros exercícios, o *Breadth-First* tem bastantes problemas quando o caminho requer passar por mais do que X arestas, sendo que este X varia consoante o número de arestas que o grafo tem, o que faz sentido tendo em conta o funcionamento do *Breadth-First*, visto que coloca, a um certo ponto, uma quantidade de informação demasiado grande na orla, o que torna o algoritmo ineficiente, pelo menos da forma como o implementei.

6.2 Pergunta 2

Selecionar apenas cidades, com uma determinada caraterística, para um determinado trajeto

Para a resolver este exercício, utilizei o algoritmo *Depth-First* e o *Breadth-First*, que não apresentam grandes alterações entre o primeiro exercício e este, apenas temos de ter em conta as **etiquetas** quando procuramos novos candidatos para adicionar à orla.

No que toca a etiquetas extra que atribuí a cada cidade, devido a restrições em termos de tempo, apenas apliquei duas: se uma cidade é **Património da UNESCO** e se ela é **Candidata a Património Mundial**. Para ambos os casos o que isto vai-se traduzir num inteiro binário, em que, caso uma cidade tenha uma destas características, será 1, e 0 caso não a tenha.

Assim sendo, na implementação dos algoritmos, o que vai acontecer de diferente é que, quando vamos adicionar à orla, vamos primeiro veririficar se, para o candidato que estamos a ver, existe um predicado *cidade/8* tal que a etiqueta em questão está a 1. Nesse caso, vamos adicioná-lo à orla. Caso contrário, ignoramo-lo.

Para invocar as travessias, podemos chamar o predicado *resolve_q2_df/4* ou *resolve_q2_bf/4*, que resolvem o problema com o *Depth-First* e com o *Breadth-First*, respetivamente, sendo que o I que recebe é 1 se quiseremos utilizar a etiqueta relacionada com ser Património, e 2 se quisermos usar a de ser Candidato.

É de notar que não há garantia de existir um caminho entre duas cidades que passe só por cidades com qualquer uma das etiquetas, uma vez que existem poucas em todo o país, cerca de 15, comparado com as quase 300 cidades que estão registadas na BC.

6.3 Pergunta 3

Excluir uma ou mais caracteristicas de cidades para um percurso

Para resolver esta pergunta, apliquei apenas o *Depth-First*, uma vez que a lógica era a mesma que na pergunta anterior, não me parecia interessante estar a repetir mais vezes do que o necessário o código da *Breadh-First*.

Assim sendo, aplico a mesma lógica que no exercício anterior, mas de modo a evitar cidades com a etiqueta dada, ou seja, procuro adicionar nodos apenas quando estes têm a etiqueta em questão a 0.

Para invocar a travessia para esta pergunta, temos o predicado *resolve_q3_df/4*. Do mesmo modo que na pergunta 2, se I for 1 cidades que são património serão evitadas, se for 2 serão excluídas cidades nomeáveis a património e, se for 3, evitaremos cidades que sejam património e nomeadas (pois uma cidade pode ser património por algo e ser de novo nomeada por outro aspeto).

6.4 Pergunta 4

Identificar num determinado percurso qual a cidade com o maior número de ligações

Esta pergunta é algo diferente das anteriores ou das próximas. O cerne desta questão não está na implementação de algoritmos de procura, pois a resposta a esta pergunta pode ser aplicada a qualquer algoritmo de procura. Por isso, para demonstrar que funciona, decidi reutilizar a implementação do *depth-first* da primeira pergunta.

O que o predicado *mais_arestas/2* vai receber é uma lista de IDs, que poderão, ou não, vir da execução de um algoritmo de procura. A lista vai ser transformada numa lista de pares, em que cada par tem o ID de uma cidade e o número de arestas que essa

cidade tem. De seguida, calcula-se o maior elemento dessa lista em termos de número de arestas, e converte-se o ID para o nome da cidade, devolvendo o número de arestas e o nome da cidade num par, que é guardado no segundo argumento do *mais_arestas/*2.

6.5 Pergunta 5

Escolher o menor percurso (usando o critério do menor número de cidades percorridas)

Para esta pergunta apenas implementei o *Breadth-First*. Isto deve-se ao facto que a resposta à mesma é imediata uma vez que o faça. O *Breadth-First* tem a característica de, devido à forma como o algoritmo trabalha, nos dar a solução mais curta em termos de arestas percorridas na primeira execução do algoritmo. Como tal, a primeira solução que o algoritmo nos dá é imediatamente a melhor soluç ão em termos de cidades percorridas.

6.6 Pergunta 6

Escolher o percurso mais rápido (usando o critério da distância)

Para realizar a resposta a esta pergunta, implementei o algoritmo **A*** e o algoritmo *Greedy*.

Na implementação dos dois algoritmos há que ter em consideração que existe, inevitavelmente, o cálculo de uma heurística. O cálculo desta heurística que eu implementei, e que não é, admitidamente, ótimo, baseia-se no cálculo da distância em linha reta entre o nodo candidato que estamos a avaliar e o nodo final. Se houver uma aresta que leve do nodo ao final, então a heurística é 0, o que ajuda o algoritmo a escolher mais rapidamente um caminho direto, uma vez que o caminho mais rápido entre dois pontos é uma linha reta, que é o que uma aresta representa. A distância entre cada ponto e calculada da mesma forma que foi descrita para a criação das arestas na sua criação no programa em Java, mas, como é óbvio, adaptado a Prolog.

Uma vez que a heurística não é necessariamente a melhor, os resultados podem não corresponder ao melhor caminho possível.

Para implementar o *Greedy* utilizei uma implementação em quase tudo igual ao A*. A diferença entre os dois vai ser na deteção do próximo nodo. No *Greedy* vamos apenas escolher com base na heurística e não com base na soma da heurística com o peso da aresta. Isto vai-nos dar qual a escolha **localmente ótima**, em vez de ser a ótima em geral.

Como podemos ver a partir da implementação que eu expliquei, as soluções vão ser apenas aproximações da melhor solução, e não as soluções ótimas.

6.7 Pergunta 7

Escolher um percurso que passe apenas por cidades "minor"

Para realizar este desafio, implementei os algoritmos *Depth-First* e *Breadth-First* uma vez mais. Neste caso, a implementação é muitíssimo semelhante à do exercício 2 e 3, mas desta vez vamos adicionar à orla apenas cidades que não sejam capitais distritais, ou seja, que no sexto parâmetro do predicado tenham um 3. Assim, garantiremos que o caminho nunca vai ter capitais distritais.

6.8 Pergunta 8

Escolher uma ou mais cidades intermédias por onde o percurso deverá obrigatoriamente passar

Para a pergunta 8, utilizei uma vez mais o *Depth-First* e o *Breadth-First*. As implementações são recicladas do primeiro exercício. O que aqui vai mudar é que, para a solução ser admissível, vai ter de incluir as cidades que são passadas como parâmetro numa lista. Para isto, a solução que for produzida pelos algortimos vai ser passada ao predicado auxiliar *lst_contains/2*, que verifica se a primeira lista está contida na segunda lista.

No entanto, o *Depth-First* raramente executa mais do que uma solução em tempo útil, pelo que é difícil vermos uma solução dada pelo mesmo. No entanto, como a implementação é reciclada do primeiro exercício, eu tenho a certeza que está certa.

7 Resultados

Para que exista um propósito para a implementação de vários algoritmos para cada uma das perguntas em que realmente existe mais do que uma implementação, há que comparar a *performance* de cada um. Nesta secção irei explorar de forma breve as vantagens e desvantagens que cada um trouxe.

Começando pelo algoritmo **Depth-First**, o primeiro que implementei e o mais rudimentar, o tempo de execução normalmente é quase imediato para o primeiro resultado. No entanto, quando experimentava testar mais vezes o algoritmo, acabava por ter problemas e o tempo de execução era demasiado elevado, sendo que era mais de vinte minutos, e nunca cheguei a obter uma resposta. O caminho que ele constrói é apenas o primeiro que encontra, o que significa que não tem em conta distância ou tamanho do caminho (na implementação que eu fiz). Em suma, é um algoritmo algo rápido mas rudimentar.

Passando agora ao algoritmo **Breadth-First**, este apresenta normalmente um caminho mais curto, mas tem problemas na implementação que eu fiz. Isto deve-se provavelmente ao facto que o número de arestas que eu crio é bastante grande, rondando as cinco mil e trezentas (sendo que o número não é fixo devido ao componente aleatório que expliquei anteriormente), e, como tal, quando queremos fazer um percurso que precise de mais de quatro nodos, o algoritmo tende a falhar e demorar tempos inaceitáveis. É, portanto, mais inteligente, mas mais lento.

O algortimo **A*** demonstra ser o melhor dos três. Não só apresenta os melhores tempos, em geral, como também está parametrizado para apenas demonstrar o melhor caminho em termos de distância. No entanto, apesar destas vantagens, a heurística que eu implementei para a estimativa utilizada em cada iteração não é necessariamente ideal, pelo que o caminho mais curto que ele encontra pode não ser necessariamente o melhor dos melhores, uma vez que ele ignora em vários casos o caminho ótimo de A para B que pode ser uma linha reta.

Por fim, o **Greedy** é em tudo igual ao A*, exceto na escolha do próximo nodo. Aqui, apenas teremos em consideração o valo heurístico de cada nodo, e não o custo da aresta ou o custo guardado até agora. Assim, as soluções não são ótimas, mas sim, como é óbvio, heurísticas. Portanto, em termos de *performance* de tempo e de espaço é muito semelhante ao A*, mas a solução é normalmente menos próxima da ideal.

Podemos encontrar a complexidade de cada algoritmo resumida na seguinte tabela:

Algoritmo	**Completo**	**Complexidade Temporal**	**Complexidade Espacial**
Depth-First	Não	O(b^d)	O(b^d)
Breadth-First	Sim	O(b^m)	O($b * m$)
A*	Sim	O(b^d)	O(b^d)
Greedy	Sim	O(b^d)	O(b^d)

8 Conclusões e Sugestões

Em geral, acredito que o meu trabalho foi um sucesso. Consegui implementar todas funcionalidades pedidas, na maioria dos casos com mais do que um algoritmo de procura. Mais ainda, consegui arranjar uma estratégia bastante adequada para traduzir os dados em algo que o Prolog consiga interpretar, e é uma implementação da qual estou bastante orgulhoso.

O único problema que o meu trabalho apresenta é o facto que os resultados de algumas *queries* não podem ser visualizados, uma vez que o tempo de execução dos algoritmos quando temos em conta o número de arestas que criei não é ideal. No entanto, eu acho que isso é algo menor quando posso fazer demonstrações em que vemos os algoritmos a funcionar.

Em suma, apesar de apresentar algumas falhas, nomeadamente em termos de tempos de execução, acredito que fiz um bom trabalho que corresponde ao pedido e esperado.

9 Referências

- https://sicstus.sics.se/documentation.html
- https://www.swi-prolog.org/pldoc/doc_for?object=section(%27packages/pldoc.html%27)
- Carlsson, Mats, et al., “SICStus Prolog User’s Manual - Release 4.5.1", Abril, 2019
- https://www.movable-type.co.uk/scripts/latlong.html
- https://en.wikipedia.org/wiki/Greedy_algorithm