

Universidade do Minho

SISTEMAS DE REPRESENTAÇÃO DE CONHECIMENTO E RACIOCÍNIO

MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA

# PROGRAMAÇÃO EM LÓGICA ESTENDIDA

(MÉTODOS DE RESOLUÇÃO DE PROBLEMAS E DE PROCURA)

**Autor**

A85227 João Azevedo (PL2)

(*E-mail*: a85227@alunos.uminho.pt)

Braga  
4 de Junho, 2020

## Resumo

O presente documento relata todo o processo de desenvolvimento do trabalho prático individual da Unidade Curricular de Sistemas de Representação de Conhecimento e Raciocínio que incidiu na aplicação de métodos de resolução de problemas de procura utilizando programação em lógica estendida, recorrendo ao *Prolog* como linguagem de programação.

Ao longo deste relatório será apresentado todo o processo de normalização e consulta de dados a partir um programa capaz de gerir um sistema de transportes do concelho de Oeiras com vista a criar recomendações sobre percursos introduzindo restrições.

Neste trabalho também foi encorajada a extensão do conhecimento extraído de todos os *datasets* fornecidos, de modo que serão indicadas todas as decisões alternativas adotadas, tanto na representação dos dados sob a forma de regras em *Prolog* como outros tipos de consultas implementados neste sistema.

# Índice

<b>1</b>	<b>Introdução</b>	<b>4</b>
<b>2</b>	<b>Preliminares</b>	<b>5</b>
2.1	Entidades do sistema . . . . .	5
2.2	Visualização, <i>Parsing</i> e normalização dos dados . . . . .	6
2.2.1	Estrutura do ficheiro GeoJSON . . . . .	6
2.2.2	Grafo do problema . . . . .	7
2.2.3	<i>Parsing</i> e normalização . . . . .	7
2.2.4	Execução e configuração do <i>parser</i> . . . . .	8
<b>3</b>	<b>Descrição do trabalho e análise de resultados</b>	<b>9</b>
3.1	Algoritmos de procura . . . . .	9
3.1.1	Pesquisa em profundidade . . . . .	9
3.1.2	Pesquisa em largura . . . . .	10
3.1.3	Pesquisa A* . . . . .	11
3.1.4	Pesquisa <i>Greedy</i> (gulosa) . . . . .	12
3.2	Consultas à base de conhecimento . . . . .	13
3.2.1	Calcular um trajeto entre dois pontos . . . . .	13
3.2.2	Selecionar apenas algumas das operadoras de transporte para um determinado percurso . . . . .	14
3.2.3	Excluir um ou mais operadores de transporte para o percurso . . . . .	14
3.2.4	Identificar quais as paragens com o maior número de carreiras num determinado percurso . . . . .	15
3.2.5	Escolher o menor percurso (usando critério menor número de paragens) . . . . .	16
3.2.6	Escolher o percurso mais rápido (usando critério da distância) . . . . .	16
3.2.7	Escolher o percurso que passe apenas por abrigos com publicidade . . . . .	17
3.2.8	Escolher o percurso que passe apenas por paragens abrigadas . . . . .	18
3.2.9	Escolher um ou mais pontos intermédios por onde o percurso deverá passar . . . . .	18
3.2.10	Outras consultas opcionais . . . . .	19
3.3	Comparação geral entre os tipos de pesquisa . . . . .	20
3.4	Estrutura do trabalho, código e documentos . . . . .	21
<b>4</b>	<b>Conclusões e Sugestões</b>	<b>22</b>

# Lista de Figuras

2.1	Representação do grafo do problema. . . . .	7
2.2	Excerto do ficheiro de dados da lista de adjacências. . . . .	8
3.1	Representação do algoritmo de pesquisa em profundidade. . . . .	9
3.2	Representação do algoritmo de pesquisa em largura. . . . .	10
3.3	Exemplo DFS para a consulta 1. . . . .	13
3.4	Exemplo A* para a consulta 1. . . . .	13
3.5	Exemplo DFS para a consulta 2. . . . .	14
3.6	Exemplo DFS para a consulta 3. . . . .	15
3.7	Exemplo DFS para a consulta 4 (de 183 a 595). . . . .	15
3.8	Exemplo menor percurso para a consulta 5. . . . .	16
3.9	Exemplo percurso mais curto para a consulta 6. . . . .	17
3.10	Exemplos DFS para a consulta 7. . . . .	17
3.11	Exemplos de procura para a consulta 8. . . . .	18
3.12	Exemplos de procura para a consulta 9. . . . .	19
3.13	Exemplos de procura para a consulta 10 (extra). . . . .	19
3.14	Exemplos de procura para a consulta 11 (extra). . . . .	19

# Capítulo 1

## Introdução

O estudo realizado teve por base a utilização de dados do sistema de transportes do concelho de Oeiras a partir de um *dataset* inicial com informações de paragens de autocarro, tais como a sua localização, as carreiras que utilizam e respetivas operadoras, entre outras propriedades que serão referidas mais à frente.

O sistema desenvolvido trata, assim, o problema em duas fases, a primeira onde se importam os dados das paragens para uma base de conhecimento, de uma forma adequada, em *Prolog*, e outra fase onde se implementam algoritmos com métodos de procura informada e não-informada para posterior consulta de caminhos possivelmente com restrições associadas, como por exemplo o custo do percurso (distância percorrida), o seu comprimento, entre outras.

A elaboração do caso prático teve então **os seguintes objetivos**:

- Calcular um trajeto entre dois pontos;
- Selecionar apenas algumas das operadoras de transporte para um determinado percurso;
- Excluir um ou mais operadores de transporte para o percurso;
- Identificar quais as paragens com o maior número de carreiras num determinado percurso.
- Escolher o menor percurso (usando critério menor número de paragens);
- Escolher o percurso mais rápido (usando critério da distância);
- Escolher o percurso que passe apenas por abrigos com publicidade;
- Escolher o percurso que passe apenas por paragens abrigadas;
- Escolher um ou mais pontos intermédios por onde o percurso deverá passar.

O sistema de transporte funciona entre as 6h00 até as 24h00 de cada dia, no entanto, nenhuma informação relativa aos horários se encontra no conjunto de dados fornecido, pelo que a variável tempo será interpretada de uma forma diferente e descrita nos próximos capítulos.

Para a realização deste caso prático foi importante ter realizado um trabalho anterior que, apesar de não incidir, desta vez, em tipos de conhecimento imperfeito, tornou mais flexível o entendimento do paradigma que o *Prolog* nos dá em termos da lógica envolvida.

## Capítulo 2

# Preliminares

O caso de estudo incide, essencialmente, no cálculo de trajetos entre locais de um sistema ligado sob a forma de um grafo, sendo cada nodo representado por uma paragem.

O processo de leitura e *parsing* dos ficheiros de dados fornecidos será descrito ainda neste capítulo, no entanto, será melhor abstrair o seu conteúdo para já indo diretamente para as considerações que foram adotadas ao nível da representação das entidades (predicados) em *Prolog*.

### 2.1 Entidades do sistema

Uma **paragem** contém a sua identificação geográfica (GeoID), as coordenadas latitude e longitude, a operadora responsável por servir os transportes daquela localização e as carreiras, i.e., a identificação dos percursos que passam naquela paragem. Outras informações como o estado de conservação da paragem, a rua, freguesia, etc., serão usados para fins informativos na representação dos caminhos calculados, assim, para um melhor entendimento desta entidade, segue-se a representação adotada em *Prolog*:

```
:- dynamic paragem/10.
paragem(GID, Latitude, Longitude, Estado, Tipo_Abrigo, Tem_Publicidade?,
        Operadora, Cod_Rua, Nome_Rua, Freguesia).
```

Este predicado representa quase na totalidade as entradas do *dataset* fornecido excluindo a carreira desta regra pois como a mesma varia, ou seja, existem para uma mesma paragem várias carreiras que passam pela mesma, decidi associar aos arcos do futuro grafo do sistema a informação da carreira. Esta estratégia será vantajosa no sentido em que nos permite encontrar percursos iguais<sup>1</sup> que passam por carreiras diferentes trabalhando estas em diferentes horários do dia.

Tendo os nodos bem concretizados em paragens precisamos de criar um predicado que associasse os diferentes IDs dos nodos, por uma determinada ordem, a um possível caminho entre duas paragens realizado por uma carreira. O predicado em questão estabelece, assim, uma **ligacao** entre um nodo<sup>2</sup> A e um nodo B através de uma carreira:

```
:- dynamic ligacao/3.
ligacao(GID_A, GID_B, Carreira).
```

---

<sup>1</sup>Percursos que passam nos mesmos nodos.

<sup>2</sup>Neste trabalho, o nodo representa o GeoID da paragem.

Esta estratégia leva a que tenhamos duas ligações entre os mesmos nodos A e B mas servidas por carreiras diferentes. Um exemplo disso, extraído do ficheiro de dados, apresenta seis carreiras diferentes (1, 7, 10, 12, 13 e 15) para ligações entre dois nodos iguais:

```
ligacao(183, 791, 1).  
ligacao(183, 791, 7).  
(etc...)
```

Assim, na altura de efetuar as consultas de trajetos o *Prolog* utilizará esse valor de carreira para identificar os diferentes caminhos.

A próxima secção será dedicada ao método de *parsing* e breve descrição do programa responsável por essa tarefa assim como apresentação de *inputs* e *outputs* esperados do mesmo.

## 2.2 Visualização, *Parsing* e normalização dos dados

Inicialmente, estava prevista a utilização de um ficheiro de dados do tipo **geojson** em que são descritos vários objetos JSON com diferentes elementos, podendo ser coordenadas, posições, geometrias e coleções de dados com etiquetas e todas as informações relativas a uma dada *feature*.

### 2.2.1 Estrutura do ficheiro GeoJSON

No nosso caso, a estrutura base do ficheiro seguia o seguinte aspeto:

```
{  
  "type": "FeatureCollection",  
  "features": [  
    {  
      "type": "Feature",  
      "properties": {  
        "gid": 298,  
        "Estado de Conservação": "Bom",  
        "Tipo de Abrigo": "Aberto dos Lados",  
        ...  
      },  
      "geometry": {  
        "type": "Point",  
        "coordinates": [ -99888.8, -105966.88 ]  
      }  
    },  
    // {...} outros nodos  
  ]  
}
```

Aqui podemos ver a representação de uma coleção onde apenas foi explicitado um nodo, estando os outros omitidos para este simples exemplo. Cada nodo é representado assim como uma *feature* de uma coleção com propriedades, as mesmas já apresentadas para uma paragem.

A utilização desta estrutura em JSON permitiu utilizar ferramentas *online* de visualização de **geojson** como o **geojson.io**, no entanto apenas consegui visualizar os pontos soltos no mapa e não um grafo conciso.

### 2.2.2 Grafo do problema

Portanto, de modo a demonstrar a dimensão do problema e o porquê de os testes neste sistema serem muito complicados de implementar decidi utilizar uma outra ferramenta recente para problemas de computação, um editor de grafos online, disponível em [csacademy.com/app/graph\\_editor/](https://csacademy.com/app/graph_editor/).

Este *website* permitiu, assim, obter o seguinte grafo, onde se torna impercetível a distinção de nodos e ligações, entre os mesmos:

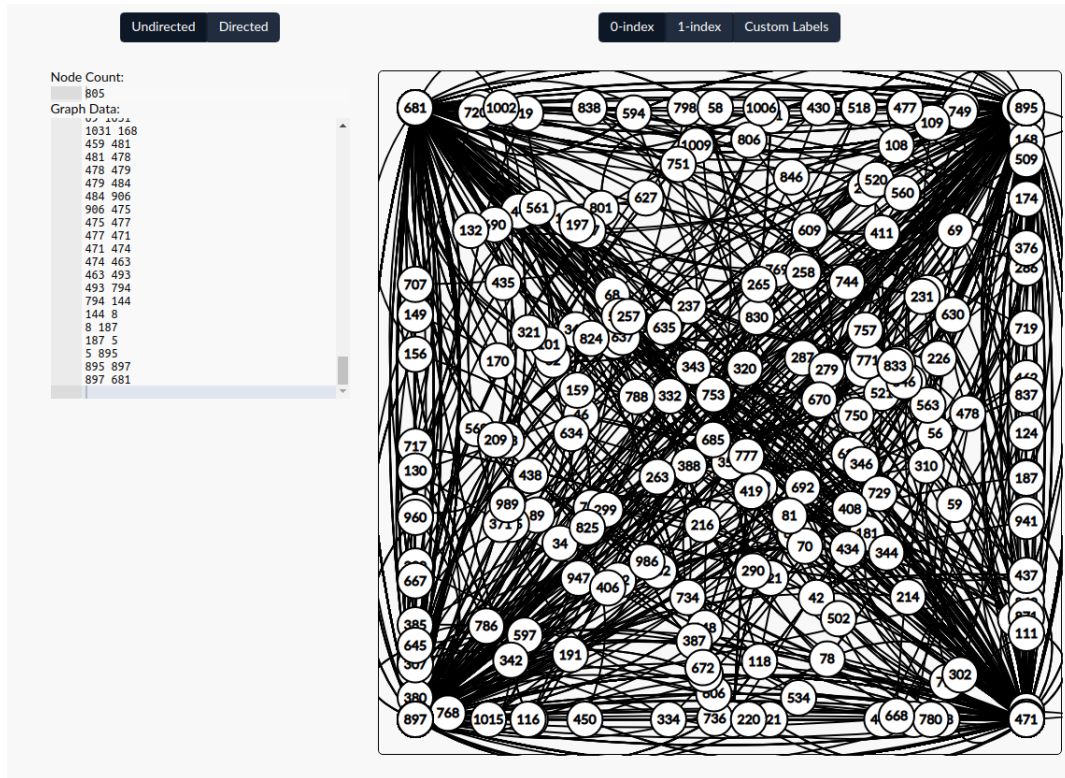


Figura 2.1: Representação do grafo do problema.

A representação do problema não é, portanto, algo que ajude muito no *debugging* da aplicação, mas permite perceber que por vezes o tempo de demora de pesquisa de um percurso é equacionado, não só com a gestão de memória e eficiência das regras, como também as inúmeras alternativas que cada nodo apresenta.

### 2.2.3 Parsing e normalização

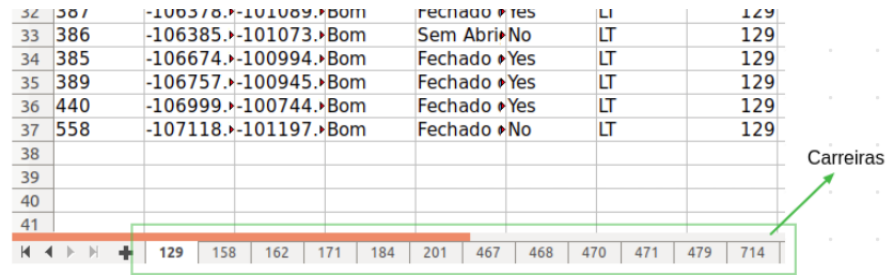
A leitura e representação de dados não é, nem devia ser a parte mais complicada na implementação, no entanto existiram certos aspetos que estiveram por detrás de alguma demora na sua normalização, como arranjar um correto *encoding*, corrigir valores vazios com valores por omissão, entre outros, algo que é esperado quando se lida com elevadas quantidades de informação.

O programa *parser* foi realizado utilizando o *Java* como ferramenta de leitura de ficheiros, armazenamento de estruturas e escrita final da informação.

Ora o primeiro passo foi estabelecer um padrão para os ficheiros de dados, que formato usar, isto é, como estão separadas as colunas, linhas, pelo que foi escolhido o formato **csv**. Passou-se então à utilização dos *datasets* presentes no **xlsx** da *lista de adjacências*, onde cada folha de cálculo era identificada por todas as paragens presentes numa dada carreira, o que



facilitou na altura de criar as ligações/arcs de cada carreira. A estrutura pode ser vista, de forma geral, na seguinte imagem:



32	387	-106378.101089	Bom	fechado	Yes	LI	129
33	386	-106385.101073	Bom	Sem Abri	No	LT	129
34	385	-106674.100994	Bom	Fechado	Yes	LT	129
35	389	-106757.100945	Bom	Fechado	Yes	LT	129
36	440	-106999.100744	Bom	Fechado	Yes	LT	129
37	558	-107118.101197	Bom	Fechado	No	LT	129
38							
39							
40							
41							
<div> <div>129</div> <div>158</div> <div>162</div> <div>171</div> <div>184</div> <div>201</div> <div>467</div> <div>468</div> <div>470</div> <div>471</div> <div>479</div> <div>714</div> </div>							

Figura 2.2: Excerto do ficheiro de dados da lista de adjacências.

Antes de converter os dados para o formato especificado foi necessário efetuar uma pequena normalização nos mesmos, corrigindo um ou outro erro de *encoding* com alguns caracteres e estabelecer valores **undefined** para campos vazios exceto no caso da latitude e longitude, onde foi estabelecido um número padrão para esses campos.

Tendo convertido cada uma das folhas de cálculo para um **csv** obtive um conjunto de ficheiros, guardados na pasta *datasets/raw/encoded*:

```
ficheiro "la_{01,...,776}.csv", total => 39 ficheiros/carreiras
```

Esses 39 ficheiros foram agora entregues ao *parser* em Java que converteu cada uma dessas linhas para um predicado em *Prolog* como o especificado na primeira secção deste capítulo. De realçar que não foram admitidas paragens iguais pelo que utilizaram-se estruturas *key-value* com *hashing* de *strings* para evitar eventuais repetições.

Para as ligações, visto que cada ficheiro já se encontrava ordenado por carreira e por ordem do percurso, foi só estabelecer uma ligação entre a linha atual e a seguinte, passando no campo carreira do arco a carreira que está no nome do ficheiro csv a ser lido.

#### 2.2.4 Execução e configuração do *parser*

O programa encontra-se em **src/parser** a partir da diretoria raiz do nosso projeto e possui uma *makefile* para que não seja necessário um IDE na sua execução/compilação e o ficheiro *make* também possui variáveis para alterar os argumentos passados ao programa, como a diretoria onde estão os dados “crus” e em que ficheiro será guardado o *output* em *Prolog*:

```
IN_DATASETS_FOLDER = ../../datasets/raw/encoded/
OUT_PROLOG_FILE = ../../datasets/processed/paragensLigacoes.pl
```

Nesta ótica pode ser utilizada a *regra make* para compilar o programa em Java e a *regra make run* para correr com as variáveis especificadas acima.

## Capítulo 3

# Descrição do trabalho e análise de resultados

No capítulo anterior pudemos perceber a dimensão do problema e quais as entidades que farão parte do sistema e como foram obtidas após leitura e normalização dos dados, resultando em dois predicados utilizados no *Prolog* **paragem** e **ligacao**.

O passo anteriormente descrito resultou num ficheiro onde constam um conjunto de factos que são a base de conhecimento que será utilizada, estando armazenada em **include/paragensLigacoes.pl**, onde os predicados serão construídos manipulando esses factos e onde chegaremos a conclusões acerca de encontrar caminhos através de pesquisas informadas e não-informadas.

### 3.1 Algoritmos de procura

Conhecer algoritmos de procura de caminho num grafo é um problema muito visto nas ciências da computação e é importante perceber não só como escrevê-los em *Prolog* ou outro paradigma qualquer, como perceber o seu funcionamento e perceber as diferenças e limites de cada um.

#### 3.1.1 Pesquisa em profundidade

Esta estratégia de pesquisa contempla as pesquisas **não-informadas** onde o algoritmo parte de um nodo específico e tenta percorrer o grafo, de cada nodo para o adjacente o mais profundo possível primeiro, antes de iniciar o *backtracking*.

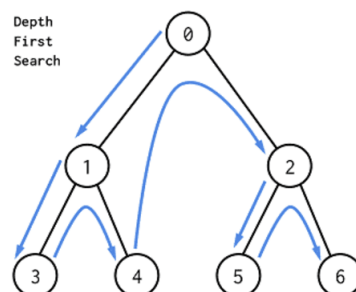


Figura 3.1: Representação do algoritmo de pesquisa em profundidade.

A complexidade é proporcional ao número de vértices somado com o número de arestas e para grafos muito grandes, onde é o caso, existem muitos caminhos onde o algoritmo

simplesmente não termina por problemas de memória, onde uma solução seria limitar a profundidade da pesquisa do problema.

O algoritmo utilizado, traduzido em *prolog*, está apresentado no seguinte excerto de código:

```

resolve_depthFirst(Paragens, Inicial, Final, [(Inicial,'Start')|Caminho]) :-
    depthFirstSearch(Paragens, Inicial, Final, Caminho).

depthFirstSearch(Paragens, Final, Final, []) :- !.
depthFirstSearch(Paragens, Nodo, Final, [(ProxNodo,Carreira)|Caminho]) :-
    adjacenteDF(Paragens, Nodo, ProxNodo, Carreira),
    depthFirstSearch(Paragens, ProxNodo, Final, Caminho).

adjacenteDF(Paragens, Nodo, ProxNodo, Carreira) :-
    ligacao(Nodo, ProxNodo, Carreira),
    (...).

```

Obviamente, este predicado já se encontra adaptado à solução adotada para este problema de paragens, onde o caminho se encontra descrito como sendo uma lista de pares (**Nodo**, **Carreira**)<sup>1</sup>, mas o funcionamento base está no predicado **depthFirstSearch** que recebe um conjunto de paragens permitidas<sup>2</sup> na procura deste trajeto e um nodo final para saber quando o caminho for encontrado.

Em cada iteração recursiva, vai procurar um nodo adjacente ao nodo atual e escolhe-o como o próximo do caminho até chegar ao caso onde o nodo é efetivamente o final, caso tenha sucesso na pesquisa.

O predicado **adjacenteDF**, assim como nos próximos algoritmos apresentados, verifica se existe uma ligação do nodo atual com um dado **ProxNodo**, guardando a referência desse nodo e da carreira presente nessa ligação nos argumentos, verificando também se esse nodo é admitido na lista de paragens disponíveis.

### 3.1.2 Pesquisa em largura

A pesquisa em largura também pertence ao leque de pesquisas sem qualquer tipo de heurística ou informação durante a pesquisa para além dos nodos e os adjacentes. Aqui a estratégia é um pouco diferente, começando na raiz, ou nodo inicial, visitamos todos os vizinhos e para cada nível da árvore visitamos os nodos desse nível até encontrar o nodo final.

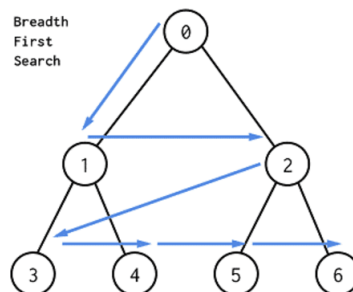


Figura 3.2: Representação do algoritmo de pesquisa em largura.

Aqui apenas será apresentado um excerto do código do algoritmo visto que o caso base e o caso de paragem acabam por ser iguais em cada um dos algoritmos, assim:

<sup>1</sup>O primeiro nodo é um caso a parte, onde o campo carreira é passado com 'Start'.

<sup>2</sup>Mais à frente será explicada a importância de seguir esta metodologia.

```

breathFirstSearch(Paragens, [[(N,C)|Caminho]|CaminhoList], Final, Solucao) :-
    bagof([(M, Carreira), (N,C)|Caminho],
        (adjacenteBF(Paragens, N, (M, Carreira)),
         \+ membro((M, Carreira), [(N,C)|Caminho])),
        NovosCaminhos),
    append(CaminhoList, NovosCaminhos, Res), !,
    breathFirstSearch(Paragens, Res, Final, Solucao);
breathFirstSearch(Paragens, CaminhoList, Final, Solucao).

```

O predicado **bagof** permite para cada nível da árvore ir buscar todos os pares (M, Carreira) onde M corresponde ao nodo adjacente do nodo N considerado no processo recursivo, onde M ainda não é considerado no conjunto de nodos a visitar, e aplicámos a recursividade nesses nodos.

### 3.1.3 Pesquisa A\*

Este é um algoritmo onde adotaremos heurísticas na procura de soluções e trata-se de uma pesquisa que produz casos ótimos na maior parte das vezes mas com um *drawback* da memória utilizada muito elevado pois guarda todos os nodos da pesquisa em memória escolhendo sempre o caminho que produz o melhor “score” na pesquisa, que no caso deste problema será a distância ao nodo final.

Este já é um algoritmo que combina a pesquisa em largura com heurísticas na pesquisa pelo que será mais difícil representar graficamente o mesmo, logo apenas será descrito o algoritmo em si, mais uma vez partindo de um excerto do mesmo:

```

astarSearch(Paragens, Caminhos, Final, SolucaoCaminho) :-
    bestPathAsearch(Caminhos, MelhorCaminho),
    seleciona(MelhorCaminho, Caminhos, OutrosCaminhos),
    expand_astar(Paragens, MelhorCaminho, Final, ExpCaminhos),
    append(OutrosCaminhos, ExpCaminhos, NovoCaminhos),
    astarSearch(Paragens, NovoCaminhos, Final, SolucaoCaminho).
(...)

```

Aqui podemos ver que o processo recursivo segue uma ordem específica, escolher o melhor caminho dos obtidos até agora, expandir esse caminho e adicioná-lo aos novos caminhos e depois chamar recursivamente para os novos caminhos. A escolha do melhor caminho contempla a heurística do custo do percurso, através da soma da estimativa do custo com o custo atual do caminho, isto é:

```

bestPathAsearch([Caminho1/Custo1/Est1, _/Custo2/Est2|Caminhos], MelhorCaminho) :-
    Custo1 + Est1 <= Custo2 + Est2, !,
    bestPathAsearch([Caminho1/Custo1/Est1|Caminhos], MelhorCaminho).
(...)

```

Expandir o nodo significa, neste caso, encontrar todos os nodos adjacentes ao caminho considerado até agora, para que depois se possam aplicar as heurísticas através do predicado **adjacenteAStar**:

```

expand_astar(Paragens, Caminho, Final, ExpCaminhos) :-
    findall(NovoCaminho, adjacenteAStar(Paragens, Caminho, Final, NovoCaminho),
    ExpCaminhos).

```

Por fim, resta-nos aplicar o predicado ajacente a cada um dos nodos do caminho atualizando a estimativa atual do custo do caminho, fazendo isto por 3 passos, buscar o nodo adjacente e a sua carreira, com o predicado **verificaLigacaoAStar**, predicado esse que foi omitido por ser igual à estratégia do *ajacenteDF/BF* indicados anteriormente, depois calcular o custo do passo para a heurística entre o nodo atual e o próximo, e por fim, guardando a estima final na referência de **Est** somando-lhe a heurística aplicada entre o próximo nodo e o final.

```
adjacenteAStar(Paragens, [(Nodo,C1)|Caminho]/Custo/_, Final,
[(ProxNodo,C2),(Nodo,C1)|Caminho]/NovoCusto/Est) :-
    verificarLigacaoAStar(Paragens, Nodo, (ProxNodo,C2)),
    \+ membro((ProxNodo,C2), Caminho),
    distanciaEuclidiana(
        Nodo, ProxNodo, PassoCusto
    ),
    NovoCusto is Custo + PassoCusto,
    distanciaEuclidiana(
        ProxNodo, Final, Est
    ).
```

### 3.1.4 Pesquisa *Greedy* (gulosa)

Este problema assemelha-se ao anterior no sentido em que procura uma solução “ótima”, no entanto, guia-se pela otimalidade local e não a procura do caminho ótimo no total, ou seja, para cada iteração escolhe o nodo com o custo mais baixo sem ter em consideração o caminho como um todo.

Apesar de não encontrar sempre a melhor solução, este algoritmo também foi pensado não exatamente para isso, mas para encontrar, num número razoável de passos, uma solução próxima da ótima, o que mesmo assim, não acontece sempre.

Assim, relativamente ao algoritmo A-estrela, a única diferença está no cálculo do melhor caminho, escolhendo aquele que, em cada iteração, apresente a melhor estimativa, calculada da mesma forma:

```
(...)
bestPathGreedy([Caminho1/Custo1/Est1, _/Custo2/Est2|Caminhos], MelhorCaminho):-
    Est1 =< Est2, !,
    bestPathGreedy([Caminho1/Custo1/Est1|Caminhos], MelhorCaminho).
(...)
```

Em geral, temos uma solução calculada de forma mais rápida que por vezes dá jeito em alguns casos, mas que neste sistema será utilizada para uma ótima mais de aplicação de diferentes algoritmos.

## 3.2 Consultas à base de conhecimento

Na introdução foram apresentados os objetivos gerais deste programa, pelo que, nas próximas secções apresentar-se-ão as estratégias utilizadas na procura de trajetos entre dois nodos.

A secção dos Algoritmos de procura serviu para, mais que saber como aplicá-los, ter um conjunto de rotinas genéricas para aplicar nas *queries* ao sistema, isto é, fornecer a esses algoritmos um conjunto de paragens válidas na procura e, de forma geral, aplicar um ou outro algoritmo apropriado a cada caso.

### 3.2.1 Calcular um trajeto entre dois pontos

Calcular o trajeto entre dois nodos do grafo será a consulta base ao sistema, pelo que a estratégia fica mais simples quando se estabeleceram os algoritmos de procura generalizada, assim:

```
calcula_trajeto(Nodo_A, Nodo_B, Caminho/CostTime) :-
findall(paragem(A,B,C,D,E,F,G,H,I,J), paragem(A,B,C,D,E,F,G,H,I,J), Paragens),
        resolve_depthFirst(Paragens, Nodo_A, Nodo_B, Caminho), CostTime = 'n/a'.
        %resolve_breathFirst(Paragens, Nodo_A, Nodo_B, Caminho), CostTime = 'n/a'.
        %resolve_astar(Paragens, Nodo_A, Nodo_B, Caminho/CostTime).
        %resolve_greedy(Paragens, Nodo_A, Nodo_B, Caminho/CostTime).
```

O primeiro passo é fornecer aos algoritmos o conjunto de paragens válidas no cálculo do percurso, ora como não existem restrições o *findall* permite obter em **Paragens** essa lista e, como para todos os casos é devolvido um **Caminho/CostTime** e os dois primeiros algoritmos não tratam disso estabeleci como padrão “n/a”.

Podemos aplicar este algoritmo então para um percurso entre o nodo 183 e 79 por exemplo, obtendo:

```
-----
| ?- calcula_trajeto(183,79,Caminho).
Caminho = [(183,'Start'),(791,1),(595,1),(182,1),(499,1),(593,1),(181,1),(180,1),(...,...)].../'n/a' ?
-----
```

Figura 3.3: Exemplo DFS para a consulta 1.

Podemos remover o comentário “%” da query para o A-estrela, por exemplo, e obter, da mesma forma, um caminho, mas desta vez com o custo especificado, para um outro exemplo, entre 183 e 181:

```
| ?- calcula_trajeto(183,181,CaminhoCusto).

Caminho:
Nodo:183,Start
Nodo:791,1
Nodo:595,1
Nodo:182,1
Nodo:181,7
Cost=20.278
CaminhoCusto = [(183,'Start'),(791,1),(595,1),(182,1),(181,7)]/20.277
```

Figura 3.4: Exemplo A\* para a consulta 1.

De realçar, mais uma vez, que para cada caminho é obtida lista de pares, contendo cada um (**Nodo**, **Carreira**).

### 3.2.2 Selecionar apenas algumas das operadoras de transporte para um determinado percurso

Esta consulta, assim como algumas que irei falar a seguir são uma especialização da anterior, onde o conjunto de dados fornecidos aos algoritmos de procura é diminuído restringindo apenas as operadoras que são passadas ao predicado como argumento:

```
calcula_trajeto_operadoras(Nodo_A, Nodo_B, ListaOperadoras) :-  
findall(paragem(A,B,C,D,E,F,G,H,I,J), paragem(A,B,C,D,E,F,G,H,I,J), Paragens),  
incluemOperadoras(ListaOperadoras, Paragens, [], ParagensOperadoras),  
    resolve_depthFirst(ParagensOperadoras, Nodo_A, Nodo_B, Caminho).  
%resolve_breathFirst(ParagensOperadoras, Nodo_A, Nodo_B, Caminho).  
%(...outros algoritmos).
```

O predicado **incluemOperadoras** vai pegar na lista total de paragens fornecida pelo **findall** e vai devolver aquelas cujo campo operadora pertencer à lista **ListaOperadoras**.

```
incluemOperadoras([], Paragens, Acc, Res) :- appendToList(Acc, [], Res).  
incluemOperadoras([Oper|Tail], Paragens, Acc, Res) :-  
    findall(paragem(A,B,C,D,E,F,Oper,H,I,J), % todas paragens com Oper  
    paragem(A,B,C,D,E,F,Oper,H,I,J), Filtered),  
    appendToList(Filtered, Acc, R1), % add ao acumulado  
    incluemOperadoras(Tail, Paragens, R1, Res). % processo recursivo
```

Aplicando este algoritmo ao percurso entre 183 e 595, que apenas utiliza a Operadora **Vimeca** podemos ver as respostas que o *Prolog* nos dá quando pedimos para incluir a mesma ou a **Carris** que não está naquele percurso:

```
| ?- calcula_trajeto_operadoras(183, 595, ['Vimeca']).  
Caminho:  
Nodo:183,Start  
Nodo:791,1  
Nodo:595,1  
yes  
| ?- calcula_trajeto_operadoras(183, 595, ['Carris']).  
no
```

Figura 3.5: Exemplo DFS para a consulta 2.

### 3.2.3 Excluir um ou mais operadores de transporte para o percurso

A consulta que se segue é semelhante à anterior, pelo que o raciocínio é o oposto desta vez, ou seja, em vez de encontrar todas as paragens que incluem uma dada lista de operadoras temos de excluir uma lista das mesmas:

```
calcula_trajeto_excluir_operadoras(Nodo_A, Nodo_B, ListaOperadoras) :-  
findall(paragem(A,B,C,D,E,F,G,H,I,J), paragem(A,B,C,D,E,F,G,H,I,J), Paragens),  
excluemOperadoras(Paragens, ListaOperadoras, [], ParagensSemOperadoras),  
    resolve_depthFirst(ParagensSemOperadoras, Nodo_A, Nodo_B, Caminho)  
%(...outros algoritmos).
```

O algoritmo de filtrar as paragens é bastante simples, vai se criando uma lista a partir de uma acumulador vazio no qual se inclui as paragens cuja *Oper*, da consulta anterior, não pertença à lista de operadoras fornecida, eis alguns exemplos:

```

| ?- calcula_trajeto_excluir_operadoras(183, 595, ['Vimeca', 'SCoTTURB']).
no
| ?- calcula_trajeto_excluir_operadoras(183, 595, ['Carris']).

Caminho:

Nodo:183,Start
Nodo:791,1
Nodo:595,1
yes

```

Figura 3.6: Exemplo DFS para a consulta 3.

### 3.2.4 Identificar quais as paragens com o maior número de carreiras num determinado percurso

Para realizar esta consulta é necessário primeiro obter o caminho entre dois nodos primeiro e após isso, para cada nodo obtido calcular quantas carreiras existem para cada paragem (nodo):

```

calcula_trajeto_nr_carreiras(Nodo_A, Nodo_B) :-
findall(paragem(A,B,C,D,E,F,G,H,I,J), paragem(A,B,C,D,E,F,G,H,I,J), Paragens),
    resolve_depthFirst(Paragens, Nodo_A, Nodo_B, Caminho),
    %(...outros algoritmos),
    calcular_paragens(Caminho, Reversed),
    invertLista(Reversed, Normal, []),
    mais_carreiras_paragens(Normal, -1, P, Res, Pres),
    displayGeneral('\nParagem com maior nr. de carreiras\n', Pres),
    displayGeneral('Nr. de carreiras\n', Res).

```

Ou seja, partimos de todas as paragens, sem restrições, ao contrário das consultas anteriores, depois utiliza-se o predicado **calcula\_paragens** para obter uma lista de paragens associada ao número de carreiras que por lá passam:

```

% (...chama o calcular_paragens_aux..)
calcular_paragens_aux([(Nodo,Carr)|RestoCaminho], Acc, Result) :-
    %(...)
    calcular_ligacoes(Nodo, [], ListaLigacoes),
    lengthList(R2, Tamanho), appendToList([Paragem/Tamanho], Acc, R1),
    calcular_paragens_aux(RestoCaminho, R1, Result).

```

Assim, resta-nos apenas escolher a paragem cujo número de carreiras seja o maior, ou seja, o predicado **mais\_carreiras\_paragens** utiliza uma lista, como a seguinte:

[Paragem/NR\_Carreiras|...tail] -> devolve: Paragem/NR\_Carreiras

```

Paragens:

paragem(183,-103678.36,-96590.26,Bom,Fechado dos Lados,Yes,Vimeca,286,Rua Aquilino Ribeiro,Carnaxide e Queijas) | Value = 6;
paragem(791,-103705.46,-96673.6,Bom,Aberto dos Lados,Yes,Vimeca,286,Rua Aquilino Ribeiro,Carnaxide e Queijas) | Value = 6;
paragem(595,-103725.69,-95975.2,Bom,Fechado dos Lados,Yes,Vimeca,354,Rua Manuel Teixeira Gomes,Carnaxide e Queijas) | Value = 7;

Paragem com maior nr. de carreiras
= paragem(595,-103725.69,-95975.2,Bom,Fechado dos Lados,Yes,Vimeca,354,Rua Manuel Teixeira Gomes,Carnaxide e Queijas)
Nr. de carreiras
= 7
yes _

```

Figura 3.7: Exemplo DFS para a consulta 4 (de 183 a 595).



### 3.2.5 Escolher o menor percurso (usando critério menor número de paragens)

A resolução deste consulta podia ser feita, a meu ver, de duas formas, uma utilizando uma heurística que seria o comprimento do percurso, utilizando o A-estrela para o efeito, ou a estratégia, que foi a adotada, de calcular, através do *findall*, todos os caminhos entre um nodo A e um nodo B e escolher aquele cujo comprimento seja o menor.

São algumas as razões para ter adotado esta estratégia, porque como o grafo se torna algo incomportável para certos casos, e porque o A-estrela em geral bloqueia muito para caminhos com mais de 5/6 nodos, optei por manter uma pesquisa *depth-first*, o que foi acontecendo também em algumas *queries* mais à frente.

Ora o algoritmo é o seguinte:

```
menor_percurso(Nodo_A, Nodo_B, Caminhos) :-
    findall(C, calcula_trajeto(Nodo_A, Nodo_B, C), Caminhos),
    menos_paragens(Caminhos, 9999, L, R1, R2), %(...).
```

Aqui, o predicado **menos\_paragens** vai apenas, para uma dada lista de caminhos, obter aquele que tem o menor comprimento, ou seja, menos paragens<sup>3</sup>, exemplificando:

```
| ?- menor_percurso(183,595,R).

Caminho com menor numero de paragens:
= [(183,Start),(791,1),(595,1)]
Numero de paragens:
= 3
R = [[(183,'Start'),(791,1),(595,1)]/'n/a',[(183,'Start'),(791,1),(595,7)]/'n/a',[(183,'Start'),(791,1),(595,10)]/'n/a',[(183,'Start'),(791,1),(595,12)]/'n/a',
[(183,'Start'),(791,1),(595,13)]/'n/a',[(183,'Start'),(791,1),(595,15)]/'n/a',[(183,'Start'),(791,7),(.....)]/'n/a',[(.....),(.....)]/'n/a',[(.....)]/'n/a',[(.....)]/'n/a',
/ 'n/a',.... / ...[...]?
yes _
```

Figura 3.8: Exemplo menor percurso para a consulta 5.

Podemos ver então, que apesar do exemplo não ser o melhor, temos em **R** os vários caminhos entre 183 e 595 e o *print* em cima mostra-nos o mais curto (um dos) assim como o número de paragens.

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

Já neste caso será mais justo tirar partido do algoritmo A-estrela com uma heurística de menor distância utilizando uma estimativa como guia de cálculo do custo dos vários caminhos e, posteriormente, do caminho final, assim, temos o seguinte excerto *Prolog*:

```
mais_rapido(Nodo_A, Nodo_B, Caminho/CostTime) :-
    %(...findall de todas as paragens),
    resolve_astar(Paragens, Nodo_A, Nodo_B, Caminho/CostTime).
    %resolve_greedy(Paragens, Nodo_A, Nodo_B, Caminho/CostTime).
```

O algoritmo A-estrela e *Greedy* já foram explicados anteriormente, tendo apenas sido descrito a heurística utilizada, podendo ela ser melhor especificada agora:

```
distanciaEuclidiana(Node1, Node2, DistanceCost) :-
    paragem(Node1,Lat1,Long1,_,_,_,_,_,_),
    paragem(Node2,Lat2,Long2,_,_,_,_,_,_),
    ValPi is pi,
```

<sup>3</sup>O valor 9999 é passado para um acumulador que obtém o número mínimo de paragens

```

Fi1 is Lat1 * (ValPi/180),
Fi2 is Lat2 * (ValPi/180),
% (...outros calculos omitidos...),
Dist is 6.371*1000 * C,
DistanceCost is Dist/1000.

```

Ou seja, obtém a distância **real**, considerando a curvatura da terra, entre dois pontos definidos pelas coordenadas latitude e longitude, para cada paragem em **km** considerando a relação **1km  $\simeq$  1 minuto**. Vejamos então alguns exemplos de aplicação dos algoritmos:

```

C = [(183,'Start'),(791,1),(595,1),(182,1)],
T = 16.590986313758854 ? ■

```

Figura 3.9: Exemplo percurso mais curto para a consulta 6.

Neste exemplo, em ambos os algoritmos de pesquisa informada encontram a mesma distância e consequente custo de 16.59 minutos ou km entre os nodos 183 e 182. Acaba por ser difícil estabelecer uma comparação entre os algoritmos pois só conseguimos obter soluções para poucos nodos e como o conhecimento não deve ser diminuído temos de nos restringir a estes testes.

### 3.2.7 Escolher o percurso que passe apenas por abrigos com publicidade

Esta *query* foi fácil de desenvolver no sentido em que foi apenas necessário alterar o *findall* inicial que vinha a ser feito, ou seja, restringir as paragens iniciais àquelas cujo campo da publicidade esteja **'Yes'**:

```

com_publicidade(Nodo_A, Nodo_B, Caminhos) :-
findall(paragem(A,B,C,D,E,'Yes',G,H,I,J), paragem(A,B,C,D,E,'Yes',G,H,I,J),
Paragens),
    resolve_depthFirst(Paragens, Nodo_A, Nodo_B, Caminho), CostTime = 'n/a'.
%resolve_breathFirst(Paragens, Nodo_A, Nodo_B, Caminho), CostTime = 'n/a',

```

Seguem-se alguns exemplos de aplicação: Aqui o caminho entre 183 e 595 tem, em todas

```

| ?- com_publicidade(183,595,C).
Caminho:
Nodo:183,Start
Nodo:791,1
Nodo:595,1
yes
| ?- com_publicidade(594,185,C).
no

```

Figura 3.10: Exemplos DFS para a consulta 7.

as paragens abrigo com publicidade daí termos obtido um caminho, já entre 594 e 185, não existe em 594 essa característica, daí a resposta ser **no**.

### 3.2.8 Escolher o percurso que passe apenas por paragens abrigadas

O algoritmo apenas faz a filtragem das paragens iniciais onde retorna na referência de **Filtrado** as paragens abrigadas:

```
apenas_abrigadas(Nodo_A, Nodo_B, Caminho) :-
    % (...Obter todas as paragens para filtragem...),
    filtrar_com_abrigo(Paragens, [], Filtrado),
    resolve_depthFirst(Filtrado, Nodo_A, Nodo_B, Caminho)
```

As paragens podem ter, pelo menos tendo visto por alto, 3 estados diferentes:

Estados: ['Fechado dos Lados', 'Aberto dos Lados', 'Sem Abrigo']

O predicado utilizado para filtrar as paragens faz então a inclusão de todas as paragens cujo campo 'Abrigo com Publicidade?' é diferente de 'Sem Abrigo', pelo que a sua implementação acaba por ser simples, seguem-se então alguns exemplos:

```
| ?- apenas_abrigadas(183,595,C).
[(183,Start),(791,1),(595,1)]
C = [(183,'Start'),(791,1),(595,1)] ?
yes
| ?- apenas_abrigadas(628,39,C).
no
- -
```

Figura 3.11: Exemplos de procura para a consulta 8.

No percurso entre 183 e 595 todas as paragens tem algum tipo de abrigo, já entre 628 e 39, a paragem 628 não cumpre os requisitos de abrigo.

### 3.2.9 Escolher um ou mais pontos intermédios por onde o percurso deverá passar

Existirão certamente várias formas de implementar esta última consulta, no entanto, adotei a estratégia de calcular todos os caminhos entre uma dada origem e destino e verificar para todos, se os nodos do caminho pertencem todos à lista de nodos excluindo o primeiro e o último.

```
passar_por(ListaNodos, Nodo_A, Nodo_B) :-
    findall(C, calcula_trajeto(Nodo_A, Nodo_B, C), Caminhos),
    map_percursos_nodos(ListaNodos, Caminhos, 0, [], NodosCaminhos),
    first_list(NodosCaminhos, First),
    print(First).
```

O predicado que vem a seguir a calcular todos os caminhos com *findall* pega em cada um desses caminhos e seleciona a lista nos quais os requisitos fornecidos acima se cumprem.

```
map_percursos_nodos(ListaNodos, [], C, Acc, Res) :- appendToList(Acc, [], Res).
map_percursos_nodos(ListaNodos, [C1|Caminhos], Counter, Acc, Res) :-
    converte_lista_nodos(C1, [], NodosList),
    (...remover 1º e ult. element pondo em SubList),
    ( membros(SubList, ListaNodos) ->
        appendToList([(Inv, Order)], Acc, R1),
        Order is Counter + 1,
        map_percursos_nodos(ListaNodos, Caminhos, Order, R1, Res)
    ; map_percursos_nodos(ListaNodos, Caminhos, Counter, Acc, Res)).
```

Exemplificando:

```

| ?- passar_por([791],183,595).
[183,791,595],36
yes
| ?- passar_por([999],183,595).
no
| ?- passar_por([183],183,595).
no
| ~

```

Figura 3.12: Exemplos de procura para a consulta 9.

Mais uma vez referindo que o caminho entre 183 e 595 é 183,791,595 logo se indicarmos 791 como nodo intermédio funciona e temos o caminho, ora com 999, nodo que não existe, não temos caminho entre 183 e 595 que passe por 999, por outro lado, indicando como nodo intermédio o nó inicio também não irá encontrar (obs.: nem com o nó final).

### 3.2.10 Outras consultas opcionais

Aqui temos alguns exemplos de outras consultas que foram desenvolvidas para completar o sistema e mesmo apresentar caminhos com informações mais caracterizadas, dando uso aos campos até agora não escolhidos.

#### Escolher percurso com paragens com Bom estado de conservação:

Será fácil de perceber que apenas foi necessário filtrar, com ajuda do *findall*, as paragens cujo campo 'Estado de Conservação' estivesse **Bom**, assim, apresentam-se alguns exemplos:

```

| ?- bom_estado(183,595,C).

Caminho:

Nodo:183,Start
Nodo:791,1
Nodo:595,1
yes
| ?- bom_estado(939,587,C).
no
| ~

```

Figura 3.13: Exemplos de procura para a consulta 10 (extra).

As paragens 939 e 587 não apresentam um Bom estado de conservação pelo que não serão escolhidas e então não existirá um caminho.

#### Apresentar as características de um percurso (Rua, Freguesia, ...):

Faltou então, para um dado percurso, saber obter as características das paragens como a rua, a freguesia, etc., pelo que foi feita uma versão 2 da consulta 1:

```

| ?- calcula_percurso_informacao(183,79, C).

Vodo -> 183, na rua Rua Aquilino Ribeiro (codigo:286) da freguesia Carnaxide e Queijas,
Vodo -> 791, na rua Rua Aquilino Ribeiro (codigo:286) da freguesia Carnaxide e Queijas,
Vodo -> 595, na rua Rua Manuel Teixeira Gomes (codigo:354) da freguesia Carnaxide e Queijas,
Vodo -> 182, na rua Rua Aquilino Ribeiro (codigo:286) da freguesia Carnaxide e Queijas,
Vodo -> 499, na rua Avenida dos Cavaleiros (codigo:380) da freguesia Carnaxide e Queijas,
Vodo -> 593, na rua Avenida dos Cavaleiros (codigo:380) da freguesia Carnaxide e Queijas,
Vodo -> 181, na rua Rua Aquilino Ribeiro (codigo:286) da freguesia Carnaxide e Queijas,
Vodo -> 180, na rua Rua Aquilino Ribeiro (codigo:286) da freguesia Carnaxide e Queijas,
Vodo -> 594, na rua Avenida Professor Dr. Reinaldo dos Santos (codigo:1116) da freguesia Carnaxide e Queijas,
Vodo -> 185, na rua Rua Manuel Teixeira Gomes (codigo:354) da freguesia Carnaxide e Queijas,
Vodo -> 89, na rua Avenida de Portugal (codigo:1113) da freguesia Carnaxide e Queijas,
Vodo -> 107, na rua Avenida de Portugal (codigo:1113) da freguesia Carnaxide e Queijas,
Vodo -> 250, na rua Avenida de Portugal (codigo:1113) da freguesia Carnaxide e Queijas,
Vodo -> 261, na rua Avenida de Portugal (codigo:1113) da freguesia Carnaxide e Queijas,
Vodo -> 597, na rua Rua Tenente-General Zeferino Sequeira (codigo:1137) da freguesia Carnaxide e Queijas,
Vodo -> 953, na rua Avenida Professor Dr. Reinaldo dos Santos (codigo:1116) da freguesia Carnaxide e Queijas,
Vodo -> 609, na rua Avenida do Forte (codigo:327) da freguesia Carnaxide e Queijas,
Vodo -> 242, na rua Avenida Tomas Ribeiro (codigo:1279) da freguesia Carnaxide e Queijas,
Vodo -> 255, na rua Avenida Tomas Ribeiro (codigo:1279) da freguesia Carnaxide e Queijas,
Vodo -> 604, na rua Rua dos Cravos de Abril (codigo:300) da freguesia Carnaxide e Queijas,

```

Figura 3.14: Exemplos de procura para a consulta 11 (extra).

### 3.3 Comparação geral entre os tipos de pesquisa

Nesta secção vai ser apresentada uma breve comparação entre algumas das consultas utilizando pesquisa informada e não informada, apesar de já terem sido demonstrados alguns exemplos de consultas usando ambas as estratégias.

Na primeira tabela conseguimos perceber alguns dos limites no cálculo dos caminhos tendo estabelecido 4 testes para cada tipo de pesquisa adotada:

- Teste 1 (T1): 183 até 595 (fácil)
- Teste 2 (T2): 183 até 593 (médio)
- Teste 3 (T3): 183 até 250 (médio-difícil)
- Teste 4 (T4): 183 até 79 (muito difícil)

A dificuldade que explicitarei advém do nível/profundidade que a árvore dos caminhos atinge, ou seja, começando em 183, o nível 79 é o mais fundo, podendo ser muito complicado para algoritmos que se baseiam em pesquisas por largura, obter o resultado num tempo aceitável. Seguem-se então os resultados dos testes:

Não-Inf.	Consulta 1 (calc. de 1 trajeto)	Inf.	Consulta 1 (calc. de 1 trajeto)
DFS	T1: 183>595, Texec = inst. T2: 183>593, Texec = inst. T3: 183>250, Texec = inst. T4: 183>79, Texec = 0.05s	A*	T1: 183>595, Texec = inst. (C=12.4) T2: 183>593, Texec = 4.23s (C = 21.6) T3: 183>250, Texec = inf? (M). T4: 183>79, Texec = inf? (M).
BFS	T1: 183>595, Texec = inst. T2: 183>593, Texec = 1.9s T3: 183>250, Texec = 42.41s T4: 183>79, Texec = inf? (M)	Greedy	T1: 183>595, Texec = inst. T2: 183>593, Texec = inst. (C = 21.6) T3: 183>250, Texec = 0.02s (C = 45.3) T4: 183>79, Texec = inf? (T)

Tabela 3.1: Comparação da consulta 1 (cálculo de trajeto) para as diferentes estratégias.

A legenda da tabela é simples, temos alguns testes onde o tempo de execução é instantâneo (inst.), o que acontece em geral para algoritmos de pesquisa em profundidade que possuem o melhor caso para nodos finais mais profundos, o que se reflete nos problemas dos outros algoritmos que se baseiam em largura.

Chega inclusive a ser impossível calcular, pelo menos em tempo aceitável, alguns casos, onde, por exemplo, no T4 do BFS do sistema operativo reportou um problema de memória (M) inclusive durante a execução do programa.

Em geral, para estes testes onde a profundidade estava a favor de certos algoritmos temos um comportamento muito bom do DFS e pouco aceitável para o BFS com 42 segundos para o Teste 3.

Os algoritmos de pesquisa informada, pelo menos o A\* teve um comportamento esperado, semelhante ao BFS nos primeiros 2 casos, mas a partir daí, devido a problemas de memória não consegui concluir a execução de T2 e T3 que já estava em mais de 4/5 minutos.

O *Greedy*, como não utiliza a estimativa mais o custo, apenas considera uma solução ótima local o que quer dizer que avança mais rápido utilizando ligeiramente menos memória e CPU que o A\*, tendo obtido valores que este último não conseguiu para o T3. Já o T4 não deu problemas de memória, mas a execução do programa já ultrapassava os 6 minutos pelo que foi interrompida.

Resumidamente, estes testes iniciais mostram que o algoritmo DFS para uma procura dentro da mesma carreira é mais rápido e fazível em tempo aceitável, pois estando na mesma carreira com uma diferença de 76 nodos (no caso do T4) este apenas tem que seguir de adjacente em adjacente.

Este mesmo par de testes para qualquer outro algoritmo testado acima produz não só um *bottle-neck* de acumulação de dados em memória como a comparação evolui de forma exponencial para os nodos adjacentes, ora então para 76 níveis de profundidade num grafo com 1522 nodos torna-se impossível testar.

Para comparar, por exemplo, no caso da consulta 6 o percurso mais rápido entre as duas estratégias de pesquisa informada é difícil arranjar um teste onde o resultado dê diferente pois até agora ambas produzem o mesmo resultado, uma mais rápido que a outra.

### 3.4 Estrutura do trabalho, código e documentos

Aproveitando esta pequena secção irei fazer uma breve descrição da organização do código do trabalho.

Os ficheiros estão divididos em 3 pastas, **documents** com o enunciado e o documento de apoio, **datasets** com o conjunto de ficheiros de dados descritos até agora e **src** contendo o código para o *parser* e a base de conhecimento em *Prolog*.

No *Prolog* temos um ficheiro principal **main.pl** contendo o código dos predicados que chamam cada uma das *queries*, já na pasta **include** temos módulos com funções auxiliares, algoritmos de procura, heurísticas e a base de dados inicial das paragens e arcos/ligacoes.

Achei por bem incluir uma *makefile* também do lado do *Prolog* para compilar e executar o programa *main* através da regra **make**. Já o **make update** permite ir buscar o ficheiro dado como *output* pelo *parser* atualizando assim a base de conhecimento.

## Capítulo 4

# Conclusões e Sugestões

A realização deste trabalho permitiu aprimorar a componente prática associada a esta Unidade Curricular que incidiu na resolução de métodos de procura num sistema fortemente ligado através de um grafo aplicado a um caso real, as paragens de autocarros do concelho de Oeiras, sendo possível relembrar conceitos relacionados com algoritmos de procura e aplicá-los em *Prolog*.

As conclusões que podemos tirar deste projeto são simples no sentido em que os testes se tornam muito complicados para a quantidade de dados utilizada, a não ser que se remova uma grande parte dos dados, o que dificultou muito a resolução das consultas que exigiam exemplos mais significativos, nomeadamente, envolvendo pesquisas não-informadas.

Apesar de tudo isso, os objetivos acabam por ser cumpridos na totalidade, tendo sido feita uma análise extensiva de soluções e alternativas ao que havia sido proposto.

Deixo em aberto que a estrutura da aplicação permite escalabilidade, na medida em que, se o problema estiver nos algoritmos de procura ou se se pretender implementar novos predicados esta está desenvolvida de forma generalizada sendo apenas necessário alterar o predicado a utilizar.