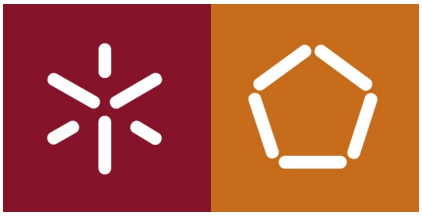

Universidade do Minho
Mestrado Integrado em Engenharia Informática
3ºano - 2º Semestre

Sistemas de Representação de Conhecimento e Raciocínio

Componente Individual

a83732 – Gonçalo Rodrigues Pinto

5 de Junho de 2020

Conteúdo

Lista de Figuras

Lista de Tabelas

1 Introdução

No 2º semestre do 3º ano do Curso de Engenharia Informática da Universidade do Minho, existe uma unidade curricular denominada por Sistemas de Representação de Conhecimento e Raciocínio, que tem como objectivo introduzir aos estudantes os paradigmas da representação do conhecimento e de raciocínio lógico, e sua aplicação na concepção e implementação de sistemas inteligentes ou de apoio à decisão, em que a qualidade da informação e o grau de confiança que é depositado na conjugação dos atributos de um predicado ou função lógica é fulcral como também tem como objectivo ajudar os estudantes a compreender a relação entre a complexidade de um modelo representação de conhecimento e as formas de raciocínio a ele associadas e o seu desempenho, utilizando esta informação na definição de uma estratégia para a sua optimização.

O presente trabalho teve como principal objectivo motivar os alunos para a utilização da Programação em Lógica, usando a linguagem de programação PROLOG, no âmbito de métodos de Resolução de Problemas e no desenvolvimento de algoritmos de pesquisa.

Neste trabalho pretendeu-se desenvolver um sistema, que permita importar os dados relativos ao conjunto das paragens de autocarro existentes no concelho de Oeiras, e representá-los numa base de conhecimento, da forma mais adequada. Posteriormente, foi desenvolvido um sistema de recomendação de transporte público para o caso de estudo.

2 Descrição do Problema

Neste trabalho o *dataset* disponibilizado foi pré-processado, tendo sido retiradas algumas inconsistências nos dados e propriedades irrelevantes para o trabalho. Este *dataset* tem a seguinte estrutura:

1. Latitude
2. Longitude
3. Gid (identificador para um ponto no mapa)
4. Estado de Conservação
5. Tipo de Abrigo
6. Abrigo com Publicidade
7. Operadora
8. Carreira
9. Código de Rua
10. Nome da Rua
11. Freguesia

A partir desta caracterização e para a realização do trabalho, construiu-se um caso prático de aplicação dos conhecimentos, que foi capaz de demonstrar as funcionalidades subjacentes à programação em lógica estendida e aos métodos de resolução de problemas e de procura.

A elaboração do caso prático foi implementada de forma a respeitar as necessidades de demonstração das seguintes funcionalidades tais como 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 e escolher um ou mais pontos intermédios por onde o percurso deverá passar

No desenvolvimento das soluções, considerou-se diferentes estratégias de pesquisa (não-informada e informada).

3 Descrição do Trabalho

3.1 Estratégias de pesquisa utilizadas

De forma a desenvolver um sistema de recomendação de transporte público para o caso de estudo, considerou-se diferentes estratégias de pesquisa, uma estratégia de pesquisa é definida escolhendo a ordem da expansão do nó.

Tal como foi estudado aplicou-se dois tipos de pesquisa, a **pesquisa não-informada (cega)** que usa apenas as informações disponíveis na definição do problema e a **pesquisa informada (heurística)** onde dá-se ao algoritmo "dicas" sobre a adequação de diferentes estados.

3.1.1 Pesquisa não-informada (cega)

Em relação a este tipo considerou-se as seguintes pesquisas:

- Pesquisa Primeiro em Largura (Breadth-first search) - onde a estratégia é que todos os nós de menor profundidade são expandidos primeiro. Basicamente, é uma boa pesquisa porque é sistemática contudo normalmente demora muito tempo e sobretudo ocupa muito espaço.

- Pesquisa Primeiro em Profundidade (Depth-FirstSearch) - onde a estratégia é expandir sempre um dos nós mais profundos da árvore. De forma abreviada, é de optar por esta pesquisa pois é muito pouca memória necessária, bom para problemas com muita soluções mas não pode ser usada para árvores com profundidade infinita, pode ficar presa em ramos errados.

3.1.2 Pesquisa informada (heurística)

Em relação a este tipo considerou-se as seguintes pesquisas:

- Pesquisa Gulosa (Greedy-Search) - a ideia é expandir o nó que parece estar mais perto da solução para isso utiliza uma função $h(n)$ que corresponde ao custo estimado para o objectivo, a designada função heurística.

- Pesquisa A* - que evita expandir caminhos que são caros, este algoritmo combina a pesquisa gulosa com a uniforme, minimizando a soma do caminho já efectuado com o mínimo previsto que falta até a solução. Usa a função $f(n) = g(n)+h(n)$ que calcula o custo estimado da solução mais barata que passa pelo nó n, onde $g(n)$ representa o custo total, até agora, para chegar ao estado n (custo do percurso).

3.2 Representação numa base de conhecimento

O sistema de representação de conhecimento e raciocínio desenvolvido caracteriza paragens de autocarro e como se pretende desenvolver um sistema de recomendação de transporte público para o caso de estudo é necessário representar as diferentes ligações que as paragens podem ter tal como uma estimativa de tempo de espera em cada paragem.

Desta forma, criou-se um programa em **Java** disponível em anexo que permitiu importar os dados relativos às paragens de autocarro. Este sistema efectuou a leitura dos diferentes *datasets* referentes às paragens e às suas ligações, processou a informação e posteriormente escreveu num ficheiro que representa a base de conhecimento.

O universo de discurso considerado para o trabalho prático é o de paragens de autocarro. Com intuito de clarificar e restringir alguns significados, considerou-se no âmbito deste trabalho as seguintes definições:

- paragem - possui a informação (apresentada no capítulo anterior) presente no *dataset* disponibilizado;
- ligacao - representa a conexão entre duas diferentes paragens através de uma carreira presentes nos diversos *datasets* que foram facultados;
- estima - corresponde a interligação entre uma paragem e um valor aleatório entre 0 e 300 segundos (0 a 5 minutos) que representa o tempo entre cada paragem;

3.2.1 Paragem

Uma paragem é caracterizada por um GID (identificador único), latitude, longitude, indicação do estado de conservação, o tipo de abrigo, se é um abrigo com publicidade, a operadora responsável por essa paragem, uma lista de carreiras a que está associada, o código de rua associado, tal como o nome da rua e por fim a freguesia. O seguintes predicados representam paragens:

```
paragem( 79,-107011.55,-95214.57,"Bom","Fechado dos Lados","Yes","Vimeca"
    ,[01],103,"Rua Damiao de Gois","Alges, Linda-a-Velha e Cruz Quebrada-
    Dafundo").
paragem( 593,-103777.02,-94637.67,"Bom","Sem Abrigo","No","Vimeca",[01],300,
    "Avenida dos Cavaleiros","Carnaxide e Queijas").
paragem( 499,-103758.44,-94393.36,"Bom","Fechado dos Lados","Yes","Vimeca"
    ,[01],300,"Avenida dos Cavaleiros","Carnaxide e Queijas").
```

Assim, podemos constatar que a paragem com o identificador 79 cuja latitude é -107011.55 e longitude é -95214.57 situada na "Rua Damiao de Gois"em "Alges, Linda-a-Velha e Cruz Quebrada-Dafundo"com código 103 possui um "Bom"estado de conservação, esta por sua vez é abrigada pois é "Fechada dos Lados"e possui publicidade ("Yes"). A operadora responsável por esta paragem é a "Vimeca"e a única carreira que passe nesta paragem é a carreira 01.

3.2.2 Ligação

Uma ligação é caracterizada por uma origem, um destino e a carreira que a efectua. Os seguintes predicados representam ligações:

```
ligacao( 183,791,1 ).
ligacao( 791,595,1 ).
ligacao( 595,182,1 ).
ligacao( 182,181,7 ).
ligacao( 181,180,15 ).
```

Desta forma é possível afirmar que entre a paragem com o identificador 183 e a paragem com o identificador 791 é possível transitar pois existe uma ligação fornecida pela carreira número 1, sendo 183 a origem e 791 o destino respectivamente.

3.2.3 Estima

Como foram consideradas diferentes estratégias de pesquisa algumas necessitam de uma função heurística para serem determinadas, com esse intuito criou-se o presente predicado que contém uma paragem e o custo aleatório de espera nessa mesma paragem até 300 segundos (5 minutos) entre o próximo autocarro. Os seguintes predicados representam exemplos de estimas definidos:

```
estima( 79,8 ).
estima( 593,35 ).
estima( 499,7 ).
```

Definindo este predicado é possível afirmar que existe uma função heurística que traduz que normalmente na paragem com o identificador número 79 demora 8 segundos a passar um autocarro. Realçar que este valor foi obtido aleatoriamente.

3.3 Funcionalidades

A partir da base de conhecimento criada construiu-se um caso prático de aplicação dos conhecimentos implementando as seguintes funcionalidades com as estratégias de pesquisa apresentadas mais apropriadas.

3.3.1 Cálculo do trajecto entre dois pontos

De forma a desenvolver este caso considerou-se todas as estratégias de pesquisa, aqui cada predicado designado *percurso* com o sufixo corresponde ao tipo de pesquisa, ou seja, *percursodf* corresponde a calcular o trajecto entre dois pontos através de uma Pesquisa Primeiro em Profundidade (Depth-First Search), *percursobf* corresponde a calcular o trajecto entre dois pontos através de uma Pesquisa Primeiro em Largura (Breadth-First Search) e assim sucessivamente. Todos estes predicados recebem três argumentos (paragem inicial, paragem final, o percurso calculado).

```
percursodf( Inicial,Final,[Inicial|Trajeto] ) :- resolvedf( Final,Inicial,
    Trajeto ).

resolvedf( Final,Paragem,[] ) :- Final is Paragem.
resolvedf( Final,Paragem,[ProxPar|Trajeto] ) :- adjacente(Paragem,ProxPar,_)
    , resolvedf( Final,ProxPar,Trajeto ).
```

Através da Pesquisa Primeiro em Profundidade, adiciona-se a paragem inicial e recorre-se à função auxiliar *resolvedf* onde se indica qual é a paragem final, esta por sua vez verifica inicialmente se a paragem que está a calcular de momento é a paragem final e caso seja indica o fim do percurso através da lista vazia contudo caso não se verifica a condição procura uma paragem adjacente, verificando se existe uma ligação (em que a paragem actual seja a origem) encontrando uma paragem como destino adiciona esta ao trajecto calculado até de momento e posteriormente invoca a chamada recursiva da função auxiliar (*resolvedf*).

```
percursobf( Inicial,Final,Solucao ) :- resolvebf( Final,[[Inicial]],S ),
    inverso(S,Solucao).

resolvebf( Final,[[Estado|Percurso]|_],[Estado|Percurso] ) :- Final is
    Estado.
resolvebf( Final,[Percurso|Historico],Solucao ) :- atualiza( Percurso,
    NovoPercurso ), junta(Historico,NovoPercurso,NovoHistorico), resolvebf(
    Final,NovoHistorico,Solucao ).

atualiza( [Estado|Percurso],NovoPercurso )  :- bagof( [NovoEstado,Estado|
    Percurso],(adjacente(Estado,NovoEstado,_),naopertence(NovoEstado,[Estado
    |Percurso])),NovoPercurso ), !.
atualiza( Percurso,[] ).
```

Através da Pesquisa Primeiro em Largura, recorre à função auxiliar *resolvebf* onde indica qual é a paragem final e o percurso de momento, após esta função terminar faz a inversão do caminho obtido. A função *resolvebf* vai inicialmente verificar se o estado actual é a paragem final caso não seja vai actualizar o percurso através de uma função auxiliar (*atualiza*) de seguida junta o percurso que já tinha calculado previamente ao novo caminho posteriormente efectua a chamada recursiva indicando o novo caminho. A função *atualiza* através da função disponibilizada pelo PROLOG *bagof* que permite extrair as informações que queremos de uma maneira mais estruturada, efectua a pesquisa por um novo estado adjacente ao actual sendo esse um novo estado que ainda não pertença ao caminho actual.

```
percursoaestrela( Inicial,Final,Caminho/Custo ) :- estima(Inicial,Estima),
    resolveaestrela( Final,[[Inicial]/1/Estima],InvCaminho/Custo/_ ),
    inverso(InvCaminho,Caminho).

resolveaestrela( Final,Caminhos,Caminho ) :- obtem_melhor( Caminhos,Caminho
    ), Caminho=[Paragem|_]/_/_, Paragem is Final.
resolveaestrela( Final,Caminhos,SolucaoCaminho ) :- obtem_melhor( Caminhos,
    MelhorCaminho ), seleciona(MelhorCaminho,Caminhos,OutrosCaminhos),
    expande_aestrela( MelhorCaminho,ExpCaminhos ), junta(OutrosCaminhos,
    ExpCaminhos,NovoCaminhos), resolveaestrela( Final,NovoCaminhos,
    SolucaoCaminho ).

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

expande_aestrela( Caminho,ExpCaminhos ) :- solucoes(NovoCaminho,seguinte(
    Caminho,NovoCaminho ),ExpCaminhos).

seguinte( [Nodo|Caminho]/Custo/_,[ProxNodo,Nodo|Caminho]/NovoCusto/Est ) :-
    adjacente(Nodo,ProxNodo,_), \+member(ProxNodo,Caminho), NovoCusto is
    Custo + 1, estima(ProxNodo,Est).
```

Através da Pesquisa A*, calcula-se a estima da paragem inicial, de seguida invoca-se a função auxiliar *resolveaestrela* passando a paragem final e que constrói um caminho inicial com apenas a paragem inicial, inicialmente esta função auxiliar vai obter o melhor caminho dos vários caminhos que possui e consequentemente vai comparar a cabeça da lista do melhor caminho se corresponde à paragem final, caso falhe vai obter o melhor caminho dos vários calculados, selecciona esse através de um predicado auxiliar *seleciona*, de seguida faz a sua expansão através do predicado *expande_aestrela*, junta esse caminho gerando novos caminhos e faz a chamada recursiva com isso.

Para obter o melhor é utilizado o predicado *obtem_melhor* que através da comparação dos custos associados a cada caminho somado com a estimação faz a selecção do melhor caminho. O predicado *expande_aestrela* vai procurar todos os caminhos que cumprem o predicado *seguinte* que verifica se duas paragens são adjacentes, se a próxima paragem não pertence ao caminho calculado e é associado o custo somando 1 unidade de forma a obter o caminho mais curto e também a estimativa do próxima paragem. Após o *resolveaestrela* ser efectuado o caminho obtido é invertido.

```
percursogulosa( Inicial,Final,Caminho/Custo ) :- estima(Inicial,Estima),
    resolvegulosa( Final,[[Inicial]/1/Estima],InvCaminho/Custo/_ ), inverso(
    InvCaminho,Caminho).

resolvegulosa( Final,Caminhos,Caminho ) :- obtem_melhor_g( Caminhos,Caminho
    ), Caminho=[Paragem|_]/_/_, Paragem is Final.
resolvegulosa( Final,Caminhos,SolucaoCaminho ) :- obtem_melhor_g( Caminhos,
    MelhorCaminho ), seleciona(MelhorCaminho,Caminhos,OutrosCaminhos),
    expande_gulosa( MelhorCaminho,ExpCaminhos ), junta(OutrosCaminhos,
    ExpCaminhos,NovoCaminhos), resolvegulosa( Final,NovoCaminhos,
    SolucaoCaminho ).

obtem_melhor_g( [Caminho],Caminho ) :- !.
obtem_melhor_g( [Caminho1/Custo1/Est1,_/Custo2/Est2|Caminhos],MelhorCaminho
    ) :- Est1 =< Est2, !, obtem_melhor_g( [Caminho1/Custo1/Est1|Caminhos],
    MelhorCaminho ). %>
obtem_melhor_g( [_|Caminhos],MelhorCaminho ) :- obtem_melhor_g( Caminhos,
    MelhorCaminho ).

expande_gulosa( Caminho,ExpCaminhos ) :- solucoes(NovoCaminho,seguinte(
    Caminho,NovoCaminho),ExpCaminhos).
```

Através da Pesquisa Gulosa, calcula-se a estima da paragem inicial, de seguida invoca-se a função auxiliar *resolvegulosa* passando a paragem final e que constrói um caminho inicial com apenas a paragem inicial, inicialmente esta função auxiliar vai obter o melhor caminho dos vários caminhos que possui e consequentemente vai comparar a cabeça da lista do melhor caminho se corresponde à paragem final, caso falhe vai obter o melhor caminho dos vários calculados, selecciona esse através de um predicado auxiliar *seleciona*, de seguida faz a sua expansão através do predicado *expande_gulosa*, junta esse caminho gerando novos caminhos e faz a chamada recursiva com isso. Para obter o melhor é utilizado o predicado *obtem_melhor_g* que através da comparação das estimações faz a selecção do melhor caminho. O predicado *expande_gulosa* vai procurar todos os caminhos que cumprem o predicado *seguinte* que verifica se duas paragens são adjacentes, se a próxima paragem não pertence ao caminho calculado e é associado o custo somando 1 unidade de forma a obter o caminho mais curto e também a estimativa do próxima paragem. Após o *resolvegulosa* ser efectuado o caminho obtido é invertido.

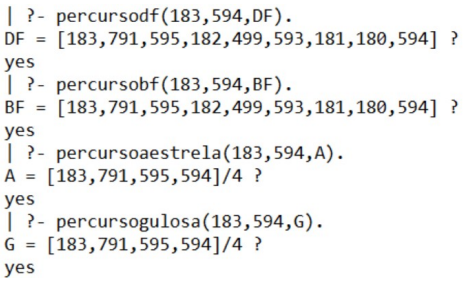

Figura 1: Trajecto entre a paragem 183 e 594 utilizando diferentes tipos de pesquisa.

3.3.2 Selecção de apenas algumas das operadoras de transporte para um determinado percurso

De forma a desenvolver este caso considerou-se todas as estratégias de pesquisa, aqui cada predicado designado *percursoOperadoras* com o sufixo corresponde ao tipo de pesquisa, ou seja, *percursoOperadorasdf* corresponde à selecção de apenas algumas das operadoras de transporte para um determinado percurso através de uma Pesquisa Primeiro em Profundidade (Depth-First Search), *percursoOperadorasbf* corresponde à selecção de apenas algumas das operadoras de transporte para um determinado percurso através de uma Pesquisa Primeiro em Largura (Breadth-First Search) e assim sucessivamente. Todos estes predicados recebem três argumentos:

- paragem inicial, paragem final, o conjunto de operadoras a seleccionar e o percurso calculado

De forma a não tornar o relatório tão longo e cansativo irei a partir daqui apresentar alternadamente o processo de raciocínio tal como implementação de um tipo de pesquisa. O processo acaba por ser semelhante para os outros tipos de pesquisa a nível de testes de predicados e sua utilização.

```
percursoOperadorasbf( Inicial,Final,Operadoras,Solucao ) :- paragem(Inicial,
    _,_,_,_,_,Op,_,_,_,_,_), pertence(Op,Operadoras), percursoOpbf( Final,
    Operadoras,[[Inicial]],S ), inverso(S,Solucao).

percursoOpbf( Final,Operadoras,[[Estado|Percurso]|_],[Estado|Percurso] ) :-
    paragem(Estado,_,_,_,_,_,_,Op,_,_,_,_,_), pertence(Op,Operadoras), Final is
    Estado.
percursoOpbf( Final,Operadoras,[Percurso|Historico],Solucao ) :- atualizaOp(
     Operadoras,Percurso,NovoPercurso ), junta(Historico,NovoPercurso,
    NovoHistorico), percursoOpbf( Final,Operadoras,NovoHistorico,Solucao ).

atualizaOp( Operadoras,[Estado|Percurso],NovoPercurso ) :- bagof( [
    NovoEstado,Estado|Percurso],(adjacente(Estado,NovoEstado,_),naopertence(
    NovoEstado,[Estado|Percurso]),paragem(NovoEstado,_,_,_,_,_,_,Op,_,_,_,_,_),
    pertence(Op,Operadoras)),NovoPercurso ), !.
atualizaOp( Operadoras,Percurso,[] ).
```

Através da Pesquisa Primeiro em Largura, é inicialmente verificado se a paragem existe como é retirado a Operadora responsável pela carreira, por conseguinte verifica-se se esta pertence ao conjunto de Operadoras passado como parâmetro e de seguida recorre-se à função auxiliar *percursoOpbf* onde se indica qual é a paragem final, as Operadoras a seleccionar e o percurso de momento após esta função terminar é efectuado a inversão do caminho obtido. A função *percursoOpbf* vai inicialmente verificar se o estado actual é a paragem final tal como se a carreira do estado actual pertence ao conjunto de Operadoras, caso não seja vai actualizar o percurso através de uma função auxiliar (*atualizaOp*) de seguida junta o percurso que já tinha calculado previamente ao novo caminho posteriormente efectua a chamada recursiva indicando o novo caminho. A função *atualizaOp* através da função disponibilizada pelo PROLOG *bagof* que permite extrair as informações que queremos de uma maneira mais estruturada, efectua a pesquisa por um novo estado adjacente ao actual sendo esse um novo estado que ainda não pertença ao caminho actual além da verificação da Operadora dele pertencer ao conjunto de Operadoras.

```
| ?- percursoOperadorasbf(318,273,["LT"],R).
R = [318,327,326,273] ?
yes
```

Figura 2: Selecção de apenas a operadora de transporte "LT"para um percurso entre a paragem 318 e a 273.

3.3.3 Exclusão de um ou mais operadores de transporte para o percurso

De forma a desenvolver este caso considerou-se todas as estratégias de pesquisa, aqui cada predicado designado *percursoNaoOperadoras* com o sufixo corresponde ao tipo de pesquisa, ou seja, *percursoNaoOperadorasdf* corresponde à exclusão de um ou mais operadores de transporte para o percurso através de uma Pesquisa Primeiro em Profundidade (Depth-First Search), *percursoNaoOperadorasbf* corresponde à exclusão de um ou mais operadores de transporte para o percurso através de uma Pesquisa Primeiro em Largura (Breadth-First Search) e assim sucessivamente. Todos estes predicados recebem três argumentos:

- paragem inicial, paragem final, o conjunto de operadoras a excluir e o percurso calculado

```
percursoNaoOperadorasaestrela( Inicial,Final,Operadoras,Caminho/Custo ) :-
    paragem(Inicial,_,_,_,_,_,_,Op,_,_,_,_,_), naopertence(Op,Operadoras),
    estima(Inicial,Estima), percursoNOpaestrela( Final,Operadoras,[[Inicial
    ]/1/Estima],InvCaminho/Custo/_ ), inverso(InvCaminho,Caminho).

percursoNOpaestrela( Final,Operadoras,Caminhos,Caminho ) :- obtem_melhor(
    Caminhos,Caminho), Caminho=[Paragem|_]/_/_, paragem(Paragem,_,_,_,_,_,_,Op
    ,_,_,_,_,_), naopertence(Op,Operadoras), Final is Paragem.
percursoNOpaestrela( Final,Operadoras,Caminhos,SolucaoCaminho ) :-
    obtem_melhor(Caminhos,MelhorCaminho), seleciona(MelhorCaminho,Caminhos,
    OutrosCaminhos), expande_nop( Operadoras,MelhorCaminho,ExpCaminhos ),
    junta(OutrosCaminhos,ExpCaminhos,NovoCaminhos), percursoNOpaestrela(
    Final,Operadoras,NovoCaminhos,SolucaoCaminho ).

expande_nop( Operadoras,Caminho,ExpCaminhos ) :- solucoes(NovoCaminho,
    proximo_nop( Operadoras,Caminho,NovoCaminho ),ExpCaminhos).

proximo_nop( Operadoras,[Nodo|Caminho]/Custo/_,[ProxNodo,Nodo|Caminho]/
    NovoCusto/Est ) :- adjacente(Nodo,ProxNodo,_), paragem(ProxNodo,_,_,_,_,_,
    _,Op,_,_,_,_,_), naopertence(Op,Operadoras), \+member(ProxNodo,Caminho),
    NovoCusto is Custo + 1, estima(ProxNodo,Est).
```

Através da Pesquisa A*, verifica-se se a Operadora da paragem inicial não pertence ao conjunto de Operadoras recebido e calcula-se a estima da paragem inicial, de seguida invoca-se a função auxiliar *percursoNOpaestrela* passando a paragem final como o conjunto de Operadoras a excluir e que constrói um caminho inicial com apenas a paragem inicial, inicialmente esta função auxiliar vai obter o melhor caminho dos vários caminhos que possui e consequentemente vai comparar a cabeça da lista do melhor caminho se corresponde à paragem final tal como a Operadora associada à cabeça não pertence ao conjunto de Operadoras, caso falhe vai obter o melhor caminho dos vários calculados, selecciona esse através de um predicado auxiliar *seleciona*, de seguida faz a sua expansão através do predicado *expande_nop*, junta esse caminho gerando novos caminhos e faz a chamada recursiva com isso. Para obter o melhor é utilizado o predicado *obtem_melhor* que através da comparação dos custos associados a cada caminho somado com a estimação faz a selecção do melhor caminho. O predicado *expande_nop* vai procurar todos os caminhos que cumprem o predicado *proximo_nop* que verifica se duas paragens são adjacentes, que a Operadora da próxima paragem não pertence ao conjunto de Operadoras recebido, se a próxima paragem não pertence ao caminho calculado e é associado o custo somando 1 unidade de forma a obter o caminho mais curto e também a estimativa do próxima paragem. Após o *percursoNOpaestrela* ser efectuado o caminho obtido é invertido.

```
| ?- percursoNaoOperadorasaestrela(318,273,["LT"],A).
no
| ?- percursoNaoOperadorasaestrela(318,273,["Vimeca"],AA).
AA = [318,327,326,273]/4 ?
yes
```

Figura 3: Exclusão de apenas a operadora de transporte "LT"e "Vimeca", respectivamente, para um percurso entre a paragem 318 e a 273.

3.3.4 Identificação da paragem com o maior número de carreiras num determinado percurso

De forma a desenvolver este caso considerou-se uma adaptação da pesquisa em profundidade aqui com predicado designado *maiorCarreiras*. Este predicado recebe:

- um percurso, a paragem com maior número de carreiras e o tamanho das carreiras

```
getCarreira( GID,C ) :- solucoes( Carreira,paragem(GID,Latitude,Longitude,
    EstadoConservacao,TipoAbrigo,AbrigoPublicidade,Operadora,Carreira,
    CodigoRua,NomeRua,Freguesia),C ).

maiorCarreiras( [Paragem],PP,TT ) :- PP is Paragem, getCarreira( Paragem,C )
    , simplifica( C,L ), comprimento( L,TT ).
maiorCarreiras( [Paragem|Percurso],P,TAM ) :- getCarreira( Paragem,C ),
    simplifica(C,L), comprimento(L,T), maiorCarreiras( Percurso,PP,TT ), (T
    >= TT -> TAM is T, P is Paragem ; TAM is TT, P is PP).
```

Neste predicado, se o percurso possuir apenas uma paragem como é óbvio a paragem com maior número de carreiras é essa, de seguida obtém-se através do predicado *getCarreira* a lista de carreiras dessa paragem fornecendo-lhe o identificador, posteriormente *simplifica* a lista obtida e é obtido o comprimento da lista de carreiras através do predicado *comprimento*. Contudo se o percurso for maior é efectuado o mesmo processo mas efectua-se uma chamada recursiva para o resto do percurso, no final é efectuado a comparação entre o tamanho de carreiras da cabeça e o obtido na chamada recursiva verificando qual o maior de forma a retornar este valor.

```
| ?- maiorCarreiras([183,791,595,182,499,593,181,180,594],PAR,TAM).
PAR = 595,
TAM = 7 ?
yes
```

Figura 4: Identificação da paragem com mais carreiras no percurso 183 a 594.

3.3.5 Escolha do menor percurso (usando critério menor número de paragens)

De forma a desenvolver este caso considerou-se a estratégias de pesquisa não informada, aqui cada predicado designado *percursoMenor* com o sufixo corresponde ao tipo de pesquisa, ou seja, *percursoMenordf* corresponde à escolha do menor percurso através de uma Pesquisa Primeiro em Profundidade (Depth-First Search) e *percursoMenorbf* corresponde à escolha do menor percurso através de uma Pesquisa Primeiro em Largura (Breadth-First Search). Em relação à pesquisa informada como o cálculo do custo é somado uma unidade no cálculo do trajecto entre dois pontos (já explicado anteriormente) este presente funcionalidade já foi desenvolvida previamente.

```
percursoMenordf( A,B,P ) :- solucoes( (S,C),(percursodf(A,B,S),comprimento(S
    ,C)),L ), minimo(L,P).

percursoMenorbf( A,B,P ) :- solucoes( (S,C),(percursobf(A,B,S),comprimento(S
    ,C)),L ), minimo(L,P).
```

Nestes predicados é utilizado o predicado auxiliar *solucoes* e é caracterizado da seguinte forma: **solucoes(X,Y,Z) :- findall(X,Y,Z).**. Este predicado procura todas as ocorrências de "X" que satisfaçam "Y" e os resultados são guardados numa lista "Z". Assim, procurou-se todos os percursos mediante o tipo de pesquisa entre "A"e "B"e calculou-se o comprimento desse percurso, guardando como par na lista "L". Posteriormente, calculou-se o menor par dessa lista através do predicado *minimo*.

```
| ?- percursoMenorbf(318,273,R).
R = ([318,327,326,273],4) ?
yes
```

Figura 5: Escolha do menor percurso, usando critério menor número de paragens, entre 318 e 273.

3.3.6 Escolha do menor percurso (usando critério da distância)

De forma a desenvolver este caso considerou-se todas as estratégias de pesquisa, aqui cada predicado designado *percursoRapido* com o sufixo corresponde ao tipo de pesquisa, ou seja, *percursoRapidodf* corresponde à escolha do menor percurso através de uma Pesquisa Primeiro em Profundidade (Depth-First Search), *percursoRapidobf* corresponde à escolha do menor percurso através de uma Pesquisa Primeiro em Largura (Breadth-First Search) e assim sucessivamente. Em relação à pesquisa não informada efectuou-se o mesmo processo de implementação para a escolha do menor percurso,

usando critério menor número de paragens. Apenas alterou-se o predicado que calcula o comprimento do percurso para o predicado *distanciaPercurso* que permite calcular a distância total de um determinado percurso.

```
getLat( GID,L ) :- solucoes( Latitude,paragem(GID,Latitude,Longitude,
    EstadoConservacao,TipoAbrigo,AbrigoPublicidade,Operadora,Carreira,
    CodigoRua,NomeRua,Freguesia),LL ), simples(LL,L).

getLon( GID,L ) :- solucoes( Longitude,paragem(GID,Latitude,Longitude,
    EstadoConservacao,TipoAbrigo,AbrigoPublicidade,Operadora,Carreira,
    CodigoRua,NomeRua,Freguesia),LL ), simples(LL,L).

getDistancia( A,B,D ) :- getLat(A,LAT1), getLat(B,LAT2), getLon(A,LON1),
    getLon(B,LON2), D is sqrt((LAT2-LAT1)*(LAT2-LAT1) + (LON2-LON1)*(LON2-
    LON1)).

percursoRapidogulosa( Inicial,Final,Caminho/Custo ) :- estima(Inicial,Estima
    ), rapidogulosa( Final,[[Inicial]/0/Estima],InvCaminho/Custo/_ ),
    inverso(InvCaminho,Caminho).

rapidogulosa( Final,Caminhos,Caminho ) :- obtem_melhor_g(Caminhos,Caminho),
    Caminho=[Paragem|_]/_/_, Final is Paragem.
rapidogulosa( Final,Caminhos,SolucaoCaminho ) :- obtem_melhor_g(Caminhos,
    MelhorCaminho), seleciona(MelhorCaminho,Caminhos,OutrosCaminhos),
    expande_rapido_g( MelhorCaminho,ExpCaminhos ), junta(OutrosCaminhos,
    ExpCaminhos,NovoCaminhos), rapidogulosa( Final,NovoCaminhos,
    SolucaoCaminho ).

expande_rapido_g( Caminho,ExpCaminhos ) :- solucoes(NovoCaminho,rapido(
    Caminho,NovoCaminho),ExpCaminhos).

rapido( [Nodo|Caminho]/Custo/_,[ProxNodo,Nodo|Caminho]/NovoCusto/Est ) :-
    adjacente(Nodo,ProxNodo,_), \+member(ProxNodo,Caminho), getDistancia(
    Nodo,ProxNodo,Dist), NovoCusto is Custo + Dist, estima(ProxNodo,Est).
```

Para o cálculo da distância entre duas paragens retira-se da base de conhecimento a latitude e longitude das duas através de predicados desenvolvidos para esse efeito. Posteriormente, é realizada a distância euclidiana entre dois pontos. Através da Pesquisa Gulosa, calcula-se a estima da paragem inicial, de seguida invoca-se a função auxiliar *rapidogulosa* passando a paragem final e que constrói um caminho inicial com apenas a paragem inicial, inicialmente esta função auxiliar vai obter o melhor caminho dos vários caminhos que possui e consequentemente vai comparar a cabeça da lista do melhor caminho se corresponde à paragem final, caso falhe vai obter o melhor caminho dos vários calculados, selecciona esse através de um predicado auxiliar *seleciona*, de seguida faz a sua expansão através do predicado *expande_rapido_g*, junta esse caminho gerando novos caminhos e faz a chamada recursiva com isso. Para obter o melhor é utilizado o predicado *obtem_melhor_g* que através da comparação das estimações faz a selecção do melhor caminho. O predicado *expande_gulosa* vai procurar todos os caminhos que cumprem o predicado *rapido* que verifica se duas paragens são adjacentes, se o próximo paragem não pertence ao caminho calculado e é associado o custo somando a distância

entre esse nodo e o próximo nodo forma a obter o caminho mais curto e também a estimativa do próxima paragem. Após o *rapidogulosa* ser efectuado o caminho obtido é invertido. Nesta pesquisa a alteração do custo não provoca nenhum impacto porque é guiado pelas estimativas.

```
| ?- percursoRapidoaestrela(318,273,R).
R = [318,299,273]/434.9004152112939 ?
yes
| ?- percursoRapidogulosa(318,273,R).
R = [318,327,326,273]/1265.3776279210856 ?
yes
```

Figura 6: Escolha do percurso mais rápido, usando critério da distância, entre 318 e 273.

Como se pode observar na pesquisa gulosa a alteração do custo não provoca nenhum impacto porque é guiado pelas estimativas contudo na pesquisa A* esse efeito é notado com um percurso mais rápido como pretendido.

3.3.7 Escolha do percurso que passe apenas por abrigos com publicidade

De forma a desenvolver este caso considerou-se todas as estratégias de pesquisa, aqui cada predicado designado *percursoPublicidade* com o sufixo corresponde ao tipo de pesquisa, ou seja, *percursoRapidodf* corresponde à escolha do percurso que passe apenas por abrigos com publicidade através de uma Pesquisa Primeiro em Profundidade (Depth-First Search), *percursoRapidobf* corresponde à escolha do percurso que passe apenas por abrigos com publicidade através de uma Pesquisa Primeiro em Largura (Breadth-First Search) e assim sucessivamente.

Todos estes predicados recebem três argumentos:

- paragem inicial, paragem final, o percurso calculado

```
percursoPublicidadedf( Inicial,Final,[Inicial|Trajeto] ) :- paragem(Inicial,
    _,_,_,_,"Yes",_,_,_,_,_), percursoPubdf( Final,Inicial,Trajeto ).

percursoPubdf( Final,Paragem,[] ) :- paragem(Paragem,_,_,_,_,"Yes",_,_,_,_,_
    ), Final is Paragem.
percursoPubdf( Final,Paragem,[ProxPar|Trajeto] ) :- adjacente(Paragem,
    ProxPar,_), paragem(ProxPar,_,_,_,_,"Yes",_,_,_,_,_), percursoPubdf(
    Final,ProxPar,Trajeto ).
```

Através da Pesquisa Primeiro em Profundidade, verifica-se inicialmente se a paragem Inicial tem no seu campo Abrigo com Publicidade igual a "Yes", caso seja é adicionada a paragem inicial ao percurso e recorre-se à função auxiliar *percursoPubdf* onde indica qual é a paragem final, esta por sua vez

verifica inicialmente se a paragem que está a calcular de momento é a paragem final como também se esta possui o campo Abrigo com Publicidade igual a "Yes"e caso seja indica o fim do percurso através da lista vazia contudo caso não se verifica a condição vai procurar uma paragem adjacente, verificando se existe uma ligação, em que a paragem actual seja a origem, efectua o mesmo teste à paragem que encontrou caso passe adiciona ao trajecto calculado até de momento e posteriormente invoca a chamada recursiva à função auxiliar.

```
| ?- percursoPublicidadedf(505,577,R).
R = [505,501,577] ?
yes
```

Figura 7: Escolha do percurso com publicidade, entre 505 e 577.

3.3.8 Escolha do percurso que passe apenas por paragens abrigadas

De forma a desenvolver este caso considerou-se todas as estratégias de pesquisa, aqui cada predicado designado *percursoAbrigado* com o sufixo corresponde ao tipo de pesquisa, ou seja, *percursoAbrigadoodf* corresponde à escolha do percurso que passe apenas por paragens abrigadas através de uma Pesquisa Primeiro em Profundidade (Depth-First Search), *percursoAbrigadobf* corresponde à escolha do percurso que passe apenas por paragens abrigadas através de uma Pesquisa Primeiro em Largura (Breadth-First Search) e assim sucessivamente. Todos estes predicados recebem três argumentos:

- paragem inicial, paragem final, o percurso calculado

Em relação a esta funcionalidade efectuou-se o mesmo processo de implementação para escolha do percurso com publicidade, apenas alterou-se a verificação do campo Abrigo com Publicidade para o campo Tipo de Abrigo ser igual "Fechado dos Lados"de forma a ser abrigado como solicitado.

```
| ?- percursoAbrigadogulosa(505,577,R).
R = [505,501,577]/3 ?
yes
```

Figura 8: Escolha do percurso abrigado, entre 505 e 577.

3.3.9 Escolha de um ou mais pontos intermédios por onde o percurso deve passar

De forma a desenvolver este caso considerou-se apenas estratégias de pesquisa não informada, aqui cada predicado designado *percursoIntermedio* com o sufixo corresponde ao tipo de pesquisa, ou seja, *percursoIntermediodf* corresponde à escolha de um ou mais pontos intermédios por onde o percurso deve passar através de uma Pesquisa Primeiro em Profundidade (Depth-First Search) e *percursoIntermediobf* corresponde à escolha e um ou mais pontos intermédios por onde o percurso deve passar através de uma Pesquisa Primeiro em Largura (Breadth-First Search). Estes predicados recebem três argumentos:

- paragem inicial, paragem final, o percurso por onde deve passar e o percurso calculado

```
percursoIntermediodf( Inicial,Final,[PI|T],P ) :- percursodf(Inicial,PI,A),
    percursoIntdf([PI|T],B), junta(A,B,C), ultimo([PI|T],D), percursodf(D,
    Final,E), junta(C,E,F), apagaRep(F,P).

percursoIntdf([H],[H]).
percursoIntdf([A,B],L) :- percursodf(A,B,L).
percursoIntdf([A,B|T],L) :- percursodf(A,B,LL), percursoIntdf([B|T],R),
    junta(LL,R,L).
```

Através da Pesquisa Primeiro em Profundidade calculou-se o percurso entre a paragem inicial e a primeira paragem do percurso por onde deve passar, de seguida calcula-se através do predicado auxiliar *percursoIntdf* o percurso que liga que os pontos intermédios recebidos através do predicado que fornece o trajecto entre dois pontos mediante o tipo de pesquisa em causa, no final esta função junta tudo num único percurso que é adicionado ao calculado inicialmente resultando num percurso único, de seguida é retirado o última paragem desse e é calculado o trajecto até ao final e mais uma vez é concatenado tudo num percurso.

```
| ?- percursoIntermediodf(183,594,[595,499],R).
R = [183,791,595,182,499,593,181,180,594] ?
yes
```

Figura 9: Escolha de pontos intermédios (595,499) por onde o percurso entre 183 e 594 deve passar.

3.4 Adição e remoção de conhecimento

Tal como referido no enunciado deste trabalho o conhecimento a tratar poderá ser estendido. Logo, a inserção e remoção de conhecimento é realizada à custa de invariantes que especificam um conjunto de restrições que devem ser verdadeiras após uma adição/remoção de conhecimento. A inserção de predicados pode ser vista como uma evolução do sistema em termos de conhecimento. De um modo análogo, uma remoção de um predicado pode ser vista como uma involução do sistema. De notar que, tanto as operações de inserção como de remoção são sempre efectuadas. No entanto, após cada uma dessas operações é verificada a consistência do sistema com recurso aos invariantes. Caso alguma operação provoque uma anomalia no sistema, esta perde o seu efeito e o sistema volta ao estado anterior à operação em questão.

3.4.1 Evolução do sistema

O processo de evolução do sistema é caracterizado por um aumento de conhecimento no sistema. Este processo é descrito por:

```
evolucao( Termo ) :- solucoes( Invariante,+Termo::Invariante,Lista ),
    insercao( Termo ), teste(Lista).

insercao( Termo ) :- assert(Termo).
insercao( Termo ) :- retract(Termo),!,fail.
```

Em que o predicado insercao caracteriza-se pela inserção de conhecimento no sistema (à custa do predicado assert) e o predicado teste pelo teste aos invariantes relativos a esse "Termo" armazenados na lista. De notar que, por conveniência, defini que um invariante relativo a uma inserção de um termo qualquer é precedido pelo símbolo de "+" no nome desse mesmo termo.

3.4.2 Involução do sistema

O processo de involução do sistema é muito semelhante ao processo de evolução e é definido do seguinte modo:

```
involucao( Termo ) :- solucoes( Invariante,-Termo::Invariante,Lista ),
    remocao( Termo ), teste(Lista).

remocao( Termo ) :- retract(Termo).
remocao( Termo ) :- assert(Termo),!,fail.
```

Em que o predicado remocao caracteriza-se pela inserção de conhecimento no sistema (à custa do predicado retract) e o predicado teste pelo teste aos invariantes relativos a esse "Termo" armazenados na lista. De notar que, por conveniência, definiu-se que um invariante relativo a uma remoção de um termo qualquer é precedido pelo símbolo de "-" no nome desse mesmo termo.

3.4.3 Paragem

A inserção de conhecimento relativa a uma paragem deve respeitar os seguintes critérios: o identificador de cada paragem de ser único, paragens com identificadores diferentes possuem diferente informação. Por sua vez, a remoção de uma paragem não é permitida quando ainda existe ligações relativos à mesma:

```
+paragem(GID,Latitude,Longitude,EstadoConservacao,TipoAbrigo,
    AbrigoPublicidade,Operadora,Carreira,CodigoRua,NomeRua,Freguesia) :: (
    solucoes(GID,paragem(GID,_,_,_,_,_,_,_,_,_,_,_),R), comprimento(R,1)).

+paragem(GID,Latitude,Longitude,EstadoConservacao,TipoAbrigo,
    AbrigoPublicidade,Operadora,Carreira,CodigoRua,NomeRua,Freguesia) :: (
    solucoes((Latitude,Longitude,EstadoConservacao,TipoAbrigo,
    AbrigoPublicidade,Operadora,Carreira,CodigoRua,NomeRua,Freguesia),
    paragem(_,Latitude,Longitude,EstadoConservacao,TipoAbrigo,
    AbrigoPublicidade,Operadora,Carreira,CodigoRua,NomeRua,Freguesia),R),
    comprimento(R,1)).

-paragem(GID,_,_,_,_,_,_,_,_,_,_,_) :: (solucoes(GID,ligacao(GID,_,_),R),
    comprimento(R,0), solucoes(GID,ligacao(_,GID,_),S), comprimento(S,0)).
```

3.4.4 Ligação

A inserção de conhecimento relativa a uma ligação deve respeitar o seguinte critério o identificador das paragens associadas a esta ligação encontram-se registados na base de conhecimento.

```
+ligacao(GID1,GID2,_) :: (solucoes(GID1, paragem(GID1,_,_,_,_,_,_,_,_,_,_,_),R
    ), comprimento(R,1), solucoes(GID2,paragem(GID2,_,_,_,_,_,_,_,_,_,_,_),S),
     comprimento(S,1)).
```

3.5 Implementação dos predicados de consulta

Após a criação dos predicados que permitem a evolução e a involução de conhecimento do sistema, assim como o desenvolvimento das demais funcionalidades, procedeu-se então à criação de predicados de consulta sobre a estrutura de conhecimento. Estes predicados foram maioritariamente desenvolvidos à custa do predicado *solucoes*.

3.5.1 Registar Paragens e Ligações

Para que seja possível a inserção de nova informação na base de conhecimento usou-se o teorema *evolucao* o qual insere na base de conhecimento de forma controlada:

```
registaParagem( GID,Latitude,Longitude,EstadoConservacao,TipoAbrigo,
    AbrigoPublicidade,Operadora,Carreira,CodigoRua,NomeRua,Freguesia ) :-
    evolucao(paragem(GID,Latitude,Longitude,EstadoConservacao,TipoAbrigo,
    AbrigoPublicidade,Operadora,Carreira,CodigoRua,NomeRua,Freguesia)).

registaLigacao( Origem,Destino,Carreira ) :- evolucao(ligacao(Origem,Destino
    ,Carreira)).
```

```
| ?- registaLigacao(681,459,776).
yes
```

Figura 10: Registo de uma nova ligação entre 581 e 459 pela carreira 776.

3.5.2 Remover Paragens e Ligações

Para que seja possível a remoção de nova informação na base de conhecimento usou-se o teorema *involucao* o qual retira da base de conhecimento de forma controlada:

```
removeParagem( GID ) :- involucao(paragem(GID,_,_,_,_,_,_,_,_,_,_,_)).

removeLigacao( Origem,Destino,Carreira ) :- involucao(ligacao(Origem,Destino
    ,Carreira)).
```

```
| ?- removeLigacao(681,459,776).
yes
```

Figura 11: Remoção da ligação entre 581 e 459 pela carreira 776.

3.5.3 Identificar uma paragem pelo seu identificador

Esta consulta permite obter a informação registada na base de conhecimento referente à paragem fornecendo para isso identificador correspondente. As strings são convertidas no seu código Ascii.

```
paragemGID( GID,R ) :- solucoes((GID,Latitude,Longitude,EstadoConservacao,
    TipoAbrigo,AbrigoPublicidade,Operadora,Carreira,CodigoRua,NomeRua,
    Freguesia), paragem(GID,Latitude,Longitude,EstadoConservacao,TipoAbrigo,
    AbrigoPublicidade,Operadora,Carreira,CodigoRua,NomeRua,Freguesia),R).
```

```
| ?- paragemGID(499,P).
P = [(499,-103758.44,-94393.36,[66,111,109],[70,101,99,104,97,100,111,32,100,111,115,32,76,97,100,111,115],[89,101,115],[86,
105,109,101,99,97],[1],300,[65,118,101,110,105,100,97,32,100,111,115,32,67,97,118,97,108,101,105,114,111,115],[67,97,114,110
,97,120,105,100,101,32,101,32,81,117,101,105,106,97,115])] ?
yes
```

Figura 12: Identificação da paragem com o identificador 499.

3.5.4 Identificar as ligações que uma determinada paragem possui como origem

Neste predicado identifica ligações que uma paragem, fornecendo o identificador, encontra-se envolvida como origem.

```
ligacoes_da_paragemOR( GID,R ) :- solucoes(ligacao(GID,Destino,Carreira),
    ligacao(GID,Destino,Carreira),R).
```

```
| ?- ligacoes_da_paragemOR(499,OR).
OR = [ligacao(499,593,1)] ?
yes
```

Figura 13: Identificação das ligações da paragem 499 é uma origem.

3.5.5 Identificar as ligações que uma determinada paragem possui como destino

Neste predicado identifica ligações que uma paragem, fornecendo o identificador, encontra-se envolvida como destino.

```
ligacoes_da_paragemDE( GID,R ) :- solucoes(ligacao(Origem,GID,Carreira),
    ligacao(Origem,GID,Carreira),R).
```

```
| ?- ligacoes_da_paragemDE(499,DE).
DE = [ligacao(182,499,1)] ?
yes
```

Figura 14: Identificação das ligações da paragem 499 é um destino.

3.5.6 Identificar as ligações que uma determinada carreira como o seu tamanho

Neste predicado identifica todas as ligações que uma determinada carreira, fornecendo para isso o número da carreira, também é calculado o total de ligações.

```
getLigacoes( Carreira,R,C ) :- solucoes(ligacao(Origem,Destino,Carreira),
    ligacao(Origem,Destino,Carreira),R), comprimento(R,C).
```

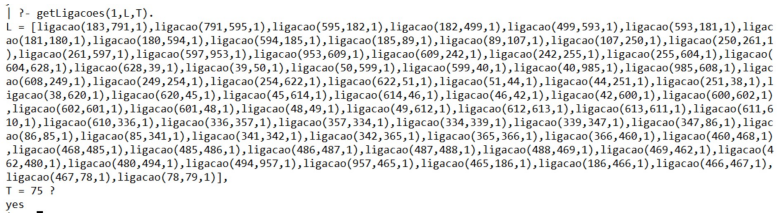

Figura 15: Identificação das ligações da carreira número 1.

3.5.7 Calcular o número total de ligações

Neste predicado identifica todas o total de ligações existem na base de conhecimento.

```
total_ligacoes(R) :- solucoes(C,ligacao(_,_,C),L), comprimento(L,R).
```

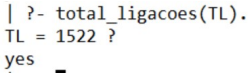

Figura 16: Número total de ligações registadas.

3.6 Resumo e Balanço das Estratégias de Pesquisa Utilizadas

Após ser destacado e explicado o processo de raciocínio e implementação para diversas funcionalidades acima apresentadas. Foi necessário avaliar os desempenhos dos diversos algoritmos de pesquisas utilizados através dos seguintes critérios.

- Completude - garante uma solução;
- Complexidade no Tempo - tempo que demora a encontrar a solução;
- Complexidade no Espaço - memória necessária para efectuar a pesquisa;
- Otimalidade - encontra logo melhor a solução.

É de realçar que o tamanho do espaço de estados possíveis é demasiado grande logo estes algoritmos de pesquisas implementados por vezes vezes acabam por não conseguir lidar com a complexidade do problema em questão.

	Pesquisa Não Informada	
Critério	**Pesquisa Primeiro em Largura**	**Pesquisa Primeiro em Profundidade**
Completude	Sim, quando o número máximo de ligações sucessoras de uma paragem é finito	Não, falha em espaços de profundidade infinita
Complexidade no Tempo	Acaba por ser exponencial	Mau, quando a máxima profundidade do espaço de estados é maior que a profundidade da melhor solução
Complexidade no Espaço	Guarda cada paragem em memória	Linear porque encontra logo a primeira ligação.
Otimalidade	Nem sempre	Não, devolve a 1º solução que encontra

Tabela 1: Tabela Comparativa da Pesquisa Não Informada

	Pesquisa Informada	
Critério	**Pesquisa Gulosa**	**Pesquisa A***
Completude	Por vezes entra em ciclos	Sim
Complexidade no Tempo	Exponencial mas poderá diminuir com uma melhor função heurística	Exponencial
Complexidade no Espaço	Mantém todas as paragens em memória	Mantém todos as paragens em memória
Otimalidade	Não encontra sempre a solução ótima	Sim

Tabela 2: Tabela Comparativa da Pesquisa Informada

4 Conclusão

O presente relatório descreveu, de forma sucinta, o desenvolvimento do sistema de representação de conhecimento e raciocínio que importou os dados relativos ao conjunto das paragens de autocarro existentes no concelho de Oeiras, que teve capacidade para representá-los numa base de conhecimento. que com a capacidade de recomendar transportes públicos.

Após a realização deste trabalho, fiquei consciente das potencialidades que a programação em lógica desenvolvida com PROLOG na medida que permite representar o conhecimento que um agente tem sobre o mundo de uma forma simples e directa, em uma linguagem de alto nível, tornando os programas mais compactos, flexíveis e inteligíveis e como também permite uma programação declarativa apenas basta especificar correctamente o problema que o motor de inferência encarrega-se de descobrir como obter sua solução.

Considero que os principais objectivos foram cumpridos, contudo tenho consciência que poderia ter apresentado melhor os dados e a função heurística poderia ter sido melhor planeada.

Senti que a realização deste trabalho prático consolidou os meus conhecimentos na programação em lógica estendida como também nos métodos de resolução de problemas e de procura.

5 Anexos

(A1) Código do programa desenvolvido em Java para processamento dos *datasets*

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Random;

public class Parser {
    private static ArrayList<Integer> paragens;

    public static void cria_estimativas(File f) throws IOException{
        FileWriter fr = new FileWriter(f, true);
        Random gerador = new Random();
        int i=0;
        fr.write("\n% estima( Paragem,TempoEspera ) \n");

        for (i = 0; i < paragens.size(); i++) {
            fr.write("estima( " + paragens.get(i) + "," + gerador.
                nextInt(300) + " ).\n" );
        }
        fr.close();
    }


    public static void processa_carreiras(File f, ArrayList<String>
        carreiras) throws IOException {
        String line = "";
        String cvsSplitBy = ";";
        FileWriter fr = new FileWriter(f, true);
        fr.write("\n% ligacao( Origem,Destino,Carreira ) \n");

        for (int i = 0; i < carreiras.size(); i++) {
            String carreiraAtual = carreiras.get(i);
            String[] path1 = carreiraAtual.split("\\\\");
            String name = path1[1].substring(0, path1[1].lastIndexOf('
                .'));
            int nameI = Integer.parseInt(name);
            int x = 0;
            int y = 0;
            BufferedReader br = new BufferedReader(new FileReader(
                carreiraAtual));
            String gidAnterior = "";

            while ((line = br.readLine()) != null) {
                String[] adjadencia = line.split(cvsSplitBy);
                if (x > 0) {
                    if (y == 0) {
                        gidAnterior = adjadencia[0];
                    } else {
                        fr.write("ligacao( " + gidAnterior + "," +
                            adjadencia[0] + "," + nameI + " ).\n");
                        gidAnterior = adjadencia[0];
                    }
```

```
                y++;
            }
            x++;
        }
        fr.write("\n");
    }
    fr.close();
}

public static void processa_paragens(File f, String csvFile)
    throws IOException {
    String line = "";
    String cvsSplitBy = ";";
    int i = 0;
    FileWriter fr = new FileWriter(f, true);
    BufferedReader br = new BufferedReader(new FileReader(csvFile)
        );

    while ((line = br.readLine()) != null) {
        String[] paragem = line.split(cvsSplitBy);
        if (paragem.length == 11 && i > 0) {
            fr.write("paragem( " + paragem[0] + "," + paragem[1] +
                 "," + paragem[2] + ",\"" + paragem[3] + "\",\"" +
                 paragem[4] + "\",\"" + paragem[5] + "\",\"" +
                paragem[6] + "\",[" + paragem[7] + "]," + paragem
                [8] + ",\"" + paragem[9] + "\",\"" + paragem[10] +
                 "\").\n");
            paragens.add(Integer.parseInt(paragem[0]));
        }
        if (i == 0) {
            fr.write("%paragem( " + paragem[0] + "," + paragem[1]
                + "," + paragem[2] + "," + paragem[3] + "," +
                paragem[4] + "," + paragem[5] + "," + paragem[6] +
                 "," + paragem[7] + "," + paragem[8] + "," +
                paragem[9] + "," + paragem[10] + " ).\n");
        }
        if (paragem.length == 10) {
            fr.write("paragem( " + paragem[0] + "," + paragem[1] +
                 "," + paragem[2] + ",\"" + paragem[3] + "\",\"" +
                 paragem[4] + "\",\"" + paragem[5] + "\",\"" +
                paragem[6] + "\",[" + paragem[7] + "]," + paragem
                [8] + ",\"" + paragem[9] + "\",
                freguesia_desconhecida).\n");
            paragens.add(Integer.parseInt(paragem[0]));
        }
        i++;
    }
    fr.close();
}

public static void main(String[] args) {
    BufferedReader br = null;
    paragens = new ArrayList<Integer>();
    ArrayList<String> carreiras = new ArrayList<>();
    carreiras.add("csv\\01.csv");
    carreiras.add("csv\\02.csv");
    carreiras.add("csv\\06.csv");
    carreiras.add("csv\\07.csv");
    carreiras.add("csv\\10.csv");
    carreiras.add("csv\\11.csv");
    carreiras.add("csv\\12.csv");
    carreiras.add("csv\\13.csv");
```

```
        carreiras.add("csv\\15.csv");
        carreiras.add("csv\\23.csv");
        carreiras.add("csv\\101.csv");
        carreiras.add("csv\\102.csv");
        carreiras.add("csv\\106.csv");
        carreiras.add("csv\\108.csv");
        carreiras.add("csv\\111.csv");
        carreiras.add("csv\\112.csv");
        carreiras.add("csv\\114.csv");
        carreiras.add("csv\\115.csv");
        carreiras.add("csv\\116.csv");
        carreiras.add("csv\\117.csv");
        carreiras.add("csv\\119.csv");
        carreiras.add("csv\\122.csv");
        carreiras.add("csv\\125.csv");
        carreiras.add("csv\\129.csv");
        carreiras.add("csv\\158.csv");
        carreiras.add("csv\\162.csv");
        carreiras.add("csv\\171.csv");
        carreiras.add("csv\\184.csv");
        carreiras.add("csv\\201.csv");
        carreiras.add("csv\\467.csv");
        carreiras.add("csv\\468.csv");
        carreiras.add("csv\\470.csv");
        carreiras.add("csv\\471.csv");
        carreiras.add("csv\\479.csv");
        carreiras.add("csv\\714.csv");
        carreiras.add("csv\\748.csv");
        carreiras.add("csv\\750.csv");
        carreiras.add("csv\\751.csv");
        carreiras.add("csv\\776.csv");
        try {
            File myObj = new File("base_conhecimento.pl");
            if (myObj.createNewFile()) {
                System.out.println("File created: " + myObj.getName())
                    ;
            } else {
                System.out.println("File already exists.");
            }
            processa_paragens(myObj, "csv\\paragens_processado.csv");
            processa_carreiras(myObj, carreiras);
            cria_estimativas(myObj);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (br != null) {
                try {
                    br.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```