Sistemas de Representação de Conhecimento e Raciocínio

Trabalho Prático Individual

Luís Miguel Ramos (A83930)

5 de Junho de 2020

Mestrado Integrado em Engenharia Informática
Universidade do Minho

Conteúdo

Resumo

Neste exercício prático foi desenvolvido um sistema de representação de conhecimento e raciocínio referente ao universo das paragens de autocarro,recorrendo a algortimos, de modo a encontrar quais as melhor opções para cada trajeto.

1 Introdução

No âmbito da unidade curricular **Sistemas de Representação de Conhecimento e Raciocínio**, resolveu-se o proposto exercício, cujo tema assenta na *Programação em lógica*, recorrendo ao PROLOG, no âmbito de métodos de resoluçao de problemas e no desenvolvimento de algoritmos de pesquisa.

2 Descrição do trabalho e análise de resultados

Tal como foi referido anteriormente, a temática deste trabalho assenta na Programação em Lógica. O trabalho desenvolvido teve como principais focus as seguintes questões:

- Tratamento de dados.
- Representação dos dados na base de conhecimento.
- Criação de algoritmos de pesquisa.
- Criação de um sistema de recomendação, capaz de responder a um conjunto de pedidos do utilizador.

2.1 Dados

O exercício começa pelo tratamento dos dados fornecidos. Neste caso, foram fornecidos ficheiros no formato excel, contendo estes informações sobre as paragens do concelho de Oeiras. Ao todo são 39 ficheiros, que correspondem a 39 carreiras diferentes. Dentro de cada carreira encontra-se um número de paragens variável. De notar, que cada linha do ficheiro corresponde a uma paragem, com a respetiva informação, sendo esta adjacente á paragem da linha seguinte e da linha anterior.

2.2 Passagem dos dados para a base de conhecimento

Sabendo já como estão armazenados os dados, foi necessário passar esses dados para a base de conhecimento. Para isso foi utilizado a linguagem *Python*.

Para isto, foi necessário converter todos os ficheiros excel, em ficheiros csv. Todo esse processo foi feito manualmente, gravando cada ficheiro no formato certo.

Depois foi carregado para o *Python* toda a informação sobre as 39 carreiras existentes.

```
[29]: lista1 = pd.read_csv("lista_adjacencias_paragens_01.csv")
      lista2 = pd.read_csv("lista_adjacencias_paragens_02.csv")
      lista3 = pd.read_csv("lista_adjacencias_paragens_06.csv")
      lista4 = pd.read_csv("lista_adjacencias_paragens_07.csv")
      lista5 = pd.read_csv("lista_adjacencias_paragens_10.csv")
```

Na seguinte figura observa-se um exemplo do formato da infromação, e dos campos que esta contém.

```
lista1.loc[0]
```

```
gid                                183
latitude                        -103678
longitude                      -96590.3
Estado de Conservacao               Bom
Tipo de Abrigo        Fechado dos Lados
Abrigo com Publicidade?             Yes
Operadora                        Vimeca
Carreira                              1
Codigo de Rua                       286
Nome da Rua        Rua Aquilino Ribeiro
Freguesia           Carnaxide e Queijas
```

A infromação foi tratada e modificada de forma adequada, criando assim o predicado que depois será passado para a base de dados. Neste caso, o predicado *paragem* e *adjacente*, pela respetiva ordem.

```
def escreve(lista):
    o = open("pro.pl","a")
    y = 0
    x = -1
    for row in lista.iterrows():
        if y==0:
            a = "paragem("+str(y)+", "+str(lista.loc[y][7])+", "+str(lista.loc[y][0])+", "+str(lista.loc[y][1])+
            ", "+str(lista.loc[y][2])+", "+"'"+str(lista.loc[y][3])+"'"+", "+"'"+str(lista.loc[y][4])+"'"+
            ", "+"'"+str(lista.loc[y][5])+"'"+", "+"'"+str(lista.loc[y][6])+"'"+", "+str(lista.loc[y][8])+
            ", "+"'"+str(lista.loc[y][9])+"'"+", "+"'"+str(lista.loc[y][10])+"'"+").\n"
            o.write(a)
        if x>=0 and (str(lista.loc[y][0])!=str(lista.loc[x][0])):
            a = "paragem("+str(y)+", "+str(lista.loc[y][7])+", "+str(lista.loc[y][0])+", "+str(lista.loc[y][1])+
            ", "+str(lista.loc[y][2])+", "+"'"+str(lista.loc[y][3])+"'"+", "+"'"+str(lista.loc[y][4])+"'"+
            ", "+"'"+str(lista.loc[y][5])+"'"+", "+"'"+str(lista.loc[y][6])+"'"+", "+str(lista.loc[y][8])+
            ", "+"'"+str(lista.loc[y][9])+"'"+", "+"'"+str(lista.loc[y][10])+"'"+").\n"
            o.write(a)
        y = y + 1
        x = x + 1
    o.write("\n")
    o.close()
```

```
def adj(lista):
    o = open("pro.pl","a")
    y = 1
    x = 0
    for row in lista.iloc[1:].iterrows():
        if (str(lista.loc[x][0])!=str(lista.loc[y][0])):
            dist = calculaDist(lista.loc[x][1],lista.loc[x][2],lista.loc[y][1],lista.loc[y][2])
            a = "adjacente(("+str(x)+","+str(lista.loc[x][7])+","+str(lista.loc[x][0])+"),
            ("+str(y)+","+str(lista.loc[y][7])+","+str(lista.loc[y][0])+"), "+str(dist)+").\n"
            b = "adjacente(("+str(y)+","+str(lista.loc[y][7])+","+str(lista.loc[y][0])+"),
            ("+str(x)+","+str(lista.loc[x][7])+","+str(lista.loc[x][0])+"), "+str(dist)+").\n"
            o.write(a)
            o.write(b)
        y = y + 1
        x = x + 1
    o.write("\n")
    o.close()
```

Por fim, dá-se a passagem de essa informação para a base de conhecimento.

```
escreve(lista1)
escreve(lista2)
escreve(lista3)
escreve(lista4)
escreve(lista5)
```

```
adj(lista1)
adj(lista2)
adj(lista3)
adj(lista4)
adj(lista5)
```

De referir, à medida que o projeto foi avançando, foi necessário fazer alterações na estrutura dos predicados, de forma a tornar o projeto mais eficiente, sendo esta referidas no ponto seguinte.

2.3 Base de conhecimento

- **paragem**: IdParagem, IdCarreira, Gid, Latitude, Longitude, Estado de Conservação, Tipo de Abrigo, Abrigo com Publicidade, Operadora, Codigo de Rua, Nome da Rua, Freguesia – {V,F}.

- **adjacente**: IdParagemI, IdCarreiraI, GidI, IdParagemF, IdCarreiraF, GidF, Distancia – {V,F}.

2.3.1 Paragem

Uma paragem é caracterizada por um *IdParagem*, sendo este único em cada carreira; um *IdCarreira*; um *Gid*, sendo este único em cada carreira, ou seja, não há paragens repetidas na mesma carreira; uma *Latitude* e uma *Longitude* que representa a posição da paragem; um *Estado de Conversação* podendo este ser 'Bom', 'Razoavel' ou 'Mau'; um *Tipo de Abrigo*, podendo este ser 'Sem Abrigo','Fechado dos Lados' ou 'Aberto dos Lados'; um *Abrigo com Publicidade* podendo este ser 'Yes' ou 'No'; uma *Operadora* podendo esta ser 'LT', 'Vimeca' ou 'SCoTTURB'; um *Codigo de Rua*; um *Nome de Rua*; uma *Freguesia*.

De referir que o *IdParagem* foi adicionado mais tarde e tem como principal função ajudar na otimização da pesquisa, ou seja, no caso de encontrar uma carreira comum entre o ponto inicial e o ponto final, através deste *IdParagem* sabemos o caminho diretamente, como irei explicar mais à frente neste relatório. Quanto à ordem da informação, esta foi colocada de modo a ser fácil de manejar e aceder a certas posições.

```
paragem(0, 1, 183, -103678.36, -96590.26, 'Bom', 'Fechado dos Lados', 'Yes', 'Vimeca', 286, 'Rua Aquilino Ribeiro', 'Carnaxide e Queijas').
paragem(1, 1, 791, -103705.46, -96673.6, 'Bom', 'Aberto dos Lados', 'Yes', 'Vimeca', 286, 'Rua Aquilino Ribeiro', 'Carnaxide e Queijas').
paragem(2, 1, 595, -103725.69, -95975.2, 'Bom', 'Fechado dos Lados', 'Yes', 'Vimeca', 354, 'Rua Manuel Teixeira Gomes', 'Carnaxide e Queijas').
paragem(3, 1, 182, -103746.76, -96396.66, 'Bom', 'Fechado dos Lados', 'Yes', 'SCoTTURB', 286, 'Rua Aquilino Ribeiro', 'Carnaxide e Queijas').
paragem(4, 1, 499, -103758.44, -94393.36, 'Bom', 'Fechado dos Lados', 'Yes', 'Vimeca', 300, 'Avenida dos Cavaleiros', 'Carnaxide e Queijas').
```

2.3.2 Adjacente

Um adjacente é caracterizado por um conjunto de um *IdParagemI*, um *IdCarreiraI* e um *GidI*, que corresponde a informação do ponto inicial; um conjunto de um *IdParagemF*, um *IdCarreiraF* e um *GidF*, que corresponde a informação do ponto final; uma *Distancia* que representa a distância entre as duas paragens.

De notar que ao conter o id da paragem e a sua carreira, permite determinar muito facilmente qual o adjacente, nessa mesma carreira, com o id anterior ou com id seguinte. Quanto à distância, seria já a pensar na pesquisa por menor distância.

```
adjacente((0,1,183), (1,1,791), 87.63541293336934).
adjacente((1,1,791), (0,1,183), 87.63541293336934).
adjacente((1,1,791), (2,1,595), 698.6929317661744).
adjacente((2,1,595), (1,1,791), 698.6929317661744).
adjacente((2,1,595), (3,1,182), 421.9863463431075).
adjacente((3,1,182), (2,1,595), 421.9863463431075).
adjacente((3,1,182), (4,1,499), 2003.3340491291042).
adjacente((4,1,499), (3,1,182), 2003.3340491291042).
adjacente((4,1,499), (5,1,593), 245.01549440800457).
adjacente((5,1,593), (4,1,499), 245.01549440800457).
```

2.4 Algoritmos de pesquisa

2.4.1 Algoritmos Não Informados

O primeiro algoritmo desenvolvido segue-se pela lógica de busca em profundidade, sendo este bastante intuitivo. Este escolhe um adjacente, verificar se já foi visitado. Caso nao tenha sido, adiciona esse à lista de visitados e esse passa a ser o ponto inicial. O caso de paragem é quando o inicial é igual ao final.

```
percurso1(InicioPercurso,FimPercurso):-
    caminho1(InicioPercurso,FimPercurso,[InicioPercurso], Percurso),
    write(Percurso),length(Percurso,R),nl,
    write(R),write(" paragens").
caminho1(Gid, Gid, _, [Gid]).
caminho1(InicioPercurso, FimPercurso, ParagensVisitadas, [InicioPercurso | Percurso]) :-
    adjacente((_,_,InicioPercurso),(_,_,Seguinte),_),
    \+memberchk(Seguinte, ParagensVisitadas),
    caminho1(Seguinte, FimPercurso, [Seguinte|ParagensVisitadas], Percurso).
```

O segundo algoritmo desenvolvido segue-se pela mesma lógica de busca em profundidade, desta vez, do ponto final para o inicial. Este escolhe um adjacente ao ponto final, verificar se já foi visitado. Caso nao tenha sido, adiciona esse à lista de visitados e esse passa a ser o ponto final da nova iteração. Caso ja tenho sido visitado, o caminho volta à paragem anterior, e procura um novo adjacente. O caso de paragem é quando o inicial é igual ao final.

```
percurso2(InicioPercurso,FimPercurso):-
    caminho2(InicioPercurso,[FimPercurso],[FimPercurso],Percurso),
    write(Percurso),length(Percurso,S),nl,
    write(S),write(" paragens").
caminho2(Gid, [Gid|Percurso], _, [Gid|Percurso]).
caminho2(InicioPercurso, [FimPercurso|Percurso], ParagensVisitadas, PercursoFinal) :-
    adjacente((_,_,Anterior),(_,_,FimPercurso),_),
    \+memberchk(Anterior, ParagensVisitadas),
    caminho2(InicioPercurso, [Anterior,FimPercurso|Percurso], [Anterior|ParagensVisitadas], PercursoFinal).
caminho2(InicioPercurso, [T,FimPercurso|Percurso], ParagensVisitadas,PercursoFinal) :-
    caminho2(InicioPercurso,[FimPercurso|Percurso],ParagensVisitadas,PercursoFinal).
```

O terceiro algoritmo desenvolvido segue-se pela lógica de busca em largura. Para cada iteração, verifica se o ponto inicial é igual ao final, caso nao seja, vai buscar todos os adjacentes ao ponto inicial. Para cada um desses adjacentes vai repetir o processo. De referir também que a cada iteração verifica se já chegou ao caminho final. O caso de paragem é quando encontra o caminho todo, sendo necessário depois inverte-lo.

```
percurso3( InicioPercurso, FimPercurso):-
    caminho3( FimPercurso, [[InicioPercurso]], Percurso),
    write(Percurso),length(Percurso,S),nl,
    write(S),write(" paragens").
caminho3(FimPercurso, [[FimPercurso|ParagensVisitadas]|_], Percurso):-
    inverte([FimPercurso|ParagensVisitadas], Percurso).
caminho3( FimPercurso, [ParagensVisitadas|Resto], Percurso) :-
    caminho3(FimPercurso, Resto, Percurso).
caminho3( FimPercurso, [ParagensVisitadas|Resto], Percurso) :-
    ParagensVisitadas = [InicioPercurso|_],
    InicioPercurso \== FimPercurso,
    findall(Seguinte,(adjacente((_,_,InicioPercurso),(_,_,Seguinte),_),\+ memberchk(Seguinte, ParagensVisitadas)),[T|Extend]),
    maplist2( consed(ParagensVisitadas), [T|Extend], VisitadosExtended),
    append2( Resto, VisitadosExtended, UpdatedLista),
    caminho3( FimPercurso, UpdatedLista, Percurso ).
```

Importante referir que estes algoritmos não funcionam para todos os casos. Por esse motivo, deu-se a criação de algoritmos informados e melhorados para a resolução dos pedidos do utilizador.

2.5 Sistema de recomendação

2.5.1 Query1

A query1 tem como objetivo calcular um trajeto entre dois pontos.

```
callquery1(InicioPercurso,FimPercurso):-
    paragem(_,_,FimPercurso,_,_,_,_,_,_,_,_),  % se a paragem final nao existir acaba logo
    query1(InicioPercurso,FimPercurso,[InicioPercurso], Percurso),
    write("\n"),
    write(Percurso),
    length(Percurso,R),
    write("\n\nO percurso contem "),
    write(R),
    write(" paragens!").

query1(FimPercurso, FimPercurso, _, [FimPercurso]).
query1(InicioPercurso,  FimPercurso, [InicioPercurso], []) :-
    paragem(_,C, FimPercurso,_,_,_,_,_,_,_,_),
    verificaIsolada(C,InicioPercurso), % ve se a paragem final é isolada
    \+paragem(_,C, InicioPercurso,_,_,_,_,_,_,_,_). % se nao pertence a mesma carreira acaba
query1(InicioPercurso,  FimPercurso, ParagensVisitadas, [InicioPercurso | Percurso]) :-
    verificaCarreira(InicioPercurso, FimPercurso,R), % se tem carreira comum
    length(R,H), H>0, buscaValor(R,CarreiraComum),  % pega na carreira
    buscaId(CarreiraComum,InicioPercurso, FimPercurso,Id), % calcula o id
    adjacente((_,CarreiraComum,InicioPercurso),(Id,CarreiraComum,X),_); % busca o adjacente com respetivo id
    adjacente((_,_,InicioPercurso),(_,_,X),_),
    \+memberchk(X, ParagensVisitadas),
    query1(X,  FimPercurso, [X|ParagensVisitadas], Percurso).
```

Para esta query, a resolução foi baseada no primeiro algoritmo de pesquisa desenvolvido. Este foi melhorado, acrescentando alguns pormenores, como é o caso de verificar se a paragem final existe, se não é isolada, e caso seja isolada, se a primeira paragem pertence a essa carreira, caso contrário termina logo pois não é possível obter um caminho. Outra das melhorias foi o caso de verificar se , em cada iteração, se a paragem inicial contém alguma carreira em comum com a paragem final, caso isto se verifique, então o caminho é direto, sendo calculado assim o id do proximo adjacente a ser escolhido, até que este chegue ao seu objetivo final.

2.5.2 Query2

A query2 tem como objetivo selecionar apenas algumas das operadoras de transporte para um determinado percurso.

```
callquery2(InicioPercurso,FimPercurso, ListaOperadoras):-
    paragem(_,_,FimPercurso,_,_,_,_,_,Operadora,_,_,_),  % se a paragem final nao existir acaba logo
    memberchk(Operadora, ListaOperadoras),
    query2(InicioPercurso,FimPercurso,[InicioPercurso], Percurso,  ListaOperadoras),
    write("\n"),
    write(Percurso),
    length(Percurso,R),
    write("\n\nO percurso contem "),
    write(R),
    write(" paragens!").

query2(FimPercurso, FimPercurso, _, [FimPercurso],_).
query2(InicioPercurso,  FimPercurso, [InicioPercurso], [],_) :-
    paragem(_,C, FimPercurso,_,_,_,_,_,_,_,_,_),
    verificaIsolada(C,InicioPercurso), % ve se a paragem final é isolada
    \+paragem(_,C, InicioPercurso,_,_,_,_,_,_,_,_,_). % se nao pertence a mesma carreira acaba
query2(InicioPercurso,  FimPercurso, ParagensVisitadas, [InicioPercurso | Percurso], ListaOperadoras) :-
    verificaCarreira(InicioPercurso, FimPercurso,R),
    (length(R,H), H>0, buscaValor(R,V), buscaId(V,InicioPercurso, FimPercurso,T) ,adjacente((_,V,InicioPercurso),(T,V,X),_);
    adjacente((_,_,InicioPercurso),(_,_,X),_)),
    paragem(_,_,X,_,_,_,_,_,Operadora,_,_,_),
    memberchk(Operadora, ListaOperadoras), %verifica se operadora pertence à lista
    \+memberchk(X, ParagensVisitadas),
    query2(X,  FimPercurso, [X|ParagensVisitadas], Percurso,  ListaOperadoras).
```

Para esta query o raciocinio é bastante parecido à query anterior, com a ligeira diferença que esta contem uma verificação para garantir que a paragem tem uma operadora que pertença à lista de operadoras permitidas.

2.5.3 Query3

A query3 tem como objetivo excluir um ou mais operadores de transporte para o percurso.

```
callquery3(InicioPercurso,FimPercurso, ListaOperadoras):-
    paragem(_,_,FimPercurso,_,_,_,_,_,Operadora,_,_,_),  % se a paragem final nao existir acaba logo
    \+memberchk(Operadora, ListaOperadoras),
    query3(InicioPercurso,FimPercurso,[InicioPercurso], Percurso,  ListaOperadoras),
    write("\n"),
    write(Percurso),
    length(Percurso,R),
    write("\n\nO percurso contem "),
    write(R),
    write(" paragens!").

query3(FimPercurso, FimPercurso, _, [FimPercurso],_).
query3(InicioPercurso,  FimPercurso, [InicioPercurso], [],_) :-
    paragem(_,C, FimPercurso,_,_,_,_,_,_,_,_),
    verificaIsolada(C,InicioPercurso), % ve se a paragem final é isolada
    \+paragem(_,C, InicioPercurso,_,_,_,_,_,_,_,_). % se nao pertence a mesma carreira acaba
query3(InicioPercurso,  FimPercurso, ParagensVisitadas, [InicioPercurso | Percurso], ListaOperadoras) :-
    verificaCarreira(InicioPercurso, FimPercurso,R),
    (length(R,H), H>0, buscaValor(R,V), buscaId(V,InicioPercurso, FimPercurso,T) ,adjacente((_,V,InicioPercurso),(T,V,X),_);
    adjacente((_,_,InicioPercurso),(_,_,X),_)),
    paragem(_,_,X,_,_,_,_,_,Operadora,_,_,_),
    \+memberchk(Operadora, ListaOperadoras), % verifica que operadora nao pertence à lista
    \+memberchk(X, ParagensVisitadas),
    query3(X,  FimPercurso, [X|ParagensVisitadas], Percurso,  ListaOperadoras).
```

Para esta query o raciocinio é idêntico à query anterior, com a diferença que desta vez, verificamos que a operadora da paragem não pertence à lista de operadoras.

2.5.4 Query4

A query4 tem como objetivo identificar quais as paragens com o maior número de carreiras num determinado percurso.

```
callquery4(Percurso, NumeroParagens) :-
    numerocarreiras(Percurso,PFinal),
    sort(PFinal,Top),
    inverte(Top,FinalTop),
    showTop(FinalTop,NumeroParagens).

numerocarreiras([H],[(R,H)]):- ncarreiras(H,R).
numerocarreiras([H|T],[(R,H)|P]):-
    ncarreiras(H,R),
    numerocarreiras(T,P).

ncarreiras(Gid,R) :-
    findall(Carreira,paragem(_,Carreira,Gid,_,_,_,_,_,_,_,_,_),L),
    sort(L,T),
    length(T,R).

showTop(_,0).
showTop([],_).
showTop([(R,T)|Y],N) :-
    N>0,
    write("A paragem "),
    write(T),
    write(" contem "),
    write(R),
    write(" carreiras!\n"),
    decrementador(N,NAnteiror),
    showTop(Y,NAnteiror).
```

A estatégia para esta query consistiu em procurar para cada paragem, o numero de carreiras dessa mesma paragem e colocar eese valor do tipo (numeroCarreiras,paragem) numa lista. Ao fazer o **sort** dessa lista, ele organiza pelo numeroCarreiras, de forma crescente, sendo necessário inverter. De notar que é possível escolher a quantidade de paragens que se pretende obter.

2.5.5 Query5

A query5 tem como objetivo escolher o menor percurso (usando critério menor número de paragens).

```
callquery5(InicioPercurso, FimPercurso) :-
    findall((NumeroParagens,Percurso), (query1(InicioPercurso,FimPercurso,[InicioPercurso], Percurso),length(Percurso,NumeroParagens)), L),
    sort(L,R),
    buscaValor2(R,J),
    write(J).
```

A estratégia nesta query era recorrer a um findall para determinar todos os caminhos possíveis, e guardar numa lista em conjunto com o numero de paragens. No final o sort organizaria por ordem crescente e o primeiro elemento dessa lista, seria o percurso mais pequeno.

2.5.6 Query6

A query6 tem como objetivo escolher o percurso mais rápido (usando critério da distância).

```
callquery6(InicioPercurso, FimPercurso) :-
    findall((Distancia,Percurso), (query1(InicioPercurso,FimPercurso,[InicioPercurso], Percurso),calculaDistancia(Percurso,Distancia)), L),
    sort(L,R),
    buscaValor2(R,J),
    write(J).

calculaDistancia([X],0).
calculaDistancia([X,Y|T],Distancia+Dist):-
    adjacente(_,_,X),(_,_,Y),Dist),
    calculaDistancia([Y|T],Distancia).
```

A estratégia nesta query era recorrer a um findall para determinar todos os caminhos possíveis, e guardar numa lista em conjunto com a distancia total desse mesmo caminho. No final o sort organizaria por ordem crescente e o primeiro elemento dessa lista, seria o percurso mais curto.

2.5.7 Query7

A query7 tem como objetivo escolher o percurso que passe apenas por abrigos com publicidade.

```
callquery7(InicioPercurso,FimPercurso, Flag):-
    paragem(_,_,FimPercurso,_,_,_,_,Publicidade,_,_,_,_),  % se a paragem final nao existir acaba logo
    memberchk(Publicidade, [Flag]),
    query7(InicioPercurso,FimPercurso,[InicioPercurso], Percurso,  [Flag]),
    write("\n"),
    write(Percurso),
    length(Percurso,R),
    write("\n\nO percurso contem "),
    write(R),
    write(" paragens!").

query7(FimPercurso, FimPercurso, _, [FimPercurso],_).
query7(InicioPercurso,  FimPercurso, [InicioPercurso], [],_) :-
    paragem(_,C, FimPercurso,_,_,_,_,_,_,_,_),
    verificaIsolada(C,InicioPercurso), % ve se a paragem final é isolada
    \+paragem(_,C, InicioPercurso,_,_,_,_,_,_,_,_). % se nao pertence a mesma carreira acaba
query7(InicioPercurso,  FimPercurso, ParagensVisitadas, [InicioPercurso | Percurso], Flag) :-
    verificaCarreira(InicioPercurso, FimPercurso,R),
    (length(R,H), H>0, buscaValor(R,V), buscaId(V,InicioPercurso, FimPercurso,Id) ,adjacente((_,V,InicioPercurso),(Id,V,Proximo),_);
    adjacente((_,_,InicioPercurso),(_,_,Proximo),_)),
    paragem(_,_,Proximo,_,_,_,_,Publicidade,_,_,_,_),
    memberchk(Publicidade, Flag),   %verifica se tem publicidade
    \+memberchk(Proximo, ParagensVisitadas),
    query7(Proximo,  FimPercurso, [Proximo|ParagensVisitadas], Percurso,  Flag).
```

Para esta query o raciocinio é parecido à query3, com a diferença que desta vez, verificamos se a paragem contem, ou não, publicidade. Para isso, o utilizador insere uma *flag* sendo esta **'Yes'** ou **'No'**, simbolizando se este quer percurso apenas com publicidade ou sem publicidade, respetivamente.

2.5.8 Query8

A query8 tem como objetivo escolher o percurso que passe apenas por paragens abrigadas.

```
callquery8(InicioPercurso,FimPercurso, 'Yes'):-
    paragem(_,_,FimPercurso,_,_,_,Abrigo,_,_,_,_,_),  % se a paragem final nao existir acaba logo
    memberchk(Abrigo, ['Aberto dos Lados','Fechado dos Lados']),
    query8(InicioPercurso,FimPercurso,[InicioPercurso], Percurso,  ['Aberto dos Lados','Fechado dos Lados']),
    write("\n"),
    write(Percurso),
    length(Percurso,R),
    write("\n\nO percurso contem "),
    write(R),
    write(" paragens!").

callquery8(InicioPercurso,FimPercurso, 'No'):-
    paragem(_,_,FimPercurso,_,_,_,Abrigo,_,_,_,_,_),  % se a paragem final nao existir acaba logo
    memberchk(Abrigo, ['Sem Abrigo']),
    query8(InicioPercurso,FimPercurso,[InicioPercurso], Percurso,['Sem Abrigo']),
    write("\n"),
    write(Percurso),
    length(Percurso,R),
    write("\n\nO percurso contem "),
    write(R),
    write(" paragens!").

query8(FimPercurso, FimPercurso, _, [FimPercurso],_).
query8(InicioPercurso,  FimPercurso, [InicioPercurso], [],_) :-
    paragem(_,C, FimPercurso,_,_,_,_,_,_,_,_,_),
    verificaIsolada(C,InicioPercurso), % ve se a paragem final é isolada
    \+paragem(_,C, InicioPercurso,_,_,_,_,_,_,_,_,_). % se nao pertence a mesma carreira acaba
query8(InicioPercurso,  FimPercurso, ParagensVisitadas, [InicioPercurso | Percurso], Flag) :-
    verificaCarreira(InicioPercurso, FimPercurso,R),
    (length(R,H), H>0, buscaValor(R,V), buscaId(V,InicioPercurso, FimPercurso,Id) ,adjacente((_,V,InicioPercurso),(Id,V,Proximo),_);
    adjacente((_,_,InicioPercurso),(_,_,Proximo),_)),
    paragem(_,_,Proximo,_,_,_,Abrigo,_,_,_,_,_),
    memberchk(Abrigo, Flag),   % verifica se tem abrigo
    \+memberchk(Proximo, ParagensVisitadas),
    query8(Proximo,  FimPercurso, [Proximo|ParagensVisitadas], Percurso,  Flag).
```

Para esta query a estratégia usada é semelhante à da query7, com a diferença que desta vez, verificamos se a paragem é, ou não, abrigada. Para isso, o utilizador insere uma *flag* sendo esta **'Yes'** ou **'No'**, simbolizando se este quer percurso apenas com paragens abrigadas ou sem abrigo, respetivamente. Como paragens não abrigadas foram consideradas aquelas que continham 'Sem Abrigo' no respetivo campo.

2.5.9 Query9

A query9 tem como objetivo escolher um ou mais pontos intermédios por onde o percurso deverá passar.

```
callquery9(InicioPercurso,FimPercurso, Lista):-
    paragem(_,_,FimPercurso,_,_,_,_,_,_,_,_,_),  % se a paragem final nao existir acaba logo
    rquery9(InicioPercurso,FimPercurso,[InicioPercurso], _, Lista, Percurso),
    write("\n"),
    write(Percurso),
    length(Percurso,R),
    write("\n\nO percurso contem "),
    write(R),
    write(" paragens!").

rquery9(InicioPercurso, FimPercurso, _ , [InicioPercurso | Percurso], [], P):-
    query1(InicioPercurso, FimPercurso, [InicioPercurso] , P).

rquery9(InicioPercurso, FimPercurso, ParagensVisitadas, [InicioPercurso | Percurso], [H|T], PFinal):-
    % se a paragem ainda nao pertence ao percurso final
    (\+(memberchk(H, ParagensVisitadas)),\+(memberchk(H,[InicioPercurso])),\+(memberchk(H,[FimPercurso])) ->
    query1(InicioPercurso, H, ParagensVisitadas, P),
    removehead(PF,PFaux),
    junta(P,PFaux,PFinal),
    rquery9(H, FimPercurso, ParagensVisitadas, [H | Percurso], T, PF);
    % se a paragem ja existe no percurso não percisa de voltar a passar lá
    rquery9(InicioPercurso, FimPercurso, ParagensVisitadas, [InicioPercurso | Percurso], T, PFinal)).
```

A abordagem a esta query foi um pouco diferente das outras todas. A estratégia utilizada foi ligar o ponto inicial e o ponto final aos pontos intermédios, garantindo assim que estes pertenciam ao percurso. Para evitar calcular percursos em excesso, foi acrescentado um if que verifica se a paragem intermédia já está contida no percurso ou se é igual à paragem inicial ou se é igual à paragem final. Caso alguma desta se confirme, então este ponto intermédio já pertence ao percurso, não sendo necessário voltar a calcular.

2.6 Resultados

De seguida foram feitos alguns testes para determinados caminhos. Percurso 1,2 e 3 representam os algoritmos não informados, pela respetiva ordem. A Query1 representa o algoritmo usada nessa mesma query. A Query11, foi um segundo algoritmo desenvolvido para a query1, de modo a comparar qual o mais eficiente.

	Percurso1	Percurso2	Percurso3	Query1	Query11
583 -> 1009	465 inferencias 0,01 segundos	2045 inferencias 0,05 segundos	**null**	3811 inferencias 0,01 segundos	3856 inferencias 0,01 segundos
609 -> 583	50473 inferencias 0,08 segundos	1181 inferencias 0,03 segundos	**null**	103755 inferencias 0,03 segundos	16593 inferencias 0,03 segundos
712 -> 219	465 inferencias 0,01 segundos	465 inferencias 0,01 segundos	**null**	74022 inferencias 0,03 segundos	35247 inferencias 0,12 segundos
609 -> 73	**null**	2250 inferencias 0,09 segundos	**null**	316 inferencias 0,01 segundos	218781 inferencias 0,66 segundos
183 -> 182	22 inferencias 0,01 segundos	18 inferencias 0,01 segundos	200 inferencias 0,01 segundos	871 inferencias 0,01 segundos	869 inferencias 0,01 segundos

Através desta tabela é possível observar que alguns casos, os algoritmos apresentam ligeiras falhas, não conseguindo obter resposta final. Também é possível verificar que o numero de inferencias varia de algoritmo para algoritmo, sendo uns mais eficientes que outros em determinados pontos.

3 Conclusões e sugestões

Com o auxílio da linguagem *Python* foi-nos possível então criar uma base de conhecimento para o exercicio proposto. Através de algoritmos de pesquisa, foi possível criar algumas funcionalidades de modo a recomendar qual o melhor caminho entre dois pontos, entre outras recomendações.

Porém, a elaboração deste trabalho não veio sem dificuldades, por exemplo na criação da base de conhecimento onde só após alguns retrocessos é que se chegou a um bom resultado, assim como na criação dos algoritmos e na melhoria desses mesmos algoritmos. Desta forma, este projeto mostrou ser uma excelente maneira de consolidar os conhecimentos transmitidos ao longo do semestre, e aperfeiçoar o manejamento do *PROLOG*.

Em futuras iterações deste trabalho, e de forma a torna-lo mais eficiente, poder-se-ia melhorar os algoritmos, passando pelo *debugging* de alguns casos que não funcionam, ou mesmo a criação de novos algortimos. Poder-se-ia também acrescentar ainda mais querys de modo a tornar o trabalho mais completo.

Concluindo, o resultado final foi bastante amargo, porque apesar de todo o esforço envolvido neste projeto, o objetivo maior não foi cumprido, pois não foram feitas todas as querys pedidas e os algoritmos criados apresentam algumas falhas, não obtendo a resposta pretendida em todos os casos.

4 Referências Bibliográficas

Referências

[1] Paulo Novais, Cesar Analide, José Neves,

"Sugestões para a Redação de Relatórios Técnicos ",

Relatório técnico, Departamento de Informática da Universidade do Minho, Portugal, 2011.