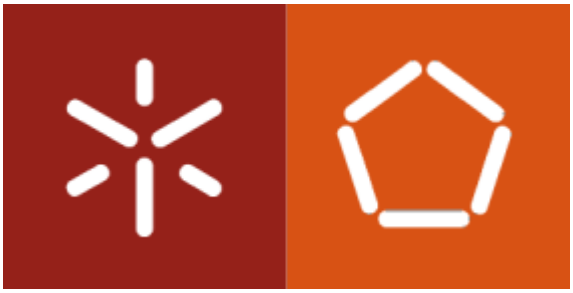

Universidade do Minho
Departamento de Informática
Mestrado Integrado em Engenharia Informática

Sistemas de Representação de Conhecimento e Raciocínio
Ano letivo 2019/2020

Rui Santos a67656
Instrumento de Avaliação
Componente Individual
Junho, 2020

1. Introdução

Este trabalho tem como objetivo a utilização de Programação em Lógica, utilizando a linguagem Prolog de modo a desenvolver algoritmos de pesquisa e resolução de problemas.

O nosso tema principal será o sistema de transportes do concelho de Oeiras, no qual teremos de determinar variados caminhos, como o mais curto, ou caminhos com certas restrições. Para tal irei criar uma base de conhecimento com os ficheiros de dados fornecidos, criando de seguida algoritmos que respondam às questões propostas.

2. Preliminares

De modo a testar os algoritmos foi necessária determinar uma maneira de determinar se caminhos existem e de calcular o caminho mais curto. Tendo em conta que o grafo é complexo demais para analisar manualmente, optei por usar uma ferramenta chamada Gephi.

Foi preciso apenas um pequeno ajuste nos meus predicados para gerar um CSV compatível, o que permitiu visualizar e confirmar soluções.

3. Descrição do Trabalho e Análise de Resultados

3.1. Base de conhecimento

A base de conhecimento inicialmente escolhida foi um simplesmente o resultado da transformação de cada uma das 39 listas de adjacência em predicados individuais. Nesta base de conhecimento não foi tida em conta conhecimento imperfeito, deste modo as paragens que não continham todos os campos foram descartadas.

Para processar os ficheiros CSV utilizei um pequeno script de Python utilizando a libraria Pandas.

```
directory = r'C:/Users/Ruca-WindowsX/Desktop/Dados/CSV'
file = open("MyFile.txt","a")

for filename in os.listdir(directory):
    indexx = 1
    file.write("%"+str(filename) )

    data = pandas.read_csv('C:/Users/Ruca-WindowsX/Desktop/Dados/CSV/' +filename).dropna()


    print (str (filename))

    for index, row in data.iterrows():

       file.write("paragem( " )

       file.write(str(indexx))
       file.write(" , ")

       file.write(str(row['Carreira']))
       file.write(" , ")

       ... Escrever todos os campos ....

       file.write("'")
       file.write(row['Freguesia'])
       file.write("'")

       file.write(").")
       file.write("\n")
       indexx+=1

    file.write("\n")

file.close()
```

Que gera predicados do género

```
paragem( 1 , 1 , 183 , -103678.36 , -96590.26 , 'Bom' , 'Fechado dos Lados' , 'Yes' , 'Vimeca' , 286 , 'Rua Aquilino Ribeiro' , 'Carnaxide e Queijas').
paragem( 2 , 1 , 791 , -103705.46 , -96673.6 , 'Bom' , 'Aberto dos Lados' , 'Yes' , 'Vimeca' , 286 , 'Rua Aquilino Ribeiro' , 'Carnaxide e Queijas').
paragem( 3 , 1 , 595 , -103725.69 , -95975.2 , 'Bom' , 'Fechado dos Lados' , 'Yes' , 'Vimeca' , 354 , 'Rua Manuel Teixeira Gomes' , 'Carnaxide e Queijas').
paragem( 4 , 1 , 182 , -103746.76 , -96396.66 , 'Bom' , 'Fechado dos Lados' , 'Yes' , 'SCoTTURB' , 286 , 'Rua Aquilino Ribeiro' , 'Carnaxide e Queijas').
paragem( 5 , 1 , 499 , -103758.44 , -94393.36 , 'Bom' , 'Fechado dos Lados' , 'Yes' , 'Vimeca' , 300 , 'Avenida dos Cavaleiros' , 'Carnaxide e Queijas').
paragem( 6 , 1 , 593 , -103777.02 , -94637.67 , 'Bom' , 'Sem Abrigo' , 'No' , 'Vimeca' , 300 , 'Avenida dos Cavaleiros' , 'Carnaxide e Queijas').
```

Durante o desenvolvimento do código deparei-me com um problema, e uma possível otimização. Quando o objetivo é encontrar um caminho, a noção de carreira é irrelevante, podendo ser até prejudicial, pois podíamos viajar da paragem X para a paragem Y através de várias carreiras diferentes, o que tornava o algoritmo muito mais ineficiente.

Deste modo decidi fazer uma segunda base, mas que guardava apenas as ligações dos nodos do grafo, uma vez mais utilizando Pandas em Python, eliminando também os repetidos. O código utilizado foi :

```
lista2=[]
directory = r'C:/Users/Ruca-WindowsX/Desktop/Dados/CSV'
file = open("MyFile.txt","a")

for filename in os.listdir(directory):
    indexx=1
    data = pandas.read_csv('C:/Users/Ruca-WindowsX/Desktop/Dados/CSV/' +filename).dropna()


    print (str (filename))
    last = 0
    for index, row in data.iterrows():

        gid = (str(row['gid']))
        if(indexx>1):
            if(not(last,gid)in(lista2)):
                lista2.append((last,gid))
        last= gid
        indexx+=1


file = open("MyFile.txt","a")
lista2.reverse()
for x in lista2:
    file.write ("adj( ")
    file.write (str (x[0]) )
    file.write (" , ")
    file.write (str(x[1]))
    file.write(").")
    file.write("\n")

file.close()
```

Que por sua vez gera predicados do tipo

```
adj( 183 , 791).
adj( 791 , 595).
adj( 595 , 182).
adj( 182 , 499).
adj( 499 , 593).
adj( 593 , 181).
adj( 181 , 180).
adj( 180 , 594).
```

Estes apenas representam os caminhos possíveis, tornando assim o meu grafo muito mais simples e eficiente de iterar. Os caminhos foram considerados num sentido apenas, ou seja, optei por um grafo direcionado. No final a lista é revertida, visto que o Prolog lê bases de conhecimento de baixo para cima.

3.2. Pesquisa não informada

3.2.1. Caminho básico

Tendo em conta que todos os objetivos do trabalho envolviam calcular algum tipo de caminho, optei por começar com um algoritmo básico de para determinar se o caminho existe, e outro para determinar o caminho em si.

```
existeCaminhoBasico(Id1,Id2) :- adj(Id1,Id2).
existeCaminhoBasico(Id1,Id2) :- adj(Id1,Idm), existeCaminhoBasico(Idm,Id2).

caminhoBasico(Id1,Id2,[Id1,Id2])  :-  adj(Id1,Id2) .
caminhoBasico(Id1,Id2,[Id1| T ])  :-  adj(Id1,Idm), caminhoBasico(Idm,Id2,T) .
```

Este nada mais faz que dizer que para existir um caminho entre A e C, ou A e C são adjacentes, ou A é adjacente a B, e existe um caminho de B para C, chamando assim o predicado recursivamente.

Embora fosse capaz de determinar alguns caminhos, rapidamente constatei que tinha problemas, ficava preso em loops no caso de grafos cíclicos, e não conseguia determinar em tempo útil alguns caminho que eram mais longos, ou que envolviam nodos em carreiras isoladas (carreiras que não ligam a nenhuma outra carreira).

Deste modo considerei um pequeno conjunto de testes para poder comparar as diferentes iterações do meu algoritmo

183 -> 499 (curto) - True - Imediato
183 -> 79 (longo) - True - Imediato
349 -> 359 (impossível e difícil) - False - 0.1s
349 -> 666 (impossível, 666 isolado) - False – 0.1s
556 -> 933 (impossível, cíclico) - Erro, Stack Limit Exceeded

Os resultados para ambos os algoritmos foram semelhantes, sendo o predicado que calcula o caminho ligeiramente mais lento.

3.2.2. Caminho sem repetidos

Visto que os meus dois algoritmos básicos dão erro em casos que podem entrar em ciclo, fiz uma versão ligeiramente diferente dos mesmos, em que guardo uma lista de visitados.

```
existeCaminhoRep(Id1,Id2)        :- existeCaminhoRepAux(Id1,Id2,[Id1]). % inciializar a lista de visitado

existeCaminhoRepAux(Id1,Id2, _ ) :- adj(Id1,Id2).
existeCaminhoRepAux(Id1,Id2,Rep) :- adj(Id1,Idm), not(member(Idm,Rep)), existeCaminhoRepAux(Idm,Id2,[Idm|Rep]).


caminhoRep(Id1,Id2,Caminho)  :- caminhoRepAux(Id1,Id2,Caminho,[Id1]) .

caminhoRepAux(Id1,Id2,[Id1,Id2], _ )   :-  adj(Id1,Id2) .
caminhoRepAux(Id1,Id2,[Id1| T ],Rep)   :-  adj(Id1,Idm) , not(member(Idm,Rep)) , caminhoRepAux(Idm,Id2,T,[Idm|Rep]) .
```

183 -> 499 (curto) - True - Imediato
183 -> 79 (longo) - True - Imediato
349 -> 359 (impossível, difícil) - False - 1.8s
349 -> 666 (impossível, difícil ,666 isolado) - False - 1.8s
556 -> 933 (impossível, cíclico) - False - Imediato

Embora o problema dos grafos cíclicos tenha sido resolvido, houve um claro aumento nos tempos de execução, principalmente nos caso difíceis, que passaram a demorar na ordem dos 2 segundos.

3.2.3. Caminho com máximo

De modo a poder começar a considerar o problema do caminho mais curto, criei uma pequena variação do algoritmo original, em que limito o tamanho máximo do caminho, de modo a forçar a procura de um caminho com esse comprimento ou menos.

```
existeCaminhoMax( _ , _ , 1 ) :- ! , fail.
existeCaminhoMax(Id1,Id2, _ ) :- adj(Id1,Id2).
existeCaminhoMax(Id1,Id2,Max) :- adj(Id1,Idm), Maxx is Max - 1, existeCaminhoMax(Idm,Id2,Maxx).

caminhoMax( _ , _ , _ , 1 )      :-  ! , fail.
caminhoMax(Id1,Id2,[Id1,Id2],_)  :-  adj(Id1,Id2) .
caminhoMax(Id1,Id2,[Id1|T],Max)  :-  adj(Id1,Idm), Maxx is Max - 1 , caminhoMax(Idm,Id2,T,Maxx) .
```

Deste modo introduzi mais dois testes no caminho 183->79, o mais curto, e um impossivelmente curto.

183 -> 499 (curto) (Max=100) - True - Imediato
183 -> 79 (longo) (Max=100) - True - Imediato
183 -> 79 (longo) (Max=33) - True - 2.6s
183 -> 79 (longo) (Max=32) - False - 2.3s
349 -> 359 (impossível, difícil) (Max=100) - False - 0.1s
349 -> 666 (impossível, difícil ,666 isolado) (Max=100) - False – 0.2s
556 -> 933 (impossível, cíclico) (Max=100) - False - Imediato

Como podemos ver este método é mais eficiente que o anterior, assumindo claro que 100 é suficiente para calcular qualquer caminho. Caso o maximo (nos casos falsos) fosse aumentado para 200 por exemplo, alguns caminhos não obtinham resposta, nem erro.

3.2.4. Caminho com máximo e repetidos

Foi ainda implementada uma versão combinada dos dois algoritmos anteriores, mas visto ser muito mais lenta, ao ponto de já não conseguir calcular alguns dos caminhos com um máximo baixo, foi descartada.

3.2.5. Conclusões

A pesquisa não informada revelou-se muito rápida de implementar, mas também muito intensiva a nível de recursos. Caminhos que não existem, cíclicos ou muito longos revelam-se um problema.

3.3. Pesquisa Informada

De modo a implementar uma pesquisa informada, optei por um algoritmo simples de depth first.

O algoritmo calcula todos os nodos adjacente à posição atual, e escolhe um que ainda não foi visitado para viajar. Se não existir nenhum voltamos para trás. Se voltarmos à origem, e não houver nenhum nodo adjacente por visitar quer dizer que já testamos todos os caminhos possíveis, logo não existe caminho.

```
caminhoInformado( Id1 , Id2 , CaminhoF) :-   caminhoInf( Id1 , Id2 , [Id1] , [Id1] , Caminho  ),   length(Caminho,X) , !, X=\=0 , reverse(Caminho,CaminhoF).

caminhoInf( Id1 , Id2 , Back , _ , [Id2|Back]) :- adj(Id1,Id2).

caminhoInf( Id1 , Id2 , [H|Back]  , Visitados , Caminho ):-
            getListaAdj(Id1,Lista),
            escolherAdjacente(Lista , Visitados, Escolhido),
            Escolhido =\= 0 ,
            caminhoInf( Escolhido , Id2 , [Escolhido,H|Back]  , [Escolhido|Visitados] , Caminho ).


caminhoInf( _ , _ , [_]  , _ , [] ):- write('Nao existe caminho \n ').


caminhoInf( _ , Id2 , [_|Back]  , Visitados , Caminho ):-
        primeiro(Back,Primeiro),
        caminhoInf( Primeiro , Id2 , Back  , Visitados , Caminho ).


getListaAdj(Id,Lista):-findall(X, adj(Id,X), Lista) .


escolherAdjacente( [] , _ , 0 ) .
escolherAdjacente([H|_], Visitados , H)       :- not(member(H, Visitados)).
escolherAdjacente([_|T], Visitados, Escolhido):- escolherAdjacente(T, Visitados , Escolhido).


primeiro( [H|_], H ).
```

Todos os testes efetuados devolveram o resultado certo, sempre com tempos muito reduzidos. A melhoria em eficiência é bastante óbvia, logo esta vai ser a solução a usar para responder aos pontos pedidos

3.4. Algoritmos pedidos

3.4.1. Pergunta 1 - Calcular um trajeto entre dois pontos

Para calcular o caminho opei por simplesmente usar a pesquisa informada

3.4.2. Pergunta 2 - Selecionar apenas algumas das operadoras de transporte

Para usar apenas algumas operadoras de transporte criei uma nova versão do caminho informado, em que a única diferença será ao escolher uma paragem adjacente, para além de não ter sido visitada, esta agora vai ter que pertencer a uma das operadoras selecionadas.

3.4.3. Pergunta 3 – Excluir uma ou mais operadoras de transporte

Uma vez mais, criei uma nova versão muito semelhante à anterior, mas em vez de garantir que está na lista de operadoras, garanto que não está

3.4.4. Pergunta 4 – Paragens com mais carreiras num percurso

Esta questão necessitou da criação de um predicado extra: quantasCarreiras (Id,N), que calcula em quantas carreiras uma determinada paragem se encontra incluída. De resto bastou apenas determinar o caminho através da pesquisa informada, e de seguida iterar o mesmo para calcular o valor para cada uma, criando assim pares. Por fim ordenamos os pares por ordem decrescente.

3.4.5. Pergunta 5 – Caminho mais curto por paragens

Este ponto revelou-se o mais difícil. Embora tenha tentado várias versões, não conseguir implementar corretamente uma procura de caminho mais curto utilizando a procura informada, deste modo optei por usar a procura informada apenas para determinar se existe caminho. Caso o caminho exista entro numa procura binária do caminho mais curto, com a utilização do predicado caminhoMax, a versão não informada de procura de caminho com limite de tamanho.

Deste modo, a procura do caminho mais curto de 183 para 79 irá ser

```
?- pergunta5(183,79,X),length(X,A).
Vou tentar 64
Vou tentar 32
Vou tentar 48
Vou tentar 40
Vou tentar 36
Vou tentar 34
Vou tentar 33
X = [183,791,595,594,185,107,250,597,953,609,599,1001,607,232,52,233,2
31,886,473,470,483,482,476,904,472,902,893,465,186,466,467,78,79],
A = 33
```

3.4.6. Pergunta 6 – Caminho mais curto por distância

Por falta de tempo, também não consegui para este ponto.

Se tivesse mais tempo iria provavelmente tratar a minha base de conhecimento e editar os predicados de adjacência para terem um terceiro campo, normalmente chamado de weight, que seria a distância pre-calculada. De seguida, numa pesquisa informada, iria fazer backtrack quando o caminho atual já fosse mais longo que um que já encontrado na pesquisa.

3.4.7. Pergunta 7 – Caminho por paragens com publicidade

Uma vez mais, optei por fazer uma pequena modificação do caminho informado, desta vez garantindo que a paragem escolhida tem publicidade.

3.4.8. Pergunta 8 – Caminho por paragens abrigadas

A mesma estratégia foi aplicada aqui, um predicado semelhante ao do caminho informado, apenas garantido que que a paragem escolhida é sempre abrigada.

3.4.9. Pergunta 9 – Caminho com pontos intermédios

Neste ponto também não considero ter chegado a uma boa solução. A estratégia adotada foi fazer sub-caminhos, ou seja: se fosse pretendido obter o caminho entre 1 e 10, a passar por 20 e 30, irei calcular o caminho de 1 a 20, de 20 a 30 e de seguida de 30 a 10.

Esta solução tem falhas óbvias:

- Se as paragens intermédias estiverem pela ordem errada, o caminho ficará impossível. Isto podia ser resolvido desenvolvendo algum algoritmo que fosse testando pares de paragens, de modo a testar todas ordens possíveis.

- Mesmo com a correção anterior, existe a hipótese de ser necessário fazer um ciclo dentro do grafo para garantir que passamos por todas as paragens intermédias, algo que o meu algoritmo informado não está pronto para fazer. Acho que seria possível corrigir este problema permitindo repetir qualquer ponto pelo menos uma por cada ponto intermédio, mas provavelmente iria levar a um aumento elevado no tempo de execução.

4. Conclusões e Sugestões

De uma maneira geral considero que fui capaz de implementar predicados que respondem aos pontos indicados razoavelmente bem, com a exceção da pergunta 6.

Uma conclusão fácil de tirar será que uma procura informada não só é mais rápida, como mais eficiente a nível de recursos. Infelizmente dediquei demasiado tempo no início do desenvolvimento a tentar melhorar a pesquisa não informada, antes de começar a desenvolver a informada, o que levou a que não conseguisse desenvolver um algoritmo de caminho mais curto utilizando a procura informada, que provavelmente teria resultados muito melhores.