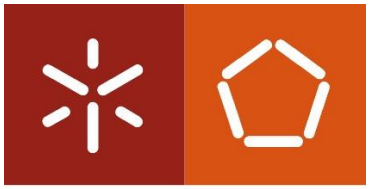

Universidade do Minho

Departamento de Informática

Sistemas de Representação de Conhecimento e Raciocínio

Trabalho Individual

José Ferreira (A83683)

5 de Junho de 2020

Conteúdo

Capítulo 1

Introdução

Este projeto prático individual foi desenvolvido no âmbito da cadeira de Sistemas de Representação de Conhecimento e Raciocínio.
O objetivo deste trabalho é aplicar os conhecimentos lecionados ao longo deste semestre num caso de estudo.
Assim, será utilizada a linguagem de programação lógica *Prolog* para criar um conjunto predicados que respondem a questões relativas a dados sobre o sistema de transportes do concelho de Oeiras.
Devido à natureza dos dados apresentados, a maneira como estes são melhor representados é através de um grafo em que os nodos são as paragens e as arestas são as ligações entre elas.
Ao longo deste relatório será descrito o processo utilizado para resolver o problema.

Capítulo 2

Base de Conhecimento

Todos os dados necessários relativos às paragens de autocarros estão disponibilizados em dois ficheiros *.xlsx*.
O primeiro ficheiro contém informação relativa a cada uma das paragens. O segundo ficheiro contém informação sobre a ligação entre as paragens.

2.1 Script

O formato de ficheiros fornecido não é suportado diretamente pelo *Prolog* por isso foi criado um *script* em *Python* (Anexo A) para converter os ficheiros de dados fornecidos para um formato suportado.
Para conseguir ler os ficheiros *.xlsx* foi utilizada a biblioteca *pandas*. Esta escolha foi feita devido à grande diversidade de formatos suportados e fácil desenvolvimento do programa pretendido.
Para obter um guia de utilização do programa basta correr:
python parser.py.

2.1.1 Nodos

Para ler a informação relativa às paragens de autocarros é carregado o ficheiro *paragens_autocarros_oeiras_processado_4.xslx*
Em seguida, a tabela é percorrida linha a linha. Para cada linha, obtém-se os seguintes dados:

- gid,
- latitude
- longitude
- Estado de Conservacao
- Tipo de Abrigo
- Abrigo com Publicidade?
- Operadora
- Carreira
- Codigo de Rua
- Nome da Rua
- Freguesia

Visto que o *Prolog* não suporta *UTF-8* e que o próprio ficheiro *.xslx* tinha alguns problemas de *encoding*, foi necessário tratar alguns dos dados obtidos.
Para aqueles que apresentavam problemas de *encoding* no próprio ficheiro de dados, foi utilizada uma biblioteca de *Python* chamada *ftfy* capaz de corrigir estes erros.
Para resolver o problema de *Prolog* não suportar *UTF-8* foi utilizada uma biblioteca chamada *unidecode* que contém um método para converter uma string de *UTF-8* para *ASCII* convertendo os caracteres com acentos para caracteres sem acentos.
Depois desta correção ser aplicada, foi detetado que algumas listas de Carreiras tinham mais do que uma vírgula seguida. Para corrigir este problema, separamos a string pelo carácter *','*, eliminamos todas as strings vazias da lista e voltamos a juntar todas as strings numa só com *','*.
Estes dados recolhidos e tratados são então impressos para *stdout* seguindo o seguinte formato:
paragem(gid,latitude,longitude,Estado de Conservacao,Tipo de Abrigo,Abrigo com Publicidade?,Operadora,[Carreira],Codi
de Rua,Nome da Rua,Freguesia).
Note se que já segue o formato de uma base de conhecimento em *Prolog*. Assim, basta redirecionar o *output* para um ficheiro e carregar esse ficheiro diretamente.

2.1.2 Arestas

O ficheiro de *.xslx* fornecido com informação sobre a ligação entre cada paragem segue o formato de ter uma *Sheet* para cada percurso de autocarro.
Assim, é necessário primeiro carregar o ficheiro *lista_adjacencias_paragens.xslx*. Depois, percorrer cada *Sheet*, e para cada *Sheet* percorrer cada linha.
Em cada linha obtém-se o *gid* atual, a carreira a que corresponde e as coordenadas da paragem. Obtém-se também o *gid* e as coordenadas da paragem na linha seguinte. Com estes dados é possível calcular qual é a distância entre as duas paragens.
O resto da informação contida neste ficheiro pode ser ignorada porque está repetida do ficheiro relativo às paragens e já foi carregada para a base de conhecimento.
Da mesma forma como foi feito para o caso das paragens, esta parte do *script* imprime *Prolog* válido para o *stdout* que segue o formato:
edge(gid_row, gid_next_row, carreira, distancia).
Assim é possível obter todas as arestas do grafo que representa o problema.

Capítulo 3

Predicados

Os predicados desenvolvidos devem ser capazes de cumprir os seguintes requisitos:

1. Calcular um trajeto entre dois pontos;
2. Selecionar apenas uma lista de operadores para um dado percurso;
3. Excluir uma lista de operadores para o percurso;
4. Identificar as paragens com mais carreiras num determinado percurso.
5. Escolher o percurso com menor número de paragens;
6. Escolher o percurso mais curto;
7. Escolher o percurso que passe apenas por abrigos com publicidade;
8. Escolher o percurso que passe apenas por paragens abrigadas;
9. Escolher um ou mais pontos intermédios por onde o percurso deverá passar.

Ao longo deste capítulo será descrita a forma como estes predicados foram implementados.

3.1 Calcular um trajeto entre dois pontos

3.1.1 Pesquisa não Informada - Depth-First

O predicado *dfs/3* implementa *Depth-First search* com uma lista de nodos visitados.
O predicado *dfsr/4* funciona como um predicado auxiliar recursivo para calcular o caminho. Quando este é chamado pela primeira vez é chamado com os parâmetros: nodo de origem, nodo de destino, uma lista de visitados com o nodo de origem e o caminho.
Sempre que este predicado é visitado, obtém-se o próximo nodo garantindo-se que este é adjacente com o predicado *adjacente/2*, garante-se que o nodo ainda não foi visitado como predicado member e por fim aplica-se recursivamente o predicado atualizando os valores.

```
1  dfs(Nodo, Destino, [Nodo|Caminho]):-
2      dfsr(Nodo, Destino, [Nodo], Caminho).
3
4  dfsr(Nodo, Destino, Visited, [Destino]):-
5      adjacente(Nodo, Destino).
6
7  dfsr(Nodo, Destino, Visited, [ProxNodo|Caminho]):-
8      adjacente(Nodo, ProxNodo),
9      \+ member(ProxNodo, Visited),
10     dfsr(ProxNodo, Destino, [Nodo|Visited], Caminho).
11
12 adjacente(Nodo,ProxNodo):-
13     edge(Nodo, ProxNodo, _, _).
```

```
?- dfs(706, 729, Path).
Path = [706, 703, 719, 718, 729] .
```

Figura 3.1: Exemplo de Depth-First

3.1.2 Pesquisa Informada - Greedy

O predicado *greedy* implementa a pesquisa gulosa. Nesta pesquisa foi utilizado como predicado para calcular o custo estimado para chegar ao destino a distância euclidiana.

```
1  greedy(Nodo, Destino, Caminho/Custo):-
2      estima(Nodo, Destino, E),
3      agreedy([[Nodo]/0/E], InvCaminho/Custo/_, Destino),
4      inverso(InvCaminho,Caminho).
5
6  agreedy(Caminhos, Caminho, Destino):-
7      get_best_g(Caminhos,Caminho),
8      Caminho = [Nodo|_]/_/_,
9      Nodo == Destino.
10
11 agreedy(Caminhos, SolucaoCaminho, Destino):-
12     get_best_g(Caminhos, MelhorCaminho),
13     seleciona(MelhorCaminho,Caminhos,OutrosCaminhos),
14     expand_greedy(MelhorCaminho,ExpCaminhos, Destino),
15     append(OutrosCaminhos,ExpCaminhos,NovoCaminhos),
16     agreedy(NovoCaminhos,SolucaoCaminho, Destino).
17
18 get_best_g([Caminho],Caminho):- !.
19
20 get_best_g([Caminho1/Custo1/Est1,_/Custo2/Est2|Caminhos], MelhorCaminho):-
21     Est1 =< Est2,
22     !,
23     get_best_g([Caminho1/Custo1/Est1|Caminhos],MelhorCaminho).
24
25 get_best_g([_|Caminhos], MelhorCaminho):-
26     get_best_g(Caminhos, MelhorCaminho).
27
28 expand_greedy(Caminho, ExpCaminhos, Destino):-
29     findall(NovoCaminho, adjacente(Caminho, NovoCaminho, Destino), ExpCaminhos).
30
31 estima(P1,P2,R):-
32         paragem(P1,X1,Y1,_,_,_,_,_,_,_,_),
33         paragem(P2,X2,Y2,_,_,_,_,_,_,_,_),
34         R is sqrt(((X1-X2)*(X1-X2))+((Y1-Y2)*(Y1-Y2))).
35
36 adjacente([Nodo|Caminho]/Custo/_, [ProxNodo,Nodo|Caminho]/NovoCusto/Est, Destino):-
37     edge(Nodo, ProxNodo, _, PassoCusto),
38     \+ member(ProxNodo, Caminho),
39     NovoCusto is Custo + PassoCusto,
40     estima(ProxNodo, Destino, Est).
```

```
?- greedy(706, 729, Path/Cost).
Path = [706, 703, 719, 718, 729],
Cost = 694.8397247015118 .
```

Figura 3.2: Exemplo de Greedy

3.1.3 Depth-First vs Greedy

Ambas os métodos de pesquisa implementados não garantem a completude da solução. No caso de *Depth-First* falha em espaços de profundidade infinita com *loops*, no caso da *Greedy* pode entrar em

ciclos e é susceptível a falsos começos.
Apesar de a complexidade temporal parecer a mesma à primeira vista ($O(b^m)$), p tempo do algoritmo *Greedy* pode ser consideravelmente diminuído com uma boa função heurística. No melhor caso porde chegar a (O(bd)).
No caso da complexidade no espaço a melhor é a *Depth-First* fazendo com que esta tenha espaço linear. Por fim, no caso da Optimalidade da solução encontrada, nenhum destes algoritmos a garante. O *Depth-First* devolve a primeira solução encontrada e o *Greedy* pode não encontrar a solução óptima.

	Completude	Complexidade no Tempo	Complexidade no Espaço	Optimalidade
Depth-First	Não	$O(b^m)$	O(bm)	Não
Greedy	Não	$O(b^m)$ / O(bd)	$O(b^m)$	Não

3.2 Selecionar apenas uma lista de operadores para um dado percurso

Para obter um percurso que só utiliza um determinado conjunto de operadores basta modificar o método de pesquisa *Depth-First* para garantir que o operador está na lista.
Desta forma, foi desenvolvido o predicado *dfs_op/4*

```
dfs_op(Nodo, Destino, Operadoras, [Nodo|Caminho]):-
    dfsr_op(Nodo, Destino, Operadoras, [Nodo], Caminho).

dfsr_op(Nodo, Destino, Operadoras, Visited, [Destino]):-
    adjacente(Nodo, Destino),
    paragem(Nodo,_,_,_,_,_,_,0,_,_,_,_),
    member(0,Operadoras).

dfsr_op(Nodo, Destino, Operadoras, Visited, [ProxNodo|Caminho]):-
    adjacente(Nodo, ProxNodo),
    \+ member(ProxNodo, Visited),
    paragem(Nodo,_,_,_,_,_,_,0,_,_,_,_),
    member(0,Operadoras),
    dfsr_op(ProxNodo, Destino, Operadoras, [Nodo|Visited], Caminho).
```

```
?- dfs_op(706, 729, ['Vimeca'], Path).
Path = [706, 703, 719, 718, 729] .
```

Figura 3.3: Exemplo de dfs_op

3.3 Excluir uma lista de operadores para o percurso

De forma análoga, para garantir que o percurso não utiliza um determinado conjunto de operadores, basta modificar o método de pesquisa *Depth-First* para garantir que o operador não está na lista. Para tal basta negar a linha criada para o predicado descrito acima.

```
dfs_noop(Nodo, Destino, Operadoras, [Nodo|Caminho]):-
    dfsr_op(Nodo, Destino, Operadoras, [Nodo], Caminho).

dfsr_noop(Nodo, Destino, Operadoras, Visited, [Destino]):-
    adjacente(Nodo, Destino).

dfsr_noop(Nodo, Destino, Operadoras, Visited, [ProxNodo|Caminho]):-
    adjacente(Nodo, ProxNodo),
    \+ member(ProxNodo, Visited),
    paragem(Nodo,_,_,_,_,_,_,0,_,_,_,_),
    \+ member(0,Operadoras),
    dfsr_noop(ProxNodo, Destino, Operadoras, [Nodo|Visited], Caminho).
```

```
?- dfs_noop(706, 729, ['Vimeca'], Path).
Path = [706, 703, 719, 718, 729] .
```

Figura 3.4: Exemplo de dfs_noop

3.4 Identificar as paragens com mais carreiras num determinado percurso

Para identificar as paragens com mais carreiras num determinado percurso foi criado o predicado *sorted_nstop/3*.
Primeiro é calculado um percurso com base num predicado à escolha. Neste caso foi usado o *dfs/3*. Em seguida utiliza-se o predicado *calc_len* para calcular uma lista de tuplos com o nodo e o número de Carreiras em cada nodo.
Por fim foi implementado uma variação do *QuickSort* que organiza em sentido decrescente com base no segundo elemento do tuplo. A única alteração que foi feita ao algoritmo foi na função que calcula o pivô fazendo com que o termo de comparação seja o segundo elemento do tuplo.

```
1  sorted_nstop(Nodo, Destino, R):-
2      dfs(Nodo, Destino, P),
3      calc_len(P, PL),
4      quick_sort(PL, R).
5
6  calc_len([], []).
7  calc_len([H|T], [(H,LC)|R]):-
8      paragem(H,_,_,_,_,_,_,C,_,_,_),
9      length(C, LC),
10     calc_len(T, R).
11
12 quick_sort([],[]).
13 quick_sort([H|T],Sorted):-
14     pivoting(H,T,L1,L2),
15         quick_sort(L1,Sorted1),
16         quick_sort(L2,Sorted2),
17     append(Sorted1,[H|Sorted2], Sorted).
18
19 pivoting(H,[], [], []).
20 pivoting((H,C1),[(X,C2)|T], L, [(X,C2)|G]):-
21     C2=<C1,
22     pivoting((H,C1),T,L,G).
23 pivoting((H, C1),[(X,C2)|T], [(X, C2)|L], G):-
24     C2>C1,
25     pivoting((H,C1), T, L, G).
```

```
?- sorted_nstop(706, 729, Path).
Path = [(706, 2), (703, 2), (718, 2), (729, 2), (719, 1)] .
```

Figura 3.5: Exemplo de sorted_nstop

3.5 Escolher o percurso com menor número de paragens

Para calcular o caminho com o menor número de paragens foi criado o predicado *dfs_len/4* que utiliza *Depth-First* e depois calcula o comprimento do caminho calculado. Este predicado é utilizado em conjunto com o *findall/3* para obter todos os caminhos possíveis da origem até ao destino.
Em seguida é utilizado o predicado *minimo/2* para obter o mínimo.
Isto é conjugado para criar o predicado *best_len/4*.

```
best_len(Nodo, Destino, S, N_stops) :-
    findall((SS, CC), dfs_len(Nodo, Destino, SS, CC), L),
    minimo(L,(S,N_stops)).

dfs_len(Nodo, Destino, C, N):-
    dfs(Nodo, Destino, C),
    length(C, N).

minimo([H|T], R):- minimoa(T, H, R).

minimoa([], Min, Min).
minimoa([(_, C) | T], (NM, CM), Min):-
    C > CM,
    minimoa(T, (NM, CM) , Min).

minimoa([(H, C) | T], (_, CM), Min):-
    C =< CM,
    minimoa(T, (H, C), Min).
```

Infelizmente este método entra em *loop* infinito possivelmente devido a informação repetida na base de conhecimento.

3.6 Escolher o percurso mais curto

De forma semelhante, o predicado *beat_cost/4* calcula o caminho com menor custo. Sendo que a única diferença quando comparado com a *best_len/4* é que calcula o custo do caminho em vez de calcular o comprimento.

```
best_cost(Nodo, Destino, S, Custo):-
    findall((SS, CC), dfs_cost(Nodo, Destino, SS, CC), L),
    minimo(L,(S,Custo)).

dfs_cost(Nodo, Destino, Path, Custo):-
    dfs(Nodo, Destino, Path),
    calc_cost(Path, Custo).

calc_cost([H,NH], C):-
    edge(H, NH, _, C).
calc_cost([H,NH|T], C):-
    edge(H, NH, _, C1),
    calc_cost([NH|T], C2),
    C is C1+ C2.
```

Infelizmente este método entra em *loop* infinito possivelmente devido a informação repetida na base de conhecimento.

3.7 Escolher o percurso que passe apenas por abrigos com publicidade

Como foi feito acima, para garantir que o percurso não utiliza abrigos sem publicidade, basta modificar o método de pesquisa *Depth-First* para garantir que a característica da paragem. Assim foi desenvolvido o predicado *dfs_pub/3*.

```
dfs_pub(Nodo, Destino, [Nodo|Caminho]):-
    dfsr_abrigado(Nodo, Destino, [Nodo], Caminho).

dfsr_pub(Nodo, Destino, _, [Destino]):-
    adjacente(Nodo, Destino).

dfsr_pub(Nodo, Destino, Visited, [ProxNodo|Caminho]):-
    adjacente(Nodo, ProxNodo),
    \+ member(ProxNodo, Visited),
    paragem(Nodo,_,_,_,_,_,_,_,P,_,_,_),
    P == 'Yes',
```

```
12      dfsr_pub(ProxNodo, Destino, [Nodo|Visited], Caminho).
```

```
?- dfs_pub(706, 720, Path).
Path = [706, 703, 719, 718, 728, 729, 724, 129, 720] .
```

Figura 3.6: Exemplo de dfs_pub

3.8 Escolher o percurso que passe apenas por paragens abrigadas

De forma análoga, para garantir que as paragens por onde passa o caminho sejam todas abrigadas, basta garantir que todos os nodos têm a propriedade a ser diferente de 'Sem Abrigo'. Desta forma, foi criado o predicado *dfs_abrigado*.

```
1  dfs_abrigado(Nodo, Destino, [Nodo|Caminho]):-
2      dfsr_abrigado(Nodo, Destino, [Nodo], Caminho).
3
4  dfsr_abrigado(Nodo, Destino, _, [Destino]):-
5      adjacente(Nodo, Destino).
6
7  dfsr_abrigado(Nodo, Destino, Visited, [ProxNodo|Caminho]):-
8      adjacente(Nodo, ProxNodo),
9      \+ member(ProxNodo, Visited),
10     paragem(Nodo,_,_,_,_,_,A,_,_,_,_),
11     A \== 'Sem␣Abrigo',
12     dfsr_abrigado(ProxNodo, Destino, [Nodo|Visited], Caminho).
```

```
?- dfs_abrigado(706, 720, Path).
Path = [706, 703, 719, 718, 728, 729, 724, 129, 720] .
```

Figura 3.7: Exemplo de dfs_abrigado

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

A maneira como foi criado o método para calcular o caminho entre dois pontos permite facilmente alterar o método de pesquisa utilizado. Neste caso o método de pesquisa utilizado foi o *Greedy*.
O predicado implementado calcula o caminho entre as duas primeiras paragens da lista, aplica o predicado *init/2* que calcula a lista menos o último elemento, calcula recursivamente o caminho para os outros pontos e concatena as duas listas calculadas.

```
1  inter([P1, P2], Curr):-
2      greedy(P1, P2, Curr/_).
3
4  inter([P1, P2 | Rest], Path):-
5      greedy(P1, P2, Curr/_),
6      inter([P2 | Rest], Tail),
7      init(Curr, Initial),
8      append(Initial, Tail, Path).
9
10 init([], []).
11 init([A], []).
12 init([H|T], [H|R]):- init(T, R).
```

Para modificar este predicado para utilizar o método de pesquisa *Depth-First* implementado bastaria alterar as linhas com o predicado *greedy/3* para:

```
1 dfs(P1, P2, Curr),
```

```
?- inter([706, 729, 720], Path).
Path = [706, 703, 719, 718, 729, 131, 129, 720] .
```

Figura 3.8: Exemplo de inter

Capítulo 4

Conclusão

Concluindo , com este trabalho foram aplicados os conhecimentos de *Prolog* lecionados ao longo deste semestre na unidade curricular de sistemas de representação de conhecimento e raciocino. Nomeadamente conhecimentos relativos a travessia de grafos.
Como trabalho futuro seria relevante implementar mais métodos de pesquisa informada, nomeadamente o A*, de forma a melhorar a qualidade dos resultados obtidos e resolver a escolha do melhor percurso com base no número de paragens e com o comprimento.

Apêndice A

Script em Python

```
import sys
import pandas as pd
from ftfy import fix_encoding
from math import sqrt
import unidecode

def fixerino(row, field):
    if row[field] == row[field]:
        a = unidecode.unidecode(fix_encoding(row[field]))
        return "'{}'".format(a.strip().replace("'", ''))
    else:
        return 'undefined'

def parse_list(s):
    if ',' in str(s):
        return ','.join(filter(None, str(s).split(',')))
    else:
        return s

#paragem(gid,latitude,longitude,Estado de Conservacao,Tipo de Abrigo,Abrigo com Publicidade?,Operadora,[Carreira],Codigo de Rua,Nome da Rua,Freguesia').
def parse_paragens():
    xl = pd.read_excel('paragem_autocarros_oeiras_processado_4.xlsx')
    for index, row in xl.iterrows():
        print("paragem({0},{1},{2},{3},{4},{5},{6},[{7}],{8},{9},{10}).".format(
            row['gid'],
            row['latitude'],
            row['longitude'],
            fixerino(row,'Estado de Conservacao'),
            fixerino(row, 'Tipo de Abrigo'),
            fixerino(row, 'Abrigo com Publicidade?'),
            fixerino(row,'Operadora'),
            parse_list(row['Carreira']),
            row['Codigo de Rua'],
            fixerino(row, 'Nome da Rua'),
            fixerino(row, 'Freguesia')
        ))

def distancia(row, n_row):
    x1 = row['latitude']
    y1 = row['longitude']
    x2 = n_row['latitude']
    y2 = n_row['longitude']

    d = sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1))
    if d == d:
        return d
    else:
        return 'undefined'

# edge(gid_row, gid_next_row, carreira, distancia).
```

```
51 def parse_graph():
52     xl = pd.ExcelFile('lista_adjacencias_paragens.xlsx')
53     for sheet in xl.sheet_names:
54         book = xl.parse(sheet)
55         for i in range(book.shape[0]-1):
56             org =  book.iloc[i]['gid']
57             dest = book.iloc[i + 1]['gid']
58             if org != dest:
59                 print("edge({0},{1},{2},{3}).".format(
60                     org,
61                     dest,
62                     book.iloc[i]['Carreira'],
63                     distancia(book.iloc[i], book.iloc[i+1])
64                 ))
65 
66 def main():
67     if len(sys.argv) == 2 and '--stops' in sys.argv[1].lower():
68         parse_paragens()
69     elif len(sys.argv) == 2 and '--edges' in sys.argv[1].lower():
70         parse_graph()
71     else:
72         print('USAGE: parse [--stops|--edges]')
73 
74 main()
```