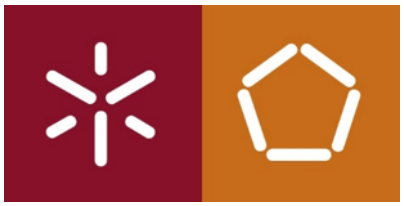

Universidade do Minho

Sistemas de Representação de Conhecimento e Raciocínio
(3º ano de curso, 2º semestre)

Exercício 2

Relatório de Desenvolvimento

Mestrado Integrado em Engenharia Informática

Hugo André Coelho Cardoso, a85006

5 de Junho de 2020

Resumo

O segundo trabalho prático realizado no âmbito da unidade curricular *Sistemas de Representação de Conhecimento e Raciocínio* teve como caso de estudo o sistema de transportes do concelho de Oeiras e o objetivo era o desenvolvimento de um programa que permitisse importar estes dados para uma base de conhecimento em Prolog e, posteriormente, desenvolver um sistema de recomendação de transporte público.

No presente relatório explicarei a forma como interpretei o problema e envisionei a solução, numa tentativa de desenvolver uma solução robusta e capaz de acomodar todos os requisitos do enunciado.

Os principais objetivos deste trabalho prático eram os seguintes: trabalhar com Lógica Extendida, aplicando Métodos de Resolução de Problemas e de Procura; aumentar a experiência de uso da linguagem de programação simbólica Prolog; aprimorar o desenho de soluções a partir de sistemas reais e a sua adaptação de forma prática e precisa, lidando com dados inconsistentes e em grande quantidade.

Conteúdo

1 Introdução **3**

2 Preliminares **4**

3 Descrição do Trabalho e Análise de Resultados **5**

3.1 Base de Conhecimento 6

3.1.1 Análise dos dados 6

3.1.2 Importação dos dados 7

3.2 Requisito 1 10

3.3 Requisitos 2, 3, 7 e 8 13

3.4 Requisito 4 15

3.5 Requisito 5 16

3.6 Requisito 6 20

3.7 Requisito 9 22

3.8 Análise de Resultados 23

3.8.1 Requisito 1 23

3.8.2 Requisitos 2, 3, 7 e 8 23

3.8.3 Requisito 4 25

3.8.4 Requisito 5 25

3.8.5 Requisito 9 25

3.9 Comparação dos Algoritmos usados 27

4 Conclusões e Sugestões **28**

5 Referências **29**

A Anexo **30**

Lista de Figuras

1 Macro para geração de ficheiros de extensão csv . . . 7
2 Programa Python para a criação da base de conhecimento . . . 8
3 Informação representada no ficheiro paragens.pl . . . 8
4 Informação representada no ficheiro adjacencias.pl . . . 9
5 Predicado do requisito 1 . . . 11
6 Predicado de pesquisa informada . . . 11
7 Predicado que verifica atalhos num percurso . . . 12
8 Verificação da condição dos requisitos no nodo atual . . . 13
9 Predicados do requisitos 2, 3, 7 e 8 . . . 14
10 Predicado do requisito 4 . . . 15
11 Predicado auxiliar contaCarreiras . . . 15
12 Predicado auxiliar invertePares . . . 15
13 Predicado de pesquisa não-informada do percurso mais curto . . . 16
14 Predicado maisCurto . . . 17
15 Predicado do requisito 5 . . . 18
16 Predicados auxiliares escolhePercursoInicial e sentidoTras . . . 18
17 Predicado auxiliar mudancasSentido . . . 19
18 Predicado auxiliar uniOuBidirecional . . . 19
19 Predicado auxiliar ordenaPercurso . . . 19
20 Predicado do requisito 6 . . . 20
21 Predicado auxiliar calculaDistancia . . . 20
22 Predicado maisProximo . . . 21
23 Predicado de pesquisa não-informada do caminho mais próximo . . . 21
24 Predicado do requisito 9 . . . 22
25 Predicado auxiliar concatPartesPercurso . . . 22
26 Percurso não existente . . . 23
27 Percurso com origem e destino na mesma carreira . . . 23
28 Percurso com origem e destino em carreiras diferentes . . . 23
29 Representação do percurso impossível com as operadoras dadas . . . 23
30 Percurso impossível com as operadoras dadas . . . 24
31 Percurso direto com as operadoras dadas . . . 24
32 Representação do percurso indireto com as operadoras dadas . . . 24
33 Percurso indireto com as operadoras dadas . . . 24
34 Aplicação do predicado do requisito 4 . . . 25
35 Percursos mais curtos unidirecionais . . . 25
36 Percurso mais curto bidirecional . . . 25
37 Percurso impossível com paragens intermédias . . . 25
38 Percurso possível com paragens intermédias . . . 26
39 Função parser do programa Python . . . 30
40 Predicado processaProxParagem . . . 30
41 Predicado escolheProxParagem . . . 31
42 Algoritmo *breadth-first* para calcular carreiras ate ao destino . . . 31

1 Introdução

O principal objetivo deste projeto é aprofundar a temática da Programação em Lógica Extendida e a aplicação de Métodos de Resolução de Problemas e de Procura, introduzidas nas aulas da unidade curricular *Sistemas de Representação de Conhecimento e Raciocínio*.

Foi-nos pedido para estudar o sistema de transportes do concelho de Oeiras, cujos dados se encontram num repositório público de dados abertos, e foram ainda pré-processados e disponibilizados pelo grupo docente em folhas Excel. Estes dados contêm informações relativas às paragens de autocarro de Oeiras, englobando várias características tais como a sua localização, as carreiras que as utilizam e respetivas operadoras, entre outras.

Para a importação e pré-processamento dos dados, desenvolvi um programa em *Python*, fazendo uso da ferramenta *Pandas* - este processo será abordado em detalhe em 3.1.2.

No que toca ao desenvolvimento do sistema de recomendação de transportes, tive em conta diferentes estratégias de pesquisa (não-informada e informada), de maneira a otimizar as travessias do grafo e respetivos tempos de execução, procurando implementar a estrátegia mais adequada a cada parte do trabalho. Os algoritmos desenvolvidos serão apresentados em secção 3, juntamente com uma tabela comparativa entre eles.

2 Preliminares

Foi realizado um trabalho de pesquisa preliminar, com o objetivo de adquirir todos os conhecimentos necessários para a realização deste trabalho prático e para a concretização dos requisitos presentes no enunciado.

Embora o tema do exercício, um sistema de tansportes públicos, seja bastante familiar, foi necessária uma análise cuidada dos dados fornecidos para entender o significado e a relevância de cada parâmetro para o presente trabalho prático. Desta forma, fiquei a entender que o utilizador identificaria as paragens pelos gids, as carreiras forneciam informação fulcral sobre a adjacência de paragens e certos parâmetros - estado de conservação, código e nome de rua e freguesia - seria irrelevantes para o projeto.

Foi também concluída a existência de bastantes erros e incoerências nos dados fornecidos, pelo que teve de se fazer um processamento cuidado e extensivo dos mesmos, como será referido em 3.1.2.

De seguida, pensando na arquitetura e estruturação da solução, decidi optar por realizar o trabalho prático no ambiente **SWI-Prolog**, uma vez que fornece um maior leque de predicados e bibliotecas extremamente úteis, bem documentados e de fácil acesso, relativamente à alternativa **SICStus Prolog**.

Quanto à matéria abrangida pelo presente projeto, foi realizado um estudo compreensivo das várias estratégias de pesquisa informadas - *depth-first*, *breadth-first*, pesquisas de custo uniforme, iterativa e bidirecional - e não-informadas - pesquisa gulosa e A* - e, após alguma reflexão, foi tomada a decisão de desenvolver um algoritmo informado de raiz que se adequasse mais ao contexto do problema, proporcionando assim mais liberdade para otimizar a solução de forma diferenciada.

Mais tarde, para a implementação dos caminhos mais curto e próximo, foram ainda implementados dois algoritmos não-informados, de maneira a executar uma pesquisa bruta mais rápida, sob certas condições, que não era possível ao algoritmo informado criado.

3 Descrição do Trabalho e Análise de Resultados

O presente capítulo está dividido em oito secções diferentes, de maneira a possibilitar a exposição organizada e bem detalhada do projeto desenvolvido.

Em primeiro lugar, é abordada a Base de Conhecimento desenvolvida, explicando o processo de importação dos dados a partir dos ficheiros fornecidos pelos docentes e o seu processamento.

De seguida, são abordadas as estratégias usadas para a solução de cada requisito do enunciado, explicando em detalhe o raciocínio por detrás da implementação e o código desenvolvido.

Na Análise de Resultados são exibidos vários testes cronometrados realizados aos predicados resolvidos para cada requisito, de maneira a mostrar a eficácia e o bom funcionamento dos mesmos.

Por fim, em 3.9, comparo os algoritmos informado e não-informados implementados segundo um certo conjunto de critérios.

Ainda no Anexo, encontram-se os predicados auxiliares usados no programa que não foram abordados em detalhe nesta secção.

3.1 Base de Conhecimento

A base de conhecimento desenvolvida consiste em dois ficheiros diferentes:

- **paragens.pl** - contém informação relativa às paragens de autocarro de Oeiras, sob a forma de predicados **paragem/13**. Cada predicado possui a seguinte estrutura: carreira atual da paragem, posição da paragem nessa carreira [*cid*], identificador geográfico da paragem [*gid*], latitude, longitude, estado de conservação, tipo de abrigo, existência de publicidade na dita paragem, respetiva operadora, lista de carreiras que passam pela paragem, código de rua, nome de rua e freguesia.

- **adjacencias.pl** - contém informação relativa às adjacências entre as paragens de autocarro, em predicados **adjacencia/3**. Cada predicado possui os gids de duas paragens adjacentes, bem como a distância entre elas, que é calculada através da fórmula da distância euclidiana entre dois pontos, tendo em conta as latitudes e longitudes das paragens.

Como foi referido em 1, desenvolvi um programa em **Python**, no ambiente **JupyterLab**, para importar os dados dos ficheiros Excel disponibilizados pelo corpo docente, fazendo uso da ferramenta de análise e manipulação de dados **Pandas**, e pré-processar os dados, retificando incoerências, antes de os escrever para ficheiros *Prolog*, onde viriam a ser a base de conhecimento.

3.1.1 Análise dos dados

O corpo docente disponibilizou dois ficheiros de dados: um com informação relativa às paragens de autocarros (nodos do grafo) e outro com informação relativa às adjacências (arcos do grafo), apresentando as paragens por carreira ordenadamente. Contudo, após a inspeção detalhada dos mesmos, concluí que o segundo ficheiro, para além das adjacências, possuía também praticamente toda a informação relativa às paragens, tornando o primeiro ficheiro praticamente obsoleto face a este.

A única informação presente exclusivamente no ficheiro dos nodos era a lista de carreiras de cada paragem, contudo este dado era incoerente, por vezes, com o ficheiro de adjacências. Logo, optei por determinar a lista das carreiras a partir do ficheiro de adjacências, acabando por não usar o ficheiro de nodos neste trabalho.

Os dados possuíam bastantes erros e incoerências:

- a omissão de certas informações de várias paragens;

- a existência de paragens repetidas na mesma carreira, em algumas folhas;

- a presença de parágrafos e vírgulas a mais em sítios onde não deviam existir - o que atrapalhava o processo de leitura do ficheiro;

- a existência de carreiras no ficheiro dos nodos para as quais não existia uma folha nos ficheiros de adjacências - 485 e 489 (as carreiras 14 e 103 também encaixam nesta descrição, mas isso não constitui um problema para estas uma vez que possuem apenas uma paragem).

Decidi não incluir a representação de conhecimento imperfeito neste trabalho prático, visto que essa matéria já foi avaliada no trabalho de grupo deste ano e só complicaria o problema atual, pelo que descarto todas as paragens que não tenham todas as informações.

3.1.2 Importação dos dados

Em primeiro lugar, procedi à geração de ficheiros de extensão **csv** a partir das folhas do ficheiro de adjacência, de maneira a poder ler os dados com o *pandas*. Para esse efeito, converti a folha de cálculo do Excel numa folha de cálculo de Excel com permissão para **Macros**, na qual executei a seguinte macro:

```
Sub exportCSV()
Dim ws As Worksheet
Dim path As String
Dim dirPath As String

dirPath = ActiveWorkbook.path & "\CSV"
MkDir dirPath

path = dirPath & "\" & Left(ActiveWorkbook.Name, InStr(ActiveWorkbook.Name, ".") - 1)
For Each ws In Worksheets
    ws.Activate
    ActiveWorkbook.SaveAs Filename:=path & "_" & ws.Name & ".csv", FileFormat:=xlCSV, CreateBackup:=False
Next
End Sub
```

Figura 1: Macro para geração de ficheiros de extensão csv

Esta macro gerou uma pasta com um ficheiro de extensão csv para cada folha do ficheiro Excel original, sendo o nome a concatenação do nome do ficheiro original com o nome da respetiva folha.

De seguida, escrevi o seguinte programa *Python*, que cria os ficheiros da base de conhecimento e lê os dados dos ficheiros csv (**read_csv**), eliminando as paragens com parâmetros em branco (**dropna()**) e as repetidas (**drop_duplicates()**).

```python
import os, glob
import math
import pandas as pd
import numpy as np

root = r"C:\Users\Hugo\Desktop\Universidade\SRCR\Avaliacao-Individual"
csv_files = glob.glob(os.path.join(root + r"\Dados\CSV","*.csv"))

paragens = open(root + r"\Prolog\paragens.pl","w",encoding="utf-8")
adjacencias = open(root + r"\Prolog\adjacencias.pl","w")

listaCSV = []
listaAdjacencias = []
dicionarioCarreiras = {}

for file in csv_files:
    carreira = pd.read_csv(file,encoding="utf-8").dropna().drop_duplicates()
    listaCSV.append(carreira)
    atualizaDicionarioCarreiras(dicionarioCarreiras,carreira)

for carreira in listaCSV:
    parser(paragens,listaAdjacencias,dicionarioCarreiras,carreira)

for adj in listaAdjacencias:
    adjacencias.write("adjacencia" + str(adj) +'.\n')

paragens.close()
adjacencias.close()
```

Figura 2: Programa Python para a criação da base de conhecimento

Aquando da leitura, o programa vai guardando as folhas de dados numa lista e atualizando um dicionário que faz a correspondência entre cada gid e as respetivas carreiras que passam por ele.

De seguida, preenche o ficheiro *Prolog* relativo às paragens com a informação lida e, à medida que faz isto, guarda os gids adjacentes e respetivas distâncias entre eles numa lista, informação com a qual preenche o ficheiro da base de conhecimento relativo às adjacências, posteriormente. O código da função **parser** pode ser consultado em A.

De seguida, é possível observar exemplos da informação representada nos dois ficheiros da base de conhecimento:

```
% Carreira 468
paragem(468, 1, 262, -107284.78, -104045.09, 'Bom', 'Fechado dos Lados', 'Yes', 'SCoTTURB', [468, 470], 1426, 'Avenida da República', 'Oeiras e São Julião da Barra, Paço de Arcos e Caxias').
paragem(468, 2, 263, -107314.88, -104013.15, 'Bom', 'Fechado dos Lados', 'Yes', 'SCoTTURB', [468, 470], 1426, 'Avenida da República', 'Oeiras e São Julião da Barra, Paço de Arcos e Caxias').
paragem(468, 3, 507, -107368.48, -103668.54, 'Bom', 'Fechado dos Lados', 'No', 'SCoTTURB', [467, 468, 479], 1426, 'Avenida da República', 'Oeiras e São Julião da Barra, Paço de Arcos e Caxias').
paragem(468, 4, 509, -107387.31, -103679.6, 'Bom', 'Fechado dos Lados', 'Yes', 'SCoTTURB', [467, 468, 479], 1426, 'Avenida da República', 'Oeiras e São Julião da Barra, Paço de Arcos e Caxias').
paragem(468, 5, 508, -107491.16, -103120.89, 'Bom', 'Fechado dos Lados', 'Yes', 'SCoTTURB', [467, 468, 479], 1426, 'Avenida da República', 'Oeiras e São Julião da Barra, Paço de Arcos e Caxias').
paragem(468, 6, 924, -107625.08, -103117.77, 'Bom', 'Fechado dos Lados', 'No', 'SCoTTURB', [467, 468, 471, 479], 1431, 'Avenida Salvador Allende', 'Oeiras e São Julião da Barra, Paço de Arcos e Caxias').
paragem(468, 7, 513, -107854.63, -102915.29, 'Bom', 'Fechado dos Lados', 'Yes', 'SCoTTURB', [467, 468, 470, 479], 1422, 'Rua da Quinta Grande', 'Oeiras e São Julião da Barra, Paço de Arcos e Caxias').
```

Figura 3: Informação representada no ficheiro paragens.pl

```
adjacencia(802, 811, 960.531).
adjacencia(811, 810, 9.964).
adjacencia(810, 841, 205.421).
adjacencia(841, 842, 16.816).
adjacencia(842, 837, 762.289).
adjacencia(837, 835, 35.132).
```

Figura 4: Informação representada no ficheiro adjacencias.pl

3.2 Requisito 1

Calcular um trajeto entre dois pontos

O primeiro requisito funcional do enunciado consistia em desenvolver um algoritmo que permitisse calcular consistentemente um trajeto entre dois pontos.

Neste sentido, eu considerei que as carreiras eram **bidirecionais**, ou seja, tanto é possível ir de A para B, como de B para A, o que torna o sistema derecomendação mais versátil.

Para este efeito, desenvolvi uma estratégia de pesquisa informada que segue a seguinte lógica:

1. O nodo atual encontra-se na mesma carreira que o destino:
    - Se houver apenas uma carreira com as duas paragens, compara os *cids* (posições na carreira) das paragens e calcula a adjacente ao nodo atual, no sentido do nodo destino.
    - Se houverem várias carreiras com as duas paragens, determina em qual das carreiras percorre menos paragens até ao destino e seleciona o adjacente nessa carreira.
2. O nodo atual e o destino encontram-se em carreiras diferentes:
    - Através da lista de carreiras de cada paragens, faz uma travessia *breadth-first* às carreiras, partindo das carreiras da paragem atual, determinando vários conjuntos de carreiras diferentes a partir dos quais é possível atingir o destino;
    - Não calcula todas as combinações possíveis porque isso seria muito exigente para a *stack* do programa; em vez disso, vai mantendo uma lista com as carreiras já visitadas ao longo da travessia, pelo que se, por exemplo, uma carreira 1 se cruzar com as carreiras 2 e 3 e ambas estas, por sua vez, estiverem ligadas a uma carreira 4, apenas uma delas (a primeira a ser verificada) incorporará a carreira 4 no seu conjunto.
    - Vai seguindo a lista de carreiras mais curta, calculando o ponto de interseção entre a carreira atual e a próxima, em cada iteração, e avançando para o nodo adjacente nesse sentido.
3. É impossível chegar ao destino a partir do nodo atual:
    - O algoritmo verifica que não se encontram na mesma carreira e tenta calcular o conjunto de carreiras entre eles, que falha também, pelo que retorna **no**, indicando que não é possível construir o percurso pretendido.

Uma vez que o algoritmo calcula um percurso direto da origem para o destino, em ambos os cenários possíveis, nunca entrará em ciclos ou por caminhos errados, pelo que não há necessidade de manter uma lista com os nodos visitados ao longo da travessia nem de alguma vez recuar no caminho (*backtracking*), ao calcular um percurso sem qualquer restrições.

Veremos que isto não é o caso em alguns dos outros requisitos mas, para este requisito, o processo encontra-se bastante otimizado, sendo as respostas sempre instantâneas.

O predicado elaborado para executar este requisito é o seguinte:

```
% Calcula o percurso com uma estrategia de pesquisa informada e verifica atalhos
percursoInformadoFinal(Origem,Destino,Tipo,Restricoes,PercursoFinal) :-
    percursoInformado(Origem,(Origem,Destino),[],[Origem],Tipo,Restricoes,Percurso),
    atalhos(Percurso,Percurso,PercursoFinal).

% Requisito 1: Calcular um trajeto entre dois pontos
percurso(Origem,Destino,Percurso):-
    percursoInformadoFinal(Origem,Destino,normal,_,Percurso).
```

Figura 5: Predicado do requisito 1

O predicado calcula um percurso através desta estratégia de pesquisa informada, cujo numero de variaveis varia conforme a estrategia que estiver a ser adotada - carreira do destino ou conjunto de carreiras ate ao destino -, invocando o respetivo predicado de escolha da proxima paragem e processando o resultado, isto é, verificando se é uma paragem válida e pode iterar o percurso normalmente, ou se precisa de calcular/mudar de caminho de carreiras, realizando a operação respetiva.

```
% Calcula o percurso da Origem ao Destino, se for possivel chegar ao Destino partindo da Origem
% Caso contrario devolve "no"
% Se uma das carreiras da Origem passar no Destino, calcula o percurso mais curto em paragens ate ao Destino
% Se nenhuma das carreiras da Origem passar no Destino, calcula os conjuntos de carreiras a partir dos quais
% e possivel chegar ao destino e faz o percurso pelo menor
percursoInformado(Destino,(_,Destino),_,PercursoInverso,_,_,Percurso) :- reverse(PercursoInverso,Percurso).

percursoInformado(Gid,(Origem,Destino),GidsProibidos,PercursoAtual,TipoTravessia,Restricoes,Percurso):-
    escolheProxParagem(Gid,Destino,GidsProibidos,TipoTravessia,Restricoes,ProxParagem),!,
    processaProxParagem(ProxParagem,(Origem,Destino),GidsProibidos,PercursoAtual,TipoTravessia,Restricoes,Percurso).

percursoInformado(Destino,(_,Destino),_,_,PercursoInverso,_,_,Percurso) :- reverse(PercursoInverso,Percurso).

percursoInformado(Gid,(Origem,Destino),GidsProibidos,CaminhosCarreiras,PercursoAtual,TipoTravessia,Restricoes,Percurso):-
    escolheProxParagem(Gid,Destino,GidsProibidos,CaminhosCarreiras,TipoTravessia,Restricoes,ProxParagem),!,
    processaProxParagem(ProxParagem,(Origem,Destino),GidsProibidos,PercursoAtual,TipoTravessia,Restricoes,Percurso).
```

Figura 6: Predicado de pesquisa informada

Estes dois predicados serão disponibilizados em A para consulta e não serão abordados em detalhe aqui porque são bastante complexos e a lógica por detrás já foi explicada acima.

e, de seguida, verifica se é possível realizar "atalhos"no resultado obtido. Por exemplo, se uma carreira passar nas paragens 1, 2 e 3, por esta ordem, e outra carreira passar na 1 e 3, é possível atalhar da paragem 1 para a 3, caso o percurso possua também a 2, reduzindo o tamanho do caminho e otimizando-o ainda mais, desta maneira.

```
atalhosAux(_,[],[]).
atalhosAux(H,[X|T],[X|T]) :- adjacencia(H,X,_); adjacencia(X,H,_).
atalhosAux(H,[_|T],L) :- atalhosAux(H,T,L).

proxAtalhos([_,H2|T],Temp,[],L) :- atalhos([H2|T],Temp,L).
proxAtalhos(_,_,Lista,L) :- atalhos(Lista,Lista,L).

atalhos([H1,H2],_,[H1,H2]).
atalhos([H1,H2|T],Temp,[H1|L]) :- atalhosAux(H1,T,Lista), proxAtalhos([H1,H2|T],Temp,Lista,L).
```

Figura 7: Predicado que verifica atalhos num percurso

Assim, o predicado **percurso** elaborado é bastante eficiente, calculando qualquer percurso possível basicamente instantâneamente, como pode ser observado em 3.8.

3.3 Requisitos 2, 3, 7 e 8

- Selecionar apenas algumas das operadoras de transporte para um determinado percurso
- Excluir uma ou mais operadoras de transporte para o percurso
- Escolher o percurso que passe apenas por abrigos com publicidade
- Escolher o percurso que passe apenas por paragens abrigadas

Todos estes requisitos consistem em calcular um percurso entre a origem e o destino dados, impondo uma certa restrição ao algoritmo:

- Requisito 2 - indica-se o conjunto das únicas operadoras que se podem usar, ou seja, o percurso calculado só passa por paragens cuja operadora pertence ao conjunto fornecido ao predicado;
- Requisito 3 - indica-se o conjunto das operadoras que não se podem usar, ou seja, o percurso calculado só passa por paragens cuja operadora não pertence ao conjunto fornecido ao predicado;
- Requisito 7 - o percurso só pode conter paragens com publicidade;
- Requisito 8 - o percurso só pode conter paragens abrigadas - abertas ou fechadas dos lados.

Para este efeito, aproveito a estratégia de pesquisa informada elaborada para o requisito 1, verificando se a operadora do nodo atual obedece à restrição imposta, a cada iteração do algoritmo. Caso a paragem atual não seja válida, o predicado seguinte vai retornar uma lista vazia, indicando que não é possível continuar a travessia atual por nenhuma carreira.

```
paragensGid(Gid,selecionarOperadoras,Restricoes,Pares) :-
    findall((Carreira,Cid), (paragem(Carreira,Cid,Gid,_,_,_,_,_,Operadora,_,_,_,_),
                             pertence(Operadora,Restricoes)), Pares).

paragensGid(Gid,excluirOperadoras,Restricoes,Pares) :-
    findall((Carreira,Cid), (paragem(Carreira,Cid,Gid,_,_,_,_,_,Operadora,_,_,_,_),
                             \+ pertence(Operadora,Restricoes)), Pares).

paragensGid(Gid,paragensAbrigadas,_,Pares) :-
    findall((Carreira,Cid), (paragem(Carreira,Cid,Gid,_,_,_,TipoAbrigo,_,_,_,_,_,_),
                             TipoAbrigo \= 'Sem Abrigo'), Pares).

paragensGid(Gid,abrigosComPublicidade,_,Pares) :-
    findall((Carreira,Cid), paragem(Carreira,Cid,Gid,_,_,_,_,'Yes',_,_,_,_,_), Pares).

paragensGid(Gid,normal,_,Pares) :-
    findall((Carreira,Cid), paragem(Carreira,Cid,Gid,_,_,_,_,_,_,_,_,_,_), Pares).
```

Figura 8: Verificação da condição dos requisitos no nodo atual

O predicado responsável por calcular o percurso recebe um argumento que indica o **tipo** de travessia que se pretende executar - **normal**, **selecionarOperadoras**, **excluirOperadoras**, **paragensAbrigadas** ou **abrigosComPublicidade** - e outro que constitui as restrições da dita travessia - no caso dos requisitos das operadoras, será a lista das operadoras indicadas, caso contrário não há nenhuma lista, pelo que é passado um apontador (_) neste argumento, dado que não será necessário. Passa depois estes argumentos ao predicado mostrado acima para fazer a verificação.

Se o primeiro percurso calculado (carreira em comum à origem e destino ou conjunto de carreiras mais pequeno calculado) não possuir nenhuma paragem que desobedeça à restrição, o resultado será igual ao obtido ao correr o predicado do requisito 1 com a origem e o destino em questão.

Se não for o caso, o predicado reinicia o percurso da origem, tentando fazê-lo pelos outros conjuntos de carreiras a partir das quais é possível chegar ao destino (sempre que um conjunto de carreiras falha, volta à origem e tenta o seguinte), verificando a restrição em cada nodo.

Caso não seja possível atingir o destino por nenhum dos conjuntos de carreiras calculado, o predicado devolve **no**, indicando que é impossível atingir o destino a partir da origem, obedecendo à restrição imposta.

```
% Requisito 2: Selecionar apenas algumas das operadoras de transporte para um determinado percurso
selecionarOperadoras(Origem,Destino,Operadoras,Percurso) :-
    percursoInformadoFinal(Origem,Destino,selecionarOperadoras,Operadoras,Percurso).

% Requisito 3: Excluir uma ou mais operadoras de transporte para o percurso
excluirOperadoras(Origem,Destino,Operadoras,Percurso) :-
    percursoInformadoFinal(Origem,Destino,excluirOperadoras,Operadoras,Percurso).

% Requisito 7: Escolher o percurso que passe apenas por abrigos com publicidade
abrigosComPublicidade(Origem,Destino,Percurso) :-
    percursoInformadoFinal(Origem,Destino,abrigosComPublicidade,_,Percurso).

% Requisito 8: Escolher o percurso que passe apenas por paragens abrigadas
paragensAbrigadas(Origem,Destino,Percurso) :-
    percursoInformadoFinal(Origem,Destino,paragensAbrigadas,_,Percurso).
```

Figura 9: Predicados do requisitos 2, 3, 7 e 8

3.4 Requisito 4

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

O predicado do quarto requisito recebe umm percurso e devolve uma lista com os gids ordenados por ordem decrescente de carreiras que passam por eles, associados ao número das mesmas.

```
% Requisito 4: Identificar quais as paragens com o maior
% número de carreiras num determinado percurso
paragensMaisCarreiras(Percurso,Pares) :-
    contaCarreiras(Percurso,ParesInv),
    keysort(ParesInv,ParesInvDecresc),
    reverse(ParesInvDecresc,ParesInvCresc),
    invertePares(ParesInvCresc,Pares).
```

Figura 10: Predicado do requisito 4

Como podemos observar na imagem acima, o predicado começa por contar as carreiras de cada paragem do percurso, através do predicado auxiliar **contaCarreiras**, devolvendo uma lista com os pares (número de carreiras, gid):

```
% Conta o numero de carreiras que passam pela paragem do gid
contaCarreiras([],[]).
contaCarreiras([Gid|T],[N-Gid|L]) :-
    paragem(_,_,Gid,_,_,_,_,_,_,Carreiras,_,_,_),
    length(Carreiras,N),
    contaCarreiras(T,L).
```

Figura 11: Predicado auxiliar contaCarreiras

De seguida, usa o **keysort** do *SWI* para ordenar a lista por ordem crescente segundo o primeiro elemento dos pares, ou seja, o número de carreiras, e inverte a lista para ficar com os elementos por ordem descrescente.

Por fim, invoca o predicado auxiliar **invertePares** para trocar os elementos de cada par, apresentando ao utilizador os gids do percurso introduzido ordenados por ordem decrescente de número de carreiras, e ainda com o número de carreiras de cada gid.

```
invertePares([],[]).
invertePares([N-Gid|T],[Gid-N|L]) :- invertePares(T,L).
```

Figura 12: Predicado auxiliar invertePares

3.5 Requisito 5

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

Para calcular o percurso mais curto entre dois pontos, foi fulcral a estratégia de pesquisa informada desenvolvida para o requisito 1, pois devolve qualquer caminho numa questão de milissegundos. O raciocínio é o seguinte:

- Calculo um caminho inicial entre os pontos dados com o algoritmo informado e determino o seu comprimento;
- Tento determinar um novo caminho através de um algoritmo não informado com, no máximo, metade do comprimento do primeiro;
- Se for possível, guardo o novo percurso e volto a tentar calcular um caminho com metade das iterações deste;
- Caso contrário, aumento o limite de iterações imposto em 50% (por exemplo, 20 passaria para 30 (20 + 20/2));
- Desta maneira, o algoritmo converge rapidamente para o número mínimo de paragens que é possível ter no percurso, devolvendo esse caminho no fim do processo.

De seguida, é apresentada a estratégia de pesquisa não-informada implementada, que se rege pelo seguinte princípio: o nodo atual ou é adjacente ao destino, ou é adjacente a um nodo que tem ligação ao destino.

```
percursoMaxIter(_,_,_,Max,Max,_) :- !,fail.

percursoMaxIter(X,Y,_,_,_,[X,Y]) :- adjacencia(Y,X,_).
percursoMaxIter(X,Y,2,_,_,[X,Y]) :- adjacencia(X,Y,_).

percursoMaxIter(X,Y,N,Iter,Max,[X|P]) :- adjacencia(Z,X,_), IterInc is Iter+1, percursoMaxIter(Z,Y,N,IterInc,Max,P).
percursoMaxIter(X,Y,2,Iter,Max,[X|P]) :- adjacencia(X,Z,_), IterInc is Iter+1, percursoMaxIter(Z,Y,2,IterInc,Max,P).
```

Figura 13: Predicado de pesquisa não-informada do percurso mais curto

O predicado incrementa um contador a cada iteração e vai comparando-o com o máximo de elementos indicado - se o superar, falha, sendo obrigado a dar *backtrack* e experimentar um caminho diferente. Se esgotar todos os caminhos possíveis sem atingir o destino naquele número de iterações, falha, dado que é impossível realizar o pedido.

Em cada iteração, verifica se o nodo atual é adjacente ao destino. Caso não seja, vai buscar o próximo adjacente e repete o processo para o mesmo. O algoritmo funciona nos dois sentidos, verificando primeiro a adjacência para "trás", ou seja, na ordem contrária à das listas de adjacências fornecidas pelo corpo docente, e, se falhar, para a "frente".

O terceiro argumento será explicado mais abaixo.

O predicado **maisCurto** implementa a lógica de *binary search tree* explicada acima, aumentando ou reduzindo o limite máximo de iterações conforme o resultado do **percursoMaxIter**:

```
maisCurto(Inicial,Final,NSentidos,Inf,_,_,Percurso) :-
    percursoMaxIter(Inicial,Final,NSentidos,1,Inf,P),
    metade(Inf,Metade),
    maisCurto(Inicial,Final,NSentidos,Metade,Inf,P,Percurso).

maisCurto(_,_,_,Inf,Sup,Percurso,Percurso) :-
    SupDec is Sup-1, Inf =:= SupDec, !.

maisCurto(Inicial,Final,NSentidos,Inf,Sup,PercursoTemp,Percurso) :-
    Dist is Sup-Inf,
    metade(Dist,Metade), InfMaisMetade is Inf+Metade,
    maisCurto(Inicial,Final,NSentidos,InfMaisMetade,Sup,PercursoTemp,Percurso).
```

Figura 14: Predicado maisCurto

Através do teste dos predicados elaborados e da análise cuidada de resultados, cheguei a várias conclusões:

- Aparentemente, se o percurso resultante do algoritmo informado, que já é bastante otimizado, for uma lista **unidirecional**, o caminho mais curto será também unidirecional;
- O algoritmo é muito mais rápido e eficiente se o caminho mais curto for unidirecional, uma vez que terá muitos menos percursos para testar;
- O algoritmo converge muito mais rapidamente a determinar os caminhos para "trás"(no sentido oposto ao das listasde adjacências) do que para a "frente";
- O algoritmo converge mais rapidamente se começar com um caminho ligeiramente maior do que o mais curto que é possível determinar com o predicado **percurso**.

Em prol destas conclusões, foram implementadas várias otimizações no código de maneira a tornar o processo mais rápido e eficiente. De seguida, é apresentado o predicado final deste requisito:

```
% Requisito 5: Escolher o menor percurso (usando critério menor número de paragens)
percursoMaisCurto(Origem,Destino,Percurso) :-
    percursoInformado(Origem,(Origem,Destino),[],[Origem],normal,_,P1), length(P1,N1),
    percursoInformado(Destino,(Destino,Origem),[],[Destino],normal,_,P2), length(P2,N2),
    escolhePercursoInicial(Origem,Destino,N1,N2,P1,P2,(N,P,Inicial,Final)),
    mudancasSentido(P,ParagensMudanca),
    metade(N,Metade),
    uniOuBidirecional(Inicial,Final,Metade,N,P,ParagensMudanca,PercursoDesord),
    ordenaPercurso(Origem,PercursoDesord,Percurso).
```

Figura 15: Predicado do requisito 5

Em primeiro lugar, como o algoritmo converge mais rápido ao começar com um caminho ligeiramente maior, começa-se por calcular os percursos da origem ao destino e do destino à origem, através do predicado **percursoInformado** em vez do predicado **percurso**, que faz a otimização final de tomar os atalhos possíveis no percurso calculado, diminuindo o seu tamanho. Calcula-se os comprimentos de ambos e seleciona-se o maior percurso dos dois, através do predicado auxiliar **escolhePercursoInicial**, para começar a procura binária.

```
escolhePercursoInicial(O,D,N1,N2,_,P,(N2,PInv,O,D)) :- N1 < N2, !, reverse(P,PInv).
escolhePercursoInicial(O,D,N1,N2,P,_,(N1,PInv,D,O)) :- N1 > N2, !, reverse(P,PInv).
escolhePercursoInicial(_,_,N,_,P1,P2,(N,P,Inicial,Final)) :- sentidoTras(P1,P2,(P,Inicial,Final)).

sentidoTras([Origem,H|_],[Destino|T2],([Destino|T2],Destino,Origem)) :- adjacencia(Origem,H,_).
sentidoTras([Origem|T1],[Destino|_],([Origem|T1],Origem,Destino)).
```

Figura 16: Predicados auxiliares escolhePercursoInicial e sentidoTras

Como é possível observar na imagem 13, o algoritmo explora as adjacências para "trás"em primeiro lugar, uma vez que converge mais rápido dessa maneira. O predicado de pesquisa informada segue a mesma norma, pelo que o caminho menor entre os dois iniciais calculados (Origem->Destino, Destino->Origem) será aquele em que se anda para "trás".

Desta maneira, o predicado **escolhePercursoInicial** encarrega-se ainda de devolver a origem e o destino na ordem pela qual são o percurso de um para o outro é feito neste sentido (se Destino estiver mais à "frente"na carreira da Origem, devolve (Destino,Origem) por esta ordem).

Por fim, como o caminho mais curto normalmente é unidirecional se o caminho calculado pelo algoritmo informado também o for, e o algoritmo não-infomado converge mais rapidamente para algoritmos unidirecionais, tomei a decisão de impingir essa restrição. Como tal, depois de escolher o percurso inicial, o predicado auxiliar **mudancasSentido** identifica todos os gids em que houve uma mudança de sentido no percurso.

```
mudancasSentido([_],_,[]).

mudancasSentido([H1,H2|T],frente,R) :- adjacencia(H1,H2,_), mudancasSentido([H2|T],frente,R).
mudancasSentido([H1,H2|T],frente,[H1|R]) :- mudancasSentido([H2|T],tras,R).

mudancasSentido([H1,H2|T],tras,R) :- adjacencia(H2,H1,_), mudancasSentido([H2|T],tras,R).
mudancasSentido([H1,H2|T],tras,[H1|R]) :- mudancasSentido([H2|T],frente,R).

mudancasSentido([H1,H2|T],R) :- adjacencia(H1,H2,_), mudancasSentido([H2|T],frente,R).
mudancasSentido([_,H2|T],R) :- mudancasSentido([H2|T],tras,R).
```

Figura 17: Predicado auxiliar mudancasSentido

Se não houver nenhuma mudança de sentido, o algoritmo não-informado é restringido a um sentido (para "trás") de maneira a otimizar bastante o tempo de execução. Caso contrário, funciona em ambos os sentidos. Isto é implementado através do predicado **uniOuBidirecional**, que indica o número de sentidos (1 ou 2) no terceiro argumento do **percursoMaxIter**.

```
uniOuBidirecional(Inicial,Final,Inf,Sup,Temp,[],Percurso) :-
    maisCurto(Inicial,Final,1,Inf,Sup,Temp,Percurso).
uniOuBidirecional(Inicial,Final,Inf,Sup,Temp,_,Percurso) :-
    maisCurto(Inicial,Final,2,Inf,Sup,Temp,Percurso).
```

Figura 18: Predicado auxiliar uniOuBidirecional

Com este sistema, é possível que o caminho mais curto retornado para um caminho unidirecional não seja efetivamente o mais curto, contudo decidi manter assim por razões de performance, uma vez que não tive tempo de otimizar mais o algoritmo bidirecional. Algo que poderia eventualmente ser feito era calcular o caminho unidirecional mais curto e, a partir desse, ir decrementando o número de iterações e procurando caminhos bidirecionais, de maneira a verificar se existia algum ainda mais curto, uma vez que julgo que, caso fosse possível, seria muito pouca a diferença no tamanho dos caminhos.

Devido à possível alteração da ordem da origem e do destino na invocação do algoritmo não-informado, no fim é necessário verificar se a ordem da lista está de acordo com os argumentos originais, invertendo-a se for o caso.

```
ordenaPercurso(Origem,[Origem|Percurso],[Origem|Percurso]).
ordenaPercurso(_,PercursoInv,Percurso) :- reverse(PercursoInv,Percurso).
```

Figura 19: Predicado auxiliar ordenaPercurso

3.6 Requisito 6

Escolher o percurso mais rápido (usando critério da distância)

Foi a pensar neste requisito que foi tomada a decisão de guardar as distâncias entre nodos nos predicados de adjacência que se encontram na base de conhecimento. Para calcular o caminho mais próximo, foi implementado o seguinte raciocínio:

- Calcula-se o percurso mais próximo possível entre a origem e o destino através do predicado **percurso**, calculando as distâncias dos dois caminhos obtidos e selecionando o mais pequeno;
- Recorrendo a uma estratégia não-informada, verifica-se se é possível obter um caminho mais próximo, ou seja, cuja distância total seja inferior à distância do primeiro caminho selecionado;
- Se for possível, guarda-se o novo percurso e repete-se o raciocínio com a distância total deste;
- Caso contrário, já foi encontrado o caminho mais próximo, pelo que se devolve o mesmo.

```
% Requisito 6: Escolher o percurso mais rápido (usando critério da distância)
percursoMaisProximo(Origem,Destino,Percurso) :-
    percurso(Origem,Destino,P1), calculaDistancia(P1,N1),
    percurso(Destino,Origem,P2), calculaDistancia(P2,N2),
    escolhePercursoInicial2(Origem,Destino,N1,N2,P1,P2,(N,P,Inicial,Final)),
    maisProximo(Inicial,Final,N,P,Percurso).
```

Figura 20: Predicado do requisito 6

A distância total de um percurso é calculada através do predicado **calculaDistancia**:

```
calculaDistancia([_],0).
calculaDistancia([H1,H2|T],D) :-
    (adjacencia(H1,H2,DD); adjacencia(H2,H1,DD)),
    calculaDistancia([H2|T],D1), D is D1+DD.
```

Figura 21: Predicado auxiliar calculaDistancia

O predicado **maisProximo** implementa a recursividade explicada acima, procurando o caminho mais próximo enquanto conseguir:

```
maisProximo(Inicial,Final,Max,Temp,Percurso) :-
    percursoMaxDist(Inicial,Final,0,Max,Dist,P),
    maisProximo(Inicial,Final,Dist,P,Percurso).
maisProximo(_,_,_,Percurso,Percurso).
```

Figura 22: Predicado maisProximo

Por fim, o predicado que implementa a estratégia de pesquisa não-informada para este requisito é apresentado a seguir. A cada iteração, o predicado verifica se a distância acumulada até ao nodo atual já superou o limite ou não, falhando, se for o caso, e fazendo *backtracking* para testar outros caminhos. Caso contrário, verifica se o nodo atual é adjacente do destino e, em caso afirmativo, certifica-se ainda que a distância total é inferior ao limite dado, falhando se não for o caso.

Caso o nodo atual também não seja adjacente ao destino, avança para o seguinte nodo adjacente, somando a distância entre eles ao acumulador, procurando explorar primeiro os adjacentes para "trás".

```
comparaDistancias(Dist,Max) :- Dist < Max, !.
comparaDistancias(_,_) :- !,fail.

percursoMaxDist(_,_,Dist,Max,_,_) :- Dist > Max, !, fail.

percursoMaxDist(X,Y,Dist,Max,DistFinal,[X,Y]) :- adjacencia(Y,X,D), DistFinal is Dist+D, comparaDistancias(DistFinal,Max).
percursoMaxDist(X,Y,Dist,Max,DistFinal,[X,Y]) :- adjacencia(X,Y,D), DistFinal is Dist+D, comparaDistancias(DistFinal,Max).

percursoMaxDist(X,Y,Dist,Max,DistFinal,[X|P]) :- adjacencia(Z,X,D), DistNew is Dist+D, percursoMaxDist(Z,Y,DistNew,Max,DistFinal,P).
percursoMaxDist(X,Y,Dist,Max,DistFinal,[X|P]) :- adjacencia(X,Z,D), DistNew is Dist+D, percursoMaxDist(Z,Y,DistNew,Max,DistFinal,P).
```

Figura 23: Predicado de pesquisa não-informada do caminho mais próximo

Contudo, o predicado deste requisito não funciona, na prática, porque a primeira linha do predicado **percursoMaxDist** não consegue forçar a falha por negação com o *cut, fail*, uma vez que estes vêm a seguir a uma comparação aritmética e não aparecem isolado a seguir ao **if**. Não fui capaz de corrigir este erro, o que é bastante frustrante uma vez que o requisito está todo feito, mas não funciona por causa deste pormenor.

3.7 Requisito 9

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

No o nono e útltimo requisito, é especificada a origem e o destino do percurso a calcular, bem como um conjunto de paragens intermédias pelas quais o percurso deve passar.

Neste requisito, considerei que o percurso devia passar pelas paragens intermédias pela ordem em que estas aparecem na lista que é dada.

Como tal, a estratégia desenvolvida passa por adicionar a origem ao início da lista das paragens intermédias e o destino ao fim, e calcular o percurso entre cada par de gids consecutivos na lista resultante, através do predicado **percurso** do requisito 1, concatenando todos os percursos calculados:

```
% Requisito 9: Escolher um ou mais pontos intermédios por onde o percurso deverá passar
paragensIntermedias(Origem,Destino,Intermedias,Percurso) :-
    reverse([Origem|Intermedias],L1), reverse([Destino|L1],L2),
    paragensIntermediasAux(L2,[],Percurso).

% Calcula um percurso que passa por todas as paragens da primeira lista
% Se nao for possivel, devolve "no"
paragensIntermediasAux([_],PercursoInvertido,Percurso) :- reverse(PercursoInvertido,Percurso).
paragensIntermediasAux([H1,H2|T],PercursoAtual,Percurso) :-
    percurso(H1,H2,PartePercurso),!,
    length(PartePercurso,N), N > 0,
    concatPartesPercurso(PartePercurso,PercursoAtual,1,P),
    paragensIntermediasAux([H2|T],P,Percurso).
```

Figura 24: Predicado do requisito 9

A concatenação dos percursos é feita com recurso ao predicado auxiliar **concatPartesPercurso**, que tem o cuidado de retirar a cabeça de todos os percursos exceto o primeiro (se calcularmos os percursos 1->2 e 2->3, os resultados serão [1,...,2] [2,...,3], logo não podemos concatenar estes percursos diretamente, senão ficariamos com um elemento repetido). O predicado devolve a lista final pela ordem contrária, pelo que o predicado **paragensIntermediasAux** inverte a lista final.

```
% [1 2 3][] -> [3 2 1]
% [4 5 6][4 3 2 1] -> [6 5 4 3 2 1]
concatPartesPercurso([],R,_,R).
concatPartesPercurso([H|T],[],1,R) :- concatPartesPercurso(T,[H],0,R).
concatPartesPercurso([_|T],L,1,R) :- concatPartesPercurso(T,L,0,R).
concatPartesPercurso([H|T],L,0,R) :- concatPartesPercurso(T,[H|L],0,R).
```

Figura 25: Predicado auxiliar concatPartesPercurso

3.8 Análise de Resultados

Nesta secção serão demonstrados testes temporizados dos predicados desenvolvidos para os vários requisitos.

3.8.1 Requisito 1

```
4 ?- time(percurso(583,681,R)).
% 15,213 inferences, 0.000 CPU in 0.007 seconds (0% CPU, Infinite Lips)
false.
```

Figura 26: Percurso não existente

```
3 ?- time(percurso(183,79,R)).
% 9,053 inferences, 0.000 CPU in 0.003 seconds (0% CPU, Infinite Lips)
R = [183, 791, 595, 594, 185, 107, 250, 597, 953, 609, 599, 40, 622, 51, 38, 620, 45, 602,
601, 48, 49, 612, 613, 611, 610, 336, 357, 334, 339, 347, 86, 85, 341, 342, 365, 366, 460,
468, 486, 487, 488, 469, 462, 480, 494, 957, 465, 186, 466, 467, 78, 79] .
```

Figura 27: Percurso com origem e destino na mesma carreira

```
2 ?- time(percurso(712,583,R)).
% 133,733 inferences, 0.031 CPU in 0.039 seconds (80% CPU, 4279456 Lips)
R = [712, 713, 714, 128, 745, 736, 147, 156, 161, 162, 172, 171, 799, 1010, 227, 230, 234,
224, 226, 232, 52, 233, 231, 241, 240, 859, 858, 360, 313, 323, 351, 352, 339, 347, 86, 85,
 341, 342, 345, 363, 335, 457, 458, 490, 491, 56, 655, 654, 59, 57, 58, 62, 63, 64, 61, 60,
 32, 33, 364, 330, 845, 846, 333, 367, 857, 856, 863, 353, 354, 983, 986, 977, 1002, 954, 9
52, 823, 818, 807, 710, 792, 947, 178, 693, 692, 817, 789, 169, 1016, 1015, 708, 738, 690,
83, 84, 157, 158, 739, 743, 151, 733, 732, 780, 751, 775, 776, 777, 778, 762, 756, 208, 797
, 191, 795, 796, 828, 284, 285, 282, 283, 281, 294, 378, 989, 869, 541, 542, 503, 516, 543,
 10, 540, 538, 310, 521, 505, 501, 577, 944, 969, 579, 581, 941, 576, 585, 584, 583] .
```

Figura 28: Percurso com origem e destino em carreiras diferentes

3.8.2 Requisitos 2, 3, 7 e 8

Como estes requisitos são todos análogos, vou demonstrar os testes de apenas um deles.

Para testar um caminho impossível com as operadoras dadas, mudei a operadora da paragem 627 para LT, e corri o seguinte teste:

Figura 29: Representação do percurso impossível com as operadoras dadas

```
2 ?- time(selecionarOperadoras(603,978,['Carris'],R)).
% 308 inferences, 0.000 CPU in 0.002 seconds (0% CPU, Infinite Lips)
false.
```

Figura 30: Percurso impossível com as operadoras dadas

O seguinte teste é de um caminho que é possível obter diretamente sem quaisquer complicações:

```
2 ?- time(selecionarOperadoras(775,583,['LT'],R)).
% 6,524 inferences, 0.000 CPU in 0.007 seconds (0% CPU, Infinite Lips)
R = [775, 776, 777, 778, 762, 756, 208, 797, 191, 795, 796, 828, 284, 285, 282, 283, 281,
294, 378, 989, 869, 541, 542, 503, 516, 543, 10, 540, 538, 310, 521, 505, 501, 577, 944, 9
69, 579, 581, 941, 576, 585, 584, 583]
```

Figura 31: Percurso direto com as operadoras dadas

Para testar um caminho em que não fosse possível atingir o destino logo no primeiro percurso escolhido, mudei a operadora da paragem 264 para LT e testei o seguinte:

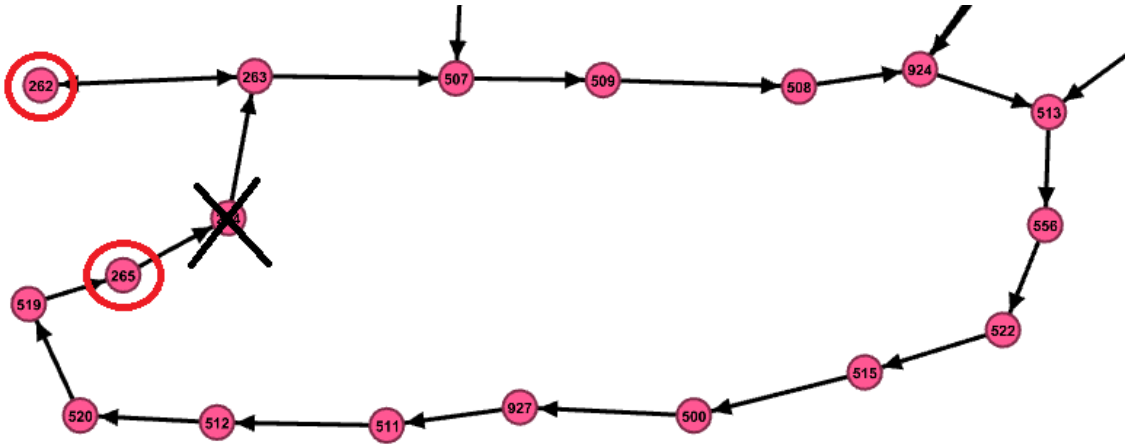

Figura 32: Representação do percurso indireto com as operadoras dadas

```
4 ?- time(selecionarOperadoras(262,265,['SCoTTURB'],R)).
% 3,765 inferences, 0.000 CPU in 0.002 seconds (0% CPU, Infinite Lips)
R = [262, 263, 507, 509, 508, 924, 513, 556, 522, 515, 500, 927, 511, 512, 520, 519, 265] .
```

Figura 33: Percurso indireto com as operadoras dadas

3.8.3 Requisito 4

```
3 ?- time(paragensMaisCarreiras([712, 713, 714, 128, 745, 736, 147, 156, 161, 162, 172, 17
1, 799, 599, 1001, 607, 232, 52, 233, 231, 886, 473, 470, 483, 482, 476, 904, 472, 902, 89
3, 465, 186, 466, 467, 78, 79],L)).
% 220 inferences, 0.000 CPU in 0.000 seconds (?% CPU, Infinite Lips)
L = [599-7, 231-4, 233-4, 52-4, 232-4, 799-4, 147-4, 736-4, 745-4, 78-3, 171-3, 172-3, 162
-3, 161-3, 156-3, 467-2, 466-2, 186-2, 465-2, 607-2, 1001-2, 128-2, 79-1, 893-1, 902-1, 47
2-1, 904-1, 476-1, 482-1, 483-1, 470-1, 473-1, 886-1, 714-1, 713-1, 712-1] .
```

Figura 34: Aplicação do predicado do requisito 4

3.8.4 Requisito 5

```
2 ?- time(percursoMaisCurto(183,79,R)).
% 3,105,045 inferences, 0.766 CPU in 0.759 seconds (101% CPU, 4055569 Lips)
R = [183, 791, 595, 594, 185, 107, 250, 597, 953, 609, 599, 1001, 607, 232, 52, 233, 231,
886, 473, 470, 483, 482, 476, 904, 472, 902, 893, 465, 186, 466, 467, 78, 79] .

3 ?- time(percursoMaisCurto(712,79,R)).
% 5,902,181 inferences, 1.453 CPU in 1.451 seconds (100% CPU, 4061716 Lips)
R = [712, 713, 714, 128, 745, 736, 147, 156, 161, 162, 172, 171, 799, 599, 1001, 607, 232,
 52, 233, 231, 886, 473, 470, 483, 482, 476, 904, 472, 902, 893, 465, 186, 466, 467, 78, 7
9] .

4 ?- time(percursoMaisCurto(219,583,R)).
% 4,279,161 inferences, 1.063 CPU in 1.079 seconds (98% CPU, 4027446 Lips)
R = [219, 220, 221, 300, 301, 308, 307, 320, 317, 319, 318, 299, 273, 274, 431, 437, 388,
387, 386, 385, 389, 440, 558, 554, 540, 538, 310, 521, 505, 501, 577, 944, 969, 579, 581,
941, 576, 585, 584, 583] .
```

Figura 35: Percursos mais curtos unidirecionais

```
6 ?- time(percursoMaisCurto(746,97,R)).
% 3,020,295 inferences, 0.781 CPU in 0.776 seconds (101% CPU, 3865978 Lips)
R = [746, 747, 165, 163, 164, 761, 753, 721, 133, 722, 727, 726, 716, 717, 98, 97] .
```

Figura 36: Percurso mais curto bidirecional

3.8.5 Requisito 9

```
4 ?- time(paragensIntermedias(183,79,[631],R)).
% 91,586 inferences, 0.047 CPU in 0.060 seconds (78% CPU, 1953835 Lips)
false.
```

Figura 37: Percurso impossível com paragens intermédias

```
8 ?- percurso(183,79,R).
R = [183, 791, 595, 594, 185, 107, 250, 597, 953, 609, 599, 40, 622, 51, 38, 620, 45, 602,
 601, 48, 49, 612, 613, 611, 610, 336, 357, 334, 339, 347, 86, 85, 341, 342, 365, 366, 460
, 468, 486, 487, 488, 469, 462, 480, 494, 957, 465, 186, 466, 467, 78, 79] .

9 ?- time(paragensIntermedias(183,79,[628,712],R)).
% 121,169 inferences, 0.047 CPU in 0.051 seconds (92% CPU, 2584939 Lips)
R = [183, 791, 595, 594, 185, 107, 250, 597, 953, 609, 82, 604, 628, 39, 50, 599, 609, 953
, 597, 250, 107, 89, 185, 594, 595, 791, 183, 170, 788, 68, 1014, 709, 154, 87, 687, 686,
742, 741, 155, 159, 734, 149, 153, 147, 736, 745, 128, 714, 713, 712, 713, 714, 128, 745,
736, 147, 153, 149, 734, 159, 155, 741, 742, 686, 687, 87, 154, 709, 1014, 68, 788, 170, 1
83, 791, 595, 594, 185, 107, 250, 597, 953, 609, 599, 40, 622, 51, 38, 620, 45, 602, 601,
48, 49, 612, 613, 611, 610, 336, 357, 334, 339, 347, 86, 85, 341, 342, 365, 366, 460, 468,
 486, 487, 488, 469, 462, 480, 494, 957, 465, 186, 466, 467, 78, 79] .
```

Figura 38: Percurso possível com paragens intermédias

3.9 Comparação dos Algoritmos usados

Base para Comparação	**Pesquisa Informada**	**Pesquisa não-Informada**
Conhecimento	Usa conhecimento dos predicados **paragem/13** para determinar o próximo nodo para onde se deve deslocar, principalmente carreiras, cids e gids	Não usa conhecimento nenhum, escolhe o próximo nodo que aparecer
Eficiência	Bastante eficiente, calcula a resposta para qualquer input quase instantaneamente	Depende do tamanho do caminho, é muito eficiente em caminhos curtos, mas para caminhos grandes é uma má escolha
Memória	Gasta muito pouca memória, basicamente só guardar os nodos do caminho atual e, se não puder prosseguir, descarta todo o caminho e volta a tentar outro percurso a partir da origem	A memória que gasta é exponencial, quanto maior for o caminho a calcular, mais memória gasta, especialmente em caminhos bidirecionais, para caminhos muito grandes bidirecionais é normal que exceda a capacidade da *stack*
Tempo	Na ordem das milésimas ou centésimas de segundos	Aumenta exponencialmente com o tamanho do caminho
Algoritmos	Se a origem e o destino se encontrarem na mesma carreira desloca-se diretamente para lá, caso contrário calcula o menor conjunto de carreiras a partir do qual é possível atingir o destino e percorre-as	Está limitado a um número de iterações (mais curto) ou uma distância máxima (mais próximo), faz uma travessia *depth-first* até atingir esses limites

4 Conclusões e Sugestões

O desenvolvimento deste projeto permitiu a consolidação da matéria teórico-prática lecionada nas aulas da unidade curricular e a eludicidação da verdadeira utilidade e poder de ferramentas como a linguagem *Prolog*, no que toca ao armazenamento de dados em grande escala em bases de conhecimento e respetivo processamento.

Estou satisfeito com a solução atingida e penso que fui bem-sucedido na criação de um sistema de recomendação de transportes públicos que se adequa às necessidades do utilizador, tendo cumprido maioritariamente tudo o que é proposto no enunciado, embora certas partes do trabalho tenham ficado áquem da minha ambição, devido a constrições de tempo.

Foi bastante frustrante lidar com os dados fornecidos, devido à presença de muitas inconsistências, contudo entendo que seja normal ao lidar com situações reais e de grande escala e, em retrospetiva, considero que foi uma boa experiência de aprendizagem e ambientação com ferramentas pouco comuns no curso.

De salientar é o facto de o algoritmo de pesquisa informada desenvolvido é extremamente eficaz, sendo capaz de responder a qualquer interrogação quase intantaneamente, pelo que os requisitos que fazem uso do mesmo disfrutam da mesma rapidez, e a estratégia de determinação do percurso mais curto, pelo menos para caminhos unidirecionais, é também muito eficiente, determinando os percursos especificados com muito pouca latência.

Em futuras iterações, seria possível otimizar os algoritmos de cálculo do menor/mais próximo percurso, resolvendo o erro mencionado do caminho mais próximo, e complementar também a estratégia dos requisitos de filtragem de resultados, que não se encontram completamente funcionais.

5 Referências

Documentação do SWI-Prolog
Disponível em: https://www.swi-prolog.org/
Acesso em: 3/06/2020

Pesquisa informada vs não-informada
Disponível em: https://pt.gadget-info.com/difference-between-informed
Acesso em: 27/05/2020

A Anexo

```python
def parser(paragens,listaAdjacencias,dicionarioCarreiras,carreira):
    paragens.write("% Carreira " + str(carreira['Carreira'][1]) + '\n')
    idCarreira = 1

    for paragem in carreira.iterrows():
        if idCarreira > 1:
            distancia = round(math.sqrt((anterior[1]-paragem[1]['latitude'])**2 + (anterior[2]-paragem[1]['longitude'])**2),3)
            adj = (anterior[0],paragem[1]['gid'],distancia)
            if adj not in listaAdjacencias:
                listaAdjacencias.append(adj)

        paragens.write("paragem(")
        paragens.write(str(paragem[1]['Carreira']) + ', ')
        paragens.write(str(idCarreira) + ', ')
        paragens.write(str(paragem[1]['gid']) + ', ')
        paragens.write(str(paragem[1]['latitude']) + ', ')
        paragens.write(str(paragem[1]['longitude']) + ', ')
        paragens.write('\'' + paragem[1]['Estado de Conservacao'] + '\', ')
        paragens.write('\'' + paragem[1]['Tipo de Abrigo'] + '\', ')
        paragens.write('\'' + paragem[1]['Abrigo com Publicidade?'] + '\', ')
        paragens.write('\'' + paragem[1]['Operadora'] + '\', ')
        paragens.write(str(dicionarioCarreiras[paragem[1]['gid']]) + ', ')
        paragens.write(str(paragem[1]['Codigo de Rua']) + ', ')
        paragens.write('\'' + paragem[1]['Nome da Rua'] + '\', ')
        paragens.write('\'' + paragem[1]['Freguesia'] + '\').\n')

        idCarreira+=1
        anterior = [paragem[1]['gid'],paragem[1]['latitude'],paragem[1]['longitude']]

    paragens.write('\n')
```

Figura 39: Função parser do programa Python

```
% Processa a proxima paragem e a lista de carreiras devolvidas pela proxParagem
% Esta numa carreira com ligacao direta ao destino, pode continuar a calcular o caminho normalmente
processaProxParagem((ProxParagem,[[]]),(Origem,Destino),GidsProibidos,PercursoAtual,TipoTravessia,Restricoes,Percurso) :-
    ProxParagem > 0, !,
    percursoInformado(ProxParagem,(Origem,Destino),GidsProibidos,[ProxParagem|PercursoAtual],TipoTravessia,Restricoes,Percurso).

% Esta numa carreira com ligacao direta ao destino mas encontrou uma paragem que nao pode usar devido as
% restricoes estabelecidas, calcula os conjuntos de carreiras a partir dos quais e possivel chegar ao destino
% e comeca a seguir o mais curto a partir da origem, caso exista algum, senao falha
processaProxParagem((GidProibidoNeg,[[]]),(Origem,Destino),GidsProibidos,_,TipoTravessia,Restricoes,Percurso) :-
    GidProibido is abs(GidProibidoNeg),
    getCarreiras(Origem,CarreirasOrigem),
    calculaCaminhosCarreiras(CarreirasOrigem,Destino,CaminhosCarreiras),
    length(CaminhosCarreiras,N), !, N > 0,
    percursoInformado(Origem,(Origem,Destino),[GidProibido|GidsProibidos],CaminhosCarreiras,[Origem],TipoTravessia,Restricoes,Percurso).

% Esta a seguir um caminho de carreiras e nao teve problemas a passar para a proxima paragem
processaProxParagem((ProxParagem,CaminhosCarreiras),(Origem,Destino),GidsProibidos,PercursoAtual,TipoTravessia,Restricoes,Percurso) :-
    ProxParagem > 0, !,
    percursoInformado(ProxParagem,(Origem,Destino),GidsProibidos,CaminhosCarreiras,[ProxParagem|PercursoAtual],TipoTravessia,Restricoes,Percurso).

% Esta a seguir um camihno de carreiras mas encontrou uma paragem que nao pode usar devido as restricoes
% estabelecidas, volta a origem e tenta seguir o caminho de carreiras seguinte
processaProxParagem((GidProibidoNeg,[_|CaminhosCarreiras]),(Origem,Destino),GidsProibidos,_,TipoTravessia,Restricoes,Percurso) :-
    GidProibido is abs(GidProibidoNeg),
    percursoInformado(Origem,(Origem,Destino),[GidProibido|GidsProibidos],CaminhosCarreiras,[Origem],TipoTravessia,Restricoes,Percurso).
```

Figura 40: Predicado processaProxParagem

```
% Tendo em conta o caminho de carreiras que esta a seguir e todos os outros parametros dados
% escolhe o gid da proxima paragem e atualiza os caminhos de carreiras conforme o resultado
escolheProxParagem(Gid,Destino,GidsProibidos,CaminhosCarreiras,TipoTravessia,Restricoes,(ProxParagem,CaminhosAtualizados)) :-
    paragensGid(Gid,TipoTravessia,Restricoes,Pares),
    ((length(Pares,0), !, ProxParagem is -Gid, cabecaListaVazia(CaminhosCarreiras,CaminhosAtualizados));
    proxParagemCaminhoCarreiras(Gid,Destino,GidsProibidos,CaminhosCarreiras,CaminhosAtualizados,ProxParagem)).

% Calcula a proxima paragem, caso haja algum caminho possivel
escolheProxParagem(Gid,Destino,GidsProibidos,TipoTravessia,Restricoes,(ProxParagem,CaminhosAtualizados)) :-
    paragensGid(Gid,TipoTravessia,Restricoes,Pares),
    passaDestinoPares(Pares,Destino,Difs),!,
    ((length(Difs,0), !, ProxParagem is -Gid, cabecaListaVazia([[]],CaminhosAtualizados));
    (tudoZeros(Difs), !,
        listaFstPares(Pares,Carreiras),
        calculaCaminhosCarreiras(Carreiras,Destino,CaminhosCarreiras),!,
        proxParagemCaminhoCarreiras(Gid,Destino,GidsProibidos,CaminhosCarreiras,CaminhosAtualizados,ProxParagem));
    (proxParagemCarreiraDestino(Pares,Difs,ProxParagem), cabecaListaVazia([[]],CaminhosAtualizados))).
```

Figura 41: Predicado escolheProxParagem

```
% Algoritmo breadth-first usado para calcular caminhos entre as carreiras dadas
% e as carreiras do destino
caminhosCarreirasAux([],_,_,_,[],R,R).

caminhosCarreirasAux([],CDestino,_,Vis,NovosCaminhos,Finais,R) :-
    listaCabecas(NovosCaminhos,Proximas),
    caminhosCarreirasAux(Proximas,CDestino,NovosCaminhos,Vis,[],Finais,R).

caminhosCarreirasAux([Atual|T],CDestino,Caminhos,Vis,TempNovosCaminhos,Finais,R) :-
    carreirasAdjacentesPorVisitar(Atual,Vis,PorVisitar),
    caminhosAteAdjacentes(Atual,CDestino,Caminhos,PorVisitar,Vis,TempNovosCaminhos,
                          Finais,(TempAtualizado,FinaisAtualizados,NovasVis)),
    caminhosCarreirasAux(T,CDestino,Caminhos,NovasVis,TempAtualizado,FinaisAtualizados,R).
```

Figura 42: Algoritmo *breadth-first* para calcular carreiras ate ao destino