Sistemas de Representação de Conhecimento e Raciocínio

Mestrado Integrado em Engenharia Informática
Universidade do Minho

Métodos de Resolução de Problemas e de Procura
Avaliação Individual

André Ferreira
A64296

Braga, 6 de junho de 2021

Sistemas de Representação de Conhecimento e Raciocínio

Métodos de Resolução de Problemas e de Procura

Resumo

Este relatório serve como documentação de todo o processo de desenvolvimento do instrumento de avaliação individual da unidade curricular de Sistemas de Representação de Conhecimento e Raciocínio, sobre a aplicação de métodos de resolução de problemas de procura a um caso de estudo real, com auxílio da linguagem de programação *Prolog*, para a implementação de programação em lógica.

O caso de estudo particular tratado ao longo deste projeto é referente a um circuito de recolha de resíduos urbanos no concelho de Lisboa. Para tal finalidade, foi necessário fazer o tratamento de dados, para posteriormente passar à sua manipulação. Todo este processo, e as decisões tomadas ao longo do mesmo, encontram-se detalhadas ao longo do presente relatório.

Conteúdo

Lista de Figuras

1. Introdução

Este projeto individual teve como objetivo a aplicação dos conhecimentos obtidos ao longo do semestre no caso de estudo da recolha de resíduos urbanos numa freguesia do concelho de Lisboa, e da geração de percursos para o mesmo. Estes percursos consistem em problemas de pesquisa, para os quais foram implementadas múltiplas estratégias e algoritmos que os permitem resolver em *Prolog*, quer de pesquisa informada, como de pesquisa não informada. Já o tratamento de dados, para facilitar o seu uso em *Prolog*, foi feito com auxilio de um *script* desenvolvido em *Python*.

No planeamento deste projeto, foi considerada uma **versão simplificada** do problema, assumindo capacidade ilimitada dos camiões, e não tendo em consideração as quantidades de resíduos nos pontos de recolha. O mesmo foi desenvolvido utilizando ***SWI-Prolog*** como ambiente de desenvolvimento *Prolog*, podendo surgir algumas diferenças na sua execução em comparação com outras versões, como *SICSTus Prolog*.

2. Tratamento de Dados

Os dados a abordar ao longo deste projeto foram fornecidos num ficheiro *.xlsx*, referentes aos múltiplos pontos de recolha disponíveis e a sua informação. Para o seu tratamento, este ficheiro foi convertido num ficheiro *.csv*, de modo a facilitar o seu uso. De seguida, foi processado por um *script* desenvolvido em *Python*, *pyparser.py*, incluído com o projeto.

2.1 Parsing

De modo a conseguir transportar os dados fornecidos para o *Prolog*, e de facilitar a sua representação, o *script pyparser.py* foi desenvolvido. Este pode ser executado (com *python pyparser.py* e, encontrando-se no mesmo local que o ficheiro *dataset.csv*, produz dois ficheiros como output: *adjacencias.pl* e *pontos.pl*. Estes consistem dos dados tratados e escritos de forma a serem lidos e aceites imediatamente pelo *Prolog*. Este *script* utiliza também a biblioteca *unidecode*, de modo a substituir caracteres unicode, para aumentar compatibilidade e facilitar as procuras. Deste modo, não existem sinais diacríticos ou outros caracteres especiais no resultado final, pelo que as procuras devem conformar com tais normas.

2.2 Formatação

2.2.1 Pontos de recolha

Os pontos de recolha são fornecidos, no ficheiro original, com a seguinte estrutura:

```
"Latitude","Longitude","OBJECTID","PONTO_RECOLHA_FREGUESIA",
"PONTO_RECOLHA_LOCAL","CONTENTOR_RESÍDUO","CONTENTOR_TIPO",
"CONTENTOR_CAPACIDADE","CONTENTOR_QT","CONTENTOR_TOTAL_LITROS"
```

Após algum estudo dos dados contidos relativos aos pontos, podemos constatar que cada entrada se refere a um contentor *OBJECTID*, com os seus respetivos valores. Podemos também ver que estes contentores estão agrupados em pontos de recolha, localizados na mesma latitude/longitude, e que os pontos de recolha em si podem ser identificados pelos números ao início do campo *PONTO_RECOLHA_LOCAL*.

Com isto foi estabelecido como seria feito o *parsing* dos dados e, consequentemente, como seriam definidos os pontos de recolha, correspondentes a nodos de um grafo, no nosso problema.

2.2.2 Nodos

Os nodos, representados no ficheiro *pontos.pl*, correspondem a pontos de recolha, visto estes terem todos a mesma localização geográfica. Ao longo da leitura do ficheiro de dados, todos os pontos são agrupados baseados no número que precede a sua rua, em *PONTO_RECOLHA_LOCAL*. Feito isto, são criados nodos com o seguinte formato:

```
:- dynamic ponto/7.
ponto(pontoID, Longitude, Latitude, Freguesia, Rua,
      [TipoContentores], CapacidadeTotal).
```

Onde *pontoID* corresponde ao número supramencionado, *TipoContentores* a uma lista dos diversos tipos de resíduos aceites pelos contentores presentes, e *CapacidadeTotal* ao total das capacidades dos contentores individuais. A informação *Rua* preserva a informação de sentido, para ruas com a mesma, visto estes pontos corresponderem a coordenadas diferentes.

2.2.3 Arestas

As arestas do nosso grafo foram geradas com base na informação presente, assumindo que cada entrada no *dataset* teria uma conexão de adjacência com a entrada imediatamente abaixo.

Para além destas, foram geradas adjacências entre os pontos que continham tal informação após a sua própria rua, do seguinte modo:

```
"15806: R do Alecrim (Impar (27->53)(->): R Ataíde - R Ferragial)"
```

Essa informação é então usada para procurar tais ruas nos dados fornecidos, e criar um adjacência com a mesma se existir. Foi também no *script* calculada a distância euclidiana entre os pontos de adjacências, para facilitar a sua verificação futura. Com isto, o resultado final tem o seguinte formato:

```
:- dynamic adjacencia/3.
adjacencia(pontoID1, pontoID2, distancia).
```

2.3 Dimensão dos Dados

Com o decorrer do projeto foi constatado que a grande quantidade de nodos e arestas, em conjunção com os métodos utilizados, causavam alguns problemas, com procuras que levavam o *Prolog* a exceder os limites da *stack*, ou que bloqueavam sem resposta, algo que se verificava ainda mais para procuras que devem devolver múltiplos resultados, ou com nodos mais distantes.

Após tentar remover quaisquer possíveis repetições de arestas com alterações à execução do *script*, este problema não foi solucionado. Como tal, e no interesse de mais facilmente correr e testar as soluções implementadas, foram criados dois ficheiros extra, *sample_adjacencias.pl* e *sample_pontos.pl*. Estes são bastante menos extensos, considerando apenas alguns dos pontos de recolha e as suas respetivas ligações. A alteração entre os conjuntos de ficheiros pode ser facilmente feita no programa principal, *main.pl*, alterando os comentários no inicio do programa:

```
%--- DATASET COMPLETO ---
%:- include('adjacencias.pl').
%:- include('pontos.pl').

%--- DATASET PARCIAL ---
:- include('sample_adjacencias.pl').
:- include('sample_pontos.pl').
```

3. Predicados e Estratégias de Procura

Concluídos os passos de análise e preparação detalhados no capítulo anterior, a base de conhecimento encontra-se pronta e disponível nos ficheiros mencionados, sendo esta importada e utilizada para fazer as pesquisas. Após isto, podemos começar a implementação dos algoritmos de procura em *Prolog*, que serão utilizados para responder às questões propostas.

3.1 Pesquisa não-informada

Os algoritmos de procura foram implementados ao início, permitindo obter trajetos entre dois nodos. Inicialmente, foram definidos os algoritmos de pesquisa não-informada, correspondentes a procuras cegas no grafo, sem utilizar informação sobre o mesmo na decisão de como se deslocar.

3.1.1 Pesquisa Primeiro em Profundidade

O primeiro algoritmo de pesquisa implementado corresponde à pesquisa primeiro em profundidade, ou *Depth-First search*. Este algoritmo corresponde a uma procura pelo grafo, tentando atingir o nodo mais profundo possível, antes de voltar aos nodos anteriores adjacentes para explorar outros possíveis caminhos.

A implementação utilizada aqui utilizada em *Prolog* toma a seguinte forma:

```
depthFirstSearch(Nodo, Destino, [Nodo|Caminho]):-
    dfsa(Nodo, Destino, [Nodo], Caminho).

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

dfsa(Destino, Destino, _, _):- !, fail.

dfsa(Nodo, Destino, Visitado, [ProxNodo|Caminho]):-
    adjacente(Nodo, ProxNodo),
    \+ memberchk(ProxNodo, Visitado),
    dfsa(ProxNodo, Destino, [Nodo|Visitado], Caminho).
```

O predicado *memberchk* garante que o nodo que visitamos ainda não foi visitado, ao passo que o predicado *adjacente*, como o nome indica, verifica a adjacência entre dois pontos, e encontra-se definido de seguinte forma:

```
adjacente(Ponto, ProxPonto):- adjacencia(Ponto, ProxPonto, _).
adjacente(Ponto, ProxPonto):- adjacencia(ProxPonto, Ponto, _).
```

Este algoritmo consegue encontrar caminhos entre dois nodos no grafo, embora estes caminhos fiquem rapidamente extensos, envolvendo muitos nodos, e com grandes quantidades de dados, frequentemente não termina ou incorre problemas de memória na sua execução.

```
?- depthFirstSearch(15805, 15812, Path).
Path = [15805, 15806, 15807, 15808, 15809, 15812] ;
Path = [15805, 15806, 15807, 15808, 15809, 15810, 15812] ;
Path = [15805, 15806, 15807, 15808, 15809, 15811, 15810, 15812]
```

Figura 3.1: Execução de uma Depth-First Search

3.1.2 Pesquisa Primeiro em Largura

A segunda estratégia de procura implementada foi também uma procura não-informada, sendo esta uma pesquisa primeiro em largura, ou uma *Breadth-first Search*. Aqui, ao invés de percorrer cada ramo do nosso grafo na sua total profundidade antes de retroceder, optamos pela estratégia oposta, percorrendo os seus ramos todos a cada nível de profundidade antes de prosseguir para o próximo nível. Este algoritmo encontra-se definido do seguinte modo:

```
breadthFirstSearch(Nodo, Destino, Caminho) :-
        bfsa([[Nodo]], Solucao, Destino),
        reverse(Solucao, Caminho).

bfsa([[Nodo|Caminho]|_], [Nodo|Caminho], Destino):-
        Nodo == Destino.

bfsa([[Nodo|Caminho]|CaminhoList], Solucao, Destino):-
        bagof([P, Nodo|Caminho],
        (adjacente(Nodo, P),
        \+ memberchk(P, [Nodo|Caminho])), NovosCaminhos),
        append(CaminhoList, NovosCaminhos, Res), !,
        bfsa(Res, Solucao, Destino);
        bfsa(CaminhoList, Solucao, Destino).
```

Comparado com o anterior, este algoritmo permite-nos fazer pesquisas que devolvem resultados envolvendo menos nodos ao inicio, cujo número vai aumentando mais progressivamente.

```
?- breadthFirstSearch(15805, 15812, Path).
Path = [15805, 15806, 15807, 15808, 15809, 15812] ;
Path = [15805, 15806, 15807, 15808, 15810, 15812] ;
Path = [15805, 21944, 15807, 15808, 15809, 15812] ;
Path = [15805, 21944, 15807, 15808, 15810, 15812] .
```

Figura 3.2: Execução de uma Breadth-First Search

3.2 Pesquisa Informada

Após estes tipos de pesquisa, foi introduzido um algoritmo de pesquisa informada. Este algoritmo, ao contrário dos anteriores, utiliza uma heurística na sua pesquisa, que lhe permite tentar fazer escolhas informadas sobre os nodos a escolher, tentando obter um resultado, em média, melhor.

3.2.1 Pesquisa Gulosa

O algoritmo de pesquisa informada implementado foi aquele conhecido como pesquisa gulosa - *Greedy search*. Este algoritmo utiliza uma heurística de escolha em que calcula um custo aos próximos nodos, estabelecido como a distância entre os pontos de recolha, e tenta optimizar a solução escolhendo, nodo a nodo, aqueles com menor distância.

O algoritmo, conforme implementado, é o seguinte:

```
greedySearch(Nodo, Destino, Caminho/Custo):-
        estima(Nodo, Destino, Est),
        greedy([[Nodo]/0/Est], InvCaminho/Custo/_, Destino),
        reverse(InvCaminho, Caminho).

greedy(Caminhos, Caminho, Destino):-
        melhorp(Caminhos, Caminho),
        Caminho = [Nodo|_]/_/_,
        Nodo == Destino.

greedy(Caminhos, SolucaoCaminho, Destino):-
        melhorp(Caminhos, MelhorCaminho),
        selecciona(MelhorCaminho, Caminhos, OutrosCaminhos),
        expandirG(MelhorCaminho, ExpCaminhos, Destino),
        append(OutrosCaminhos, ExpCaminhos, NovoCaminhos),
        greedy(NovoCaminhos, SolucaoCaminho, Destino).
```

Este algoritmo é algo mais complexo, mas consegue encontrar soluções aceitáveis nas suas tentativas de minimização de custo, mesmo que não garanta soluções ótimas, e utiliza os seguintes predicados extra para efetuar as suas pesquisas:

```
melhorp([Caminho], Caminho):- !.

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

melhorp([_|Caminhos], MelhorCaminho):-
        melhorp(Caminhos, MelhorCaminho).

expandirG(Caminho, ExpCaminhos, Destino):-
        findall(NovoCaminho, adjacente(Caminho, NovoCaminho, Destino),
        ExpCaminhos).
```

```
?- greedySearch(15805, 15812, Path).
Path = [15805, 15806, 15807, 15808, 15809, 15812]/247 ;
Path = [15805, 15806, 15807, 15808, 15810, 15812]/288 ;
Path = [15805, 15806, 15807, 15808, 15809, 15810, 15812]/298 .
```

Figura 3.3: Execução de uma Greedy search

3.3 Comparação de Algoritmos

Implementadas as diferentes estratégias de procura, foram feitas múltiplas pesquisas, e comparados os algoritmos a nível de complexidade temporal e espacial, bem como a nível de garantias de estes completarem com uma resposta, e se a resposta dada é, ou não, a melhor solução.

A nível da pesquisa ser completa, podemos constatar que o algoritmo de pesquisa Primeiro em Profundidade não o garante, e por vezes não se obtém resposta, possivelmente devido a loops e repetições no qual fica retido. O algoritmo de pesquisa *Greedy* também não o garante, entrando similarmente em ciclos, e tendo problemas com falsos começos. Já o algoritmo de pesquisa Primeiro em Largura garante, em teoria, que completa. O problema aqui, neste caso de estudo, acaba por ser a nível da sua complexidade de espaço, e com os dados que são fornecidos, tende a esgotar o espaço que lhe é alocado antes de dar uma resposta, para as questões nas quais os outros algoritmos também falham.

A nível da solução ser ótima, nenhum destes algoritmos o garante. Os de pesquisa não-informada devolvem o primeiro resultado encontrado, ao passo que o algoritmo de pesquisa *Greedy* utiliza uma simples heurística, que não garante a melhor solução, algo que é possível ver após algumas execuções do mesmo.

Para uma análise do tempo real de execução dos algorítmos de pesquisa, foram executadas pesquisas escolhendo dois pontos suficientemente distanciados, obtendo o tempo de execução graças ao predicado *profile/1*. Os resultados apresentados referem-se a uma pesquisa entre os nodos 15811 → 19407.

	Completo	Melhor Solução	Tempo (Complexidade)	Espaço (Complexidade)	Tempo (Segundos)
Breadth-first	Sim	Não	$O(b^d)$	$O(b^d)$	4.92
Depth-first	Não	Não	$O(b^m)$	$O(bm)$	0.01
Greedy	Não	Não	Melhor caso: $O(bd)$ Pior caso: $O(b^m)$		Stack Exceeded

De um ponto de vista geral, estes algoritmos eram instantâneos na primeira resposta para pesquisas entre pontos com pouca profundidade entre as suas ligações e de baixa dificuldade, e progrediam, no geral, da forma demonstrada na tabela, com o algoritmo *Depth-first* a ser bastante rápido ou quase instantâneo a apresentar resposta mesmo para pontos bastante distantes, para os quais os outros dois começavam a demonstrar problemas; o algoritmo *Breadth-first* apresentava demoras bastante mais longas, por vezes não obtendo resposta após largos períodos de espera, e o *Greedy* ficando também parado na sua execução, por vezes devido a exceder o limite de memória da *stack*.

Tendo feito esta análise, e comparando os resultados reais obtidos com os dados que estão em uso, e particularmente considerando que muitas das questões aplicadas ao caso de estudo requerem que se obtenha e processe não *um* circuito, mas todos os que obedecessem a um certo critério, foi feita a escolha de implementar primariamente o algoritmo de pesquisa Primeiro em Profundidade para grande parte dos predicados que tentam dar resposta a estas questões. Esta escolha foi feita tendo em conta que este algoritmo apresenta bastante melhor performance para os grafos de grande tamanho, como o que estamos a lidar neste caso prático. Mesmo assim, para a maioria das respostas apresentadas, seria bastante fácil simplesmente alterar o algoritmo principal de procura, para que estas passassem a executar com um outro qualquer à escolha, se tal fosse desejável.

3.4 Respostas ao Caso Prático

Com estas considerações, passamos à análise do caso prático, utilizando os algoritmos definidos para dar resposta às questões apresentadas. Devido às interações entre a distância e algumas destas questões, os pontos inicias e finais foram considerados como nodos existentes do percurso, a serem providenciado nos predicados. Com a base de conhecimento utilizada, isto significa que os mesmos são pontos de recolha; porém, isto poderia facilmente ser alterado para utilizar, por exemplo, garagens, sendo inseridas as mesmas na base de conhecimento com as suas próprias coordenadas de localização.

3.4.1 Circuitos de recolha indiferenciada e seletiva

O primeiro problema com que nos deparamos trata de gerar os circuitos de recolha, tanto indiferenciada como seletiva, que abrangem uma área específica. A sua solução foi dada através de dois predicados, um para apresentar os circuitos de recolha indiferenciada, e outro de recolha seletiva.

Para o primeiro, simplesmente é feita uma pesquisa usando um dos algoritmos anteriores, neste caso com uma *Depth-first search*. Utiliza o predicado *findall* de modo a gerar todos estes circuitos, e apresentá-los como resposta.

```
circuitoIndiferenciado(Nodo, Destino, Circuitos):-
        findall(Caminho,
        depthFirstSearch(Nodo, Destino, Caminho),
        Circuitos).
```

```
?- circuitoIndiferenciado(15811, 15809, N).
N = [[15811, 15809], [15811, 15810, 15809], [15811, 15810, 15812, 15809], [15811, 15810, 15808, 15809]].
```

Figura 3.4: Circuitos indiferenciados

Para a pesquisa por circuitos diferenciados, foi utilizada uma implementação da *Depth-first search*, onde apenas aceitamos um nodo após confirmar que este tem contentores com o tipo de resíduo que fornecemos ao predicado. Novamente, um *findall* é usado para obter o conjunto de todos estes circuitos.

```
circuitoSeletivo(Nodo, Destino, Tipo, Circuitos):-
        findall(Caminho,
        dfscsa(Nodo, Destino, Tipo, Caminho), Circuitos).

dfscsa(Nodo, Destino, Tipo, [Nodo|Caminho]):-
    dfscs(Nodo, Destino, [Nodo], Tipo, Caminho).

dfscs(Nodo, Destino, _, Tipo, [Destino]):-
    adjacente(Nodo, Destino),
    ponto(Destino, _, _, _, _, T, _),
    member(Tipo, T).

dfscs(Destino, Destino, _, _, _):- !, fail.

dfscs(Nodo, Destino, Visitado, Tipo, [ProxNodo|Caminho]):-
    adjacente(Nodo, ProxNodo),
    \+ memberchk(ProxNodo, Visitado),
    ponto(ProxNodo, _, _, _, _, T, _),
```

```
    memberchk(Tipo, T),
    dfscs(ProxNodo, Destino, [Nodo|Visitado], Tipo, Caminho).
```

```
?- circuitoSeletivo(15805, 15889, 'Papel e Cartao', Path).
Path = [[15805, 15806, 15807, 15889]].
```

Figura 3.5: Circuitos seletivos

3.4.2 Circuitos com mais pontos de recolha

Aqui temos como objetivo identificar os circuitos com mais pontos de recolha, por tipo de resíduo a recolher. Para tal, foi criado um predicado que faz uso do anterior, de modo a obter todos os circuitos que sabemos ter os tipos de resíduos dados, entre um ponto de inicio e fim. Depois, com o auxilio de outro predicado, obtemos quais os circuitos mais longos desses, ou seja, com mais pontos de recolha, para obter a solução final.

```
maisPontosRecolha(Nodo, Destino, Tipo, Circuitos):-
        findall(MLCircuitos,
        (circuitoSeletivo(Nodo, Destino, Tipo, CircuitosTodos),
        maisLongas(CircuitosTodos, MLCircuitos)), Circuitos).

maxlen([],[]).
maxlen([H|T], [LH|LHT]) :-
        length(H, LH),
        maxlen(T, LHT).

lengthLongest(ListofLists, Max):-
        maxlen(ListofLists, Ls),
        max_list(Ls, Max).

maisLongas(ListofLists, ML):-
        lengthLongest(ListofLists, Len),
        member(ML, ListofLists),
        length(ML, Len).
```

```
?- maisPontosRecolha(15805, 15810, 'Lixos', Paths).
Paths = [[15805, 15806, 15807, 15808, 15809, 15811, 15810], [15805, 15806, 15807, 15808, 15809, 15812, 15810]].
```

Figura 3.6: Circuitos com mais pontos do tipo Ïixo

3.4.3 Comparar circuitos com Indicadores de Produtividade

Atendendo a que não estamos a considerar a capacidade do veículo coletor, para esta questão utilizamos como indicador de produtividade a distância média entre dois pontos. Está a ser feita uma procura *Depth-first*, tendo sido criado um predicado para o cálculo das distâncias totais, calculando depois a média de distâncias, e apresentando-as em conjunção com os circuitos em si, para comparação. Aqui aproveitamos a distância entre nodos que está nas suas adjacências, não tendo de as recalcular, com o predicado *adjacenteD*.

```
indicadorProdutividade(Nodo, Destino, Result):-
```

```
        findall((PP, DD), dfsp(Nodo, Destino, PP, DD), Result).

dfsp(Nodo, Destino, Caminho, CustoMedio):-
        depthFirstSearch(Nodo, Destino, Caminho),
        distanciaTotal(Caminho, Custo),
        length(Caminho, Total),
        CustoMedio is Custo/Total.

distanciaTotal([P1,P2], D):-
        adjacenteD(P1,P2,D).

distanciaTotal([P1,P2|T], D):-
        adjacenteD(P1, P2, D1),
        distanciaTotal([P2|T], D2),
        D is D1+D2.
```

```
?- indicadorProdutividade(15805, 15808, Paths).
Paths = [([15805, 15806, 15807, 15808], 44.5), ([15805, 15806, 21944, 15807, 15808], 37.4), ([15805, 21944, 1
5806, 15807, 15808], 52), ([15805, 21944, 15807, 15808], 44.75)].
```

Figura 3.7: Circuitos com indicadores de produtividade

3.4.4 Circuito mais rápido (menor distância)

São gerados todos os circuitos entre dois pontos neste predicado, de modo a dar resposta à questão, com a soma total de distância percorrida associada a cada um, sendo depois escolhido o circuito cujo valor de distância é menor entre eles.

```
caminhoMaisRapido(Nodo, Destino, Result):-
        findall((PP, DD),
        dfsml(Nodo, Destino, PP, DD), Caminhos),
        minimo(Caminhos, Result).

dfsml(Nodo, Destino, Caminho, Custo):-
        depthFirstSearch(Nodo, Destino, Caminho),
        distanciaTotal(Caminho, Custo).

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

minima([], M, M).

minima([(_, C)|T], (NM, CM), M):-
        C>CM, !,
        minima(T, (NM, CM), M).

minima([(H, C)|T], (_, _), M):-
        minima(T, (H,C), M).
```

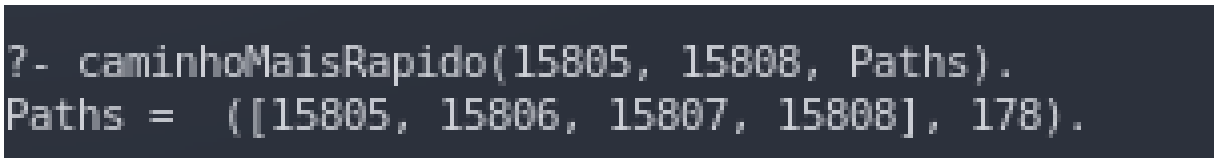

Figura 3.8: Circuito mais rápido entre dois nodos

3.4.5 Circuito mais eficiente

Para poder gerar o circuito mais eficiente, é necessário um critério de eficiência. Para o propósito desta solução, atendendo aos dados que temos disponíveis sobre cada ponto de recolha (e sobre as suas respetivas adjacências), foi definido como critério de eficiência o valor de (Capacidade total do Circuito)/Distância, gerando como métrica o circuito mais "denso"a nível de resíduos a serem coletados.

```
caminhoMaisEficiente(Nodo, Destino, Result):-
        findall((PP, DD),
        dfsme(Nodo, Destino, PP, DD), Caminhos),
        maximo(Caminhos, Result).

dfsme(Nodo, Destino, Caminho, Eficiencia):-
        depthFirstSearch(Nodo, Destino, Caminho),
        distanciaTotal(Caminho, Custo),
        capacidadeTotal(Caminho, Capacidade),
        Eficiencia is Capacidade/Custo.

capacidadeTotal([P1,P2], C):-
        ponto(P1,_,_,_,_,_,C1),
        ponto(P2,_,_,_,_,_,C2),
        C is C1+C2.

capacidadeTotal([P1,P2|T], C):-
        ponto(P1,_,_,_,_,_,C1),
        distanciaTotal([P2|T],C2),
        C is C1+C2.
```

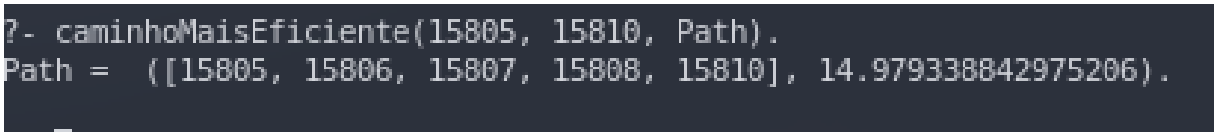

Figura 3.9: Circuito mais eficiente entre dois nodos

4. Conclusão e Comentários Finais

Ao longo da realização deste trabalho foram aprofundados os conhecimentos obtidos na unidade curricular de Sistemas de Representação de Conhecimento e Raciocínio, em particular com relação a métodos e problemas de procura, permitindo melhor compreender como implementar e aplicar a casos reais os algoritmos de procura que foram demonstrados ao longo do semestre.

Com este projeto foi possível ver como os diferentes algoritmos respondem quando a quantidade de dados é bastante grande, gerando grafos muito complexos, e os problemas que surgem para tais casos, como se verificou em muitas das pesquisas. Embora as soluções criadas funcionem bem para quantidades pequenas e controladas de dados, com a utilização do *dataset* completo frequentemente reparamos que se torna quase impossível fazer algumas destas pesquisas, na tentativa de obter todos os percursos entre dois pontos com profundidades bastante diferentes. Isto complicou e dificultou o processo de desenvolvimento e resolução, tendo sido tentadas várias alternativas. Ainda assim foram apresentadas respostas aos problemas, como aqui detalhado, embora estas nem sempre escalem bem para a quantidade de dados apresentada.

Como trabalho futuro seria útil desenvolver as estratégias de procura que não foram aqui implementadas, nomeadamente com um algoritmo de Busca Iterativa Limitada em Profundidade, como pesquisa não-informada, e um algoritmo de pesquisa A*, como alternativa à pesquisa *Greedy* para pesquisa informada. Isto permitiria obter mais resultados a nível das comparações entre algoritmos e ver como se comportam para os dados que utilizamos, sendo que uma pesquisa Limitada em Profundidade poderia-se adaptar bem para o problema apresentado.