Universidade do Porto
Faculdade de Ciências
Departamento de Ciências de Computadores

Caminhos ótimos e eficientes para transportadora internacional

Unidade curricular: Programação em Lógica

Autor
Nuno Bernardes

7 de dezembro de 2020

Conteúdo

Capítulo 1

Introdução

Para completar o processo de avaliação da unidade curricular de programação em lógica, propus a resolução dum problema que é alvo de uma grande discussão online, algoritmia e procura de caminhos ótimos e eficientes em grafos de elevada complexidade.

Dentro de um vasto mundo de temas que poderia abordar, decidi avançar com a resolução de caminhos ótimos para uma transportadora internacional. A ideia de todo o trabalho foi dividido em duas fases, em que a primeira trata de analisar os caminhos ótimos de um país para outro e a segunda trata dos caminhos ótimos já dentro de um distrito.

A inspiração para a realização deste trabalho foi obtida dum enorme fascínio que tenho por uma das maiores empresas de transportes do mundo, a Amazon. Com o confinamento, esta empresa monstruosa aumentou o volume de vendas astronomicamente, e, mesmo com esse aumento, conseguiu cumprir os prazos de entrega para todas as encomendas que eu próprio fiz no website, e isso fez-me procurar um pouco mais sobre como funcionam os algoritmos de entregas.

Este trabalho foi o primeiro em que apliquei as metodologias que aprendi na empresa onde atualmente trabalho, a Coletiv Studio, onde a primeira semana da execução de trabalho eu configurei um repositório no GitHub, dividi as duas tarefas mães em tarefas mais curtas, de menor complexidade, e criei um painel de projeto no mesmo repositório onde segui a metodologia de Scrum Sprint para maior organização de código e sensação de progresso em relação ao projeto. Aplicando este método de trabalho, consegui concluir o projeto com mais facilidade e rapidez.

Capítulo 2

Pesquisa de caminhos para entrega de encomendas segundo restrições de tempo e preços

2.1 Objetivo

A distribuidora tem uma série de centros de distribuição em capitais de países da Europa de onde pode encaminhar as encomendas para outros centros por via terrestre e/ou aérea. Um ficheiro Excel guarda uma tabela com as distâncias por vias terrestre e aéreas entre as diferentes cidades aonde a distribuidora tem representação. Esta fase do trabalho tem por objetivo:

1. Descrever a informação nas tabelas sob a forma de factos em Prolog.
2. Determinar, dada uma origem, um destino, data, hora e peso, qual o percurso mais barato e mais rápido e respetivos tempos de duração, preço associado e tipos de transporte utilizados.
    - O valor cobrado por via terrestre é de 0.01€ por km e kg de peso enquanto que o preço cobrado por via aérea é de 0.05€ por km e kg de peso.
    - Assume-se uma velocidade média de 80kms/h por via terrestre e de 500kms/h por via aérea.
    - O transporte por via terrestre é feito todos os dias às 7.00 e às 12.00.
    - O transporte por via aérea é feito às 3as, 5as e sábados às 6.00.

2.2 Desenvolvimento

Para resolução desta tarefa começei por, como sugeri acima, apresentar as informações nas tabelas na forma de factos. Para isso, criei um ficheiro exlusivo só para os factos, para não misturar a lógica do exercício com os dados. Para interligar o ficheiro de factos ao da lógica usei a importação de módulos de prolog. Para cumprir com as retrições do exercício de disponibilidades, também juntei a este módulos de factos as funções de verificação de

disponibilidade. Estas funções tratam de analisar para uma determinada altura do dia, quais são os transportes que estão disponíveis.

Por curiosidade e para facilitar a abstração de onde as encomendas serão direcionadas, decidi elaborar a representação gráfica do grafo, resultando (grafo elaborado em Miro)

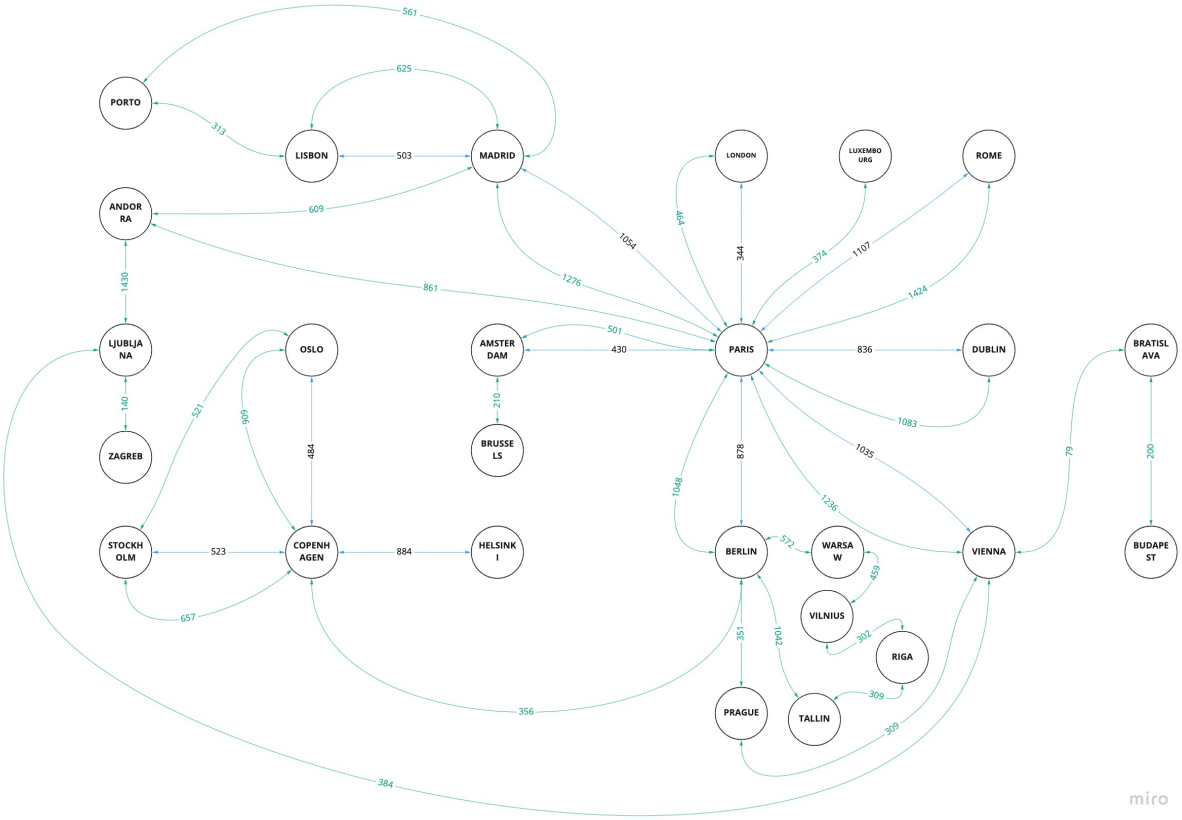

Figura 2.1: Representação gráfica do grafo da tarefa 1

Para a lógica desta tarefa decidi avançar com um algoritmo proposto por Ailsa Land e Alison Harcourt em 1960, o Branch & Bound. Este algoritmo é muito conhecido para resolução de vários problemas no ramo da pesquisa em grafos e árvores binárias.

A razão que me levou a escolher este algoritmo comparativamente a outros muito conhecidos tais como A*, Mini Max ou até algoritmos genéticos foi que, para o número de nós e ligações que temos para este grafo, não vamos comprometer muito na utilização de recursos a usar um algoritmo que tem uma complexidade mais elevada como o Branch & Bound. Para um grafo de mundo real, em que poderão haver milhares ou até milhões de nós, talvez a melhor estratégia seria um algoritmo que segue uma heurística mais restrita que a do Branch & Bound.

Nota: Todo o código está claramente documentado e para o caso das funções que respondem ao problema, são apresentados comentários com casos de teste. Para experimentar, deve aceder ao diretório da tarefa em causa e executar o ficheiro de resolução com o comando

```
$ swipl -l task.pl
```

2.2.1 Percurso mais barato

Para o percurso mais barato a lógica desenvolvida foi a seguinte

```
% Algoritmo Branch & BOund - Tarefa 1 Método 1 Mais barato
percursoMaisBarato(Orig, Dest, Peso, Perc) :-
    go1([((0,0), [Orig])], Dest, Peso, P),
    reverse(P, Perc).

% Caso final
go1([((C, Y), Prim)| _], Dest, Peso, Prim) :-
    Prim = [Dest| _],
    formatar_custo(C, Euros, Centimos),
    formatar_duracao(Y, Horas, Minutos),
    write('=========================[ PERCURSO ENCONTRADO ]=========================='), nl,
    write('Percurso Mais Barato: '), nl,
    write('Peso Encomenda: '), write(Peso), write(' Kg'), nl,
    write('Custo: '), write(Euros), write(' Euros e '), write(Centimos), write(' Centimos'), nl,
    write('Tempo de duração: '), write(Horas), write(' horas e '), write(Minutos), write(' minutos'), nl.

% Interromper, Continuar a pesquisa no Resto
go1([(_, _), [Dest| _]| Resto], Dest, Peso, Perc) :- ! , go1(Resto, Dest, Peso, Perc).
go1([((C, Y), [Ult| T])| Outros], Dest, Peso, Perc) :-
    findall(
        ((NC, NY), [Z, Tipo, Ult| T]),
        (proximo_no(Ult, T, Z, C1, Tipo),
        ((Tipo == 'car', NC is ((C + C1 * 0.01) + (Peso * 0.01)), NY is Y + (C1/80));
        (Tipo == 'flight', NC is ((C + C1 * 0.05) + (Peso * 0.05)), NY is Y + (C1/500)))),
        Lista),
    append(Outros, Lista, NPerc),
    sort(NPerc, NPerc1),
    go1(NPerc1, Dest, Peso, Perc).

% Função auxiliar para salta de nós método 1
proximo_no(X, T, Z, C, Tipo) :- trip(Tipo, X, Z, C),  \+ member(Z, T).

formatar_custo(C, Euros, Centimos) :-
    Euros is integer(C),
    Centimos is integer((C - Euros) * 100).

formatar_duracao(Y, Horas, Minutos) :-
    Horas is integer(Y / 0.6),
    Minutos is integer((Horas - (Y / 0.6)) * 100).
```

Figura 2.2: Código referente à primeira task da tarefa 1

Este código irá invocar a função proximo_no que vai verificar todos os caminhos para uma dada origem, destino e tipo de transporte. Quando é invocado no findall, o método próximo_no vai ficar instanciado com a origem, destino, distância e tipo de transporte do percurso que encontrou. De seguida, é verificado qual o tipo, e, conforme o mesmo, é calculado o custo associado aquele nó do percurso correspondente e este é somado com o custo associado aos nós anteriormente encontrados. No final, é apresentado o custo total para o percurso final, ou seja, o percurso com o menor custo associado, e o tempo de duração do mesmo.

Para experimentar este método, deve iniciar o programa e, já no terminal de execução de prolog, executar o seguinte comando (substitua Origem e Destino por alguma cidade presente no grafo de cidades e substitua o PesoDaEncomenda por um valor de peso para a encomenda)

```
?- percursoMaisBarato(Origem, Destino, PesoDaEncomenda, Percurso).
```

2.2.2 Percurso mais rápido

Para o percurso mais rápido a lógica desenvolvida foi a seguinte

```
% Tarefa 1 Método 2 mais rápido
percursoMaisRapido(Orig, Dest, DataHora, Perc) :-
    date_time_stamp(DataHora, Segundos),
    SegundosFinal = Segundos,
    go2([(0, [Orig])], Dest, Segundos, SegundosFinal, P),
    reverse(P, Perc).

% Caso final
go2([(C, Prim)| _], Dest, Segundos, _, Prim) :-
    Prim = [Dest| _], nl,
    write('Percurso Mais Rápido: '), nl,
    SFinal is Segundos + C, stamp_date_time(SFinal, DataFinal, 0),
    date_time_value(year, DataFinal, Ano),
    date_time_value(month, DataFinal, Mes), date_time_value(day, DataFinal, Dia),
    write('Data de Chegada: '), write(Dia), write('/'), write(Mes), write('/'), write(Ano),
    date_time_value(hour, DataFinal, Hora), date_time_value(minute, DataFinal, Minuto), nl,
    write('Horas de Chegada: '), write(Hora), write(':'), write(Minuto), nl,
    write('========================[ BARATO / RÁPIDO ]========================'), nl.

% Interromper, Continuar a pesquisa no Resto
go2([(_, [Dest| _])| Resto], Dest, Segundos, SegundosFinal, Perc) :-
    !, go2(Resto, Dest, Segundos, SegundosFinal, Perc).
go2([(C, [Ult| T])| Outros], Dest, Segundos, _, Perc) :-
    findall(
        (NC, [Z, Tipo, Ult| T]),
        (proximo_noh(Ult, T, Z, C1), SegundosAux is Segundos  + C,
        verificaDisponibilidade(SegundosAux, Count, Tipo),
        ((Tipo == 'car', NC is (C * ((C1/80) * 3600) + Count));
        (Tipo == 'flight', NC is (C + ((C1/500) * 3500) + Count)))),
        Lista),
    append(Outros, Lista, NPerc),
    sort(NPerc, NPerc1),
    go2(NPerc1, Dest, Segundos, Segundos, Perc).

% Função auxiliar para saltar de nós método 2
proximo_noh(X, T, Z, C) :- trip(_, X, Z, C), \+ member(Z, T).
```

Figura 2.3: Código referente à segunda task da tarefa 1

Este código irá invocar a função proximo_no e a verificaDisponibilidade, essas é que vão verificar se o próximo nó do percurso se encontra disponível e, dependendo da disponibilidade de cada tipo de transporte, depois de o método ser invocado com a hora pretendida, irá ser intanciado com o tempo de espera resultante e com o tipo de transporte que se encontrava disponível. Posteriormente, é calculado o tempo de duração para cada nó do percurso ao qual é somado o tempo de espera atual e o valor dos nós anteriores. No final, são apresentados a data e horas de chegada ao destino, juntamente com o percurso resultante.

Para experimentar este método, deve iniciar o programa e, já no terminal de execução de prolog, executar o seguinte comando (substitua Origem e Destino por alguma cidade presente no grafo de cidades e substitua a Data por um valor no formato de date de prolog (date(YYYY, MM, DD, HH, MM, SS, 0, _, _)).

```
?- percursoMaisRapido(Origem, Destino, Data, Percurso).
```

2.2.3 Execução de ambas as tasks

Por fim, como é para calcular para um mesmo percurso tanto o percurso mais barato como o mais rápido, fiz uma função para chamar ambos os métodos.

```
% Testar a tarefa:
% tarefa(porto, lisbon, date(2020, 11, 30, 18, 0, 0, 0, _, _), 1, Barato, Rapido).
% tarefa(oslo, warsaw, date(2020, 11, 30, 7, 30, 0, 0, _, _), 1, Barato, Rapido).

tarefa(Orig, Dest, Data, Peso, Perc, Perc1):-
    percursoMaisBarato(Orig, Dest, Peso, Perc),
    percursoMaisRapido(Orig, Dest, Data, Perc1).

% Percurso Mais Barato: preco utilizado + tempo utilizado, transporte utilizado
% Percurso Mais Rápido: tempo de duracao + transporte utilizado
```

Figura 2.4: Como executar toda a tarefa

2.2.4 Exemplo de output

```
?- tarefa(porto, lisbon, date(2020, 11, 30, 18, 0, 0, 0, _, _), 1, Barato, Rapido).
==========================[ PERCURSO ENCONTRADO ]==========================
Percurso Mais Barato:
Peso Encomenda: 1 Kg
Custo: 3 Euros e 14 Centimos
Tempo de duração: 7 horas e 48 minutos

Percurso Mais Rápido:
Data de Chegada: 1/12/2020
Horas de Chegada: 6:36
==========================[ BARATO / RÁPIDO ]==========================
Barato = [porto, car, lisbon],
Rapido = [porto, flight, lisbon].
```

Figura 2.5: Exemplo de execução dos métodos de Porto a Lisboa

2.3 Wrap up da tarefa

A complexidade desta tarefa não era muito grande, pois a unidade curricular de Desenho e Análise de Algoritmos já me teria dado algumas bases desta teoria, e não tive dificuldades para encontrar uma solução para o problema proposto. Como já tinha referido acima, se o grafo fosse mais complexo, o algoritmo que usei só iria funcionar em computadores com hardware muito específico para este tipo de execução. Para a execução num computador normal, o desenvolvimento de algoritmos com uma heurística mais reservada iria ser a melhor solução.

Capítulo 3

Determinar o percurso ótimo para a entrega de encomendas numa dada localidade

3.1 Objetivo

A entrega de encomendas a partir do ponto de distribuição mais próximo da morada de destino é feita por via terrestre por um estafeta da empresa que conduz um veículo de mercadorias que transporta as encomendas. Nesta fase terei que desenvolver um programa que dê ao estafeta o percurso que deve percorrer de forma a minimizar a distância percorrida e voltando no final das entregas ao centro de distribuição de onde partiu. Para esta parte, utilizará como exemplo o mapa da cidade do porto dividido em subáreas correspondentes às freguesias. O objetivo é modelar o mapa e implementar um gerador de percursos tendo em conta as retrições enunciadas anteriormente. O percurso deve ter 15 pontos de entrega definidos.

Figura 3.1: Mapa da cidade do Porto a considerar.

3.2 Desenvolvimento

Para resolução desta tarefa começei pela modulação do mapa da cidade do Porto. Para isso, como no exercício anterior, elaborei o grafo corresponde a este mapa, com o auxílio da ferramente Miro. O grafo resultante foi o seguinte

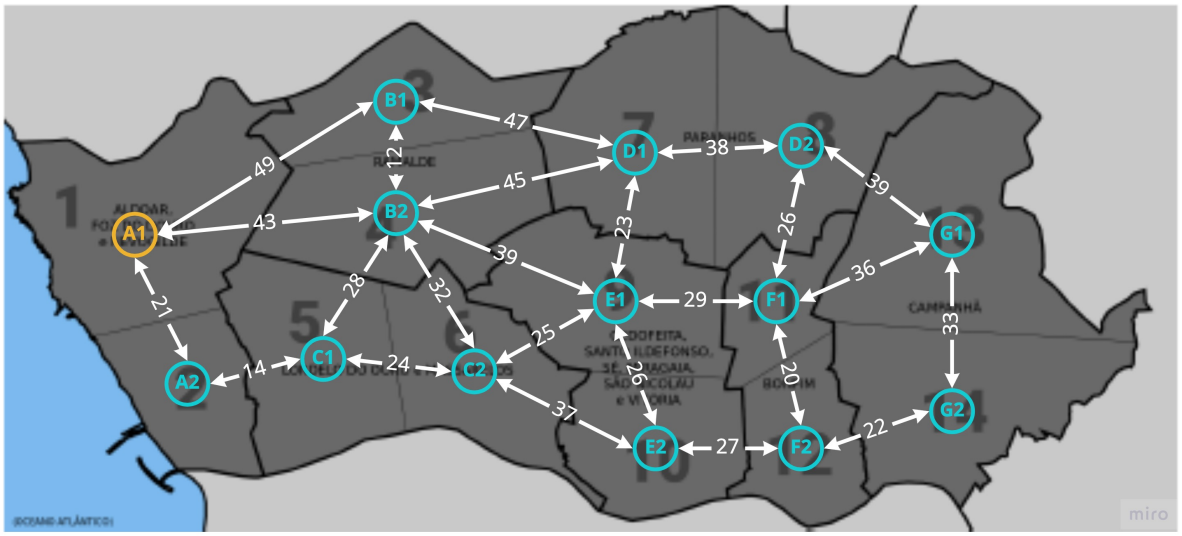

Figura 3.2: Mapa da cidade do Porto a considerar com grafo.

Quando começei a pensar como iria resolver o problema, pensei que iria ser muito semelhante ao problema da primeira tarefa, mas não. Acabei por descobrir que este problema já é explorado à bastantes anos, e encontra-se dentro dos Vehicle Routing Problems (VRP). A dificuldade deste tipo de problemas é, mesmo tendo um mapa com várias estradas, querermos só descobrir o percurso ótimo por onde o estafeta tem de passar para entregar todas as encomendas sem necessariamente ter de passar por todas as conexões. Inicialmente pensei que seria muito semelhante aos Travelling Salesman Problems, mas, nesse tipo de problemas, o caixeiro viajante irá visitar todos os nós do grafo.

Para este problema decidi arriscar a abordagem de Edsger Dijkstra, um cientista de computação muito conhecido no mundo da algoritmia. Claro que, logo de partida, para este tipo de problemas, seria mais correto o uso de abordagem A* ou até algoritmo genético, mas, após debater com um colega de curso, chegamos à conclusão que para o número de nós que tenho, o algoritmo de Dijkstra iria conseguir resolver o problema. Mais uma vez, para um grafo maior, ou compensávamos com melhoria drástica de hardware, ou teríamos de analisar algoritmos com heurísticas mais complexas.

Por curiosidade, a Google já explorou problemas de otimização na ferramenta de OR-Tools, nos quais constam maioria das variantes de TSP e ensinam muito bem como atacá-las (desvantagem é que não suportam Prolog na API deles).

Como no problema anterior, dividi o programa em dois ficheiros, facts.pl e task.pl. A escrita dos factos foi semelhante à do exercício anterior, mas, desta vez, fiz uma função que regista a bidirecionalidade dos nós para evitar a repetição de factos. Além disso, neste ficheiro, consta uma função que analisa o nó minimo, que é um fator importante no algoritmo de Dijkstra.

3.3 Algoritmo de Dijkstra com alteração de procura de solução

Para a tarefa, a lógica desenvolvida foi a seguinte

```
% Carregar modulo de factos
:- use_module(facts).

% Função principal
% Testes:
%   estafeta([])
%   estafeta([b1, d1, d2, g2])
%   estafeta([d1, f2, c1, b1, g2])
estafeta([]) :- format('Preencha a lista de encomendas com valores'),  halt().
estafeta(Encomendas) :-
      % Afirmação: O destino mais longe do centro de distribuição será o último a ser entregue.
      entrega_mais_longe(Encomendas, 0, Destino),
      write(Destino),
      % Procurar o caminho mais curto desde o centro até ao destino calculado encima.
      % Para cada caminho retornado verificar se o Percurso contêm todas as moradas das
      % Encomendas. Se contiver, retornar ao centro de distribuição.
      caminho_mais_curto(centro_distribuicao, Destino, PercursoIda, DistanciaIda),
      verificar_solucao(Encomendas, PercursoIda),
      % Caminho mais curto e ótimo desde o destino até à origem (retornar à base)
      caminho_mais_curto(Destino, centro_distribuicao, [HVolta| PercursoVolta], DistanciaVolta),
      % Concatenar caminho de entregas e de retorno à base e respetivas distâncias
      append(PercursoIda, PercursoVolta, PercursoTotal),
      DistanciaTotal is DistanciaIda + DistanciaVolta,
      % Retornar resultados
      format('+~`-t~78|+ ~n', []),
      format('Percurso de ida -> ~w, Distância de ida -> ~w ~n', [PercursoIda, DistanciaIda]),
      format('Percurso de volta -> ~w, Distância de volta -> ~w ~n ~n', [[HVolta| PercursoVolta], DistanciaVolta]),
      format('Percurso total -> ~w, Distância total -> ~w ~n', [PercursoTotal, DistanciaTotal]),
      format('+~`-t~78|+ ~n', []), !.

% Função que investiga qual dos nós onde é necessário entregar
% encomendas está mais longe do centro de distribuição
% entrega_mais_longe(List, X) :- reverse(List, [X|_]).
entrega_mais_longe([], _, _).
entrega_mais_longe([Encomenda| Encomendas], DistanciaMaxima, _) :-
       caminho_mais_curto(centro_distribuicao, Encomenda, _, DistanciaAtual),
       DistanciaAtual >= DistanciaMaxima,
       entrega_mais_longe(Encomendas, DistanciaAtual, Encomenda).
entrega_mais_longe([Encomenda| Encomendas], DistanciaMaxima, NoMaximo) :-
       caminho_mais_curto(centro_distribuicao, Encomenda, _, DistanciaAtual),
       DistanciaAtual < DistanciaMaxima,
       entrega_mais_longe(Encomendas, DistanciaMaxima, NoMaximo).

% Função que verifica se no conjunto de nós visitados a solução existe
verificar_solucao([], _).
verificar_solucao([Encomenda|Encomendas], Visitados) :-
    member(Encomenda, Visitados),
    verificar_solucao(Encomendas, Visitados).

% Função que itera pelos vários percursos
percurso(A,B,Path,Len) :-
       proximo_no(A,B,[A],Q,Len),
       reverse(Q,Path).

% Função que "viaja" para o nó seguinte do nó onde estamos
proximo_no(A,B,P,[B|P],L) :-
       caminhos(A,B,L).
proximo_no(A,B,Visited,Path,L) :-
       caminhos(A,C,D),
       C \== B,
       \+member(C,Visited),
       proximo_no(C,B,[C|Visited],Path,L1),
       L is D+L1.

% Função que retorna todas as soluções organizadas de forma ótima
% (caminho mais curto)
caminho_mais_curto(A,B,Path,Length) :-
   setof([P,L],percurso(A,B,P,L),Set),
   Set = [_|_], % falha se vazio
   minimo(Set,[Path,Length]).
```

Figura 3.3: Código referente à tarefa 2.

Defini como heurística que, da lista de encomendas, a última a ser completa será a que está mais longe do centro de distribuição (o centro de distribuição corresponde ao ponto a1 no grafo) para ser possível aplicar o algoritmo da forma mais eficaz possível.

Para este exercício, apliquei o algoritmo de Dijkstra e, como o Prolog tem a capacidade de

me ir retornando iterativamente os caminhos mínimos ordernados pela distância, acrescentei uma função de verificar a solução, que procura se, no percurso dado, o estafeta parou em todas as encomendas. Se isso acontecer, retorna essa solução, se não acontecer, passa para o próximo caminho ótimo.

Depois de obter esse caminho, aplico mais um Dijkstra desde o último ponto de entrega até ao contro de distribuição obtendo assim o caminho ótimo para o estafeta voltar à base. Desta forma, todo o percurso fica completo da forma mais ótima possível.

Para experimentar este método, deve iniciar o programa e, já no terminal de execução de prolog, executar o seguinte comando (substitua Encomendas por uma lista de posições do mapa, excepto a a1 que corresponde ao centro de distribuição).

```
?- estafeta(Encomendas).
```

3.3.1 Exemplo de output

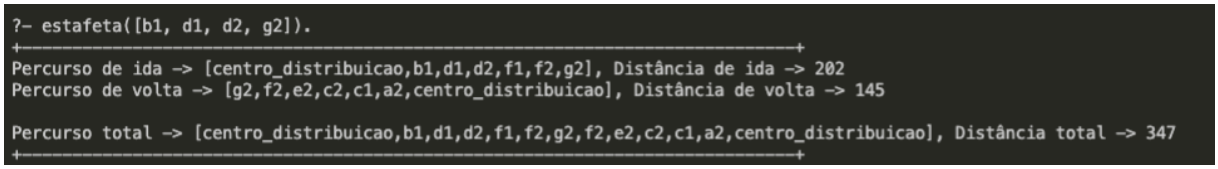

Figura 3.4: Exemplo de execução da tarefa.

3.4 Wrap up da tarefa

Esta tarefa envolveu mais investigação do que código. O algoritmo em si já o tinha explorado também na unidade curricular de Desenho e Análise de Algoritmos, pelo que, a sua aplicabilidade não foi de todo difícil. O processo de investigação é que demorou mais, estava com bastantes dificuldades em entender que tipo de problema se tratava e qual seria a melhor forma para o atacar. O código está bem escrito e da primeira para a segunda tarefa aprendi novos conceitos que me permitiram reduzir bastante o código.

Capítulo 4

Conclusão e notas finais

Sem dúvida que este trabalho conseguiu responder a várias questões que tinha de como o processo de decisão de caminhos funciona, não só no código que desenvolvi mas também em todos os artigos que li ao longo do percurso. O trabalho no fim cumpriu com as minhas expectativas e consegui aplicar os conceitos mais importantes de Prolog. Algo que tinha gostado de aplicar era a OR-Tools da Google, pois parece-me uma ferramente muito poderosa. Também teria sido interessante procurar soluções para grafos que estariam mais aproximados de casos reais, em que o número de nós é muito maior, mas só o futuro dirá se não será o tema da minha Dissertação ☺.

Sobre a linguagem Prolog, não tenho nenhum tipo de comentário negativo sobre a mesma. O meu dia a dia na Coletiv Studio é programar em Elixir onde vários conceitos são muito aproximados de Prolog (apesar do paradigma de Elixir ser funcional). Senti que a comunidade de Prolog é muito restrita, pois Python veio substituir a performance de Prolog e introduz mais capacidades que Prolog não tem. Mesmo com isto, consegui encontrar uma comunidade pequena de Prolog no Discord onde consegui apresentar muitas dúvidas ao longo do desenvolvimento deste projeto e onde vários membros da comunidade conseguiram auxiliar-me ao longo do desafio.