UNIVERSIDADE DO MINHO

TRABALHO PRÁTICO- 1º PARTE

MESTREDO INTEGRADO EM ENGENHARIA BIOMÉDICA
Ramo Informática Médica

PROGRAMAÇÃO EM LÓGICA CONHECIMENTO E RACIOCÍNIO
(1º Semestre- 2014/2015)

Ana Sofia Quintas, 65078

Carmina Azevedo 61777

Margarida Costa, 65037

Braga
Novembro, 2014

Sumário

O presente relatório surge no âmbito da Unidade Curricular de Programação em Lógica, Conhecimento e Raciocínio e teve como principal objetivo a motivação dos alunos para a utilização da linguagem de programação em Lógica PROLOG, e o desenvolvimento de competências para a construção de mecanismos de raciocínio e posterior resolução de problemas. Assim, neste contexto pretende-se desenvolver um sistema de representação de conhecimento e raciocínio com capacidade para caracterizar um universo de discurso relativo à temática localização de serviços. Isto é, dado um conjunto de locais (cidades e/ou regiões), estes devem ser caraterizados por um conjunto de serviços.

Numa primeira fase, foram definidas as bases de conhecimento, de seguida desenvolveu-se funcionalidades obrigatórias de forma a operarem sobre a base de conhecimento, posteriormente procurou-se implementar novas funcionalidades quer ao nível da capacidade de representação de conhecimento quer ao nível das faculdades de raciocínio.

Por último, foram tecidos, no final deste relatório, alguns comentários, críticas e propostas de melhoria ao trabalho efetuado.

Índice

Introdução

Este trabalho escrito surge como resultado do desenvolvimento de um sistema de representação de conhecimento e raciocínio com capacidade para caracterizar um universo de discurso com o qual se pretende abordar a temática de localização de serviços.

Com base no que foi anteriormente descrito, foi desenvolvido uma base de conhecimento caracterizada por locais, que podem ser cidades ou regiões, sendo que em cada cidade ou região existem serviços. Existe, ainda a ligar essas mesmas cidades, estradas, podendo estas ser do tipo auto-estrada, nacional ou caminhos rupestres. Para além disto, foi ainda definido um custo, uma distancia e um tempo para cada ligação em questão.

Para abordagem deste problema recorreu-se à Teoria dos Grafos, descrita mais à frente, em que os locais são modos e ligações dadas pelos arcos.

De forma a interagir com o universo de discurso elaborado, foi construído um caso com aplicação prática de modo a ilustrar as competências representação de conhecimento e a aptidão para resolução de problemas pela construção de mecanismos de raciocínio desenvolvidas pelo grupo. A elaboração deste caso prático respeitou as seguintes funcionalidades:

- Identificar os serviços existentes num determinado local ou região;
- Identificar os locais onde esteja presente um determinado serviço ou conjunto de serviços;
- Identificar os serviços que não se podem encontrar numa determinada região;
- Determinar o caminho entre dois serviços;
- Determinar o caminho entre dois locais;
- Determinar o caminho que permite percorrer uma sequência de serviços.

Para além das funcionalidades já descritas, foram incluídas novas funcionalidades quer ao nível quer ao nível das capacidades de representação de conhecimento quer ao nível das faculdades de raciocínio.

Preliminares

O PROLOG (PROgramação em LÓGica) surgiu na década de 70 como uma linguagem apoiada no raciocínio matemático, que integra o paradigma declarativo/descritivo da programação. Este paradigma sugere a implementação de uma descrição do problema como método para a sua resolução, ao contrário das linguagens procedimentais que implementam um conjunto de instruções para o mesmo efeito, tal como é o caso do Java e C/C++, por exemplo [1].

Por se tratar de uma forma radicalmente diferente de programar, o Prolog não teve, inicialmente, muita adesão. Contudo, ao longo dos tempos, veio a ganhar popularidade em determinadas áreas, nomeadamente na demonstração de teoremas matemáticos; nos domínios da computação simbólica e da linguagem natural [2].

Programa - Conjunto de especificações em cláusulas de Horn (factos e regras) que estabelecem relações entre objetos. Um programa descreve conhecimento e pressupostos necessários à resolução de um problema. Chama-se a isto a base de conhecimento de um programa Prolog;

Problema - Asserção lógica a ser demonstrada - objetivo;

Execução - Demonstração do objetivo, através da dedução de consequência lógicas da base de conhecimento. Esta dedução é efetuada por um ambiente de execução, que é por norma um interpretador (na elaboração deste projeto utilizou-se o SICSTUS), que interpreta expressões como perguntas às quais se procura responder [3].

A linguagem Prolog herda da lógica de predicados três elementos básicos: factos, regras e consultas. Um predicado é um identificador de relações e é composto por uma ou mais cláusulas de Horn. Um predicado pode, então, ser um facto ou uma regra. Um facto tem valor verdadeiro incondicionalmente. Trata-se de uma cláusula sem condições, isto é, de corpo vazio [4]. Por exemplo:

```
pai( manuel,joaquim ).
pai( manuel,filipe ).
filho( joaquim,manuel ).
```

```
filho( filipe,manuel ).
homem( manuel ).
```

Os primeiros quatro factos enunciados têm aridade 2 e devem ler-se, respetivamente: “O Manuel é pai do Joaquim”; “O Manuel é pai do Filipe”; “O Joaquim é filho do Manuel" e “O Filipe é filho do Manuel”. O último facto *homem* tem aridade 1 e deve ler-se “O Manuel é um homem”.

A aridade de um predicado é uma caraterística que indica o seu número de argumentos. É de especial importância já que predicados com nomes iguais mas aridades diferentes são predicados distintos [5]. Note-se que os predicados são escritos com letra minúscula propositadamente. O predicado pai que se segue, por sua vez, é uma regra (cláusula de verdade condicional, composta por uma ou mais condições):

```
pai( X,Y ) :- filho( Y,X ), homem( X ).
```

Deve ler-se “X é pai de Y se Y é filho de X e X é homem”. O símbolo :- lê-se “se” e antecede o corpo da cláusula, constituído por condições. A vírgula que separa essas condições lê-se “e”, indicando conjunção. Se se pretendesse uma disjunção (”ou") o símbolo correto seria o ; (ponto e vírgula).

Para que uma regra seja verdadeira, todas as condições que a constituem devem ser verdadeiras [3].

Resta apenas especificar as cláusulas do predicado filho:

```
filho( Y,X ) :- pai( X,Y ).
```

Uma vez comunicado ao programa ao sistema Prolog, este é capaz de responder a perguntas que o utilizador lhe possa colocar, através do consultas (queries) digitadas no “prompt” do interpretador Prolog:

```
?- homem( manuel ).
yes

?- pai( manuel,joaquim ).
yes
```

No primeiro caso, o utilizador está a perguntar: “O Manuel é homem?”. Após consultar a base de conhecimento, o Prolog encontra a regra homem(manuel), respondendo ”yes”. No segundo caso, o utilizador está a consultar se o tuplo (manuel,joaquim) satisfaz o preficado pai. O ponto final indica o final da consulta. Como seria esperado, o Prolog verifica que o facto pertence à sua base de conhecimento e, portanto, responde afirmativamente.

```
?- homem( joaquim ).
no
```

O Prolog verifica que não tem informação suficiente na sua base de conhecimento para deduzir a resposta e, consequentemente, responde que não.
As consultas tornam-se mais apelativas quando são empregues variáveis:

```
?- pai( manuel,X ).
X = joaquim ? no
X = filipe? yes
yes
```

Repare-se que o utilizador questionou qual o valor X que torna a consulta uma consequência lógica do programa. O interpretador verifica a base de conhecimento e responde com a primeira resposta encontrada (por ordem de cláusulas definidas). Contudo, quando há mais que uma resposta para uma consulta, o utilizador pode interferir digitanto ”yes” ou “no”, consoante esteja satisfeito ou não, com a resposta encontrada. Em caso negativo, tal como sucede na primeira etapa do exemplo, o Prolog é forçado a apresentar as próximas respostas possíveis, por um mecanismo de *backtracking*. Isto é, quando não existem mais metas a ser percorridas numa linha de resolução, o Prolog utiliza um sistema de retrocesso. Tal pode ser entendido como uma procura numa árvore, realizada de cima para baixo, a partir de um elemento raiz: quando se atinge a profundidade máxima de um ramo, deve-se retroceder à raiz para procurar um novo ramo de alternativa [6].

Quando o Prolog responde “yes” a uma consulta diz-se que ocorreu uma unificação (*matching*). Dois termos unificam numa de duas situações:

- Quando são idênticos (é o caso da primeira consulta de exemplo efetuada);

- Quando há uma instanciação para as variáveis dos dois termos que os torna iguais (como é o caso da última consulta de exemplo efetuada: o termo X unifica com o termo filipe) [6].

A Sintaxe Prolog

Um predicado designa-se também por estrutura (ou elemento complexo) e é um dos quatro termos manipulados em Prolog. Os restantes termos podem ser átomos, números ou variáveis.

Os átomos e os números pertencem ao grupo das constantes. Por sua vez, as constantes em conjunto com as variáveis formam os objetos simples do Prolog. [4]

Um átomo pode ser:

- Uma *string* de carateres iniciada por letra minúscula;
- Uma sequência arbitrária de carateres contida entre os carateres "e", por exemplo: "Vicente";
- Uma string de carateres especiais, como por exemplo =, ; e :- (como já foi mencionado, algumas destas *strings* têm significado especial).

Os números reais não têm particular importância em Prolog. Por outro lado, os números inteiros são úteis para serem manipulados, por exemplo, em funções de contagem de elementos de uma lista;

- Uma variável é uma *string* iniciada por letra maiúscula ou por um *underscore* no caso de se tratar de uma variável anónima

Finalmente, é possível constatar que na forma de uma estrutura, que é a seguinte:

função(argumento_1,..., argumento_n).

A função deverá ser um átomo, enquanto que os argumentos podem ser qualquer tipo de termo Prolog, inclusivé estruturas. [5]

Teoria de Grafos

Os grafos são uma das estruturas mais importantes em programação. As listas e árvores, por exemplo, são casos particulares de grafos. A manipulação de grafos tem grande aplicação nas áreas da produção industrial, investigação operacional, inteligência artificial, entre outras [7].

Um grafo é representado por uma coleção de pontos e um conjunto de ligações entre esses pontos [6]. Porém, por vezes a sua representação gráfica não é relevante. Por exemplo, os grafos da figura 1 são o mesmo grafo.

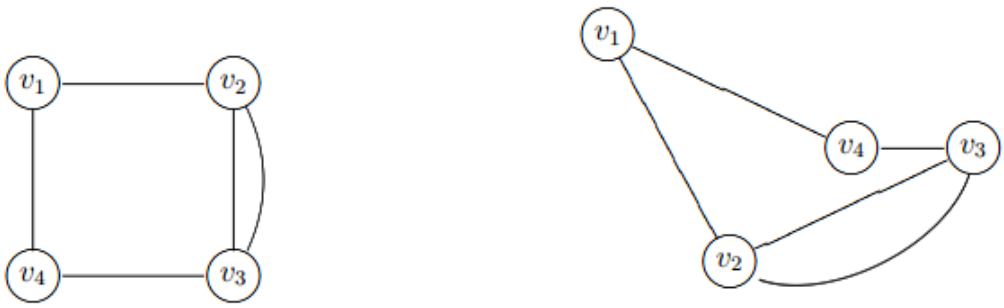


Figura 1 – Grafos com aspeto visual distinto mas que contêm a mesma informação

Os grafos podem ser tipo dirigido ou não dirigido. Num grafo dirigido (ou orientado) o sentido de a para b é semanticamente diferente do de b para a, e a ligação é representada por uma seta, indicando o sentido do arco. Já num grafo não dirigido, o sentido de a para b é semanticamente igual ao de b para a (sendo a e b dois nodos extremos de uma aresta). Na elaboração deste projeto utiliza-se, maioritamente, grafos não dirigidos.
Em linguagem Prolog, um grafo pode ser representado de duas formas:

1. Como um só objeto, por exemplo:

G=(V,A,g).

Onde:

G - Designação do grafo;

V - Conjunto não-vazio de vértices (nodos);

A – Arestas/ligações/arcos;

g - função que associa cada aresta a um par de vértices (extremos da aresta).

2. Pela representação de cada aresta/arco como uma cláusula, por exemplo:

arco(a,b,3).
arco(b,c,4).
arco(d,b,2). [7]

Os grafos são extremamente úteis na representação de caminhos, sendo considerados os locais como nodos e as ligações entre eles, os arcos.

Descrição do Problema

O problema desenvolvido baseou-se na existência de três cidades, o Porto, Lisboa e Braga, constituída por um conjunto de serviços, hospitais, farmácias e praias. Este universo de discurso é, simultaneamente, caracterizado, pelas ligações entre os locais. Estas ligações incluem informação sobre o tipo de ligação (auto-estrada, via municipal ou caminho rupestre), sobre os locais de origem e destino e, ainda, sobre uma medida do custo dessa ligação (distância, valor monetário e período de tempo). O serviço farmácias é ainda caracterizado por um horário de funcionamento especifico para cada farmácia, onde existem farmácias que estão permanentemente abertas, e outras têm apenas um horário de funcionamento das 9h às 21h.

Com base no enunciado do problema, foi então criado o seguinte caso prático: "A D.Manuela, tem 75 anos, e vive no centro do Porto, apesar da sua idade avançada é uma pessoa independente que gosta de viajar e visitar os seus dois filhos, o António que vive no centro de Braga, e o Rui que vive no centro de Lisboa. Preocupados com a sua saúde, e o bem estar da sua mãe, os seus dois filhos resolveram criar um mapa onde assinalaram os seus locais de residências, as farmácias das suas cidades, assim como os hospitais, de modo a que a sua mãe se pudesse orientar melhor num situação de emergência. Mais tarde acrescentaram as praias favoritas da sua mãe onde ela passa a maioria do seu tempo." Este mapa encontra-se representado na imagem 1 em anexo.

Com o caso prático anteriormente referido, é assim possível desenvolver funcionalidades, úteis, e com aplicação ao caso prático criado

Base de Conhecimento

- **Conhecimento relativo aos locais**

No conhecimento relativos aos locais, implementou-se dois factos relativos às regiões e às cidades. A primeira recebe como argumento "centro" ou "norte", sendo que a segunda recebe "braga", "porto" e "lisboa". Ainda foi adicionado o facto que relaciona a regiao com a cidade, sendo o pertence.

```
regiao( centro ).
regiao( norte ).

cidade( braga ).
cidade( porto ).
cidade( lisboa ).

pertence(braga,norte).
pertence(porto,norte).
pertence(lisboa,centro).
```

- **Conhecimento relativo aos serviços**

Relativamente aos serviços, foram implementados três tipos de serviços: a "farmacia", o "hospital" e a "praia".

```
servico( farmacia ).
servico( hospital ).
servico( praia ).
```

Ainda foi criado o facto *horario_farmacia* que tem como argumentos a hora, o nome da farmácia, e o estado ("aberto" ou "fechado").

```
horario_farmacia(H:M, farmacia_antunes, fechado):- H>=20, H <24, M>|=00, M <60.
```

horario_farmacia(H:M, farmacia_antunes, aberto):- H >=9, H <20, M>=00, M <60.
horario_farmacia(H:M, farmacia_antunes, invalido):- H>=24.
horario_farmacia(H:M, farmacia_antunes, invalido):- M >=60.

horario_farmacia(H:M, farmacia_dias, aberto):- H>00, H <24, M>=00, M <60.
horario_farmacia(H:M, farmacia_dias, invalido):- H>=24.
horario_farmacia(H:M, farmacia_dias, invalido):- M >=60.

horario_farmacia(H:M, farmacia_gago,fechado):- H>=20, H <24, M>=00, M <60.
horario_farmacia(H:M, farmacia_gago, aberto):- H>=9, H <20, M>=00, M <60.
horario_farmacia(H:M, farmacia_gago, invalido):- H>=24.
horario_farmacia(H:M, farmacia_gago, invalido):- M >=60.

horario_farmacia(H:M, farmacia_fernandes, aberto):- H>00, H <24, M>=00, M <60.
horario_farmacia(H:M, farmacia_fernandes, invalido):- H>=24.
horario_farmacia(H:M, farmacia_fernandes, invalido):- M >=60.

horario_farmacia(H:M, farmacia_natercia,fechado):- H>=20, H <24, M>=00, M <60.
horario_farmacia(H:M,farmacia_natercia, aberto):- H>=9, H <20, M>=00, M <60.
horario_farmacia(H:M, farmacia_natercia, invalido):- H>=24.
horario_farmacia(H:M, farmacia_natercia, invalido):- M >=60.

- **Conhecimento relativo às coordenadas**

Relativamente ao conhecimento relativo às coordenadas, foi criado o facto *ponto_descricao*, que possui argumentos relativos ao tipo de ponto, ou seja se é um serviço ou cidade, seguido do tipo de serviço, ou no caso, posteriormente o tipo de ponto e por ultimo as duas coordenadas no X e no Y de um plano.

ponto_descricao(servico, praia, praia_mira_sol, braga, 3,3).
ponto_descricao(servico, farmacia, farmacia_natercia, braga, 10,1).
ponto_descricao(servico, hospital, hospital_braga, braga, 8,7).
ponto_descricao(cidade, centro, centro_braga, braga, 13, 5).

ponto_descricao(cidade, centro, centro_porto, porto, 5,9).
ponto_descricao(servico, praia, praia_esposende, porto, 15,15).
ponto_descricao(servico, farmacia, farmacia_gago, porto, 8,13).
ponto_descricao(servico, hospital, hospital_porto, porto, 7,11).
ponto_descricao(servico, farmacia, farmacia_fernandes, porto, 2, 14).

ponto_descricao(cidade, centro, centro_lisboa, lisboa, 5, 18).
ponto_descricao(servico, farmacia, farmacia_antunes,lisboa, 3, 22).
ponto_descricao(servico, hospital, hospital_lisboa, lisboa, 11, 21).
ponto_descricao(servico, farmacia, farmacia_dias, lisboa, 14, 17).

- **Conhecimento relativo aos arcos**

Por último, no conhecimento relativo aos arcos, os factos que se seguem, pretendem caracterizar o arco em questão, sendo que primeiramente temos a origem do arco, seguido do destino, o tipo de estrada pelo qual é possível passar, o nome da estrada e por último a velocidade máxima a que é possível transitar nessa estrada.

arco(farmacia_antunes, hospital_lisboa,nacional, n1 ,80).
arco(hospital_lisboa, centro_lisboa, nacional, n2 ,80).
arco(farmacia_antunes, centro_lisboa,rupestre, r1 ,50).
arco(hospital_lisboa,farmacia_dias, rupestre, r2 ,50).
arco(farmacia_dias, praia_esposende,rupestre, r3 ,50).
arco(farmacia_antunes, farmacia_fernandes, auto_estrada, a2 ,120).
arco(centro_lisboa, hospital_porto, auto_estrada, a3 ,120).
arco(hospital_lisboa, farmacia_gago, auto_estrada, a4 ,120).
arco(farmacia_gago, hospital_porto, rupestre, r4, 50).
arco(farmacia_fernandes, hospital_porto, nacional,n3, 80).
arco(praia_esposende, hospital_braga,auto_estrada, a5,120).
arco(praia_esposende, centro_braga, auto_estrada, a6,120).
arco(hospital_porto, hospital_braga, nacional, n4, 80).
arco(hospital_porto, centro_porto, rupestre, r5, 50).
arco(farmacia_fernandes, praia_mira_sol, auto_estrada, a7, 120).
arco(centro_porto, praia_mira_sol, nacional, n5, 80).

arco(hospital_braga, praia_mira_sol, rupestre, r6, 50).

arco(hospital_braga, farmacia_natercia, nacional, n6, 80).

arco(praia_mira_sol, farmacia_natercia, nacional, n7, 80).

arco(farmacia_natercia, centro_braga, rupestre, r7,50).

Funcionalidades de Serviços e Caminhos

Identificar os serviços existentes num determinado local ou região

servicos_num_local: L, LS -> {V,F}

Onde:

[L] - Local (que pode ser uma cidade ou uma região);
[LS] - Lista de serviços existentes no local L.

servicos_num_local(L,LS) :-
findall(S,tem_servico(L,S),Lista),
elimina_duplicados(Lista,LS).

```
| ?-
| ?- servicos_num_local(porto,LS).
LS = [praia,hospital,farmacia] ?
```

Figura 2: Teste ao predicado *serviços_num_local*

Dado um local **L**, é utilizado um procedimento *built-in* denominado *findall* que devolve uma Lista que contém todos os elementos **S** (serviço), que satisfaçam o predicado *tem_servico(L,S)*. Dado que, em alguns casos, existe mais do que uma variante de cada tipo de serviço num só local, recorreu-se seguidamente ao predicado auxiliar *elimina_duplicados(Lista,Ls)* que elimina todas as repetições de elementos na Lista, devolvendo a lista final de serviços LS que existem no local **L**.

Identificar os locais onde esteja presente um determinado serviço ou conjunto de serviços

cidades_com_servico: S, C -> {V,F}

Onde:

[S] - Serviço;
[LC] - Lista de cidades que contêm o serviço S.

cidades_com_servico(S,LC) :-
findall(C,(cidade(C),tem_servico(C,S)),Lista),
elimina_duplicados(Lista,LC).

```
| ?- cidades_com_servico(farmacia,LC).
LC = [braga,porto,lisboa] ?
```

Figura 3: Teste ao predicado *cidades_com_servico*

Dado um serviço **S** é utilizado o procedimento *built-in findall* que devolve uma lista (**Lista**) que contém todos os elementos **C** (cidade), que satisfaçam o predicado *tem_servico(C,S)*. Dado que, em alguns casos, existe mais do que uma variante de cada tipo de serviço num só local, recorreu-se seguidamente ao predicado *elimina_duplicados(Lista,Ls)* que elimina todas as repetições de elementos na Lista, devolvendo a lista final de cidades **LC** que contêm o serviço **S**, tal como é possível visualizar na imagem 3.

Identificar os serviços que não se podem encontrar numa determinada região

servicos_que_nao_existem: L, LS -> {V,F}

Onde:

[L] - Local;
[LS] - Lista de serviços que não existem no local L.

servicos_que_nao_existem(L,LS) :-
findall(S,servico(S),LA1),
servicos_num_local(L,LA2),
elim12(LA2,LA1,LS).

```
| ?- servicos_que_nao_existem(lisboa,LS).
LS = [praia] ?
```

Figura 4: Teste ao predicado *serviços_que_nao_existem*

Dado um serviço **L** é utilizado o procedimento *built-in findall* que devolve uma lista auxiliar1, **LA1**, que contém todos os elementos **S** (serviço) que satisfazem o predicado *servico(C,S)*. Seguidamente é utilizado o predicado *servicos_num_local* que devolve uma lista auxiliar2, **LA2**, que contém todos os serviços existentes no local **L**. Finalmente, é utilizado o predicado auxiliar *elim12* que recebe as listas **LA2** e **LA1** e devolve a lista **LS**. Desta forma, tal como observado na figura 4 é possível verificar os serviços que não existem numa determinada cidade.

Identificar as farmácias abertas, a uma determinada hora, para uma dada Cidade.

farmacia: H:M,Cidade,Estado,Nome -> {V,F}

Onde:

[H]: Hora;
[M]: Minutos;
[C]: Cidade onde se se pretende verificar a existência de farmácias abertas;
[E]: Estado. Pode aceitar os átomos "aberto", ou "fechado", dependendo daquilo que se pretende consultar;
[NF]: Nome da farmácia, que corresponde aos dados anteriormente inseridos pelo utilizador;

```
farmacia( H:M,C,E,NF ) :-
    ponto_descricao(servico, farmacia, NF, C,_,_),
        horario_farmacia( H:M,NF,E ).
```

```
| ?- farmacia(23:00, porto, aberto, NF).
NF = farmacia_fernandes ? yes
yes
```

Figura 5: Teste ao predicado *farmacia*

Dada, uma certa hora, no formato Hora:Minuto, **H:M**, é possível verificar para uma determinada Cidade, **C**, quais as farmácias num determinado estado, **E**, (aberto ou fechado), e receber o nome das farmácias, **NF**, que correspondem a estes dados introduzidos, uma a uma. Para isso, foi necessário recorrer aos predicados *ponto_descricao* e *horario_farmacia* que permitem satisfazer as condições inseridas no predicado *farmacia*. Só de notar, que no predicado *ponto_descricao*, foi introduzido "serviço" e "farmácia" como átomo para que a informação resultante desse mesmo predicado fosse apenas referente ao serviço farmácias.
Neste exemplo foi introduzida como hora 23:00, como C "porto", e como E "aberto". Assim às 23:00h a farmácia aberta no porto é a Farmácia Fernandes.

Com o intuito de facilitar a consulta do predicado anteriormente descrito, foi acrescentado ainda um outro que introduzidos dos dados **H:M**, **C, E** e **NF**, em vez de retornar as farmácias, uma a uma, fornece diretamente uma lista com o nome das farmácias. Assim, foi construído o seguinte predicado *lista_farmacias*, que recorrendo ao procedimento *built-in* denominado *findall* devolve uma lista das farmácias **LF** que contém todos os elementos, tal como mostra a figura 6.

```
lista_farmacias( H:M,C,E,LF ) :-
        findall( NF,farmacia( H:M,C,E,NF ),LF ).
```

```
| ?- lista_farmacias(11:00, lisboa, aberto, LF).
LF = [farmacia_antunes,farmacia_dias] ? yes
yes
| ?-
```

Figura 6: Teste ao predicado *lista_farmacias*

Onde:

[H]: Hora;
[M]: minutos;
[C]: Cidade onde se se pretende verificar a existência das farmácias abertas;

[E]: Estado. Pode tomar o valor de aberto, ou fechado, dependendo daquilo que se pretende consultar;

[LF]: Lista das farmácias, que vão ao encontro dos dados introduzidos pelo utilizador;

Determinar o caminho entre dois locais ou serviços

caminho: O, D, Cam -> {V,F}

Onde:

[O] - Local de origem;

[D] - Local de destino;

[Cam] - Lista que contém o caminho, cujos elementos são os nomes das estradas que permitem o deslocamento.

caminho(O,D,Cam) :-
evita_ciclos(O,D,Cam,[ ]).

```
| ?- caminho(farmacia_dias,hospital_porto,Cam).
Cam = [r3,a5,n4] ? no
Cam = [r3,a5,r6,a7,n3] ? yes
yes
| ?-
```

Figura 7: Teste ao predicado *caminho*

Dado um local de origem **O** e de destino **D** atribuídos pelo utilizador, o predicado caminho calcula um percurso **Cam**, através da manipulação do conhecimento *evita_ciclos*. A extensão deste predicado auxiliar encontra-se na secção seguinte.

Determinar o caminho que permite percorrer uma sequência de serviços

seq_servicos_caminho: LS, Cam -> {V,F}

Onde:

[LS] - Lista de serviços;

[Cam] - Caminho (lista de estradas) necessário para percorrer uma determina lista de serviços;

```
seq_servicos_caminho([H|[ ]],[ ]).

seq_servicos_caminho( [H | [H1 | T1]],[H2 \| T2] ) :-
        caminho( H,H1,H2 ),
        apagarPri( [H | [H1 | T1]],LA ),
        seq_servicos_caminho( LA,T2 ).
```

A primeira cláusula deste predicado salvaguarda a situação em que a lista de serviços **LS** inserida pelo utilizador é uma lista de um só elemento, ou seja, contém um único serviço. Assim, o caminho para percorrer **LS** é uma lista vazia.

A segunda cláusula aplica-se quando **LS** é uma lista composta por dois ou mais elementos. A determinação do caminho realizado para a percorrer consiste, primeiramente, na determinação do caminho entre os dois primeiros serviços (elementos) que compõe a lista, que são **H** e **H1**, efetuado através da manipulação do conhecimento caminho. Esse caminho será então a cabeça **H2** da lista **Cam**. Em seguida é utilizado o predicado auxiliar *apagarPri* que recebe a lista de serviços **LS** e devolve a lista auxiliar **LA**, que consiste na lista **LS** sem o primeiro elemento **H**. Por fim, utiliza-se, recursivamente, o predicado *seq_servicos_caminho* que recebe a lista de serviços LA e devolve o caminho que lhe corresponde. Este, por sua vez, será a cauda **T2** da lista **Cam**. O resultado deste predicado está evidenciado na figura 8.

```
| ?- seq_servicos_caminho([praia_esposende,hospital_porto,farmacia_fernandes],
     Cam).
Cam = [[a5,n4],[n3]] ?
yes
```

Figura 8: Teste ao predicado *seq_servicos_caminho*

Identificar os caminhos, que permitem ir de um dado serviço ou local, para um outro local ou serviço, sem passar por um determinado, tipo de estrada

nao_passa_numa_estrada: O, D, E, P, Dis ->{V,F}

Onde:

[O]: Origem;

[D]:Destino;

[E]: Tipo de estrada;

[P]: Lista dos percursos obtidos.

[X]: Lista dos percursos obtidos.

```
nao_passa_numa_estrada(O, D , E, P):-
   findall(X, nao_passa(O, D , E, X  ), K ),
   elimina_todos_niveis([ ], K, P).
```

```
| ?- nao_passa_numa_estrada(farmacia_natercia, praia_mira_sol, nacional, X).
X = [[r7,a6,a5,r6],[r7,a6,r3,r2,a4,r4,a3,r1,a2|...]] ?
```

Figura 9: Teste ao predicado *nao_passa_numa_estrada*

```
nao_passa(O, D , E, X  ):-
   caminho(O,D,CAM ),
   findall(NE, arco2(_,_,E ,NE ,_ ), Lista),
   pertenceListaRetornaVazia(CAM, Lista, X).
```

```
| ?- nao_passa(farmacia_natercia, praia_mira_sol, nacional, X).
X = [] ? no
X = [r7,a6,a5,r6] ? no
X = [] ? no
X = [] ? no
X = [] ?
```

Figura 10: Teste ao predicado *nao_passa*

As duas funções anteriores funcionam em conjunto, de modo a determinar os possíveis caminhos entre dois pontos, sem passar por um determinado tipo de estrada. Para tal, o predicado *nao_passa*, recorre a três predicados. No primeiro predicado caminho, é obtida a lista **CAM** com todos os caminhos possíveis entre a origem e o destino pretendido. Posteriormente é utilizado um procedimento *built-in* denominado *findall* de modo a encontrar os nomes das estradas **NE** , que corresponde ao tipo de estrada introduzido, sendo os nomes das estradas guardado dentro da Lista. Por ultimo,

as listas **CAM** e Lista são usadas no predicado *pertenceListaRetornaVazia* que faz as comparações entre as duas listas, e caso estas tenham elementos comuns o resultado é uma lista vazia, caso não haja comuns o resultado é a própria lista, Lista. Desta forma, sempre este predicado *nao_passa* é usado, se o caminho encontrado contiver a estrada pelo qual não se quer passar, ele retorna uma lista vazia, por outro lado, se o caminho não contiver esse mesmo tipo de estrada, ele surge como uma lista normal, tal como se verifica a imagem 10.

Tendo em conta o predicado anteriormente descrito, o predicado *nao_passa_numa_estrada*, serviu para fazer um *findall* de todos os caminhos resultantes da função *nao_passa*, aglomerando-os como elementos da lista **K**. Como este *findall* encontrava as tais listas vazias, que eram listas que continham os caminhos com o tipo de estrada pelo qual não se queria passar, assim como os caminhos que são úteis, recorreu-se ao predicado auxiliar *elimina_todos_niveis*, que eliminou todas as listas vazias encontradas, dentro da lista **K**, e colocando o resultado na lista **P**, tal como se pode ver na imagem 9.

Determinar a distância entre dois serviços ou locais

distanciaCaminho: O, D, Cam1, Cam2, Dis ->{V,F}

Onde:
[O] - Serviço/local de origem;
[D] - Serviço/local de destino;
[Cam1] - Cabeça da lista que devolve o caminho;
[Cam2] - Cauda da lista que devolve o caminho;
[Dis] - Distância entre os dois locais/serviços.

```
distanciaCaminho( O,O,[ ],0 ).

distanciaCaminho( O,D,[Cam1 | Cam2],Dis ) :-
   distancia2pontosconsecutivos( O,A,Cam1,Dis1),
   distanciaCaminho( A,D,Cam2,Dis2 ),
   Dis is Dis1 + Dis2.
```

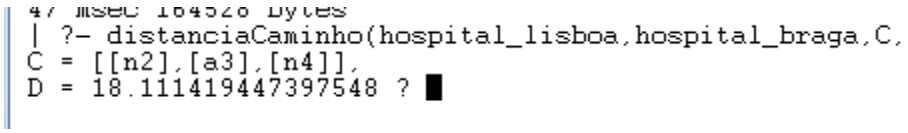


Figura 11: Teste ao predicado *distancia_caminho*

O predicado *distanciaCaminho*, como o nome indica, retribui a distância total a percorrer entre um local/serviço de origem **O** e outro de destino **D**, assim como o respetivo caminho percorrido. Caso se apresente o mesmo ponto de origem e de destino **A**, a distância percorrida será, obviamente, 0 km e o caminho uma lista vazia ([ ]). Caso contrário, dado um ponto de origem e outro de destino, a função calculará, para o primeiro arco utilizado no caminho, recorrendo à função distancia2pontosconsecutivos, a distância percorrida. Através da recursividade, a distância será, de igual modo, calculada para os restantes arcos utilizados no caminho. A distância total percorrida é, então, dada pela soma das distâncias percorridas em cada arco.

.

Determinar o caminho mais curto entre dois serviços/locais

caminho_mais_curto2: O, D, Cam, Dis -> {V,F}

Onde:

[O] - Serviço/ local de origem;

[D] - Serviço/ local de destino;

[Cam] - Caminho mais curto;

[Dis] - Comprimento do caminho mais curto.

```
caminho_mais_curto2( O,D,Cam,Dis ) :-
    distanciaCaminho(O,D,Cam,Dis),!.
```

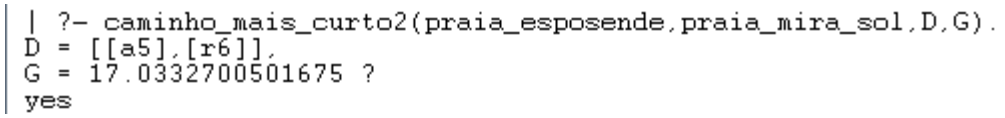


Figura 12: Teste ao predicado *caminho_mais_curto*

O predicado *caminho_mais_curto2* tem como objetivo encontrar o caminho de menor comprimento entre um local/serviço de origem e outro de destino, fornecidos pelo utilizador. Uma vez que a função *distanciaCaminho* retribui as distâncias por ordem crescente, a primeira distância que retribuir corresponderá ao caminho mais curto. Desta forma aplicou-se o comando *cut* (!) para indicar ao *Prolog* que não necessitam de ser considerados os restantes comprimentos, abortando o processo de *backtracking*. No final, o predicado retribui o primeiro comprimento fornecido pela função *distanciaCaminho* e o caminho que lhe corresponde, o caminho mais curto, tal como é visível na figura 12.

Determinar o custo entre dois serviços/locais

custoCaminho: O, D, Dis, Cam, Cus -> {V,F}

Onde:

[O] - Serviço/ local de origem;

[D] - Serviço/ local de destino;

[Dis] - Distância entre os dois locais/serviços.

[Cam] - Lista que devolve o caminho;

[Cus] - Custo do caminho;

```
custoCaminho( O,O,[ ],0 ).
custoCaminho( O,D,Cam,Cus ) :-
  distanciaCaminho( O,D,Cam,Dis ),
  Cus is Dis * 0.091.
```

```
| goal:  user:custocaminho(hospital_lisboa,hospital_brag
| ?- custoCaminho(hospital_lisboa,hospital_braga,C,D).
C = [[n2],[a3],[n4]],
D = 1.6481391697131769 ? yes
yes
| ?-
```

Figura 13: Teste ao predicado *caminhoCusto*

O predicado *custoCaminho* devolve o caminho percorrido, de um local/serviço de origem até outro de destino, e o custo dessa viagem em euros. Para desenvolver esta função recorreu-se à função *distânciaCaminho* e considerou-se um preço médio do gasóleo de 1,3€/l e um consumo médio de gasóleo de 7l/100km, resultando um custo

médio por Km de 0,091€/km. A utilização deste predicado está evidenciado na Figura 13.

Determinar o caminho mais barato entre dois serviços/locais

caminho_mais_barato2: O, D, Cam, Cus -> {V,F}

Onde:

[O] - Serviço/ local de origem;

[D] - Serviço/ local de destino;

[Cam] - Caminho mais barato;

[Cus] - Custo do caminho mais barato.

```
caminho_mais_barato2( O,D,Cam,Cus ) :-
    custoCaminho(O,D,Cam,Cus),!.
```

```
| ?- caminho_mais_barato2(praia_esposende,praia_mira_sol,D,G).
D = [[a5],[r6]],
G = 1.5500275745652423 ? NO
```

Figura 14: Teste ao predicado *caminho_mais_barato_2*

O *predicado caminho_mais_barato2* tem como objetivo encontrar o caminho de menor custo entre um local/serviço de origem e outro de destino, fornecidos pelo utilizador. Uma vez que a função *custoCaminho* retribui os tempos por ordem crescente, o primeiro tempo que retribuir corresponderá ao caminho mais barato. Desta forma aplicou-se o comando *cut* (!) para indicar ao *Prolog* que não necessitam de ser considerados os restantes tempos, abortando o processo de *backtracking*. No final, o predicado retribui o primeiro tempo fornecido pela função *custoCaminho* e o caminho que lhe corresponde, o caminho mais barato.

Determinar o tempo necessário para a deslocação entre dois serviços/locais

tempoCaminho: O, D, Cam1, Cam2, Tem1, Tem2, Tempo -> {V,F}

Onde:

[O] - Serviço ou local de origem;

[D] - Serviço ou local de destino;

[Cam1] - Cabeça da lista que devolve o caminho;

[Cam2] - Cauda da lista que devolve o caminho;

[Tempo] - Tempo entre os dois locais/serviços.

```
tempoCaminho(O,O,[ ],0).
    tempoCaminho(O,D,[Cam1 | Cam2],Tempo):-
        tempo2pontosconsecutivos(O,A,Cam1,Tem1),
        tempoCaminho(A,D,Cam2,Tem2),
        Tempo is Tem1 + Tem2.
```

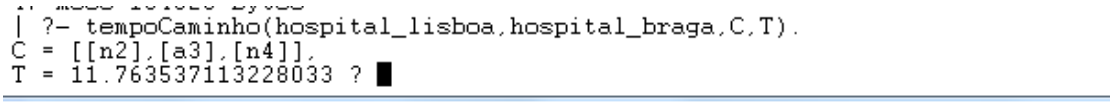


Figura 15: Teste ao predicado *tempoCaminho*

O predicado *tempoCaminho*, retribui ao utilizador o caminho a percorrer, entre dois pontos de origem e destino, e o tempo dessa viagem (em horas). Para o primeiro arco utilizado no caminho, recorre-se ao predicado auxiliar *tempo2pontosconsecutivos*, para determinar o tempo do caminho. Através da recursividade, o tempo será, de igual modo, calculado para todos os restantes arcos utilizados no caminho. **Tem1** e **Tem2** são variáveis auxiliares do tempo que armazenam temporariamente os tempos que demora a percorrer arcos do caminho. Da soma destes valores resulta o tempo total demorado, **Tempo,** desde o ponto de origem até ao ponto de destino. O teste a este predicado está evidenciado na figura 15.

Determinar o caminho mais rápido entre dois serviços/locais

caminho_mais_rapido2: O, D, Cam, Tem ->{V,F}

Onde:

[O] - Serviço/ local de origem;

[D] - Serviço/ local de destino;

[Cam] - Caminho mais rápido;

[Tem] - Tempo que demora a percorrer o caminho mais rápido.

```
caminho_mais_rapido2( O,D,Cam,Tem ) :-
    tempoCaminho(O,D,Cam,Tem),!.
```

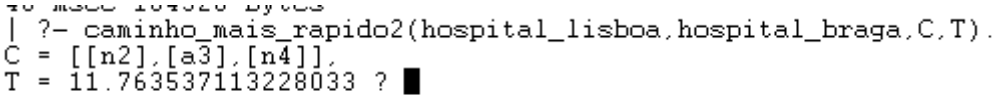


Figura 16: Teste ao predicado *caminho_mais_rapido2*

O predicado *caminho_mais_rapido2* tem como objetivo encontrar o caminho que demora menos tempo entre um local/serviço de origem e outro de destino, fornecidos pelo utilizador. Uma vez que a função *tempoCaminho* retribui os tempos por ordem crescente, o primeiro tempo que retribuir corresponderá ao caminho mais rápido. Desta forma aplicou-se o comando *cut* (!) para indicar ao *Prolog* que não necessitam de ser considerados os restantes tempos, abortando o processo de *backtracking*. No final, o predicado retribui o primeiro tempo fornecido pela função *tempoCaminho* e o caminho que lhe corresponde, o caminho mais rápido. Este predicado foi testado como demonstra a imagem 16.

Predicados auxiliares

- **Verificar se determinado elemento é um local**

Local: A -> {V,F}

```
local( A ) :-
```

```
        cidade( A ).
local( A ) :-
        regiao( A ).
```

Este predicado dita que determinado elemento **A** é um local se uma cidade ou uma região, consultando a base de conhecimento, nomeadamente as regras cidade e região.

- **Verficar se determinado local tem determinado serviço**

tem_servico: R,S -> {V,F}

```
tem_servico( R,S ) :-
        regiao( R ),
   pertence( C,R ),
        ponto_descricao(servico,S,_,C,_,_).

tem_servico(C,S):-
   ponto_descricao(servico,S,_,C,_,_).
```

A primeira cláusula do predicado *tem_servico* salvaguarda o caso em que o local **L** é uma região **R**. Assim, primeiramente, verifica que se trata de uma região através da regra regiao. Seguidamente é verificado se uma dada cidade **C** pertence a essa região **R.** Por último, é consultada a base de conhecimento, nomeadamente, a regra *ponto_descricao,* indicando explicitamente que o serviço **S** em questão, é um tipo de serviço que se localiza na cidade **C**.

A segunda cláusula é direccionada para quando o local **L** se trata de uma cidade, recorrendo diretamente à base de conhecimento, nomeadamente à regra *ponto_descricao*, indicando, explicitamente que o serviço S em questão, é um tipo de serviço que se localiza na cidade **C**.

- **Verificar se determinado elemento pertence a uma lista**

```
pertenceLista(A, B , []):-
        intersec_conj(A,B, [L ]),  !.
```

pertenceLista(A,B, A).

Dado um elemento qualquer **A** e uma lista, o predicado pertence retorna um de dois valores: verdadeiro/falso.

A primeira cláusula deste predicado indica que o elemento **A** pertence à lista se esse elemento se encontra na cabeça dessa lista.

A segunda cláusula indica que, caso o elemento **A** não pertença à cabeça **H** da lista, então pertencerá à lista apenas se pertencer à sua cauda **T**. Para isto, o elemento **A** terá que ser diferente do elemento **H**.

- **Eliminar elementos duplicados de uma lista**

elimina_duplicados: Lista1, Lista2 -> {V,F}

elimina_duplicados([ ], [ ]).

elimina_duplicados[H |T],Lista) :-
pertence(H,T),
elimina_duplicados(T,Lista).

elimina_duplicados([H | T], [H | T1]) :-
nao_pertence(H,T),
elimina_duplicados(T,T1).

O predicado *elimina_duplicados* recebe uma lista e devolve outra sem nenhum elemento repetido.

A primeira cláusula indica que se a lista recebida for vazia, então a lista resultante será também vazia.

A segunda cláusula decompõe a lista recebida em cabeça **H** e cauda **T** e manipula o conhecimento pertence, indicando que se a cabeça da lista recebida pertencer à cauda, isto é, se se encontrar repetida na cauda **T**, então, por recursividade, é utilizado o predicado *elimina_duplicados* que removerá essa repetição da cauda **T**, resultando a **Lista** sem elementos duplicados.

A segunda cláusula assegura os casos em que a cabeça **H** da lista recebida não consta da cauda **T**, manipulando, para isso, o conhecimento *nao_pertence*. Assim, não

pertencendo à cauda **T**, o elemento **H** constará da lista resultante, sendo também a cabeça dessa lista. **A** eliminação de elementos duplicados ocorrerá, então, na cauda **T**, resultando a cauda **T1** da nova lista.
A extensão do predicado *nao_pertence* encontra-se na secção seguinte.

- **Verificar se determinado elemento não pertence a uma lista**

nao_pertence: S,L -> {V,F}

```
nao_pertence( L,[] ).
nao_pertence( L,[H|T] ) :-
        L \= H,
        nao_pertence( L,T ).
```

Dado um elemento qualquer **A** e uma lista, **Lista**, o predicado *pertence* retorna um de dois valores: verdadeiro/falso.

A primeira cláusula deste predicado indica que o elemento **A** não pertence à lista se esta se encontrar vazia.

A segunda cláusula indica que o elemento **A** não pertencerá à lista se for diferente da cabeça **H** da lista, bem como se não pertencer à sua cauda T**,** utilizando-se para tal, recursividade.

- **Eliminar de uma lista os elementos que estão contidos noutra lista**

elim12: L1,L2,L3 -> {V,F}

```
elim12( [],Lista,Lista ) :-!.

elim12( [H|T],LA1,Lista ) :-
        elimina_todos_niveis( H,LA1,LA2 ),
        elim12( T,LA2,Lista ).
```

Dada uma lista 1 e uma lista 2, o predicado *elim12* devolve uma terceira lista, **Lista**, que elimina da **Lista2** todos os elementos que estão contidos na **Lista1.**

A primeira cláusula deste predicado indica que se a primeira lista for vazia, então a **Lista2** não sofrerá nenhuma alteração, sendo devolvida à saída da função nas mesmas condições. Nesta cláusula é implementado o cut que impede que seja efetuado o mecanismo de *backtracking*.

A segunda cláusula indica que, caso a **Lista1** não seja vazia, então a eliminação na **Lista2** dos elementos da **Lista1** resultará numa terceira Lista se e só se: a cabeça **H** da **Lista1** for removida a todos os níveis da **Lista2**, resultando numa lista auxiliar **LA**; e se, por recursividade, os elementos da cauda **T** da **Lista1** forem eliminados dessa lista auxiliar **LA**, sendo o resultado dessa eliminação a **Lista** final que se pretende obter.
A extensão do predicado auxiliar *elimina_todos_niveis* encontra-se de seguida.

- **Eliminar a todos os níveis um elemento de uma lista**

elimina_todos_niveis: A, T, H, Lista, LA1, LA2 -> {V,F}

```
elimina_todos_niveis( A,[],[] ):-!.
elimina_todos_niveis( A,[A],[] ):-!.

elimina_todos_niveis( A,[A|T],Lista ) :-
        elimina_todos_niveis( A,T,Lista ),!.

elimina_todos_niveis( A,[H|T],Lista ) :-
        verifica_lista( H ),
        elimina_todos_niveis( A,H,LA1 ),
        elimina_todos_niveis( A,T,LA2 ),
        concatenar( [LA1],LA2,Lista ),!.

elimina_todos_niveis( A,[H|T],Lista ):-
        elimina_todos_niveis( A,T,LA1 ),
        concatenar( [H],LA1,Lista ).
```

O predicado auxiliar *elimina_todos_niveis* remove um dado elemento **A** de uma **Lista1** resultando numa nova lista, **Lista2**. Esta remoção é efetuada a todos os níveis, isto é, caso a **List1** contenha sublistas, então o elemento **A** será removido dessas sublistas também.

Este predicado é composto por cinco cláusulas. A primeira indica que a eliminação do elemento **A** de uma lista vazia, resultará numa lista vazia. A segunda cláusula salvaguarda a situação em que o elemento que se pretende eliminar da lista, é o único elemento da lista. O resultado será, novamente, uma lista vazia.

A terceira cláusula salvaguarda a situação do elemento A que se pretende remover se encontra à cabeça da **Lista1**. Neste caso o resultado será uma **Lista2** (com cabeça diferente de A), se e só se, por recursividade se eliminar da cauda **T** da **Lista1** todos os elementos **A** que lhe pertençam.

A quarta cláusula adequa-se quando o elemento **A** é diferente da cabeça da lista, **Lista1**. Neste caso a eliminação de **A** da **Lista1** ocorrerá, retornando a **Lista2**, se e só se verificarem três condições. Primeiramente, o elemento cabeça **H** da Lista1 deverá, por si só for uma lista, manipulando-se o conhecimento *verifica_lista*. Por recursividade é efetuada a eliminação de **A** nessa lista H o que resulta numa lista auxiliar **LA1** Analogamente, é efetuada a eliminação de **A** na cauda **T** da **Lista1**, o que resulta numa outra lista auxiliar, **LA2**. Por último é manipulado o conhecimento concatenar que recebe as listas auxiliares **LA1** e **LA2** e devolve uma nova lista **Lista2** que é a junção das duas listas auxiliares.

- **Verificar se uma estrutura é uma lista**

verifica_lista: L -> {V,F}

```
verifica_lista( [] ) :- !.
verifica_lista( [H|T] ) :-
        verifica_lista( T ).
```

O predicado *verifica_lista*, tal como o nome indica, verifica se uma determinada estrutura é uma lista. A primeira cláusula indica que se a estrutura for uma lista vazia, então é uma lista. A segunda cláusula indica que se a estrutura for composta por uma cabeça **H** e uma cauda **T**, então será uma lista se a cauda **T** for, por sua vez, uma lista.

- **Concatenar duas listas**

concatenar: Lista1,Lista2,Resultado -> {V,F}

```
concatenar( [],Lista,Lista ) :- !.
concatenar( [H|T],LA1,[H|T1] ) :-
        concatenar( T,LA1,T1 ).
```

O predicado concatenar efetua a junção de duas listas **Lista1** e **Lista2**, devolvendo uma terceira lista, **Lista3** como resultado. A primeira cláusula indica que o resultado da concatenação uma lista vazia com uma não vazia é a lista não vazia.

A segunda cláusula é direccionada para a concatenação de duas listas não vazias [H | T] e uma **Lista2**. A lista resultante terá a cabeça **H** da **Lista1** e a cauda **T1** se e só se **T1** for o resultado da concatenação da cauda **T** da **Lista1** com a **Lista2**.

- **Apagar o primeiro elemento de uma lista**

apagarPri: L1, L2 -> {V,F}

```
apagarPri([],[]).
apagarPri([H|T],T).
```

O predicado *apagarPri* apaga o primeiro elemento de uma lista, devolvendo a lista sem esse elemento.

A primeira cláusula deste predicado, salvaguarda a situação em que a lista é vazia. Então, nesse caso, o resultado será, também uma lista vazia.
A segunda cláusula assume a lista como não vazia. O resultado será então a cauda **T** da lista já que o primeiro elemento de uma lista é a sua cabeça, **H**.

- **Evitar ciclos ao percorrer um grafo**

```
evita_ciclos( O,D,[NE],_ ) :-
        arco2( O,D,_,NE,_ ).
evita_ciclos( O,D,[H|T],Vis ) :-
        arco2( O,A,_,H,_ ),
        nao_pertence( A,Vis ),
        evita_ciclos( A,D,T,[A|Vis]),
        A \= D.
```

O predicado *evita_ciclos* recebe um nó de origem **O** e um nó de destino **D** e devolve os caminhos possíveis entre esses dois nós sem que haja repetições de arcos, ou seja, sem que ocorram ciclos, guardando, para isso os elementos visitados numa lista **Vis**.

A primeira cláusula assume que o local de origem e destino são os extremos de um arco, logo o caminho entre eles será uma lista de apenas um elemento que é o nome da estrada **NE** que efetua essa ligação. Como o significado da variável **Vis** (nós visitados) não tem sentido neste contexto, então, ela é uma variável anónima.

A segunda cláusula é aplicada no caso dos nós **O** e **D** não se tratarem dos extremos de um arco. Assim, o caminho entre os nós será uma lista constituída pela cabeça **H** e **T** de tal modo que **H** é o caminho (de um só elemento) que liga a origem a um ponto consecutivo **A**, sendo que este ponto **A** não pode pertencer à lista de locais visitados **Vis** e é diferente de **D**. Por recursividade, é determinado o caminho que evita ciclos entre o ponto A e o ponto **D**, sabendo que **A** é agora um nó visitado, sendo a cabeça da lista de elementos visitados da função recursiva. **O** caminho auxiliar encontrado corresponderá à cauda **T** do caminho entre os nós **O** e **D**.

- **Tornar um grafo não orientado**

arco2: A,B,_,TE,_ -> {V,F}

```
arco2( A,B,TE,NE,Vel ) :-
        arco( A,B,TE,NE,Vel );
        arco( B,A,TE,NE,Vel ).
```

O predicado *arco2* torna possível percorrer um arco no sentido de um nó **A** para um nó **B** mas também no sentido inverso, consultando para tal a base de conhecimento correpondente à regra arco.

- **Interseção entre duas listas**

intersec_conj: [Cabeca | Cauda], L, U - > {V,F}

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

```
intersec_conj([Cabeca | Cauda],L, [Cabeca | U ]):-
   pertence(Cabeca, L ), !, intersec\_conj(Cauda, L, U ).

intersec_conj([ _ | Cauda], L , U ):- intersec_conj(Cauda, L, U ).
```

Neste predicado auxiliar, o objetivo é encontrar os elementos comuns entre duas listas e colocá-los numa terceira lista. Para tal a primeira cláusula, diz que se receber uma lista vazia, os elementos comuns entre essa e uma outra qualquer lista é uma lista vazia. A segunda clausula que a interseção entre a lista **[Cabeca | Cauda]** com a lista **L**, é a lista **[Cabeca | U],** se **Cabeca** pertencer a **L** e **U** for a interseção da **Cauda** com L. A última clausula diz que a interseção da lista [_ | Cauda], com a lista L é a interseção da **Cauda** com **L**, mas esta clausula apenas será usada se o primeiro elemento de da primeira lista não pertencer a **L**.

- **Se uma lista tiver elementos comuns a outras é gerada uma lista vazia**

```
pertenceListaRetornaVazia(A, B , []):-
        intersec_conj(A,B, [L]),  !.
pertenceListaRetornaVazia(A,B,A ).
```

Este predicado auxiliar foi utilizado no predicado *nao_passa*, de modo, a que caso a interseção entre duas listas **A** e **B** seja uma segunda lista com elementos, **L** esta seja substituída por uma outra lista vazia. Caso contrário, ou seja não haja interseção o resultado será a própria lista A.

- **Distância entre dois pontos consecutivos**

```
distancia2pontosconsecutivos( A,B,Cam,Dis ) :-
        caminho( A,B,Cam ),
        ponto_descricao(_,_,A,_,XA,YA),
        ponto_descricao(_,_,B,_,XB,YB),
        Dis is sqrt((XB - XA) **2 + (YB - YA) **2),!.
```

Esta função utiliza o teorema de pitágoras. Assim, dadas as coordenadas entre dois pontos, será "construído" um triangulo cujos catetos são respetivamente, a diferença entre as abcissas e as ordenadas dos dois pontos, calculando-se, de seguida, a hipotenusa do mesmo que corresponde à distancia entre os dois pontos.

- **Tempo entre dois pontos consecutivos**

```
tempo2pontosconsecutivos(A,B,Cam,Tempo) :-
        arco2( A,B,_,_,Vel),
        distancia2pontosconsecutivos( A,B,Cam,Dis ),
        Tempo is (Dis/Vel) * 60,!.
```

A função *tempo2pontosconsecutivos* começa por verificar a velocidade média correspondente à estrada (arco) em questão. De seguida, utiliza a função *distancia2pontosconsecutivos*, que por sua vez, calcula a distância entre os dois pontos, divide-a pela velocidade e, finalmente, multiplica o resultado por 60. Deste modo a função retribui o tempo necessário para percorrer determinado arco, em minutos.

- **Valor mínimo de uma lista**

min: A, H, T, B -> {V,F}

```
min([A],A).
min([H|T], H) :- min(T,B), H =< B.
min([H|T], B) :- min(T,B), H > B.
```

A função *min* devolve o menor valor de uma lista. Dada uma lista com apenas um elemento **A**, devolve o próprio elemento **A**. Dada uma lista com uma cabeça (primeiro elemento da lista) e uma cauda (restantes elementos da lista), **[H|T]**, a função devolve **H** se a cabeça da lista for menor que os restantes elementos da cauda e devolve **B**, caso um dos elementos da cauda seja o menor elemento da lista.

Conclusões e Sugestões

Este trabalho prático foi o primeiro contacto desenvolvido pelo grupo com o paradigma da Programação Lógica, tendo sido uma experiencia bastante trabalhosa, mas interessante. Á medida que se explorou a ferramenta PROLOG foi-se percebendo as suas potencialidades e a sua forte aplicação em Sistemas Baseados no Conhecimento. PROLOG é uma boa ferramenta para representação de factos do mundo real, que difícil ou até impossivelmente, seriam recriados em linguagens imperativas. A estruturação e o aspeto funcional desta linguagem revelaram-se fundamentais e de fácil compreensão. Assim sendo, resolveu-se um problema de Localização de Serviços, completamente aplicável ao nosso dia-a-dia, tal como, o caminho para ir de um local para outro, e o seu respectivo custo, tempo e distância.

Como seria de esperar, surgiram alguns obstáculos ao longo do desenvolvimento do código. Deste modo, alguns dos objectivos aos quais o grupo se propôs, não foram totalmente alcançados, no entanto foram maioritariamente contornados. Um dos problemas surgidos deu-se com o predicado distanciaCaminho. Apesar de todos os esforços do grupo terem sido reunidos no sentido de resolver a situação, neste momento, esta função só funciona no sentido Sul-Norte. O grupo não conseguiu perceber qual o motivo deste erro, uma vez que o predicado caminho, manipulado pela função distanciaCaminho, funciona nos dois sentidos e a base de conhecimento arco2 garante que os arcos são bidireccionais. Esta problemática implicou que o predicado custoCaminho sofra do mesmo problema acima referido. O predicado caminho, por sua vez, também tem um problema: quando a pesquisa falha, ou é encontrado um nó terminal da árvore, entra em funcionamento o mecanismo de backtracking. Esse procedimento faz com que o sistema retorne pelo mesmo caminho percorrido com a finalidade de encontrar soluções alternativas. Alguns dos predicados que manipulam esta função, tal como nao_passa_estrada, poderão estar a ser afetados. Por último, os predicados caminho_mais_rapido, caminho_mais_barato e caminho_mais_curto não estavam a funcionar, por isso criaram-se as funções caminho_mais_rapido2, caminho_mais_barato2 e caminho_mais_curto2 com as quais obtivemos os resultados pretendidos, no entanto de um modo pouco correto.

Bibliografia

[1] P. Brna, Prolog Programming. 2001.

[2] J. Lu, Prolog - A Tutorial Introduction. Bucknell University.

[3] A. de Andrade Barbosa, Apositla: Introdução à Programação em Lógica. Universidade Federal de Campina Grande, 2006.

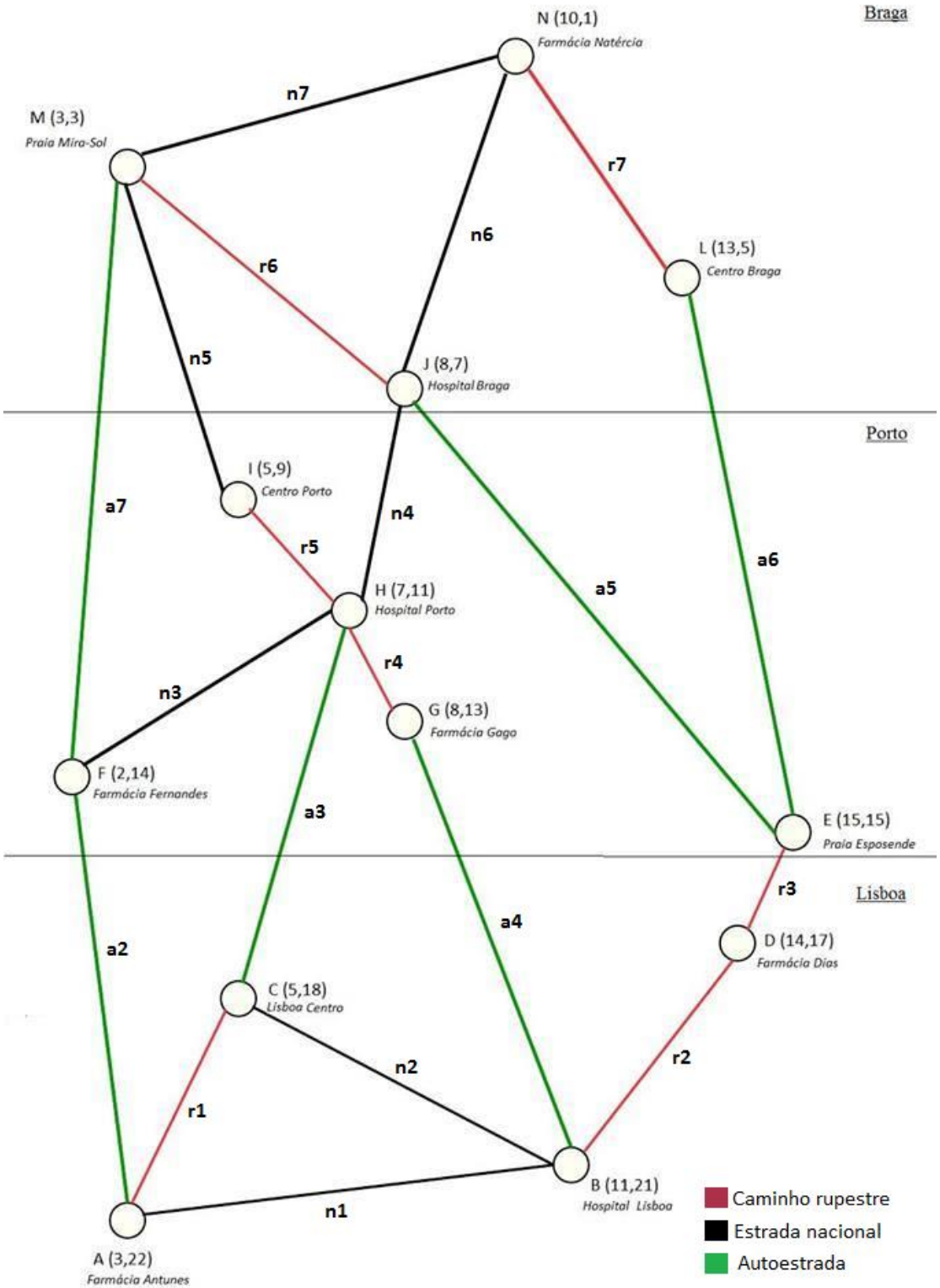
[4] S. Lago, Introdução à Linguagem PROLOG.

[5] K. S. Patrick Blackburn, Johan Bos, Learn Prolog Now! 2001.

[6] I. Bratko, Prolog Programming for Arti_cal Intelligence. Addison Wesley.

[7] E. M. Moreira, Teoria dos Grafos. Universidade Federal de Itajub.

Anexo I


Braga
N (10,1)
Farmácia Natércia
M (3,3)
Praia Mira-Sol
n7
r7
n6
r6
n5
L (13,5)
Centro Braga
J (8,7)
Hospital Braga
Porto
I (5,9)
Centro Porto
a7
n4
r5
a6
a5
H (7,11)
Hospital Porto
r4
n3
G (8,13)
Farmácia Gago
F (2,14)
Farmácia Fernandes
a3
E (15,15)
Praia Esposende
r3
Lisboa
a4
a2
D (14,17)
Farmácia Dias
C (5,18)
Lisboa Centro
n2
r2
r1
B (11,21)
Hospital Lisboa
n1
A (3,22)
Farmácia Antunes
Caminho rupestre
Estrada nacional
Autoestrada

Anexo II

```
%Extensão do predicado caminho_mais_curto

caminho_mais_curto( A,A,[ ],0 ).

    caminho_mais_curto( O,D,Cam,Dis ) :-
       findall( A,distanciaCaminho( O,D,B,A ),LA1 ),
       min( LA1,Dis ),
       distanciaCaminho( O,D,Cam,Dis ).

%Extensão do predicado caminho_mais_barato

    caminho_mais_barato( A,A,[],0 ).

    caminho_mais_barato( O,D,Cam,Cus ) :-
       findall( A,custoCaminho( O,D,B,A ),LA1 ),
       min( LA1,Cus ),
       custoCaminho( O,D,Cam,Cus ).

%Extensão do predicado caminho_mais_rapido

    caminho_mais_rapido( A,A,[],0 ).

    caminho_mais_rapido( O,D,Cam,Tem ) :-
       findall( A,tempoCaminho( O,D,B,A ),LA1 ),
       min( LA1,Tem ),
       tempoCaminho( O,D,Cam,Tem ).
```

Estes predicados tinham como objetivo encontrar o caminho de menor distância,custo ou tempo entre um local (ou serviço) de origem e outro de destino, fornecidos pelo utilizador. A ideia inicial era utilizar o meta-predicado findall sobre as funções distanciaCaminho, custoCaminho, tempocaminhoCaminho que devolveria

uma lista LA1 com todos os comprimentos,custos,tempos possíveis, dos caminhos entre O e D. De seguida, utilizar-se-ia a função min para encontrar o menor valor contido nesta lista. No final, o predicado retribuiria esta distância,custotempo e o caminho que lhe corresponderia, o caminho mais curto,barato,rápido.