Universidade do Minho
Escola de Engenharia - Departamento de Informática

Relatório do Trabalho Prático - 2ª Fase

Mestrado Integrado em Engenharia Informática

Sistemas de Representação de Conhecimento e Raciocínio

A81451 – Alexandre Rzepecki Rodrigues

A82145 - Filipa Correia Parente

A81896 – Nuno Afonso Gonçalves Solha Moreira Valente

A81403 – Pedro Henrique de Passos Ferreira

Braga, abril 2019

Resumo

O seguinte relatório visa elucidar os procedimentos efetuados para construir um sistema de representação de conhecimento e raciocínio, utilizando o pressuposto de mundo aberto para representar conhecimento imperfeito.

Em primeiro lugar, começa-se por apresentar uma pequena fundamentação teórica com os conceitos na qual nos baseamos para a realização das funcionalidades propostas.

Seguidamente, será apresentada o modo de resolução de cada funcionalidade pedida, com recurso a imagens ilustrativas dos predicados utilizados.

Finalmente, na secção Conclusões e Sugestões, será feita uma pequena análise do trabalho realizado, num computo geral, bem como as dificuldades sentidas durante a sua realização.

É de referir também que na secção Anexos se encontram todos os predicados auxiliares utilizados para a realização das funcionalidades bem como a exemplificação dos resultados no interpretador.

Índice

Índice de figuras

Introdução

Com este trabalho pretende-se consolidar conhecimento da unidade curricular de Sistemas de Representação de Conhecimento e Raciocínio, no âmbito da representação de conhecimento imperfeito. Tal como no exercício anterior, o trabalho consiste na utilização da linguagem de programação em lógica PROLOG para representar o conhecimento e raciocínio na área de cuidados de saúde.

Partindo do enunciado começamos com os predicados: utente, prestador e cuidados. O objetivo será introduzir no sistema o potencial de conhecimento negativo e imperfeito através das mecânicas estudadas na disciplina de SRCR.

Preliminares

Para a realização deste trabalho é necessário ter conhecimento da linguagem PROLOG e os seus fundamentos consolidados.

Os fundamentos teóricos estudados para realizar o trabalho anterior foram:

- Facto;
- Regra;
- Pressuposto do mundo fechado;
- Inserção e remoção da base de conhecimento;
- Invariante e sua estrutura;

A realização deste trabalho necessita também de conhecimento sobre pressuposto de mundo aberto, conhecimento negativo, conhecimento imperfeito (seja este incerto, impreciso e interdito) e valores nulos.

Descrição do trabalho e Análise de resultados

Antes de entrar diretamente na resolução das alíneas propostas irá ser feita uma pequena fundamentação teórica que ainda não foi abordada anteriormente e que será usada para a elaboração deste trabalho prático.

Fundamentação teórica

Pressuposto do mundo aberto:

O pressuposto de mundo aberto admite que pode haver mais conhecimento do que está representado na base de conhecimento.

Partindo deste pressuposto qualquer questão pode ter uma de 3 respostas diferentes:

- **Verdadeiro:** Existe conhecimento que comprova a sua veracidade na BC.
- **Falso ou negação forte:** Existe conhecimento explícito para negar a questão.
- **Desconhecido ou negação por falha**: não existe conhecimento, isto é, informação na BC, que permita retirar uma conclusão.

O pressuposto de mundo aberto, de uma forma geral, permite distinguir o que é realmente falso de algo que não está presente na base de conhecimento.

Para representar este tipo de conhecimento, a programação em lógica mostra-se um recurso limitado, uma vez que as respostas às questões colocadas são, apenas, de dois tipos: **verdadeiro** ou **falso**. Neste contexto surge como uma solução, a criação de uma extensão do mesmo que permite a inclusão de respostas do tipo **desconhecido**.

Extensão à Programação em Lógica

Uma extensão de um programa de lógica envolve 2 componentes:

- Representação de conhecimento, completo e incompleto/imperfeito;
- Sistema de inferência que permita a interpretação de conhecimento tanto perfeito, como imperfeito.

Conhecimento Imperfeito:

Numa extensão de um programa de lógica o conhecimento incompleto/imperfeito, é representado sob a forma de valores nulos.

Existem 3 tipos de valores nulos:

- **Incerto**: Desconhecido, de um conjunto indeterminado de hipóteses;
- **Impreciso**: Desconhecido, mas de um conjunto determinado de hipóteses;
- **Interdito**: Desconhecido e não é permitido conhecer.

Sistema de Inferência

Em PROLOG, o sistema de inferência utilizado corresponde a uma extensão de um predicado, representado na seguinte imagem:

```
% Extensao do meta-predicado si: Questao,Resposta -> {V,F}
%                           Resposta = { verdadeiro,falso,desconhecido }

si( Questao,verdadeiro ) :-
    Questao.
si( Questao,falso ) :-
    -Questao.
si( Questao,desconhecido ) :-
    nao( Questao ),
    nao( -Questao ).
```

Figura 1 - sistema de inferência utilizado

Note-se que para representar a negação forte, neste programa, é utilizado o "-" e a negação por falha é representada através do predicado "nao".

Após a retenção destes conceitos, partimos para a inserção de conhecimento útil para a resolução das funcionalidades propostas.

Resolução das funcionalidades propostas

1. Representar conhecimento positivo e negativo

De forma a inserir o conhecimento necessário para a resolução das restantes funcionalidades propostas, decidimos construir os factos que a seguir se apresentam:

Utentes

```
% utente: IdUt, Nome, Idade, Morada ~ { V, F, D }
utente(1,'Ana Martins' ,30,porto).
utente(2,'Beatriz Costa',21,lisboa).
utente(3,'Catarina Furtado',45,lisboa).
utente(4,'Diogo Picarra',27,xptoLocal).
utente(5,xptoNome,34,viana).
```

Figura 2 - Inserção do conhecimento relativo aos utentes

Prestadores

```
% prestador: IdPrest, Nome, Especialidade, Instituição ~ { V, F, D }
prestador(1,'Emanuel Santos',ortopedia,'Hospital Particular de Viana').
prestador(2,'Filipe Ferreira',fisioterapia,'Hospital Lusiadas').
prestador(3,'Guilherme Lima',cardiologia,xptoMorada).
prestador(4,'Helena Gomes',dermatologia,'Hospital Lusiadas').
```

Figura 3 - Inserção do conhecimento relativo aos prestadores

Cuidados

```
% cuidado: Data, IdUt, IdPrest, Descrição, Custo ~ { V, F, D }
cuidado(1,1,1,consulta_ortopedia,20 ).
cuidado(2,2,2,massagem,100).
cuidado(3,3,1,consulta_ortopedia,20).
cuidado(4,4,3,eletrocardiograma,50).
cuidado(4,1,4,massagem,100).
cuidado(5,3,2,xptoCusto).
```

Figura 4 - Inserção do conhecimento relativo aos cuidados

Datas

```
% data: IdData, Ano, Mês, Dia ~ { V, F, D }
data(1,2014,1,1).
data(2,2012,1,5).
data(3,2018,2,5).
data(4,2013,2,6).
data(5,2011,2,7).
```

Figura 5 - Inserção do conhecimento relativo às datas

O conhecimento relativo às datas permitiu ter uma granularidade maior, sendo um conhecimento auxiliar aos cuidados.

Para a **representação do conhecimento negativo**, recorremos à seguinte representação:

```
-utente(Idu,N,Idd,M) :- nao(utente(Idu,N,Idd,M)),
                        nao(excecao(utente(Idu,N,Idd,M))).

-prestador(Idp,Nome,Esp,Inst) :- nao(prestador(Idp,Nome,Esp,Inst)),
                                 nao(excecao(prestador(Idp,Nome,Esp,Inst))).

-cuidado(Data,Idu,Idp,Prio,Desc,Custo) :- nao(cuidado(Data,Idu,Idp,Prio,Desc,Custo)),
                                          nao(excecao(cuidado(Data,Idu,Idp,Prio,Desc,Custo))).
```

Figura 6 - representação de conhecimento negativo para os utentes, prestadores e cuidados

2. Representar casos de conhecimento imperfeito, pela utilização de valores nulos de todos os tipos estudados

Conhecimento imperfeito: Valores nulos do tipo incerto

A inserção de conhecimento inserto é feita através de factos que utilizam valores pré-definidos para este tipo de conhecimento.

```
excecao(utente(Idu,_,Idd,Morada)) :- utente(Idu,xpto1,Idd,Morada).
excecao(utente(Idu,N,_,Morada)) :- utente(Idu,N,xpto2,Morada).
excecao(utente(Idu,N,Idd,_)) :- utente(Idu,N,Idd,xpto3).

excecao(prestador(Idu,_,Idd,Morada)) :- prestador(Idu,nome,Idd,Morada).
excecao(prestador(Idu,Nome,_,Morada)) :- prestador(Idu,Nome,idade,Morada).
excecao(prestador(Idu,Nome,Idd,_)) :- prestador(Idu,Nome,Idd,morada).
```

Figura 7 - representação de valores do tipo incerto para utentes e prestadores

Neste sistema, o utente que tenha o nome xpto1, é possível descobrir o nome do mesmo. A mesma situação acontece para os utentes que tenham a idade a xpto2 ou a Morada a xpto3.

Conhecimento imperfeito: Valores nulos do tipo impreciso

A inserção de conhecimento impreciso ocorre quando utentes ou utilizadores com o mesmo id tem dois ou mais registos com campos diferentes.

```
% a joana tem 20 ou 21 anos
excecao(utente(100,joana,20,braga)).
excecao(utente(100,joana,21,braga)).

% o dr paulo Especializou-se em cardiologia ou pediatria
excecao(prestador(100,paulo,cardiologia,sao_joao)).
excecao(prestador(100,paulo,pediatria,sao_joao)).
```

Figura 8 - inserção de conhecimento impreciso, relativo aos utentes e aos prestadores

Na figura 8, podemos observar que a utente com o nome "joana", possui uma idade que se situa entre os 20 e os 21 anos, não se sabendo ao certo a idade. Também não se sabe precisamente em que especialidade atua o prestador de cuidados com o nome "paulo".

Conhecimento imperfeito: Valores nulos do tipo interdito

```
excecao(utente(Id,_,Idade,Local)):- utente(Id,xptoNome,Idade,Local).
excecao(utente(Id,Nome,Idade,_)):- utente(Id,Nome,Idade,xptoLocal).
excecao(prestador(Id,Nome,Especialidade,_)):- prestador(Id,Nome,Especialidade,xptoMorada).
excecao(cuidado(_,IDut,Idp,Descricao,Custo)) :- cuidado(xptoData,IDut,Idp,Descricao,Custo).
excecao(cuidado(Data,IDut,Idp,Descricao,_))  :- cuidado(Data,IDut,Idp,Descricao,xptoCusto).
nulo(xptoNome).
nulo(xptoLocal).
nulo(xptoMorada).
nulo(xptoData).
nulo(xptoCusto).
```

Figura 9 - inserção de valores nulos do tipo interdito para os utentes, cuidados e prestadores

Na figura 9, verifica-se que o utente com o nome "xptoNome", nunca se poderá saber o nome desse utente, uma vez que se trata de um valor do tipo interdito. Também no caso do utente ter como morada, o valor "xptoLocal", nunca se poderá saber a morada do mesmo, pelos mesmos motivos.

O mesmo acontece no caso do prestador que tiver a instituição com o valor xptoMorada, bem como do cuidado que tiver como data "xptoData", ou como custo o valor "xptoCusto".

Note-se que, uma vez que este tipo de conhecimento é interdito, nunca será possível conhecer o valor do mesmo. Por esse motivo, é necessário acrescentar invariantes que impeçam a inserção de conhecimento desse tipo na BC, como se poderá observar de seguida.

3. Manipular invariantes que designem restrições à inserção e à remoção de conhecimento do sistema

Para garantir esta funcionalidade, recorreu-se aos seguintes invariantes:

- **Invariantes de Inserção**

```
% ------------------------------------------------------------------
% Invariantes de Conhecimento Positivo:  nao permitir a insercao de conhecimento repetido
+utente(Id,_,_,_) :: (solucoes(Id,(utente(Id,_,_,_)),S),
                                        comprimento( S,N ),
                                        N==1).

+prestador(Id,_,_,_) :: (solucoes(Id,(prestador(Id,_,_,_)),S),
                                        comprimento( S,N ),
                                        N==1).

+cuidado(Data, IdU,IdP, Descricao,Custo) :: (solucoes( ( Data, IdU,IdP, Descricao,Custo), cuidado( Data, IdU,IdP, Descricao,Custo), S ),
                                    comprimento( S,N ),
                                            N==1).

+data(Id,Ano,Mes,Dia) :: ( solucoes(Id,data(Id,Ano,Mes,Dia),S),
                        comprimento(S,1)).
```

Figura 10 - invariantes de inserção de conhecimento

Os invariantes representados na figura 10 garantem que não é inserida na BC, conhecimento repetido. Note-se que tanto para o utente como para o prestador para a data o valor que diferencia é apenas o Id. Também foram adicionados invariantes de inserção para a inserção de conhecimento negativo.

```
% Não permitir a inserção de conhecimento contraditório (com neg)
+prestador( Id,Nome,Especialidade,Instituicao) :: ( solucoes( (Id), (-prestador( Id,Nome,Especialidade,Instituicao)), S ),
                        comprimento( S, N ),
                        N == 0 ).

+utente( Id,Nome,Idade,Morada ) :: ( solucoes( (Id), (-utente( Id,Nome,Idade,Morada )), S ),
                        comprimento( S, N ),
                        N == 0 ).

+cuidado(Data,Idu,Idp,Descricao,Custo ) :: ( solucoes( ( Data,Idu,Idp,Descricao,Custo), (-cuidado( Data,Idu,Idp,Descricao,Custo)), S ),
                        comprimento( S, N ),
                        N == 0 ).
```

Figura 11 - invariantes de inserção na BD relativo a conhecimento contraditório

A figura 11, mostra os invariantes que garantem que não é inserido conhecimento contraditório ao existente na BC.

```
% Invariante nao permitir a adicao de conhecimento interdito
+utente(Id,Nome,Idade,Morada) :: (
        solucoes((Id,Nome,Idade,Morada),(utente(5,Nome,Idade,Morada),nao(nulo(Nome))),LR),
        solucoes((Id,Nome,Idade,Morada),(utente(4,Nome,Idade,Morada),nao(nulo(Morada))),L),
        comprimento(L,0),
        comprimento(LR,0)
).

+prestador(Id,Nome,Especialidade,Instituicao) :: (
        solucoes(Instituicao,(prestador(3,Nome,Especialidade,Instituicao),nao(nulo(Instituicao))),LR),
        comprimento(LR,0)
).

+cuidado(Data,Idu,Idp,Descricao,Custo) :: (
        solucoes(Custo,(cuidado(6,Idu,Idp,Descricao,Custo),nao(nulo(Custo))),LH),
        comprimento(LH,0)
).
```

Figura 12 - invariante relativo à inserção de conhecimento interdito

Na figura 12, estão representados os invariantes estruturais que garantem que, nenhum conhecimento que tenha valores do tipo interdito seja inserido na BC, como já explicado no ponto 2.

```
% ------------------------------------------------------
% Invariante Referencial:  os ids sao inteiros

+utente(Id,_,_,_) :: (
    integer(Id)
).

+prestador(Id,_,_,_) :: (
    integer(Id)
).

+data(Id,_,_,_) :: (
    integer(Id)
).

% Invariante Referencial:  a idade é um inteiro
+utente(_,_,Id,_) :: (
    integer(Id)
).

%------------------------------ - - - - - - - - - - - - - - - -
% Invariante Referencial: nao admitir cuidados que nao tenham um prestador e um utente incorporado na BD

+cuidado( _,IdUt,IdSer,_,_ ) :: (
        utente(IdUt,_,_,_),
        prestador(IdP,_,_,_)
).
```

Figura 13 - Invariantes referenciais de inserção

A figura 13, apenas mostra os invariantes impostos pelos requisitos do sistema a ser representado.

- **Invariantes de Remoção**

```
% Invariantes Estruturais: nao remover utentes/prestadores/cuidados que nao estejam na BD
-utente( Id,Nome,Idade,Morada ) :: (
        solucoes((Id,Nome,Idade,Morada),(utente(Id,Nome,Idade,Morada)),S),
        comprimento( S,N ),
        N == 0
).


-prestador( Id,Nome,Especialidade,Instituicao) :: (
        solucoes((Id,Nome,Especialidade,Instituicao),(prestador(Id,Nome,Especialidade,Instituicao)),S),
        comprimento( S,N ),
        N == 0
).


-cuidado(Data,Idu,Idp,Descricao,Custo) :: (
        solucoes((Data,Idu,Idp,Descricao,Custo),(cuidado(Data,Idu,Idp,Descricao,Custo)),S),
        comprimento( S,N ),
        N == 0
).

-data(Id,Ano,Mes,Dia) :: (solucoes((Id,Ano,Mes,Dia),data(Id,Ano,Mes,Dia),L),
                          comprimento(L,0)).
```

Figura 14 - invariantes estruturais de remoção de conhecimento

Os invariantes representados na figura 14 garantem que não é removida da BC, conhecimento que não esteja lá presente.

```
% ------------------------------------------------------
% Invariante Referencial:  não permitir a remoção de utentes/prestadores que têm um cuidado associado
-utente(IdUt,_,_,_) :: (solucoes( (IdUt),(cuidado(_,IdUt,_,_,_)),S ),
                        comprimento( S,N ),
                                    N==0).

-prestador(IdPrest,Nome,Especialidade,Local) :: (solucoes( (IdPrest), cuidado(_,_,IdPrest,_,_),S ),
                                                comprimento( S,N ),
                                                        N==0).
```

Figura 15 - Invariantes referenciais de remoção

A figura 15, apenas mostra os invariantes impostos pelos requisitos do sistema a ser representado.

4. Lidar com a problemática da evolução do conhecimento, criando os procedimentos adequados

Para lidar com a problemática da evolução do conhecimento, foi necessário ter em conta a presença de conhecimento imperfeito na base de conhecimento. Também foi necessário arranjar mecanismos de alteração de valores nulos imprecisos e incertos, no caso de, numa posterior evolução, essa informação fosse atualizada.

Para isso, decidimos criar 3 tipos de evolução, cada uma responsável pela inserção de conhecimentos de tipo diferente.

- Inserção de conhecimento imperfeito do tipo incerto, feita através do predicado "evolução_incertos";

```
evolucao_incertos( Termo ) :-
        tratamento_incertos(Termo).
```

Figura 16 - predicado evolucao_incertos

- Inserção de conhecimento imperfeito do tipo impreciso, feita através do predicado "evolução_imprecisos";

```
evolucao_imprecisos( Termo ) :-
        tratamento_imprecisos(Termo).
```

Figura 17 - predicado evolucao_imprecisos

- Inserção de conhecimento perfeito, feita através do predicado "evolução_perfeitos";

```
%------------------------------ - - - - - - - - - - - - - - - - - -
% Extensao do predicado que permite a evolucao do conhecimento perfeito
evolucao_perfeitos( Termo ) :-
        solucoes(Invariante,+Termo::Invariante,Lista),
        insercao(Termo),
        teste(Lista).
```

Figura 18 - predicado evolucao_perfeitos

É de referir que os predicados de evolução do conhecimento imperfeito recorreram a um predicado auxiliar, que trata cada tipo de maneira diferente:

Tratamento do conhecimento perfeito do tipo incerto:

```
%Extensão do predicado tratamento_incertos que trata de tornar o conhecimento incerto em perfeito
tratamento_incertos(utente(ID,N,I,M)) :- removeIncerto(utente(ID,_,_,_)),
                        solucoes(Inv, +utente(ID,N,I,M) :: Inv, S),
                        insercao(utente(ID,N,I,M)),
                        teste(S).


tratamento_incertos(prestador(ID,N,E,IDi)) :- removeIncerto(prestador(ID,N,E,IDi)),
                        solucoes(Inv, +prestador(ID,N,E,IDi) :: Inv, S),
                        insercao(prestador(ID,N,E,IDi)),
                        teste(S).
```

Figura 19 - predicado tratamento_incertos

O predicado representado na figura 19, remove da BC todos os factos incertos com o que tenham o id do facto dado como argumento. Posteriormente, procede ao processo normal de evolução do conhecimento.

Tratamento do conhecimento perfeito do tipo impreciso:

```
%Extensão do predicado tratamento_imprecisos que trata de tornar o conhecimento impreciso em perfeito

tratamento_imprecisos(utente(ID,N,I,M)) :- solucoes(Inv, +utente(ID,N,I,M) :: Inv, S),
                                    excecao(utente(ID,N,I,M)),
                                    insercao(utente(ID,N,I,M)),
                                    teste(S),
                                    removeImpreciso(utente(ID,_,_,_)).

tratamento_imprecisos(prestador(ID,N,E,IDi)) :- solucoes(Inv, +prestador(ID,N,E,IDi) :: Inv, S),
                        excecao(prestador(ID,N,E,IDi)),
                        insercao(prestador(ID,N,E,IDi)),
                        teste(S),
                        removeImpreciso(prestador(ID,_,_,_)).
```

Figura 20 - predicado tratamento_imprecisos

O predicado representado na figura 20, verifica na base de conhecimento se existe uma "excecao" com esse conhecimento. Posteriormente, procede ao processo normal de evolução do conhecimento. Por fim procede à eliminação das exceções relativas ao conhecimento impreciso da BC.

5. Desenvolver um sistema de inferência capaz de implementar os mecanismos de raciocínio inerentes a estes sistemas

Para este requisito, o decidimos ampliar o sistema de inferência, de modo a poder suportar conjunções, disjunções, e a responder a uma série de questões num só predicado.

Para isso, definiu-se 2 operadores, para representar os operadores de conjunção e disjunção, e a 4 predicados, como se pode ilustrar na figura 21.

```
%Extensão do predicado conjuncao: Valor,Valor,Resposta -> {V,F}

conjuncao(V1,V2,falso) :- V1 == falso; V2 == falso.
conjuncao(verdadeiro,verdadeiro,verdadeiro).
conjuncao(verdadeiro,desconhecido,desconhecido).
conjuncao(desconhecido,V,desconhecido) :- V == desconhecido; V == verdadeiro.

%Extensão do predicado disjuncao: Valor,Valor,Resposta -> {V,F}

disjuncao(V1,V2,verdadeiro) :- V1 == verdadeiro; V2 = verdadeiro.
disjuncao(falso,falso,falso).
disjuncao(falso,desconhecido,desconhecido).
disjuncao(desconhecido,V,desconhecido) :- V == desconhecido; V == falso.

%Extensão do meta-predicado siList: Questions,Answers -> {V,F}

siList([],[]).
siList([Q|T],[A|R]) :- si(Q,A), siList(T,R).

%Extensão do meta-predicado siComp: composicao_questoes,Resposta -> {V,F}

siComp(Q1 && CQ, R) :- si(Q1,R1), siComp(CQ,R2), conjuncao(R1,R2,R).
siComp(Q1 or CQ, R) :- si(Q1,R1), siComp(CQ,R2), disjuncao(R1,R2,R).
siComp(Q1, R) :- si(Q1,R).
```

Figura 21 - predicados utilizados para ampliar o sistema de inferência

É de referir que os predicados conjunção e disjunção correspondem às disjunções e conjunções da tabela de verdade da lógica tradicional. O predicado “siComp”, pega no tipo de operador dado como argumento, bem como as questões a serem tratadas, e aplica as regras de disjunção ou conjunção. Por sua vez o predicado “siList” pega numa lista de questões e devolve as devidas respostas também numa lista.

Requisitos adicionais

Para testar o sistema de conhecimento foram criadas as seguintes funcionalidades adicionais:

- Listar utentes, prestadores e cuidados.
- Identificar cuidados prestados por instituição/custo.
- Identificar os utentes de um prestador/instituição.
- Cuidados prestados por prestador/instituição.
- Calcular o custo total dos cuidados de saúde por utente/data.
- Cuidados de saúde realizados por utente.

Estas funcionalidades encontram-se presentes no ficheiro “requisitos.pl”.

Conclusões e sugestões

Com a realização deste trabalho, foi possível adquirir uma maior prática na utilização da linguagem de programação PROLOG para construir sistemas que tiram partido do pressuposto de mundo aberto para representar não só conhecimento perfeito, mas também conhecimento imperfeito.

Parte do trabalho tira proveito do conhecimento já adquirido no trabalho anterior, onde se tornou possível reaproveitar partes como funções auxiliares.

Finalmente, considerámos que os objetivos do trabalho foram atingidos, resultando num sistema com maior capacidade na representação de conhecimento do que o sistema do trabalho anterior.

Referências

[Analide, 2001] ANALIDE, Cesar, NOVAIS, Paulo, NEVES, José,

"Sugestões para a Elaboração de Relatórios", Relatório Técnico, Departamento de Informática, Universidade do Minho, Portugal, 2001.

[Bratko, 2000] BRATKO, Ivan,

"PROLOG: Programming for Artificial Intelligence", 3rd Edition, Addison-Wesley Longman Publishing Co., Inc., 2000

Anexos

Predicados auxiliares utilizados

```
%------------------------------- - - - - - - - - - - - - - - - - - -
% Extensao do predicado comprimento: ListaSolucoes,Valor -> {V,F}

comprimento([],0).
comprimento([Head],1).
comprimento([Head|Tail],G):-
    comprimento(Tail,T),
    G is T+1.

%------------------------------- - - - - - - - - - - - - - - - - - -
% Extensao do predicado teste: Lista -> {V,F}

teste([]).
teste( [R|LR] ):-
    R,
    teste(LR).

%------------------------------- - - - - - - - - - - - - - - - - - -
% Extensao do predicado insercao : Termo -> {V,F}

insercao( Termo ) :- assert( Termo ).

insercao( Termo ) :- retract( Termo ),!,fail.
```

Figura 22 - predicados auxiliares utilizados para definição dos invariantes e dos predicados "evolucao" e "involucao"

```
%------------------------------- - - - - - - - - - - - - - - - - - -
% Extensao do predicado remocao : Termo -> {V,F}

remocao( Termo ) :- retract( Termo ).

remocao( Termo ) :- assert( Termo ),!,fail.

%------------------------------- - - - - - - - - - - - - - - - - - -
% Extensao do predicado solucoes: Formato,Relacao,ListaSolucoes -> {V,F}

solucoes(F,R,LS) :-
            R,assert(tmp(F)),fail.

solucoes(F,R,LS) :-
            construir([],LS).

%------------------------------- - - - - - - - - - - - - - - - - - -
% Extensao do predicado construir: ListaSolucoes,Solucao -> {V,F}

construir(LS,S):-
        retract(tmp(X)),
        !,
        construir([X|LS],S).

construir(S,S).
```

Figura 23 - predicados auxiliares utilizados para definição dos predicados "evolucao" e "involucao"

```
pertence(H,[H|T]).
pertence(X,[H|T]) :-
    X \= H,
    pertence(X,T).

elimina(_,[],[]).
elimina(X,[X|T],R):- elimina(X,T,R).
elimina(X,[H|T],[H|R]) :- elimina(X,T,R).

remove_duplicates([],[]).
remove_duplicates([H|T], [H|R]) :-
                                elimina(H,T,R1),
                                remove_duplicates(R1,R).
```

Figura 24 - predicados auxiliares "pertence", "elimina" e "remove_duplicates"

```
concat([],L,L).
concat([X|T1],L2,[X|T2]):-concat(T1,L2,T2).
```

Figura 25 - predicados auxiliares "concat" e "servicos"

```
utentes([],[]).
utentes([H],L):- listUtentes(id,H,L).
utentes([H|T],LS) :- listUtentes(id,H,X),
                     utentes(T,L1),
                     concat(X,L1,LS).
```

Figura 26 - predicado auxiliar "utentes"

```
% remove conhecimento impreciso da base de conhecimento
removeImpreciso(utente(Id,Nome,Idd,M)) :- solucoes(utente(Id,Nome,Idd,M), (excecao(utente(Id,Nome,Idd,M)),
                                          Nome \= xpto1, Idd \= xpto2, M \= xpto3), S),
                                         remove_imprecisos(S).

removeImpreciso(prestador(Id,N,E,Idi)) :- solucoes(prestador(Id,N,E,Idi), (excecao(prestador(Id,N,E,Idi)),
                                          N \= xpto4, E \= xpto5, Idi \= xpto6), S),
                                          remove_imprecisos(S).
```

Figura 27 - predicado auxiliar removeImpreciso

```
%Extensão do predicado remove_imprecisos que, dada uma lista de termos, remove a respetiva exceção de conhecimento impreciso

remove_imprecisos([]).
remove_imprecisos([Termo]) :- retract(excecao(Termo)).
remove_imprecisos([Termo|T]) :- retract(excecao(Termo)), remove_imprecisos(T).


% extesão do predicado que remove todos os elementos de uma lista

removeAllFromLista([]).
removeAllFromLista([H]) :- retract(H).

% remove todo o conhecimento incerto associado a um fato
removeIncerto(utente(ID,_,_,_)) :- solucoes(utente(ID,N,I,M,D), (utente(ID,N,I,M), (N == xpto1; I == xpto2; M == xpto3)), S),
                                   removeAllFromLista(S).

removeIncerto(prestador(ID,_,_,_)) :- solucoes(prestador(ID,N,E,IDi), (prestador(ID,N,E,IDi), (N == xpto4; E == xpto5; IDi == xpto6)), S),
                                      removeAllFromLista(S).
```

Figura 28 - predicados auxiliares remove_imprecisos, removeAllFromLista, e removeIncerto