Universidade do Minho
Escola de Engenharia

TRABALHO PRÁTICO EXERCÍCIO 2

Mestrado Integrado em Engenharia Informática

Sistemas de Representação de Conhecimento e Raciocínio

(2º semestre, 3º ano)

A78322 André Filipe Ferreira de Mira Vieira

A77048 Eduardo Gil Ribeiro Rocha

A78764 Ricardo André Araújo Neves

Data:

18 de março de 2018

Braga,

20 de abril de 2018

Índice

Resumo

Este relatório serve como complemento ao segundo exercício do trabalho prático da Unidade Curricular de **Sistemas de Representação de Conhecimento e Raciocínio**.

O universo de dados tratado neste trabalho prático é composto por 3 entidades referentes à área de cuidados de saúde.

Aqui iremos expressar a nossa linha de pensamento e como implementamos cada um dos predicados, seguida de uma breve conclusão crítica sobre o trabalho realizado pelo grupo.

Introdução

Este **segundo exercício** do trabalho prático tem em vista a implementação, na linguagem PROLOG, de um sistema de representação de conhecimento e raciocínio, onde a informação não é completa, contrariamente ao que pudemos observar na primeira parte.

Assim, além dos valores **Verdadeiro** e **Falso**, é agora introduzido o **Desconhecido**.

Para além destes valores de verdade, o **Conhecimento** deixa também de ser apenas **Positivo**. Ou seja, ocorre agora outros tipos de Conhecimento, como o **Negativo** e o **Imperfeito**, que iremos abordar mais tarde.

Deste modo, ao longo deste relatório, iremos refletir sobre todos os predicados implementados, de modo a trabalhar com toda esta informação.

Preliminares

Como já foi referido, foi utilizada a **linguagem de programação PROLOG**, que permite inserir, remover e manipular informação.

Com toda a primeira parte do projeto pronta, o grupo decidiu guiar-se sobre esta, o que facilitou um pouco este segundo exercício.

Representação das Entidades

Como já foi referido na primeira parte deste projeto, este Universo é composto por 3 entidades, o **Utente**, o **Prestador** e os **Cuidados** de Saúde.

Para representar o **Utente**, foi criada uma entidade com 4 atributos: **IdUt** (ID único que caracteriza um e um só utente), **Nome**, **Idade** e **Morada**.

Na representação do **Prestador**, foi criada outra entidade com o mesmo número de atributos: **IdPrest** (ID único que caracteriza um e um só prestador de cuidados de saúde), **Nome**, **Especialidade** e **Instituição** à qual está designado.

Por último, foi necessário criar a entidade **Cuidado**, desta vez, com 5 atributos: **Data** em que foi prestado o serviço, **IdUt** (ID do utente que foi atendido), **IdPrest** (ID do prestador que prestou o serviço), **Descrição** e **Custo**.

```
% utente(IdUt, Nome, Idade, Morada).
% prestador(IdPrest, Nome, Especialidade, Instituicao).
% cuidado(Data, IdUt, IdPrest, Descricao, Custo).
```

Deste modo, para nos ajudar no desenvolvimento das funcionalidades a seguir definidas, criamos uma **situação de teste**, onde demos exemplos a cada um dos elementos acima. Assim, à medida que desenvolvíamos cada uma das seguintes "querys", fomos testando, de modo a corrigir possíveis erros de implementação.

```
utente(1, ricardo, 20, porto).
utente(2, andre, 20, porto).
utente(3, gil, 20, braga).
utente(4, joao, 14, braga).
utente(5, maria, 34, porto).
prestador(1, carlos, medicinaGeral, porto).
prestador(2, manuel, fisioterapia, braga).
prestador(3, david, medicinaGeral, porto).
prestador(4, paula, fisioterapia, porto).
prestador(5, luis, medicinaGeral, braga).
cuidado(16032018, 1, 1, consulta, 20).
cuidado(17032018, 1, 2, fisioterapia, 20).
cuidado(16032018, 5, 4, fisioterapia, 19).
cuidado(16032018, 2, 3, consulta, 15).
cuidado(18032018, 3, 4, fisioterapia, 50).
cuidado(20032018, 1, 5, consulta, 23).
```

Como podemos observar na imagem acima, podemos retirar uma série de factos:

- O Utente com ID número 1 chama-se Ricardo, tem 20 anos e é natural do Porto.
- O Prestador com ID número 5 é especialista em Medicina Geral, trabalha na Instituição de Braga, e chama-se Luís.
- No dia 18-03-2018, o prestador com ID 5 (Luís) prestou um cuidado de saúde do utente com ID 3 (Gil), com a descrição X e um custo de 50 euros.

Para além destes exemplos, acrescentamos **conhecimento negativo** ao nosso sistema.

```
-utente(6, hugo, 67, braga).
-utente(7, carla, 29, braga).
-prestador(6, bernardo, porto).
```

Esta informação quer dizer, com certeza, que não existe na base de conhecimento, por exemplo, um utente chamado Hugo, com 67 anos e que vive em Braga.

Conjunções

O grupo utilizou as seguintes conjunções de valores de verdade, que irão ser necessárias num predicado, mais tarde.

```
conjuncoes(verdadeiro, verdadeiro, verdadeiro).
conjuncoes(verdadeiro, falso, falso).
conjuncoes(falso, verdadeiro, falso).
conjuncoes(falso, falso, falso).
conjuncoes(verdadeiro, desconhecido, desconhecido).
conjuncoes(desconhecido, verdadeiro, desconhecido).
conjuncoes(falso, desconhecido, falso).
conjuncoes(desconhecido, falso, falso).
conjuncoes(desconhecido, desconhecido, desconhecido).
```

Em cima, podemos observar que o valor "Verdadeiro" é o valor mais "fraco", uma vez que, na presença de outro valor, o primeiro é anulado.

Pelo contrário, o valor de falsidade pode ser o considerado o valor de verdade mais "forte", uma vez que, qualquer que seja o outro valor, a conjunção destes é sempre falsa.

Invariantes

Nesta segunda parte do trabalho prático, o grupo decidiu alterar e acrescentar uma série de invariantes, de modo a tornar a inserção e remoção de informação mais eficaz.

Estes são os responsáveis por permitir ou impedir a **evolução/involução** do conhecimento.

De seguida, iremos apresentar todos os invariantes implementados pelo grupo de trabalho. Em alguns destes invariantes, iremos poder observar um pequeno exemplo onde o invariante é usado.

1. Invariante que não permite inserir um utente repetido:

```
+utente(IdUt,Nome,Idade,Morada) :: (findall((IdUt, Nome, Idade, Morada), utente(IdUt, Nome, Idade, Morada),S),
                                    comprimento(S,N),
                                    N == 1).
```

O utente introduzido já se encontra na base de conhecimento, logo não é possível introduzi-lo de novo.

```
| ?- evolucao(utente(1, ricardo, 20, porto)).
no
```

2. Não permite a inserção de um utente com ID repetido, uma vez que este ID é necessário para a marcação de cuidados de saúde:

```
+utente(IdUt, Nome, Idade, Morada) :: (findall((IdUt), utente(IdUt, N, I, M),S),
                                       comprimento(S,N),
                                       N == 1).
```

Já existe um utente com o ID igual 3, logo terá de ser atribuído outro IdUt.

```
| ?- evolucao(utente(3, jose, 54, braga)).
no
```

3. Não permite remover um cliente que já foi sujeito a um cuidado de saúde:

```
-utente(IdUt, Nome, Idade, Morada) :: ( findall((IdUt), cuidado(Data, IdUt, IdPrest, Descricao, Custo), S),
                                      comprimento(S, N),
                                      N == 0).
```

O utente André já foi sujeito a um cuidado de saúde, logo não poderá ser removido.

```
| ?- involucao(utente(2, andre, 20, porto)).
no
```

4. Não permite conhecimento positivo contraditório para o predicado utente:

```
+(-utente(IdUt, Nome, Idade, Morada)) :: (findall((IdUt), utente(IdUt, Nome, Idade, Morada), S),
                                         comprimento(S,N),
                                         N == 0).
```

5. Permite a inserção de um utente, mesmo não se souber ao certo a sua idade:

```
+utente(IdUt, Nome, Idade, Morada) :: (findall( (excecao(utente(IdUt, Nome, I, Morada))),
                                      excecao(utente(IdUt, Nome, I, Morada)), S),
                                      comprimento(S,N),
                                      N == 0).
```

6. Permite a inserção de um utente mesmo não sabendo ao certo a sua morada:

```
+utente(IdUt, Nome, Idade, Morada) :: (findall( (excecao(utente(IdUt, Nome, Idade, M))),
                                      excecao(utente(IdUt, Nome, Idade, M)), S),
                                      comprimento(S, N),
                                      N == 0).
```

7. Impede a inserção de um utente com conhecimento interdito relativamente ao seu ID, ou seja, não se conhece o seu ID único:

```
+utente(IdUt, Nome, Idade, Morada) :: (findall((IdUt, Nome, Idade, Morada), (utente(idNull, Nome, Idade, Morada), null(idNull)), S),
                                      comprimento(S, N),
                                      N == 0).
```

```
+(-utente(IdUt, Nome, Idade, Morada)) :: (findall((IdUt, Nome, Idade, Morada), (utente(idNull, Nome, Idade, Morada), null(idNull)), S),
                                         comprimento(S, N),
                                         N == 0).
```

8. Conhecimento negativo para o predicado utente:

```
-utente(IdUt, Nome, Idade, Morada) :- nao(utente(IdUt, Nome, Idade, Morada)),
                                      nao(excecao(utente(IdUt, Nome, Idade, Morada))).
```

9. Não permite a inserção de exceções repetidas:

```
+(excecao(utente(IdUt, Nome, Idade, Morada))) :: (findall(excecao(utente(IdUt, Nome, Idade, Morada)), excecao(utente(IdUt, Nome, Idade, Morada)), S),
                                                  comprimento(S, N),
                                                  N == 1).
```

10. Não permite a inserção de um prestador repetido:

```
+prestador(IdPrest,Nome,Especialidade,Instituicao) :: (findall((IdPrest, Nome, Especialidade, Instituicao),(prestador(IdPrest, Nome, Especialidade, Instituicao)),S),
                                                       comprimento(S,N),
                                                       N == 1).
```

O prestador introduzido já se encontra na base de conhecimento, logo não é possível introduzi-lo de novo.

```
| ?- evolucao(prestador(2, manuel, fisioterapia, braga)).
no
```

11. Não permite a inserção de um prestador com ID repetido, semelhante ao que acontece com os Utentes:

```
+prestador(IdPrest, Nome, Especialidade, Instituicao) :: (findall( (IdPrest),prestador(IdPrest, N, E, I),S),
                                                          comprimento(S,N),
                                                          N == 1).
```

Já existe um prestador com ID igual a 2, logo é necessário atribuir um novo IdPrest.

```
| ?- evolucao(prestador(2, francisco, pediatria, porto)).
no
```

12. Permite a inserção de um prestador mesmo não sabendo a sua Especialidade:

```
+prestador(IdPrest, Nome, Especialidade, Instituicao) :: (findall( (excecao(prestador(IdPrest, Nome, E, Instituicao))),
                                                          excecao(prestador(IdPrest, Nome, E, Instituicao)), S),
                                                          comprimento(S, N),
                                                          N == 0).
```

13. Impede a inserção de um prestador com conhecimento interdito relativamente ao seu ID:

```
+prestador(IdPrest, Nome, Especialidade, Instituicao) :: (findall((IdPrest, Nome, Especialidade, Instituicao), (prestador(idNull, Nome, Especialidade, Instituicao), null(idNull)), S),
                                                          comprimento(S, N),
                                                          N == 0).
```

```
+(-prestador(IdPrest, Nome, Especialidade, Instituicao)) :: (findall((IdPrest, Nome, Especialidade, Instituicao), (prestador(idNull, Nome, Especialidade, Instituicao), null(idNull)), S),
                                                             comprimento(S, N),
                                                             N == 0).
```

14. Não permite remover um prestador que já prestou um cuidado de saúde:

```
-prestador(IdPrest, Nome, Especialidade, Instituicao) :: ( findall((IdPrest), cuidado(Data, IdUt, IdPrest, Descricao, Custo), S),
                                                          comprimento(S, N),
                                                          N == 0).
```

A prestadora Paula já prestou um cuidado de saúde, logo não pode ser removida da base de conhecimento.

```
| ?- involucao(prestador(4, paula, fisioterapia, porto)).
no
```

15. Não permite conhecimento positivo contraditório para o predicado prestador:

```
+(-prestador(IdPrest, Nome, Especialidade, Instituicao)) :: (findall((IdPrest), prestador(IdPrest, Nome, Especialidade, Instituicao), S),
                                                            comprimento(S,N),
                                                            N == 0).
```

16. Conhecimento negativo para o predicado prestador:

```
-prestador(IdPrest, Nome, Especialidade, Instituicao) :- nao(prestador(IdPrest, Nome, Especialidade, Instituicao)),
                                                         nao(excecao(prestador(IdPrest, Nome, Especialidade, Instituicao))).
```

17. Só é possível inserir um Cuidado se o Utente e o Prestador existirem, ou seja, se existirem os seus Ids (IdUt e IdPrest):

```
+cuidado(Data, IdUt, IdPrest, Descricao, Custo) :: (findall(IdUt), utente(IdUt, Nome, Idade, Morada), SU),
                                                   comprimento(SU, NU),
                                                   NU == 1,
                                                   (findall(IdPrest), prestador(IdPrest, Nome, Especialidade, Instituicao), SP),
                                                   comprimento(SP, NP),
                                                   NP == 1.
```

Não existe nenhum utente com IdUt igual a 99, logo não é possível inserir o cuidado pretendido.

```
| ?- evolucao(cuidado(12062017, 99, 2, fisioterapia, 50)).
no
```

18. Permite a inserção de um Cuidado mesmo se não houver conhecimento relativamente à Descrição:

```
+cuidado(Data, IdUt, IdPrest, Descricao, Custo) :: ( findall(( excecao(cuidado(Data, IdUt, IdPrest, D, Custo))),
                                                     excecao(cuidado(Data, IdUt, IdPrest, D, Custo)), S),
                                                     comprimento(S, N),
                                                     N == 0).
```

19. Conhecimento negativo para o predicado Cuidado:

```
-cuidado(Data, IdUt, IdPrest, Descricao, Custo) :- nao(cuidado(Data, IdUt, IdPrest, Descricao, Custo)),
                                                   nao(excecao(cuidado(Data, IdUt, IdPrest, Descricao, Custo))).
```

20. Não permite a inserção de exceções repetidas para o predicado Cuidado:

```
+(excecao(cuidado(Data, IdUt, IdPrest, Descricao, Custo))) :: (findall(excecao(cuidado(Data, IdUt, IdPrest, Descricao, Custo)), excecao(cuidado(Data, IdUt, IdPrest, Descricao, Custo)), S),
                                                              comprimento(S, N),
                                                              N == 0).
```

Como podemos observar, em alguns dos invariantes acima, constatamos a presença do predicado "comprimento", que calcula o número de elementos existentes numa lista.

```
comprimento([], 0).
comprimento([X|L], C) :- comprimento(L, N), C is 1+N.
```

Registo de Utentes, Prestadores e Cuidados

Neste sistema, é possível registar um novo Utente, Prestador ou Cuidado de Saúde no sistema. Para isto, foi criado o predicado "evolução", que testa os Invariantes criados, insere o Termo no sistema, com o método "assert", e corre o teste final.

```
evolucao( Termo ) :- findall(I, +Termo::I, Li),
                     insercao(Termo),
                     teste(Li).
```

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

```
teste([]).
teste([H|S]) :- H, teste(S).
```

Exemplo:

```
| ?- evolucao(utente(6, Xavier, 8, aveiro)).
yes
```

Foi inserido um novo Utente, cujo ID único é 6, chamado Xavier, com 8 anos de idade e natural da cidade de Aveiro.

Neste segundo exercício, o grupo decidiu implementar também uma evolução que aceitasse uma lista de termos e inserisse essa informação.

```
evolucaoLista([]).
evolucaoLista([H|T]) :- evolucao(H),
                        evolucaoLista(T).
```

Remoção de Utentes, Prestadores e Cuidados

Para ser possível remover registos de utentes, prestadores de serviços ou cuidados de saúde, foi também implementada essa funcionalidade. Assim, com o predicado "involução", remove o Termo do sistema, com o método "retract", correndo o teste final.

```
involucao( Termo ) :- Termo,
                      findall(I, -Termo::I, Li),
                      remocao(Termo),
                      teste(Li).
```

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

Exemplo:

```
| ?- involucao(utente(6, Xavier, 8, aveiro)).
yes
```

Foi removido o Utente, cujo ID único é 6, chamado Xavier, com 8 anos de idade e natural da cidade de Aveiro.

Aqui, também implementamos uma involução que apagasse todos os termos incluídos numa lista.

```
involucaoLista([]).
involucaoLista([H|T]) :- involucao(H),
                         involucaoLista(T).
```

Demo

Neste segundo exercício, o grupo implementou um sistema de inferência capaz de dar resposta a ***querys*** do utilizador, utilizando as conjunções que já foram referidas em cima.

```
demoListaAux( Q, [], Q).
demoListaAux( verdadeiro, [H|T], R ) :- demoListaAux(H, T, R).
demoListaAux( falso, [H|T], falso).
demoListaAux( desconhecido, [H|T], R) :- demoListaAux(H, T, R1), conjuncoes(desconhecido,R1,R).

demoLista([], verdadeiro).
demoLista([H], R) :- demo(H, R).
demoLista( [H|T], R) :- demo(H, R1), demoLista(T, R2), conjuncoes(R1, R2, R).
```

```
demo( Questao,verdadeiro ) :- Questao.

demo( Questao,falso ) :- -Questao.

demo( Questao,desconhecido ) :- nao( Questao ),
                                nao( -Questao ).
```

```
nao( Questao ) :- Questao, !, fail.

nao( Questao ).
```

Assim, dando uma lista de questões, o sistema é capaz de fornecer a resposta às mesmas, como podemos observar nos exemplos.

É verdadeiro que existe o utente com ID igual a 3 chamado Gil.

```
| ?- demo(utente(3, gil, 20, braga), R).
R = verdadeiro ?
yes
```

É verdadeiro que existem todos estes utentes na base de conhecimento.

```
| ?- demoLista([utente(1, ricardo, 20, porto), utente(2, andre, 20, porto),
 utente(3, gil, 20, braga)], R).
R = verdadeiro ?
yes
```

Existe o prestador Manuel, mas é falso que exista o prestador Guilherme, com ID igual a 6.

```
| ?- demoLista([prestador(2, manuel, fisioterapia, braga),
 prestador(6, guilherme, secretaria, porto)], R).
R = falso ?
yes
```

Conhecimento Imperfeito Incerto

O primeiro tipo de Conhecimento Imperfeito é o **Incerto**.

Este tipo de conhecimento representa informação a que **não se tem acesso,** ou seja**,** não se sabe nada acerca da mesma. Por exemplo:

Não se sabe a idade do utente Xavier.

```
utente(8, xavier, idadeInc, braga).
excecao( utente(IdUt, Nome, Idade, Morada)) :- utente(IdUt, Nome, idadeInc, Morada).
```

Não se sabe a morada da utente Margarida.

```
utente(9, margarida, 43, moradaInc).
excecao( utente(IdUt, Nome, Idade, Morada)) :- utente(IdUt, Nome, Idade, moradaInc).
```

Não se conhece a especialidade do prestador António.

```
prestador(7, antonio, especialidadeInc, braga).
excecao( prestador(IdPrest, Nome, Especialidade, Instituicao)) :- prestador(IdPrest, Nome, especialidadeInc, Instituicao).
```

Conhecimento Imperfeito Impreciso

O segundo tipo de Conhecimento Imperfeito é o Impreciso.

Aqui, temos acesso a uma série ou **intervalo de alternativas** para a mesma informação, no entanto, apenas uma está correta. Por exemplo:

O utente Tiago tem 4 ou 5 anos.

```
excecao(utente( 10, tiago, 4, porto)).
excecao(utente( 10, tiago, 5, porto)).
```

A utente Cristina mora no Porto ou em Braga.

```
excecao(utente( 11, cristina, 37, porto)).
excecao(utente( 11, cristina, 37, braga)).
```

Não se sabe se o cuidado prestado foi consulta ou fisioterapia.

```
excecao(cuidado( 21032018, 2, 3, consulta, 22)).
excecao(cuidado( 21032018, 2, 3, fisioterapia, 22)).
```

Conhecimento Imperfeito Interdito

O terceiro e último de Conhecimento Imperfeito é o Interdito.

Este é o tipo de conhecimento mais "agressivo", onde **nunca se conseguirá obter a informação** correta sobre um dado predicado, permanecendo para sempre uma incógnita. Por exemplo:

Nunca se saberá a idade do utente Renato.

```
utente( 12, renato, idadeInt, porto).
excecao( utente( IdUt, Nome, Idade, Morada)) :- utente(IdUt, Nome, idadeInt, Morada).
null(idadeInt).
```

Nunca se saberá a especialidade do prestador Mário.

```
prestador( 8, mario, especialidadeInt, braga).
excecao(prestador(IdPrest, Nome, Especialidade, Instituicao)) :- prestador(IdPrest, Nome, especialidadeInt, Instituicao).
null(especialidadeInt).
```

Neste tipo de Conhecimento Imperfeito, utilizamos o predicado null, fazendo com que, com o invariante respetivo, não seja possível alterar esta informação.

Conclusões e Sugestões

Em conclusão, este trabalho permitiu aprofundar os nossos conhecimentos em relação à linguagem PROLOG, aos vários tipos de Conhecimento e invariantes.

Durante a realização deste trabalho prático, as nossas maiores dificuldades surgiram na implementação dos invariantes mais exigentes, onde tivemos que recorrer a várias pesquisas, acabando por conseguir completa-los.

De resto, concordamos que terminamos este trabalho com sucesso, estando preparados para as próximas fases.

Referências

[Leite, 1978] LEITE, J. A. Américo,

"Metodologia de Elaboração de Teses",

McGraw-Hill do Brasil, São Paulo, 1978.

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

"Sugestões para a Elaboração de Relatórios",

Relatório Técnico, Departamento de Informática, Universidade do Minho, Portugal, 2001.