Sistema de Representação de Conhecimento e Raciocíneo

Exercício II

Relatorio de Desenvolvimento

André Gonçalves (A75625)

Bruno Cancelinha (A75428)

José Silva (A74576)

Marcelo Miranda (A74817)

9 de Abril de 2017

Universidade do Minho

Conteúdo

Resumo

Depois de completo e exercício 1, onde foi descrito um sistema capaz de representar conhecimento perfeito, vamos agora desenvolver um sistema semelhante embora com a capacidade extra de representar conhecimento imperfeito.

Capítulo 1

Introdução

Completada a tarefa proposta na unidade curricular de Sistemas de Representação de Conhecimento e Reconhecimento, foi desenvolvido um sistema capaz de representar o conhecimento de uma instituição médica.

Este sistema, no entanto, limitava-se a representar conhecimento concreto, não havendo possibilidade de descrever conhecimento imperfeito. É neste ponto que nos vamos focar neste segundo exercício, o qual, depois de completo, deverá ser capaz de representar conhecimento incerto, impreciso e interdito relativo ao domínio dos cuidados de saúde.

Capítulo 2

Análise Teórica

No exercício anterior, terminamos com um sistema capaz de representar conhecimento perfeito não sendo capaz de representar conhecimento imperfeito. Tal sucede-se porque o sistema era baseava no ***pressuposto do mundo fechado***, onde todo o conhecimento que se encontra na base de conhecimento é verdadeiro, e tudo o que não se encontra é, necessariamente, falso.

Na vida real seria irresponsável considerar que tudo o que não consta no nosso conhecimento seria imediatamente falso, e que tudo o que consta é necessariamente verdade. Se não se sabe algo será melhor considerar simplesmente que se desconhece a verdade. Entra aqui a idéia do ***mundo aberto***, onde assumimos que nem sempre podemos saber toda a informação. Assim vamos dar ao nosso sistema o privilégio do desconhecimento.

2.1 Mundo Aberto

Partindo do pressuposto do mundo aberto, vamos fazer uma forte distinção entre a *negação por falha*, a qual representa algo que não se encontra na base de conhecimento, e *negação forte* que é uma prova de que uma dada proposição é falsa.

Tal como foi referido acima, segundo o pressuposto do mundo aberto, para além de podermos ter respostas verdadeiras e falsas podemos ainda obter como resposta o desconhecido. Assim, usando a *negação por falha* e a *negação forte*, podemos definir uma resposta à questão q como:

Verdadeira $\exists x : q(x)$

Falsa $\exists x : \neg q(x)$

Desconhecido $\neg\exists x : q(x) \vee \neg q(x)$

2.1.1 Conhecimento Imperfeito

Uma proposição $q(x)$ pode ser dita ***conhecimento perfeito*** se a sua resposta é ou Verdadeira ou Falsa, ou ***conhecimento imperfeito*** se a resposta for desconhecido. Pela sua própria natureza, temos vários tipos de ***conhecimento imperfeito***.

Incerto Representa conhecimento que é completamente desconhecido como, por exemplo, o número de raças extra-terrestres no universo.

Impreciso Um conhecimento é impreciso quando existe uma noção do que é falso, mas não do que é verdadeiro. Um exemplo deste conhecimento seria o número de espécies diferentes que existem no planeta terra, podemos dizer com toda a certeza que não são 10 nem 20 nem 100, mas não sabemos o número exato.

Interdito Um conhecimento interdito é conhecimento que será para sempre desconhecido.

Apesar de estarmos a lidar com o pressuposto do mundo aberto, por vezes é necessário definir que uma negação de uma prova $p(x)$ é verdade se não houver nenhuma prova $p(x)$, ou seja, $\neg p(x) \leftarrow \neg \exists x : p(x))$. Porém, já dizia o ditado "não há regra sem exceção", e podemos necessitar de definir um predicado $p(x)$ que é falso para a grande maioria dos casos mas havendo exceções, portanto não podemos deixar a nossa definição de $\neg p(x)$ anterior sem a expandir, terminamos com a seguinte definição:

$$\neg p(x) \leftarrow \neg \exists x : p(x) \vee excecao(p(x))$$

Que se pode ler: $p(x)$ é falso se não houver prova verdadeira de $p(x)$ nem houver nenhuma exceção de $p(x)$. Vamos encontrar muitas destas regras ao longo do projeto que estamos a apresentar.

2.2 Lógica Ternária

Com a introdução do pressuposto do mundo fechado foi necessário adicionar ao ***verdadeiro*** e ***falso*** da lógica clássica o ***desconhecido***, nascendo assim a necessidade de definir a conjução, disjunção e negação destes três termos.

$\wedge$	**V**	**D**	**F**
V	V	D	F
D	D	D	F
F	F	F	F

Tabela 2.1: Tabela de conjunção da lógica ternária.

$\neg$	
V	F
D	D
F	V

Tabela 2.3: Tabela de negação da lógica ternária.

$\vee$	**V**	**D**	**F**
V	V	V	V
D	V	D	D
F	V	D	F

Tabela 2.2: Tabela de disjunção da lógica ternária

2.3 Evolução do Conhecimento

Dada a necessidade de representar conhecimento imperfeito, é necessário adaptar o processo de evolução da base de conhecimento desenvolvido no primeiro exercício deste projeto. Anteriormente este processo consistia na definição de um conjunto de invariantes, constituindo assim as regras que garantiam a consistência da base de conhecimento, bem como na declaração dos predicados ***evolução*** e ***involução***. Estes dois predicados eram responsáveis por inserir e remover conhecimento, assegurando que a realização destas operações não violam os invariantes definidos.

Atualmente, devido ao ***pressuposto do mundo aberto***, há ainda a necessidade de inserir conhecimento imperfeito recorrendo à temática dos valores nulos estudados: ***incerto***, ***impreciso*** e ***interdito***. Estes valores nulos podem ser facilmente representados através de exceções, como veremos mais tarde. No caso do valor nulo do tipo interdito é ainda necessário indicar os invariantes necessários para garantir que o conhecimento não pode evoluir.

Dada a natureza deste tipo de conhecimento, ao longo do período de vida do sistema, este pode ainda evoluir reduzindo assim a imprecisão do desconhecimento ou até mesmo passar a conhecimento perfeito. Mesmo nesta fase, é importante ainda garantir que a evolução do desconhecimento não viola os invariantes.

Capítulo 3

Implementação da Solução

Tal como já foi referido na introdução, o principal objetivo deste exercício é a representação de conhecimento sobre o pressuposto do mundo aberto, afastando-nos da lógica clássica e entrando na lógica extendida, onde para além das respostas ***verdadeiro*** e ***falso***, temos também a resposta ***desconhecido***. Vamos agora descrever como usamos a linguagem PROLOG para Programação Lógica Extendida.

Os três tipos de respostas para uma questão $q(x)$ foram definidos usando lógica clássica no capítulo *Análise Teórica* na secção *Mundo Aberto* da seguinte maneira:

Verdadeiro $\exists x : q(x)$

Falso $\exists x : \neg q(x)$

Desconhecido $\neg\exists x : q(x) \vee \neg q(x)$

Como PROLOG é uma linguagem de programação que segue a lógica clássica, será necessário desenvolver um predicado baseado na definição acima que seja capaz de associar uma resposta a uma dada questão q. No nosso trabalho, esse predicado tem o nome `demo` de "demonstração", e é definido da seguinte maneira:

```
1 % demo : Q -> {V, F, D}
2 demo( Q, verdadeiro) :- Q.
3 demo( Q, falso) :- -Q.
4 demo( Q, desconhecido) :- nao(Q), nao(-Q).
```

Este predicado é apenas capaz de avaliar uma única questão, mas será também útil ter um predicado capaz de avaliar uma lista de predicados, aplicando quer a conjução, quer a disjunção a todos eles. Para isso temos o `demoC` e o `demoD` para cada respetiva operação. Estes são definidos em PROLOG da seguinte maneria:

```
% demoC : [Q] -> {V, F, D}
demoC(L, R) :- maplist(demo, L, Rs), and(Rs, R).

% and : [R] -> {V, F, D}
and([], verdadeiro).
and([Q | Qs], R) :- and(Qs, Rs), R eq Q && Rs.

% demoD : [Q] -> {V, F, D}
demoD(L, R) :- maplist(demo, L, Rs), or(Rs, R).

% or : [R] -> {V, F, D}
or([], falso).
or([Q | Qs], R) :- or(Qs, Rs), R eq Q $$ Rs.
```

Foi necessário definir o operador para lógica ternária $\wedge$ como `&&` e $\vee$ como `$$`, para além do operador `eq` que é semelhante ao operador `is` do PROLOG.

3.1 Representação de Conhecimento Imperfeito

Vamos relembrar a definição da negação para um predicado $p(x)$ descrito anteriormente.

$$\neg p(x) \leftarrow \neg \exists x : p(x) \vee excecao(p(x))$$

Esta definição é facilmente convertida para PROLOG da seguinte maneira:

```
-p(X) :- nao( p(X)), nao( excecao( p(X))).
```

Esta definição é usada no nosso trabalho para representar a negação dos predicados `utente`, `servico`, `ato` e `data`.

3.1.1 Conhecimento Incerto

Para representar conhecimento incerto tomamos partido da definição da negação descrito acima e do predicado exceção. Assim, se queremos registar o conhecimento incerto $p(x)$, é necessário primeiro declarar $p(x)$ e seguidamente declarar uma exceção de $p(x)$. Por exemplo, registar um utente para o qual sabemos a idade e a morada mas não o nome será feito da seguinte maneira:

```
utente(2, ut02_nome, 49, "S. Nicolau").

excecao( utente(Id, _, I, M)) :- utente(Id, ut02_nome, I, M).
```

No nosso trabalho, desenvolvemos um predicado dedicado a declarar conhecimento imperfeito incerto. Assim, para o exemplo anterior, bastaria fazer o seguinte:

```
utente(2, ut02_nome, 49, "S. Nicolau").

utente_desconhecido(2, nome).
```

Podemos testar o sistema para comprovar que respeita o predicado do mundo aberto, deixando desconhecido qual o verdadeiro nome do utente enquanto que garante a sua existência.

```
?- demo(utente(2, ut02_nome, 49, "S. Nicolau"), R).
R = Verdadeiro.

?- demo(utente(2, "Joao", 49, "S. Nicolau"), R).
R = Desconhecido.
```

Na eventualidade de ser conhecido o verdadeiro nome do utente 2, seria necessário substituir o valor `ut02_nome` pelo seu nome real.

3.1.2 Conhecimento Impreciso

Para representar conhecimento impreciso definimos três predicados auxiliares que poderiam vir a tornar-se úteis, estes são o `proximo_de`, `cerca_de` e `quinzena`, que retornam um intervalo que geralmente é usado para definir qual o conhecimento é desconhecido e qual é falso.

Tomando novamente partido do predicado `excecao`, podemos representar, por exemplo, um utente com idade proxima dos 14 anos da seguinte maneira:

```
utente(11, "Vitoria Alves", ut11_idade, "Guimaraes").

excecao(utente(11, _, Idade, _)) :- proximo_de(14, Min, Max),
                                    Idade >= Min,
                                    Idade =< Max.
```

Tal como fizemos para o conhecimento incerto, definimos um predicado para simplificar este processo, assim o código acima podia ser escrito da seguinte maneira:

```
utente(11, "Vitoria Alves", ut11_idade, "Guimaraes").

utente_desconhecido(11, idade, Idade, (proximo_de(14, Min, Max),
                                       Idade >= Min, Idade =< Max)).
```

Podemos assim verificar que o utente existe no sistema, comprovando este está registado e que, embora não conheçamos a sua idade real, conseguimos constatar que este não tem, por exemplo, 90 anos.

```
?- demo(utente(11, "Vitoria Alves", ut11_idade, "Guimaraes").
R = Verdadedeiro.

?- demo(utente(11, "Vitoria Alves", 90, "Guimaraes"), R).
R = Falso.

?- demo(utente(11, "Vitoria Alves", 15, "Guimaraes"), R).
R = Desconhecido.
```

3.1.3 Conhecimento Interdito

Como já foi descrito no capítulo anterior, conhecimento interdito é aquele que está para sempre destinado a ter valor ***desconhecido***. É portanto necessário marca-lo para que esse conhecimento nunca seja evoluído. Para isso temos o predicado `nulo` que proíbe que esse conhecimento seja expandido. Este predicado será usado num invariante para garantir que nunca será evoluído.

Para declarar que o utente 5 tem um nome e morada que nunca poderá ser descoberto fazemos o seguinte:

```
utente(5, ut05_nome, ut05_idade, ut05_morada).

nulo(ut05_nome).
nulo(ut05_morada).

+utente(_, Nome, _, _) :: (
        findall(Nome, (utente(5, Nome, _, _), nao(nulo(Nome))), []) ).

+utente(_, _, _, Morada) :: (
        findall(Morada, (utente(5, _, _, Morada), nao(nulo(Morada))), [])).
```

Novamente, desenvolvemos um predicado para simplificar este processo, assim o código acima podia simplesmente ser:

```
utente(5, ut05_nome, ut05_idade, ut05_morada).

utente_interdito(5, nome, Nome, (
        findall(Nome, (utente(5, Nome, _, _), nao(nulo(Nome))), []) ).

utente_interdito(5, morada, Morada, (
        findall(Morada, (utente(5, _, _, Morada), nao(nulo(Morada))), [])).
```

3.2 Evolução de Conhecimento Imperfeito

A evolução do conhecimento imperfeito para conhecimento perfeito é efetuado substituindo um facto que possuí um símbolo, representantivo de conhecimento imperfeito, por um facto com

um valor concreto no seu lugar, mantendo os restantes valores inalterados. É de notar que a inserção do novo facto, agora com um valor concreto, é sujeito à validação dos invariantes, sendo que a mudança pode não ocorrer.

Também neste caso foram definidos predicados para auxiliar o processo de evolução de conhecimento. Caso verificassemos o resultado final desta demonstração observariamos que a morada do utente foi corretamente alterada. Já o nome, sendo um valor nulo do tipo interdito não pode ser alterado, pelo que o predicado *conhecer_utente* falha no processo de validação dos invariantes, não havendo alterações na base de conhecimento.

```
?- evolucao( utente(5, ut05_nome, ut05_idade, ut05_morada) ).
true.

?- utente_desconhecido(5, morada).
true.

?- utente_interdito(5, nome, Nome, (
          findall(Nome, (utente(5, Nome, _, _), nao(nulo(Nome))), []) ).
true.

?- conhecer_utente(morada, 5, "S. Vitor").
true.

?- conhecer_utente(nome, 5, "Manelito").
false.
```

3.3 Mecanismos de Raciocínio

Com todas as mudanças realizadas na representação de conhecimento, torna-se também necessário alterar os predicados previamente definidos que operavam sobre a base de conhecimento. É necessário, durante a realização do predicado, verificar se estamos, ou não, perante conhecimento perfeito, evitando assim raciocinar sobre dados que não se encontram bem definidos.

Imaginemos, por exemplo, o predicado *idade_media* que calcula a média das idades dos utentes registados. Não fazendo sentido incluir os utentes cujas idades não são conhecidas, começamos por exclui-los, reduzindo assim o domínio sobre o qual o predicado vai atuar. Noutros casos, no entanto, a situação poderia ser diferente, podendo haver interesse em incluir os dados desconhecidos no domínio do problema.

```
idade_media(R)
    :- findall(Idade, (utente(_, _, Idade, _), integer(Idade)), L),
       length(L, Comprimento),
       sumlist(L, Soma),
       R is Soma / Comprimento.
```

Capítulo 4

Conclusão

Agora completo o trabalho, temos um sistema capaz de representar conhecimento perfeito e imperfeito, bem como mecanismos de raciocínio capazes de inferir sobre os vários tipos de conhecimento. Conseguimos expandir com sucesso a lógica clássica, suportando assim o valor lógico *desconhecido*, invariantes, assim como vários tipos de negação. Procuramos ainda simplificar o processo de declaração de conhecimento imperfeito assim como a sua evolução, tendo para isso implementado um conjunto de meta-predicados.