

# MAC0444 - Sistemas Baseados em Conhecimento

## Lista de Exercícios No. 2

Mateus Agostinho dos Anjos  
NUSP 9298191

2 de Outubro de 2019

1 -

Predicados:

$fezEx(x)$  = x fez os exercícios

$vaiBem(x)$  = x vai bem na prova

$mediaAlta(x)$  = x fica com media alta

$aprovado(x, y)$  = x é aprovado em y

Formalizando as sentenças do enunciado chegamos em:

$$\forall x (fezEx(x) \rightarrow vaiBem(x))$$

$$\forall y (vaiBem(y) \rightarrow mediaAlta(y))$$

$$\forall z (mediaAlta(z) \rightarrow aprovado(z, mac444))$$

$$fezEx(Jo\tilde{a}o)$$

$$vaiBem(Maria)$$

Base de conhecimento (KB):

1.  $[\neg fezEx(x), vaiBem(x)]$
2.  $[\neg vaiBem(y), mediaAlta(y)]$
3.  $[\neg mediaAlta(z), aprovado(z, mac444)]$
4.  $[fezEx(Jo\tilde{a}o)]$
5.  $[vaiBem(Maria)]$
6.  $[\neg aprovado(Jo\tilde{a}o, mac444)]$

Veja que inserimos  $[\neg aprovado(Jo\tilde{a}o, mac444)]$  na base de conhecimento, pois é a negação do nosso objetivo. Sendo assim, se chegarmos na cláusula vazia a partir desta base de conhecimento estará provado que  $aprovado(Jo\tilde{a}o, mac444)$  é consequência lógica das sentenças do enunciado.

Utilizando a **resolução SLD** temos:

$$\begin{array}{ll}
 \neg aprovado(Jo\tilde{a}o, mac444) & \text{(resolve com 3. e } z/Jo\tilde{a}o) \\
 \downarrow & \\
 \neg mediaAlta(Jo\tilde{a}o) & \text{(resolve com 2. e } y/Jo\tilde{a}o) \\
 \downarrow & \\
 \neg vaiBem(Jo\tilde{a}o) & \text{(resolve com 1. e } x/Jo\tilde{a}o) \\
 \downarrow & \\
 \neg fezEx(Jo\tilde{a}o) & \text{(resolve com 4.)} \\
 \downarrow & \\
 [] & 
 \end{array}$$

Sendo assim provamos que:  $KB \cup \{\neg aprovado(Jo\tilde{a}o, mac444)\}$  é insatisfazível, portanto  $aprovado(Jo\tilde{a}o, mac444)$  é consequência lógica de nossa base de conhecimento.

A **resolução SLD** será semelhante para Maria, portanto temos:  
Base de conhecimento (KB):

1.  $[\neg fezEx(x), vaiBem(x)]$
2.  $[\neg vaiBem(y), mediaAlta(y)]$
3.  $[\neg mediaAlta(z), aprovado(z, mac444)]$
4.  $[fazEx(Jo\~{a}o)]$
5.  $[vaiBem(Maria)]$
6.  $[\neg aprovado(Maria, mac444)]$

Utilizando a **resolução SLD** temos:

$$\begin{array}{ll}
 \neg aprovado(Maria, mac444) & \text{(resolve com 3. e } z/Maria) \\
 \downarrow & \\
 \neg mediaAlta(Maria) & \text{(resolve com 2. e } y/Maria) \\
 \downarrow & \\
 \neg vaiBem(Maria) & \text{(resolve com 5.)} \\
 \downarrow & \\
 [ ] & 
 \end{array}$$

**2 -**

Temos a Base de Conhecimento (KB) reescrita com variáveis renomeadas para evitar confusões na resolução do exercício:

1.  $[\neg A_1(x), \neg A_2(x), P(x)]$
2.  $[\neg B_1(y), \neg B_2(y), A_1(y)]$
3.  $[\neg B_3(z), \neg B_4(z), A_2(z)]$
4.  $[B_1(a)]$
5.  $[B_2(a)]$
6.  $[B_3(a)]$
7.  $[B_4(a)]$

a)

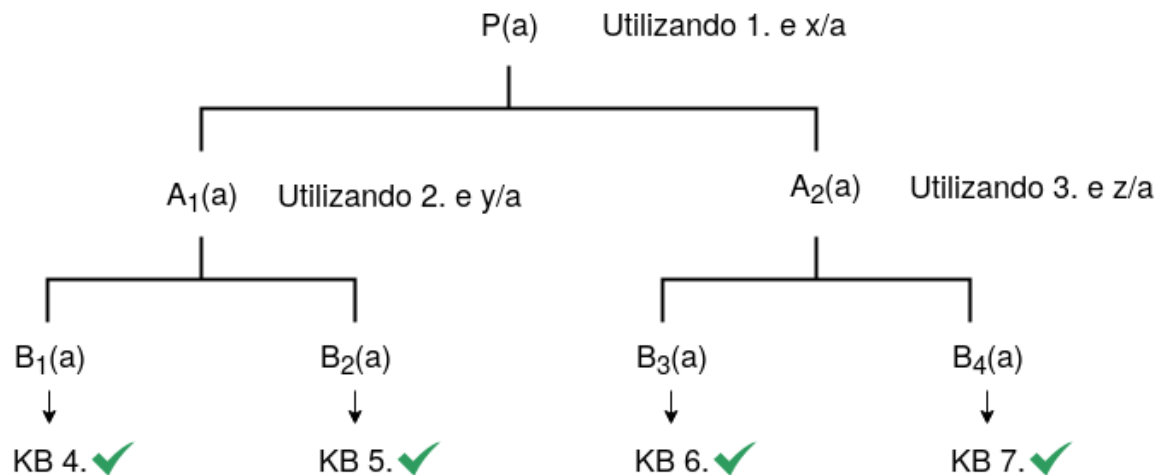
Para mostrar o passo a passo do procedimento de encadeamento para trás (backward chaining) devemos começar identificando as implicações da Base de Conhecimento.

Seguindo a ordem acima temos:

(note que utilizamos  $\leftarrow$  nas implicações)

1.  $\forall x(P(x) \leftarrow A_1(x) \wedge A_2(x))$
2.  $\forall y(A_1(y) \leftarrow B_1(y) \wedge B_2(y))$
3.  $\forall z(A_2(z) \leftarrow B_3(z) \wedge B_4(z))$
4.  $B_1(a)$
5.  $B_2(a)$
6.  $B_3(a)$
7.  $B_4(a)$

A partir destas implicações o passo a passo pode ser mostrado a partir da figura abaixo, sendo que cada passo gera pelo menos uma sub-árvore.



Todas as folhas da árvore estão na base de conhecimento em uma cláusula unitária, portanto são verdadeiras. Sendo assim podemos marcá-las com um ✓.

A partir disso podemos concluir que  $A_1(a)$  e  $A_2(a)$  estão provados e depois que  $P(a)$  está provado, portanto mostramos que o encadeamento para trás (backward chaining) produz resposta SIM com objetivo  $P(a)$ .

b)

Utilizando a **resolução SLD** com a base de conhecimento definida no início da questão, iniciamos com  $\neg P(a)$ , que é a negação do nosso objetivo, e buscaremos a cláusula vazia.

Obtemos o seguinte:

$$\begin{array}{ll}
 [\neg P(a)] & \text{(resolve com 1. e x/a)} \\
 \downarrow & \\
 [\neg A_1(a), \neg A_2(a)] & \text{(resolve com 2. e y/a)} \\
 \downarrow & \\
 [\neg B_1(y), \neg B_2(y), \neg A_2(a)] & \text{(resolve com 3. e z/a)} \\
 \downarrow & \\
 [\neg B_1(a), \neg B_2(a), \neg B_3(a), \neg B_4(a)] & \text{(resolve com 4.)} \\
 \downarrow & \\
 [\neg B_2(a), \neg B_3(a), \neg B_4(a)] & \text{(resolve com 5.)} \\
 \downarrow & \\
 [\neg B_3(a), \neg B_4(a)] & \text{(resolve com 6.)} \\
 \downarrow & \\
 [\neg B_4(a)] & \text{(resolve com 7.)} \\
 \downarrow & \\
 [ ] & 
 \end{array}$$

Como chegamos na cláusula vazia a partir de  $\neg P(a)$ , então está provado que  $P(a)$  é consequência desta base de conhecimento.

**3 -**

a)

Após ter carregado o programa a resposta do Prolog para a consulta:

? - result([a, b, c, d, e, f, g], X).

será:

$X = [b, d, f]$

b)

Considerando que a lista é enumerada a partir da posição 1, o programa recebe uma lista e elimina os elementos das posições ímpares, devolvendo apenas os elementos das posições pares.

Veja o exemplo de consulta que os elementos da lista coincidem com o número de sua posição:

? – *result*([1, 2, 3, 4, 5, 6], *X*).

*X* = [2, 4, 6]

Para fazer isso o programa possui um fato, *result*(*\_*, [*\_*]), que cobre os casos em que o primeiro argumento é uma lista vazia ou uma lista com apenas 1 elemento, pois nestes casos não é possível extrair 2 elementos da lista como a primeira regra exige (o corte impede a utilização do fato quando a lista tem 2 ou mais elementos).

Definida a base do programa a partir deste fato, chamadas recursivas serão feitas tentando equiparar, inicialmente, a lista passada como argumento e a primeira regra, veja a execução do exemplo:

Chamada inicial	<i>result</i> ([1, 2, 3, 4, 5, 6], <i>X</i> )
Casa com	<i>result</i> ( <i>_</i> , <i>E</i>   <i>L</i> ], [ <i>E</i>   <i>M</i> ])
Com valoração	<i>_</i> = 1, <i>E</i> = 2, <i>L</i> = [3, 4, 5, 6], <i>X</i> = [2 M <sub>1</sub> ]
Faz chamada recursiva	<i>result</i> ( <i>L</i> , <i>M</i> <sub>1</sub> )

Chamada	<i>result</i> ([3, 4, 5, 6], <i>M</i> <sub>1</sub> )
Casa com	<i>result</i> ( <i>_</i> , <i>E</i>   <i>L</i> ], [ <i>E</i>   <i>M</i> ])
Com valoração	<i>_</i> = 3, <i>E</i> = 4, <i>L</i> = [5, 6], <i>M</i> <sub>1</sub> = [4 M <sub>2</sub> ]
Faz chamada recursiva	<i>result</i> ( <i>L</i> , <i>M</i> <sub>2</sub> )

Chamada	<i>result</i> ([5, 6], <i>M</i> <sub>2</sub> )
Casa com	<i>result</i> ( <i>_</i> , <i>E</i>   <i>L</i> ], [ <i>E</i>   <i>M</i> ])
Com valoração	<i>_</i> = 5, <i>E</i> = 6, <i>L</i> = [], <i>M</i> <sub>2</sub> = [6 M <sub>3</sub> ]
Faz chamada recursiva	<i>result</i> ( <i>L</i> , <i>M</i> <sub>3</sub> )

Chamada	<i>result</i> ([], <i>M</i> <sub>3</sub> )
Casa com o fato	<i>result</i> ( <i>_</i> , [ <i>_</i> ])
Com valoração	<i>_</i> = [], <i>M</i> <sub>3</sub> = []

Após essa execução devemos obter o valor de  $X$ , para isso temos que reconstruí-lo a partir de  $M_3$ ,  $M_2$  e  $M_1$ , veja:

$$\begin{aligned} M_3 &= [\ ] &= [\ ] \\ M_2 &= [6|M_3] &= [6] \\ M_1 &= [4|M_2] &= [4, 6] \\ X &= [2|M_1] &= [2, 4, 6] \end{aligned}$$

Esta execução única só é possível, pois o corte (!) na primeira linha do programa faz com que não seja possível criar ramificações para obter diferentes respostas casando as chamadas recursivas intermediárias com o fato, uma vez que já foi utilizado a primeira regra de casamento (que possui a instrução de corte).

Sendo assim, o corte impede a alternativa de resposta em que o programa casa a chamada  $result([3, 4, 5, 6], M_1)$  com o fato  $result(., [\ ])$  ( $\_ = [3, 4, 5, 6]$  e  $M_1 = [\ ]$ ) e obtém a resposta  $X = [2]$ , por exemplo.

Note que foi utilizado a variável anônima ( $\_$ ), pois não queremos saber qual o valor do elemento que foi atribuído a ela durante o processo de obtenção do valor de  $X$ , queremos somente que exista um valor possível a ser atribuído.

4 -

a)

$$\begin{aligned} avof(Mul, Pess) &: - mae(Mul, Y), mae(Y, Pess). \\ avof(Mul, Pess) &: - mae(Mul, Y), pai(Y, Pess). \end{aligned}$$

b)

$$\begin{aligned} avom(Hom, Pess) &: - pai(Hom, Y), mae(Y, Pess). \\ avom(Hom, Pess) &: - pai(Hom, Y), pai(Y, Pess). \end{aligned}$$

c)

$$\begin{aligned} bisavom(Hom, Pess) &: - pai(Hom, Y), avom(Y, Pess). \\ bisavom(Hom, Pess) &: - pai(Hom, Y), avof(Y, Pess). \end{aligned}$$

d)

Primos de primeiro grau de P1 são os filhos e filhas dos tios de P1.

Por isso devemos definir quem é pai ou mãe de P1, chamado de X e quem é irmão ou irmã de X, chamado de Y, depois verificamos se Y é pai ou mãe de P2.

$primo\_1(P1, P2) : - pai(X, P1), irmaos(X, Y), pai(Y, P2).$   
 $primo\_1(P1, P2) : - pai(X, P1), irmaos(X, Y), mae(Y, P2).$   
 $primo\_1(P1, P2) : - mae(X, P1), irmaos(X, Y), pai(Y, P2).$   
 $primo\_1(P1, P2) : - mae(X, P1), irmaos(X, Y), mae(Y, P2).$

e)

Se X é primo de n-grau de Y e Z é filho de Y, então X é primo de (n+1)-grau de Z.

Devemos saber, portanto, quem são os descendentes dos primos de primeiro grau. Por isso devemos definir filho/2 e descendente/2.

$primo(P1, P2) : - irmaos(X, Y), descendente(P1, X), descendente(P2, Y).$   
 $filho(X, Y) : - pai(Y, X).$   
 $filho(X, Y) : - mae(Y, X).$   
 $descendente(X, Y) : - filho(X, Y).$   
 $descendente(X, Y) : - filho(X, Z), descendente(Z, Y).$

f)

$maior\_de\_idade(Pess) : - idade(Pess, X), X \geq 18.$

g)

Assumindo que uma pessoa ou é homem ou é mulher, definimos pessoa:

$pessoa(Pess) : - homem(Pess).$   
 $pessoa(Pess) : - mulher(Pess).$

Utilizando o comando findall, temos a lista de pessoas dada por:

$pessoas(Lista) : - findall(Pess, pessoa(Pess), Lista).$



h)

Utilizaremos o operador  $\backslash +$  para fazer uma busca em todas as pessoas definidas com idade/2, pegando X e procurando se existe algum  $Y > X$ , caso existir iremos comparar a nova maior idade X com todos os elementos que tem idade definida.

$$mais\_velho(Pess) : - idade(Pess, X), \backslash + (idade(_, Y), Y > X).$$

i)

Para criar a lista de pessoas de um determinado sexo com as respectivas idades devemos verificar se o Sexo passado é m (homem) ou f (mulher), depois disso utilizamos o comando findall com template "[Pess, X]" para armazenar a pessoa "Pess" e sua idade "X", utilizado como parâmetro de consulta (goal) se "Pess" tem idade "X" definida na base de conhecimento e se "Pess" é homem ou mulher dependendo se "Sexo" é "m" ou "f".

Desta forma:

$$lista\_pessoas(Lista, Sexo) : - Sexo = m, \\ findall([Pess, X], (idade(Pess, X), homem(Pess)), Lista).$$

$$lista\_pessoas(Lista, Sexo) : - Sexo = f, \\ findall([Pess, X], (idade(Pess, X), mulher(Pess)), Lista).$$

j)

Primeiro verificaremos se não há algum parentesco entre "X" e "Y" definindo a regra:

$$sem\_parentesco(X, Y) : - homem(X), mulher(Y), not(pai(X, Y)), \\ not(mae(Y, X)), not(irmao\_de(X, Y)), not(avof(Y, X)), not(avom(X, Y)), \\ not(bisavom(X, Y)), not(primo(X, Y)).$$

Agora criaremos uma regra para evitar que pessoas casadas sejam adequadas:

$$sem\_traicao(X, Y) : - not(casados(X, _)), not(casados(_, Y)).$$

Por fim podemos criar a regra "adequado" (note que "Z" é a idade do homem e "W" a idade da mulher):

*adequados*(*Hom*, *Mul*) :  $\neg$  *homem*(*Hom*), *mulher*(*Mul*),  
*sem\_parentesco*(*Hom*, *Mul*),  
*sem\_traicao*(*Hom*, *Mul*), *idade*(*Hom*, *Z*), *idade*(*Mul*, *W*),  
*not*( $Z < W - 2$ ), *not*( $Z > W + 10$ ).