

Inteligência Artificial – ACH2016

Aula 13 – *Backward Chaining* e Programação Lógica

Norton Trevisan Roman
(norton@usp.br)

11 de abril de 2019

L.P.O. – Backward Chaining

Funcionamento

- Começando do objetivo, “voltamos” nas regras de modo a encontrar fatos conhecidos que suportem a prova

L.P.O. – Backward Chaining

Funcionamento

- Começando do objetivo, “voltamos” nas regras de modo a encontrar fatos conhecidos que suportem a prova
 - Trata de regras do tipo *antecedente* \Rightarrow *consequente*

Funcionamento

- Começando do objetivo, “voltamos” nas regras de modo a encontrar fatos conhecidos que suportem a prova
 - Trata de regras do tipo *antecedente* \Rightarrow *consequente*
 - Fatos são tratados como uma implicação com um consequente e nenhum antecedente

Funcionamento

- Começando do objetivo, “voltamos” nas regras de modo a encontrar fatos conhecidos que suportem a prova
 - Trata de regras do tipo *antecedente* \Rightarrow *consequente*
 - Fatos são tratados como uma implicação com um consequente e nenhum antecedente
- Usado em programação lógica

L.P.O. – Backward Chaining

Algoritmo

Função *Backward*(*BC*, *obj*, θ): conjunto de substituições

resposta $\leftarrow \{\}$

se *obj* estiver vazia **então**

retorna $\{\theta\}$

$q' \leftarrow (\text{Primeiro}(\text{obj}))\theta$

para cada sentença *s* em *BC* onde são aplicados com sucesso

$\text{Padroniza}(s) = (p_1 \wedge \dots \wedge p_n \Rightarrow q)$ e $\theta' \leftarrow \text{Unifica}(q, q')$ **faça**

novoObj $\leftarrow [p_1, \dots, p_n | \text{Resto}(\text{obj})]$

resposta $\leftarrow \text{Backward}(BC, \text{novoObj}, \text{Composição}(\theta', \theta)) \cup \text{resposta}$

retorna *resposta*

L.P.O. – Backward Chaining

Algoritmo

Função *Backward*(*BC*, *obj*, θ): conjunto de substituições

resposta $\leftarrow \{\}$

se *obj* estiver vazia **então**

retorna $\{\theta\}$

Base de Conhecimento

$q' \leftarrow (\text{Primeiro}(\text{obj})) \theta$

para cada sentença *s* em *BC* onde são aplicados com sucesso

$\text{Padroniza}(s) = (p_1 \wedge \dots \wedge p_n \Rightarrow q)$ e $\theta' \leftarrow \text{Unifica}(q, q')$ **faça**

novoObj $\leftarrow [p_1, \dots, p_n | \text{Resto}(\text{obj})]$

resposta $\leftarrow \text{Backward}(\text{BC}, \text{novoObj}, \text{Composição}(\theta', \theta)) \cup \text{resposta}$

retorna *resposta*

L.P.O. – Backward Chaining

Algoritmo

Função *Backward*(*BC*, *obj*, θ): conjunto de substituições

$resposta \leftarrow \{\}$

se *obj* estiver vazia **então**

└ **retorna** $\{\theta\}$

Lista dos elementos que formam a query (θ já aplicado)

$q' \leftarrow (\text{Primeiro}(\text{obj})) \theta$

para cada sentença *s* em *BC* onde são aplicados com sucesso

$\text{Padroniza}(s) = (p_1 \wedge \dots \wedge p_n \Rightarrow q)$ e $\theta' \leftarrow \text{Unifica}(q, q')$ **faça**

└ $\text{novoObj} \leftarrow [p_1, \dots, p_n | \text{Resto}(\text{obj})]$

└ $resposta \leftarrow \text{Backward}(BC, \text{novoObj}, \text{Composição}(\theta', \theta)) \cup resposta$

└ **retorna** *resposta*

L.P.O. – Backward Chaining

Algoritmo

Função *Backward*(*BC*, *obj*, θ): conjunto de substituições

resposta $\leftarrow \{\}$

se *obj* estiver vazia **então**

retorna $\{\theta\}$

A substituição atual
(inicialmente $\{\}$)

$q' \leftarrow (\text{Primeiro}(\text{obj})) \theta$

para cada sentença *s* em *BC* onde são aplicados com sucesso

$\text{Padroniza}(s) = (p_1 \wedge \dots \wedge p_n \Rightarrow q)$ e $\theta' \leftarrow \text{Unifica}(q, q')$ **faça**

novoObj $\leftarrow [p_1, \dots, p_n | \text{Resto}(\text{obj})]$

resposta $\leftarrow \text{Backward}(BC, \text{novoObj}, \text{Composição}(\theta', \theta)) \cup \text{resposta}$

retorna *resposta*

L.P.O. – Backward Chaining

Algoritmo

Função *Backward*(*BC*, *obj*, θ): **conjunto de substituições**

$resposta \leftarrow \{\}$

se *obj* estiver vazia **então**

└ **retorna** $\{\theta\}$

Conjunto de todas as substituições que satisfazem a query

$q' \leftarrow (Primeiro(obj)) \theta$

para cada sentença *s* em *BC* onde são aplicados com sucesso

$Padroniza(s) = (p_1 \wedge \dots \wedge p_n \Rightarrow q)$ e $\theta' \leftarrow Unifica(q, q')$ **faça**

└ $novoObj \leftarrow [p_1, \dots, p_n | Resto(obj)]$

└ $resposta \leftarrow Backward(BC, novoObj, Composição(\theta', \theta)) \cup resposta$

└ **retorna** *resposta*

L.P.O. – Backward Chaining

Algoritmo

Função *Backward*(*BC*, *obj*, θ): conjunto de substituições

resposta $\leftarrow \{\}$

se *obj* estiver vazia **então**

└ **retorna** $\{\theta\}$

Conjunto de substituições

$q' \leftarrow (\text{Primeiro}(\text{obj})) \theta$

para cada sentença *s* em *BC* onde são aplicados com sucesso

Padroniza(*s*) = ($p_1 \wedge \dots \wedge p_n \Rightarrow q$) e $\theta' \leftarrow \text{Unifica}(q, q')$ **faça**

└ *novoObj* $\leftarrow [p_1, \dots, p_n | \text{Resto}(\text{obj})]$

└ *resposta* $\leftarrow \text{Backward}(\text{BC}, \text{novoObj}, \text{Composição}(\theta', \theta)) \cup \text{resposta}$

└ **retorna** *resposta*

L.P.O. – Backward Chaining

Algoritmo

Função *Backward*(*BC*, *obj*, θ): conjunto de substituições

resposta $\leftarrow \{\}$

se *obj* estiver vazia **então**

retorna $\{\theta\}$

Retira o primeiro objetivo em *obj*

$q' \leftarrow (\text{Primeiro}(\text{obj})) \theta$

para cada sentença *s* em *BC* onde são aplicados com sucesso

$\text{Padroniza}(s) = (p_1 \wedge \dots \wedge p_n \Rightarrow q)$ e $\theta' \leftarrow \text{Unifica}(q, q')$ **faça**

novoObj $\leftarrow [p_1, \dots, p_n | \text{Resto}(\text{obj})]$

resposta $\leftarrow \text{Backward}(BC, \text{novoObj}, \text{Composição}(\theta', \theta)) \cup \text{resposta}$

retorna *resposta*

L.P.O. – Backward Chaining

Algoritmo

Função *Backward*(*BC*, *obj*, θ): conjunto de substituições

$resposta \leftarrow \{\}$

se *obj* estiver vazia **então**

└ **retorna** $\{\theta\}$

$q' \leftarrow (\text{Primeiro}(\text{obj})) \theta$

para cada *sentença s* em *BC* onde são aplicados com sucesso

Padroniza(*s*) = ($p_1 \wedge \dots \wedge p_n \Rightarrow q$) e $\theta' \leftarrow \text{Unifica}(q, q')$ **faça**

└ $\text{novoObj} \leftarrow [p_1, \dots, p_n | \text{Resto}(\text{obj})]$

└ $resposta \leftarrow \text{Backward}(\text{BC}, \text{novoObj}, \text{Composição}(\theta', \theta)) \cup resposta$

└ **retorna** *resposta*

Verifica toda cláusula na
BC cujo conseqüente se
unifica com esse objetivo

L.P.O. – Backward Chaining

Algoritmo

Função *Backward*(*BC*,*obj*, θ): conjunto de substituições

$resposta \leftarrow \{\}$

se *obj* estiver vazia **então**

└ **retorna** $\{\theta\}$

$q' \leftarrow (Primeiro(obj))\theta$

para cada sentença *s* em *BC* onde são aplicados com sucesso

$Padroniza(s) = (p_1 \wedge \dots \wedge p_n \Rightarrow q)$ e $\theta' \leftarrow Unifica(q, q')$ **faça**

└ $novoObj \leftarrow [p_1, \dots, p_n | Resto(obj)]$

└ $resposta \leftarrow Backward(BC, novoObj, Composição(\theta', \theta)) \cup resposta$

└ **retorna** *resposta*

Cada uma dessas cláusulas
cria uma nova chamada
recursiva ao algoritmo

L.P.O. – Backward Chaining

Algoritmo

Função *Backward*(*BC*, *obj*, θ): conjunto de substituições

$resposta \leftarrow \{\}$

se *obj* estiver vazia **então**

└ **retorna** $\{\theta\}$

$q' \leftarrow (\text{Primeiro}(\text{obj}))\theta$

para cada sentença *s* em *BC* onde são aplicados com sucesso

$\text{Padroniza}(s) = (p_1 \wedge \dots \wedge p_n \Rightarrow q)$ e $\theta' \leftarrow \text{Unifica}(q, q')$ **faça**

└ $\text{novoObj} \leftarrow [p_1, \dots, p_n | \text{Resto}(\text{obj})]$

└ $resposta \leftarrow \text{Backward}(\text{BC}, \text{novoObj}, \text{Composição}(\theta', \theta)) \cup resposta$

└ **retorna** *resposta*

Suas premissas (antecedentes)
são adicionadas a (*obj* – *q'*) e
passadas a essa nova chamada

Algoritmo

Função *Backward*(*BC*, *obj*, θ): conjunto de substituições

resposta $\leftarrow \{\}$

se *obj* estiver vazia **então**

retorna $\{\theta\}$

$q' \leftarrow (\text{Primeiro}(\textit{obj})) \theta$

para cada sentença *s* em *BC* onde são aplicados com sucesso

$\text{Padroniza}(s) = (p_1 \wedge \dots \wedge p_n \Rightarrow q)$ e $\theta' \leftarrow \text{Unifica}(q, q')$ **faça**

novoObj $\leftarrow [p_1, \dots, p_n | \text{Resto}(\textit{obj})]$

resposta $\leftarrow \text{Backward}(\textit{BC}, \textit{novoObj}, \text{Composição}(\theta', \theta)) \cup \textit{resposta}$

retorna *resposta*

- $\text{Composição}(\theta', \theta)$ é a substituição cujo efeito é idêntico ao de aplicar cada uma das substituições em sequência

Algoritmo

Função *Backward*(*BC*,*obj*, θ): conjunto de substituições

resposta $\leftarrow \{\}$

se *obj* estiver vazia **então**

└ **retorna** $\{\theta\}$

$q' \leftarrow (\text{Primeiro}(\textit{obj})) \theta$

para cada sentença *s* em *BC* onde são aplicados com sucesso

Padroniza(*s*) = ($p_1 \wedge \dots \wedge p_n \Rightarrow q$) e $\theta' \leftarrow \text{Unifica}(q, q')$ **faça**

└ *novoObj* $\leftarrow [p_1, \dots, p_n | \text{Resto}(\textit{obj})]$

└ *resposta* $\leftarrow \text{Backward}(\textit{BC}, \textit{novoObj}, \text{Composição}(\theta', \theta)) \cup \textit{resposta}$

└ **retorna** *resposta*

- *Composição*(θ', θ) é a substituição cujo efeito é idêntico ao de aplicar cada uma das substituições em sequência

- Ou seja, $(p) \text{Composição}(\theta', \theta) = ((p) \theta') \theta$

L.P.O. – Backward Chaining

Exemplo

1	$Compatriota(x) \wedge Arma(y) \wedge Vender(x, y, z) \wedge$ $Hostil(z) \Rightarrow Criminoso(x)$	axioma
2	$Inimigo(Pit)$	axioma
3	$Míssil(M)$	axioma
4	$Ter(Pit, M)$	axioma
5	$Míssil(x) \wedge Ter(Pit, x) \Rightarrow Vender(Os, x, Pit)$	axioma
6	$Míssil(x) \Rightarrow Arma(x)$	axioma
7	$Inimigo(x) \Rightarrow Hostil(x)$	axioma
8	$Compatriota(Os)$	axioma

L.P.O. – Backward Chaining

Exemplo

1	$Compatriota(x) \wedge Arma(y) \wedge Vender(x, y, z) \wedge$ $Hostil(z) \Rightarrow Criminoso(x)$	axioma
2	$Inimigo(Pit)$	axioma
3	$Míssil(M)$	axioma
4	$Ter(Pit, M)$	axioma
5	$Míssil(x) \wedge Ter(Pit, x) \Rightarrow Vender(Os, x, Pit)$	axioma
6	$Míssil(x) \Rightarrow Arma(x)$	axioma
7	$Inimigo(x) \Rightarrow Hostil(x)$	axioma
8	$Compatriota(Os)$	axioma

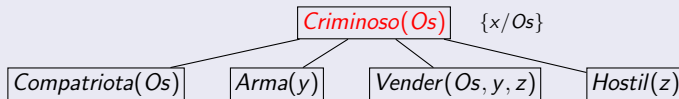
$Criminoso(Os)$

$\theta = \{\}$

L.P.O. – Backward Chaining

Exemplo

1	$Compatriota(x) \wedge Arma(y) \wedge Vender(x, y, z) \wedge Hostil(z) \Rightarrow Criminoso(x)$	axioma
2	$Inimigo(Pit)$	axioma
3	$Míssil(M)$	axioma
4	$Ter(Pit, M)$	axioma
5	$Míssil(x) \wedge Ter(Pit, x) \Rightarrow Vender(Os, x, Pit)$	axioma
6	$Míssil(x) \Rightarrow Arma(x)$	axioma
7	$Inimigo(x) \Rightarrow Hostil(x)$	axioma
8	$Compatriota(Os)$	axioma

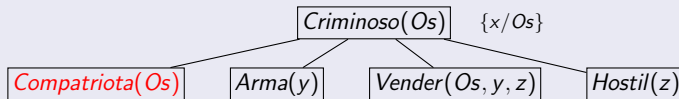


$$\theta = \{x/Os\}$$

L.P.O. – Backward Chaining

Exemplo

1	$Compatriota(x) \wedge Arma(y) \wedge Vender(x, y, z) \wedge Hostil(z) \Rightarrow Criminoso(x)$	axioma
2	$Inimigo(Pit)$	axioma
3	$Missil(M)$	axioma
4	$Ter(Pit, M)$	axioma
5	$Missil(x) \wedge Ter(Pit, x) \Rightarrow Vender(Os, x, Pit)$	axioma
6	$Missil(x) \Rightarrow Arma(x)$	axioma
7	$Inimigo(x) \Rightarrow Hostil(x)$	axioma
8	$Compatriota(Os)$	axioma

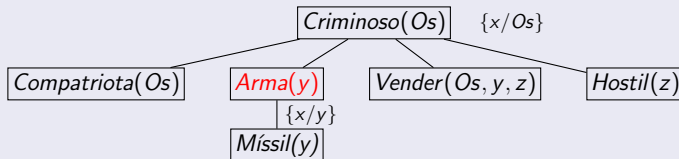


$$\theta = \{x/Os\}$$

L.P.O. – Backward Chaining

Exemplo

1	$Compatriota(x) \wedge Arma(y) \wedge Vender(x, y, z) \wedge Hostil(z) \Rightarrow Criminoso(x)$	axioma
2	$Inimigo(Pit)$	axioma
3	$Míssil(M)$	axioma
4	$Ter(Pit, M)$	axioma
5	$Míssil(x) \wedge Ter(Pit, x) \Rightarrow Vender(Os, x, Pit)$	axioma
6	$Míssil(x) \Rightarrow Arma(x)$	axioma
7	$Inimigo(x) \Rightarrow Hostil(x)$	axioma
8	$Compatriota(Os)$	axioma

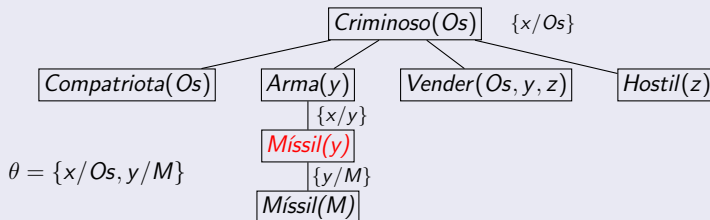


$$\theta = \{x/Os\}$$

L.P.O. – Backward Chaining

Exemplo

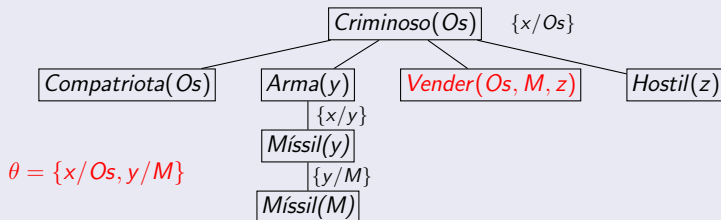
1	$Compatriota(x) \wedge Arma(y) \wedge Vender(x, y, z) \wedge Hostil(z) \Rightarrow Criminoso(x)$	axioma
2	$Inimigo(Pit)$	axioma
3	$Míssil(M)$	axioma
4	$Ter(Pit, M)$	axioma
5	$Míssil(x) \wedge Ter(Pit, x) \Rightarrow Vender(Os, x, Pit)$	axioma
6	$Míssil(x) \Rightarrow Arma(x)$	axioma
7	$Inimigo(x) \Rightarrow Hostil(x)$	axioma
8	$Compatriota(Os)$	axioma



L.P.O. – Backward Chaining

Exemplo

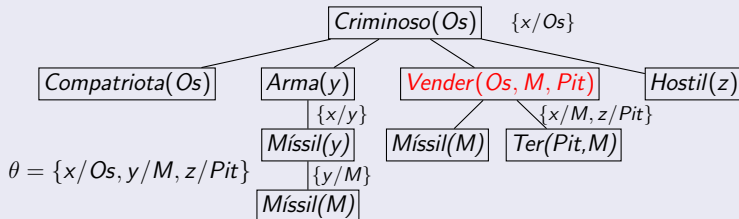
1	$Compatriota(x) \wedge Arma(y) \wedge Vender(x, y, z) \wedge Hostil(z) \Rightarrow Criminoso(x)$	axioma
2	$Inimigo(Pit)$	axioma
3	$Míssil(M)$	axioma
4	$Ter(Pit, M)$	axioma
5	$Míssil(x) \wedge Ter(Pit, x) \Rightarrow Vender(Os, x, Pit)$	axioma
6	$Míssil(x) \Rightarrow Arma(x)$	axioma
7	$Inimigo(x) \Rightarrow Hostil(x)$	axioma
8	$Compatriota(Os)$	axioma



L.P.O. – Backward Chaining

Exemplo

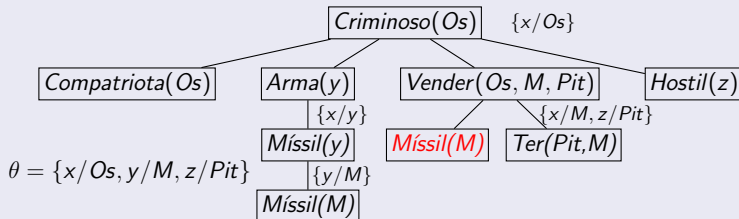
1	$Compatriota(x) \wedge Arma(y) \wedge Vender(x, y, z) \wedge Hostil(z) \Rightarrow Criminoso(x)$	axioma
2	$Inimigo(Pit)$	axioma
3	$Míssil(M)$	axioma
4	$Ter(Pit, M)$	axioma
5	$Míssil(x) \wedge Ter(Pit, x) \Rightarrow Vender(Os, x, Pit)$	axioma
6	$Míssil(x) \Rightarrow Arma(x)$	axioma
7	$Inimigo(x) \Rightarrow Hostil(x)$	axioma
8	$Compatriota(Os)$	axioma



L.P.O. – Backward Chaining

Exemplo

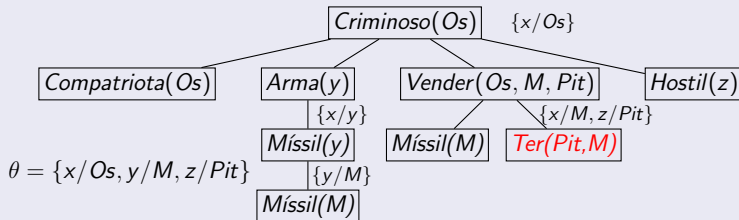
1	$Compatriota(x) \wedge Arma(y) \wedge Vender(x, y, z) \wedge Hostil(z) \Rightarrow Criminoso(x)$	axioma
2	$Inimigo(Pit)$	axioma
3	$Míssil(M)$	axioma
4	$Ter(Pit, M)$	axioma
5	$Míssil(x) \wedge Ter(Pit, x) \Rightarrow Vender(Os, x, Pit)$	axioma
6	$Míssil(x) \Rightarrow Arma(x)$	axioma
7	$Inimigo(x) \Rightarrow Hostil(x)$	axioma
8	$Compatriota(Os)$	axioma



L.P.O. – Backward Chaining

Exemplo

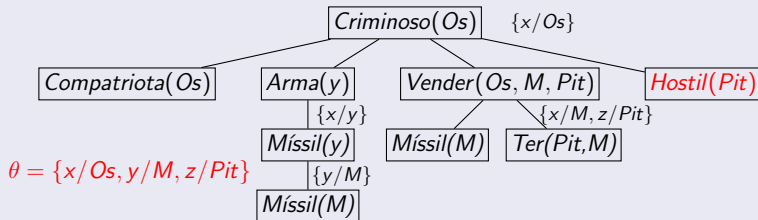
1	$Compatriota(x) \wedge Arma(y) \wedge Vender(x, y, z) \wedge Hostil(z) \Rightarrow Criminoso(x)$	axioma
2	$Inimigo(Pit)$	axioma
3	$Míssil(M)$	axioma
4	$Ter(Pit, M)$	axioma
5	$Míssil(x) \wedge Ter(Pit, x) \Rightarrow Vender(Os, x, Pit)$	axioma
6	$Míssil(x) \Rightarrow Arma(x)$	axioma
7	$Inimigo(x) \Rightarrow Hostil(x)$	axioma
8	$Compatriota(Os)$	axioma



L.P.O. – Backward Chaining

Exemplo

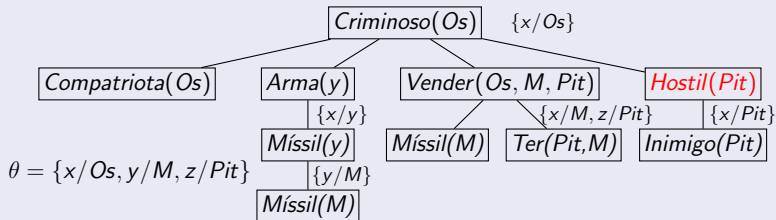
1	$Compatriota(x) \wedge Arma(y) \wedge Vender(x, y, z) \wedge Hostil(z) \Rightarrow Criminoso(x)$	axioma
2	$Inimigo(Pit)$	axioma
3	$Míssil(M)$	axioma
4	$Ter(Pit, M)$	axioma
5	$Míssil(x) \wedge Ter(Pit, x) \Rightarrow Vender(Os, x, Pit)$	axioma
6	$Míssil(x) \Rightarrow Arma(x)$	axioma
7	$Inimigo(x) \Rightarrow Hostil(x)$	axioma
8	$Compatriota(Os)$	axioma



L.P.O. – Backward Chaining

Exemplo

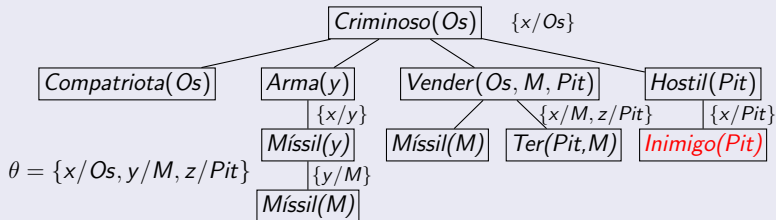
1	$Compatriota(x) \wedge Arma(y) \wedge Vender(x, y, z) \wedge Hostil(z) \Rightarrow Criminoso(x)$	axioma
2	$Inimigo(Pit)$	axioma
3	$Míssil(M)$	axioma
4	$Ter(Pit, M)$	axioma
5	$Míssil(x) \wedge Ter(Pit, x) \Rightarrow Vender(Os, x, Pit)$	axioma
6	$Míssil(x) \Rightarrow Arma(x)$	axioma
7	$Inimigo(x) \Rightarrow Hostil(x)$	axioma
8	$Compatriota(Os)$	axioma



L.P.O. – Backward Chaining

Exemplo

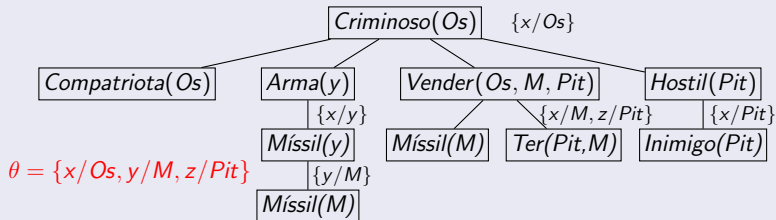
1	$Compatriota(x) \wedge Arma(y) \wedge Vender(x, y, z) \wedge Hostil(z) \Rightarrow Criminoso(x)$	axioma
2	$Inimigo(Pit)$	axioma
3	$Míssil(M)$	axioma
4	$Ter(Pit, M)$	axioma
5	$Míssil(x) \wedge Ter(Pit, x) \Rightarrow Vender(Os, x, Pit)$	axioma
6	$Míssil(x) \Rightarrow Arma(x)$	axioma
7	$Inimigo(x) \Rightarrow Hostil(x)$	axioma
8	$Compatriota(Os)$	axioma



L.P.O. – Backward Chaining

Exemplo

1	$Compatriota(x) \wedge Arma(y) \wedge Vender(x, y, z) \wedge Hostil(z) \Rightarrow Criminoso(x)$	axioma
2	$Inimigo(Pit)$	axioma
3	$Míssil(M)$	axioma
4	$Ter(Pit, M)$	axioma
5	$Míssil(x) \wedge Ter(Pit, x) \Rightarrow Vender(Os, x, Pit)$	axioma
6	$Míssil(x) \Rightarrow Arma(x)$	axioma
7	$Inimigo(x) \Rightarrow Hostil(x)$	axioma
8	$Compatriota(Os)$	axioma



Características

- É uma busca em profundidade recursiva

Características

- É uma busca em profundidade recursiva
 - Cláusulas são testadas na ordem em que ocorrem na base

Características

- É uma busca em profundidade recursiva
 - Cláusulas são testadas na ordem em que ocorrem na base
- Sofre com presença de laços

L.P.O. – Backward Chaining

Características

- É uma busca em profundidade recursiva
 - Cláusulas são testadas na ordem em que ocorrem na base
- Sofre com presença de laços
 - Ex: Encontre um caminho de A a C , a partir da base (testando as alternativas em ordem)

1	$Ligação(A,B)$
2	$Ligação(B,C)$
3	$Ligação(x,z) \Rightarrow Caminho(x,z)$
4	$Ligação(y,z) \wedge Caminho(x,y) \Rightarrow Caminho(x,z)$

L.P.O. – Backward Chaining

Características

- $\text{Caminho}(A,C)?$

1	$\text{Ligação}(A,B)$
2	$\text{Ligação}(B,C)$
3	$\text{Ligação}(x,z) \Rightarrow \text{Caminho}(x,z)$
4	$\text{Ligação}(y,z) \wedge \text{Caminho}(x,y) \Rightarrow \text{Caminho}(x,z)$

L.P.O. – Backward Chaining

Características

- $\text{Caminho}(A, C)?$

1	$\text{Ligação}(A, B)$
2	$\text{Ligação}(B, C)$
3	$\text{Ligação}(x, z) \Rightarrow \text{Caminho}(x, z)$
4	$\text{Ligação}(y, z) \wedge \text{Caminho}(x, y) \Rightarrow \text{Caminho}(x, z)$
<div>$\text{Caminho}(A, C)$</div>	

L.P.O. – Backward Chaining

Características

- $Caminho(A,C)?$

1	$Liga\tilde{c}\tilde{a}o(A,B)$
2	$Liga\tilde{c}\tilde{a}o(B,C)$
3	$Liga\tilde{c}\tilde{a}o(x,z) \Rightarrow Caminho(x,z)$
4	$Liga\tilde{c}\tilde{a}o(y,z) \wedge Caminho(x,y) \Rightarrow Caminho(x,z)$

$Caminho(A,C)$

$Liga\tilde{c}\tilde{a}o(A,C)$

L.P.O. – Backward Chaining

Características

- $\text{Caminho}(A,C)?$

1	$\text{Ligação}(A,B)$
2	$\text{Ligação}(B,C)$
3	$\text{Ligação}(x,z) \Rightarrow \text{Caminho}(x,z)$
4	$\text{Ligação}(y,z) \wedge \text{Caminho}(x,y) \Rightarrow \text{Caminho}(x,z)$

$\text{Caminho}(A,C)$

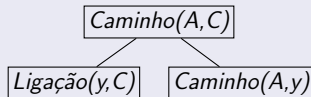
$\text{Ligação}(A,C)$

L.P.O. – Backward Chaining

Características

- $\text{Caminho}(A,C)?$

1	$\text{Ligação}(A,B)$
2	$\text{Ligação}(B,C)$
3	$\text{Ligação}(x,z) \Rightarrow \text{Caminho}(x,z)$
4	$\text{Ligação}(y,z) \wedge \text{Caminho}(x,y) \Rightarrow \text{Caminho}(x,z)$

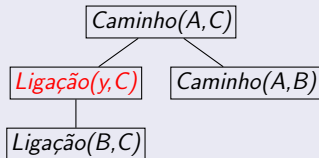


L.P.O. – Backward Chaining

Características

- $\text{Caminho}(A,C)?$

1	$\text{Ligação}(A,B)$
2	$\text{Ligação}(B,C)$
3	$\text{Ligação}(x,z) \Rightarrow \text{Caminho}(x,z)$
4	$\text{Ligação}(y,z) \wedge \text{Caminho}(x,y) \Rightarrow \text{Caminho}(x,z)$

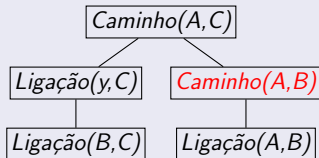


L.P.O. – Backward Chaining

Características

- $\text{Caminho}(A,C)?$

1	$\text{Ligação}(A,B)$
2	$\text{Ligação}(B,C)$
3	$\text{Ligação}(x,z) \Rightarrow \text{Caminho}(x,z)$
4	$\text{Ligação}(y,z) \wedge \text{Caminho}(x,y) \Rightarrow \text{Caminho}(x,z)$

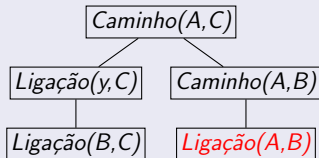


L.P.O. – Backward Chaining

Características

- $\text{Caminho}(A,C)?$

1	$\text{Ligação}(A,B)$
2	$\text{Ligação}(B,C)$
3	$\text{Ligação}(x,z) \Rightarrow \text{Caminho}(x,z)$
4	$\text{Ligação}(y,z) \wedge \text{Caminho}(x,y) \Rightarrow \text{Caminho}(x,z)$

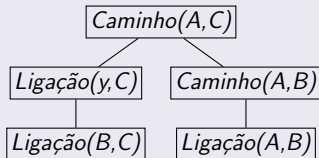


L.P.O. – Backward Chaining

Características

- $\text{Caminho}(A,C)?$

1	$\text{Ligação}(A,B)$
2	$\text{Ligação}(B,C)$
3	$\text{Ligação}(x,z) \Rightarrow \text{Caminho}(x,z)$
4	$\text{Ligação}(y,z) \wedge \text{Caminho}(x,y) \Rightarrow \text{Caminho}(x,z)$



- OK

L.P.O. – Backward Chaining

Características

- $\text{Caminho}(A,C)?$

1	$\text{Ligação}(A,B)$
2	$\text{Ligação}(B,C)$
3	$\text{Ligação}(y,z) \wedge \text{Caminho}(x,y) \Rightarrow \text{Caminho}(x,z)$
4	$\text{Ligação}(x,z) \Rightarrow \text{Caminho}(x,z)$

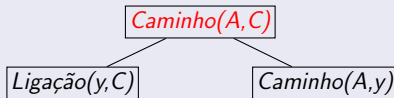
$\boxed{\text{Caminho}(A,C)}$

L.P.O. – Backward Chaining

Características

- $\text{Caminho}(A,C)?$

1	$\text{Ligação}(A,B)$
2	$\text{Ligação}(B,C)$
3	$\text{Ligação}(y,z) \wedge \text{Caminho}(x,y) \Rightarrow \text{Caminho}(x,z)$
4	$\text{Ligação}(x,z) \Rightarrow \text{Caminho}(x,z)$

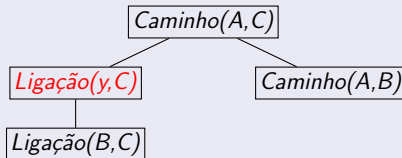


L.P.O. – Backward Chaining

Características

- $\text{Caminho}(A,C)?$

1	$\text{Ligação}(A,B)$
2	$\text{Ligação}(B,C)$
3	$\text{Ligação}(y,z) \wedge \text{Caminho}(x,y) \Rightarrow \text{Caminho}(x,z)$
4	$\text{Ligação}(x,z) \Rightarrow \text{Caminho}(x,z)$

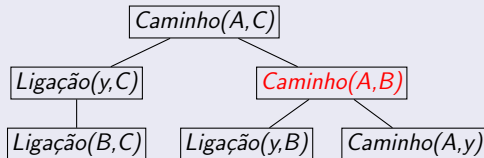


L.P.O. – Backward Chaining

Características

- $\text{Caminho}(A,C)?$

1	$\text{Ligação}(A,B)$
2	$\text{Ligação}(B,C)$
3	$\text{Ligação}(y,z) \wedge \text{Caminho}(x,y) \Rightarrow \text{Caminho}(x,z)$
4	$\text{Ligação}(x,z) \Rightarrow \text{Caminho}(x,z)$

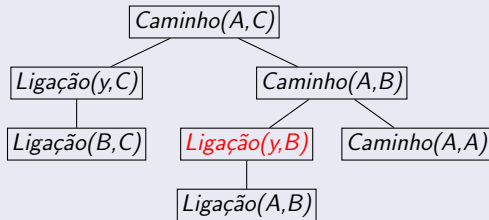


L.P.O. – Backward Chaining

Características

- $\text{Caminho}(A,C)?$

1	$\text{Ligação}(A,B)$
2	$\text{Ligação}(B,C)$
3	$\text{Ligação}(y,z) \wedge \text{Caminho}(x,y) \Rightarrow \text{Caminho}(x,z)$
4	$\text{Ligação}(x,z) \Rightarrow \text{Caminho}(x,z)$

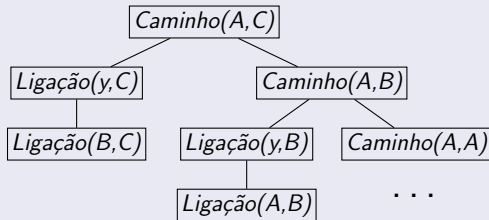


L.P.O. – Backward Chaining

Características

- $\text{Caminho}(A,C)?$

1	$\text{Ligação}(A,B)$
2	$\text{Ligação}(B,C)$
3	$\text{Ligação}(y,z) \wedge \text{Caminho}(x,y) \Rightarrow \text{Caminho}(x,z)$
4	$\text{Ligação}(x,z) \Rightarrow \text{Caminho}(x,z)$

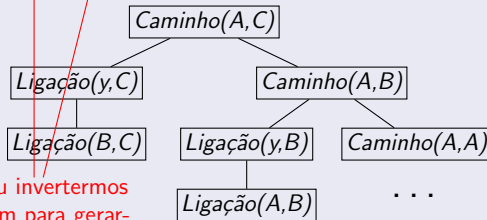


L.P.O. – Backward Chaining

Características

- $\text{Caminho}(A,C)?$

1	$\text{Ligação}(A,B)$
2	$\text{Ligação}(B,C)$
3	$\text{Ligação}(y,z) \wedge \text{Caminho}(x,y) \Rightarrow \text{Caminho}(x,z)$
4	$\text{Ligação}(x,z) \Rightarrow \text{Caminho}(x,z)$



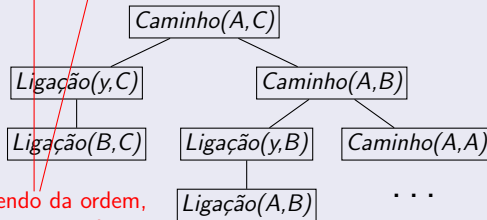
Bastou invertermos
a ordem para gerarmos
mais tentativas.

L.P.O. – Backward Chaining

Características

- $\text{Caminho}(A,C)?$

1	$\text{Ligação}(A,B)$
2	$\text{Ligação}(B,C)$
3	$\text{Ligação}(y,z) \wedge \text{Caminho}(x,y) \Rightarrow \text{Caminho}(x,z)$
4	$\text{Ligação}(x,z) \Rightarrow \text{Caminho}(x,z)$



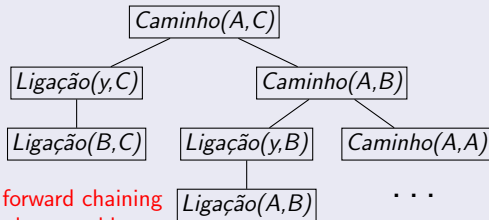
Dependendo da ordem,
até infinitas tentativas...

L.P.O. – Backward Chaining

Características

- $\text{Caminho}(A,C)?$

1	$\text{Ligação}(A,B)$
2	$\text{Ligação}(B,C)$
3	$\text{Ligação}(y,z) \wedge \text{Caminho}(x,y) \Rightarrow \text{Caminho}(x,z)$
4	$\text{Ligação}(x,z) \Rightarrow \text{Caminho}(x,z)$



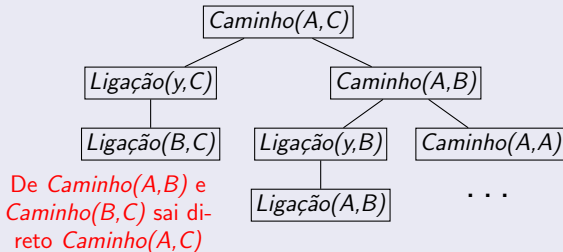
Note que forward chaining
não sofre desse problema

L.P.O. – Backward Chaining

Características

- $\text{Caminho}(A,C)?$

1	$\text{Ligação}(A,B)$
2	$\text{Ligação}(B,C)$
3	$\text{Ligação}(y,z) \wedge \text{Caminho}(x,y) \Rightarrow \text{Caminho}(x,z)$
4	$\text{Ligação}(x,z) \Rightarrow \text{Caminho}(x,z)$



Características

- Sofre com computações redundantes

Características

- Sofre com computações redundantes
 - Solução: Guarde resultados já obtidos para sub-objetivos e reuse-os.

Programação Lógica

Programação Lógica

Características

Programação Lógica

Características

- Programas são construídos pela expressão do conhecimento em uma linguagem formal

Programação Lógica

Características

- Programas são construídos pela expressão do conhecimento em uma linguagem formal
 - São conjuntos de cláusulas definidas

Programação Lógica

Características

- Programas são construídos pela expressão do conhecimento em uma linguagem formal
 - São conjuntos de cláusulas definidas
- Problemas são resolvidos via inferência a partir desse conhecimento

Programação Lógica

Características

- Programas são construídos pela expressão do conhecimento em uma linguagem formal
 - São conjuntos de cláusulas definidas
- Problemas são resolvidos via inferência a partir desse conhecimento
- Alguns usos:

Programação Lógica

Características

- Programas são construídos pela expressão do conhecimento em uma linguagem formal
 - São conjuntos de cláusulas definidas
- Problemas são resolvidos via inferência a partir desse conhecimento
- Alguns usos:
 - Linguagem de prototipação, Análise sintática de língua natural, Sistemas especialistas etc

Programação Lógica

Características

- Programas são construídos pela expressão do conhecimento em uma linguagem formal
 - São conjuntos de cláusulas definidas
- Problemas são resolvidos via inferência a partir desse conhecimento
- Alguns usos:
 - Linguagem de prototipação, Análise sintática de língua natural, Sistemas especialistas etc
- Prolog é a linguagem mais usada, mas não a única

Programação Lógica

Programação Lógica × Não Lógica

Lógica

Identifique o problema

Reúna a informação

(Pausa para o café)

Codifique a informação na base

Codifique instâncias do
problema como fatos

Faça perguntas (queries)

Encontre fatos falsos

Não Lógica

Identifique o problema

Reúna a informação

Identifique soluções

Programe a solução

Codifique instâncias do
problema como dados

Aplique o programa aos dados

Corrija erros de procedimento

Programação Lógica – Prolog

Notação

Programação Lógica – Prolog

Notação

- Variáveis: iniciam com letras maiúsculas

Programação Lógica – Prolog

Notação

- Variáveis: iniciam com letras maiúsculas
- Contantes: escritas com letras minúsculas

Notação

- Variáveis: iniciam com letras maiúsculas
- Contantes: escritas com letras minúsculas
- Cláusulas são escritas com o conseqüente antes dos antecedentes:

Notação

- Variáveis: iniciam com letras maiúsculas
- Contantes: escritas com letras minúsculas
- Cláusulas são escritas com o consequente antes dos antecedentes:
 - $\text{criminoso}(X) \text{ :- compatriota}(X), \text{ arma}(Y), \text{ vender}(X,Y,Z), \text{ hostil}(Z).$
 $(\text{compatriota}(x) \wedge \text{arma}(y) \wedge \text{vender}(x, y, z) \wedge \text{hostil}(z) \Rightarrow \text{criminoso}(x))$

Notação

- Variáveis: iniciam com letras maiúsculas
- Contantes: escritas com letras minúsculas
- Cláusulas são escritas com o consequente antes dos antecedentes:
 - $\text{criminoso}(X) \text{ :- compatriota}(X), \text{ arma}(Y), \text{ vender}(X,Y,Z), \text{ hostil}(Z).$
$$(\text{compatriota}(x) \wedge \text{arma}(y) \wedge \text{vender}(x, y, z) \wedge \text{hostil}(z) \Rightarrow \text{criminoso}(x))$$
- Queries são feitas diretamente ao interpretador

Notação

- Variáveis: iniciam com letras maiúsculas
- Contantes: escritas com letras minúsculas
- Cláusulas são escritas com o consequente antes dos antecedentes:
 - $\text{criminoso}(X) \text{ :- compatriota}(X), \text{ arma}(Y), \text{ vender}(X,Y,Z), \text{ hostil}(Z).$
$$(\text{compatriota}(x) \wedge \text{arma}(y) \wedge \text{vender}(x, y, z) \wedge \text{hostil}(z) \Rightarrow \text{criminoso}(x))$$
- Queries são feitas diretamente ao interpretador
 - Sim... é interpretada, mas há versões compiladas

Programação Lógica – Prolog

Características

- A execução de um programa é feita via *backward chaining*, usando busca em profundidade

Programação Lógica – Prolog

Características

- A execução de um programa é feita via *backward chaining*, usando busca em profundidade
 - Cláusulas são testadas na ordem em que foram escritas

Características

- A execução de um programa é feita via *backward chaining*, usando busca em profundidade
 - Cláusulas são testadas na ordem em que foram escritas
- A base de conhecimento (ou seja, regras e fatos) só pode ser lida de arquivos

Características

- A execução de um programa é feita via *backward chaining*, usando busca em profundidade
 - Cláusulas são testadas na ordem em que foram escritas
- A base de conhecimento (ou seja, regras e fatos) só pode ser lida de arquivos
 - Via “[nome_do_arquivo].”

Características

- A execução de um programa é feita via *backward chaining*, usando busca em profundidade
 - Cláusulas são testadas na ordem em que foram escritas
- A base de conhecimento (ou seja, regras e fatos) só pode ser lida de arquivos
 - Via “[nome_do_arquivo’].”
 - Repare o ‘.’ ao final → Todo comando em prolog termina em ‘.’

Programação Lógica – Prolog

Funcionalidades externas à inferência lógica:

Programação Lógica – Prolog

Funcionalidades externas à inferência lógica:

- Funções aritméticas

Programação Lógica – Prolog

Funcionalidades externas à inferência lógica:

- Funções aritméticas
 - Ex: “X is 7+4.”, “X is 3*5+2.” etc

Programação Lógica – Prolog

Funcionalidades externas à inferência lógica:

- Funções aritméticas
 - Ex: “X is 7+4.”, “X is 3*5+2.” etc
 - “Provadas” pela execução de código, em vez de inferência

Funcionalidades externas à inferência lógica:

- Funções aritméticas
 - Ex: “X is 7+4.”, “X is 3*5+2.” etc
 - “Provadas” pela execução de código, em vez de inferência
- Predicados de entrada e saída

Programação Lógica – Prolog

Funcionalidades externas à inferência lógica:

- Funções aritméticas
 - Ex: “X is 7+4.”, “X is 3*5+2.” etc
 - “Provadas” pela execução de código, em vez de inferência
- Predicados de entrada e saída
 - “write(termo)” e “read(variável)”

Programação Lógica – Prolog

Funcionalidades externas à inferência lógica:

- Funções aritméticas
 - Ex: “X is 7+4.”, “X is 3*5+2.” etc
 - “Provadas” pela execução de código, em vez de inferência
- Predicados de entrada e saída
 - “write(termo)” e “read(variável)”
 - assert/retract: modificam a base de conhecimento

Programação Lógica – Prolog

Funcionalidades externas à inferência lógica:

- Funções aritméticas
 - Ex: “X is 7+4.”, “X is 3*5+2.” etc
 - “Provadas” pela execução de código, em vez de inferência
- Predicados de entrada e saída
 - “write(termo)” e “read(variável)”
 - assert/retract: modificam a base de conhecimento
- Igualdade (operador =)

Programação Lógica – Prolog

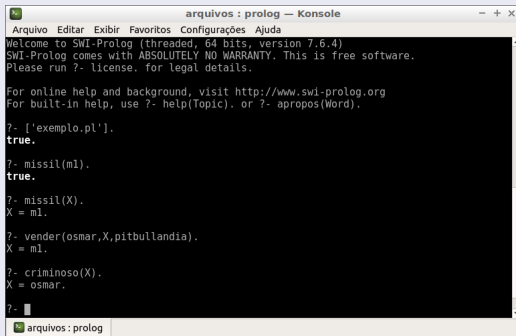
Funcionalidades externas à inferência lógica:

- Funções aritméticas
 - Ex: “X is 7+4.”, “X is 3*5+2.” etc
 - “Provadas” pela execução de código, em vez de inferência
- Predicados de entrada e saída
 - “write(termo)” e “read(variável)”
 - assert/retract: modificam a base de conhecimento
- Igualdade (operador =)
 - Verifica se ambos os termos são unificáveis (em vez de referenciarem o mesmo objeto)

Programação Lógica – Prolog

Exemplo

```
criminoso(X) :- compatriota(X), arma(Y), vender(X,Y,Z), hostil(Z).  
arma(X) :- missil(X).  
hostil(X) :- inimigo(X).  
vender(osmar,X,pitbullandia):- missil(X), ter(pitbullandia,X).  
ter(pitbullandia,m1).  
missil(m1).  
compatriota(osmar).  
inimigo(pitbullandia).
```



The screenshot shows a Prolog console window titled "arquivos : prolog — Konsole". The window contains the following text:

```
Arquivo Editar Exibir Favoritos Configurações Ajuda  
Welcome to SWI-Prolog (threaded, 64 bits, version 7.6.4)  
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.  
Please run ?- license. for legal details.  
  
For online help and background, visit http://www.swi-prolog.org  
For built-in help, use ?- help(Topic). or ?- apropos(Word).  
  
?- ['exemplo.pl'].  
true.  
  
?- missil(m1).  
true.  
  
?- missil(X).  
X = m1.  
  
?- vender(osmar,X,pitbullandia).  
X = m1.  
  
?- criminoso(X).  
X = osmar.  
  
?-
```

Negação

- E se quisermos dizer que algo não é verdade?

Negação

- E se quisermos dizer que algo não é verdade?
- Problemas:

Negação

- E se quisermos dizer que algo não é verdade?
- Problemas:
 - A regra resultante não é uma cláusula definida (literais devem ser positivos)

Negação

- E se quisermos dizer que algo não é verdade?
- Problemas:
 - A regra resultante não é uma cláusula definida (literais devem ser positivos)
 - Em LPO, não temos como concluir uma negação → Somente fatos positivos são derivados

Negação

- Em programação lógica, tipicamente assumimos um mundo fechado

Negação

- Em programação lógica, tipicamente assumimos um mundo fechado
 - Sabemos tudo que há para saber sobre o domínio

Negação

- Em programação lógica, tipicamente assumimos um mundo fechado
 - Sabemos tudo que há para saber sobre o domínio
 - Se não sabemos algo (ou não podemos prová-lo), então deve ser falso

Negação

- Em programação lógica, tipicamente assumimos um mundo fechado
 - Sabemos tudo que há para saber sobre o domínio
 - Se não sabemos algo (ou não podemos prová-lo), então deve ser falso
- Que fazer então?

Negação

- Em programação lógica, tipicamente assumimos um mundo fechado
 - Sabemos tudo que há para saber sobre o domínio
 - Se não sabemos algo (ou não podemos prová-lo), então deve ser falso
- Que fazer então?
 - Simular a negação usando a falha em prová-la

Programação Lógica – Prolog

Simulando Negação

- Se não conseguimos provar, então é falso

Programação Lógica – Prolog

Simulando Negação

- Se não conseguimos provar, então é falso
 - `legal(X) :- \+ ilegal(X)`

Programação Lógica – Prolog

Simulando Negação

- Se não conseguimos provar, então é falso
 - `legal(X) :- \+ ilegal(X)`
 - `\+` objetivo será verdadeiro se o objetivo não puder ser provado

Simulando Negação

- Se não conseguimos provar, então é falso
 - `legal(X) :- \+ ilegal(X)`
 - `\+` objetivo será verdadeiro se o objetivo não puder ser provado
- O interpretador fará uma busca em todas as coisas ilegais, comparando-as com X

Simulando Negação

- Se não conseguimos provar, então é falso
 - `legal(X) :- \+ ilegal(X)`
 - `\+` objetivo será verdadeiro se o objetivo não puder ser provado
- O interpretador fará uma busca em todas as coisas ilegais, comparando-as com X
 - Se nenhum fato ilegal for o mesmo que X, X é legal

Simulando Negação

- Se não conseguimos provar, então é falso
 - `legal(X) :- \+ ilegal(X)`
 - `\+` objetivo será verdadeiro se o objetivo não puder ser provado
- O interpretador fará uma busca em todas as coisas ilegais, comparando-as com X
 - Se nenhum fato ilegal for o mesmo que X, X é legal
- Tudo que eu não puder provar ilegal, é legal

Simulando Negação

- Se não conseguimos provar, então é falso
 - `legal(X) :- \+ ilegal(X)`
 - `\+` objetivo será verdadeiro se o objetivo não puder ser provado
- O interpretador fará uma busca em todas as coisas ilegais, comparando-as com X
 - Se nenhum fato ilegal for o mesmo que X, X é legal
- Tudo que eu não puder provar ilegal, é legal
 - Perigoso, se não soubermos tudo sobre o domínio

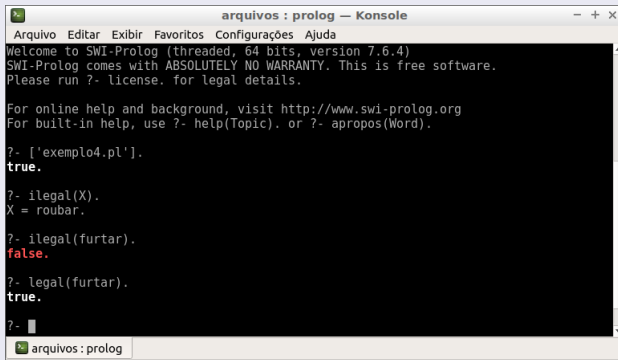
Simulando Negação

- Se não conseguimos provar, então é falso
 - `legal(X) :- \+ ilegal(X)`
 - `\+` objetivo será verdadeiro se o objetivo não puder ser provado
- O interpretador fará uma busca em todas as coisas ilegais, comparando-as com X
 - Se nenhum fato ilegal for o mesmo que X, X é legal
- Tudo que eu não puder provar ilegal, é legal
 - Perigoso, se não soubermos tudo sobre o domínio
 - Em uma base vazia, qualquer coisa pode ser derivada assim

Programação Lógica – Prolog

Simulando Negação – Exemplo

```
legal(X) :- \+ ilegal(X).  
ilegal(roubar).
```



```
arquivos : prolog — Konsole  
Arquivo Editar Exibir Favoritos Configurações Ajuda  
Welcome to SWI-Prolog (threaded, 64 bits, version 7.6.4)  
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.  
Please run ?- license. for legal details.  
  
For online help and background, visit http://www.swi-prolog.org  
For built-in help, use ?- help(Topic). or ?- apropos(Word).  
  
?- ['exemplo4.pl'].  
true.  
  
?- ilegal(X).  
X = roubar.  
  
?- ilegal(furtar).  
false.  
  
?- legal(furtar).  
true.  
  
?-
```

Programação Lógica – Prolog

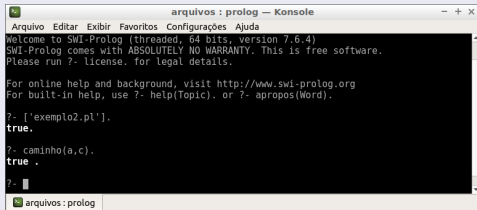
E o calcanhar de Aquiles...

```
caminho(X,Z) :- ligacao(X,Z).  
caminho(X,Z) :- caminho(X,Y), ligacao(Y,Z).  
ligacao(a,b).  
ligacao(b,c).
```

Programação Lógica – Prolog

E o calcanhar de Aquiles...

```
caminho(X,Z) :- ligacao(X,Z).  
caminho(X,Z) :- caminho(X,Y), ligacao(Y,Z).  
ligacao(a,b).  
ligacao(b,c).
```

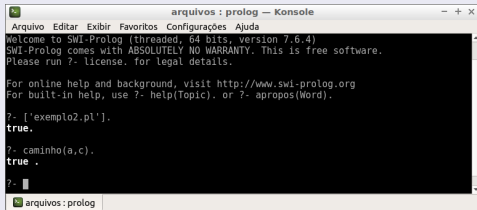


```
arquivos: prolog - Konsole  
Arquivo Editar Exibir Favoritos Configurações Ajuda  
Welcome to SWI-Prolog (threaded, 64 bits, version 7.6.4)  
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.  
Please run ?- license. for legal details.  
  
For online help and background, visit http://www.swi-prolog.org  
For built-in help, use ?- help(Topic). or ?- apropos(Word).  
  
?- [\'exemplo2.pl\'].  
true.  
  
?- caminho(a,c).  
true.  
  
?-
```

Programação Lógica – Prolog

E o calcanhar de Aquiles...

```
caminho(X,Z) :- ligacao(X,Z).  
caminho(X,Z) :- caminho(X,Y), ligacao(Y,Z).  
ligacao(a,b).  
ligacao(b,c).
```



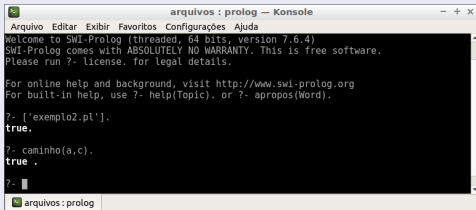
```
arquivos: prolog - Konsole  
Arquivo Editar Exibir Favoritos Configurações Ajuda  
Welcome to SWI-Prolog (threaded, 64 bits, version 7.6.4)  
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.  
Please run ?- license. for legal details.  
  
For online help and background, visit http://www.swi-prolog.org  
For built-in help, use ?- help(Topic). or ?- apropos(Word).  
  
?- [\'exemplo2.pl\'].  
true.  
  
?- caminho(a,c).  
true.  
  
?-
```

```
caminho(X,Z) :- caminho(X,Y), ligacao(Y,Z).  
caminho(X,Z) :- ligacao(X,Z).  
ligacao(a,b).  
ligacao(b,c).
```

Programação Lógica – Prolog

E o calcanhar de Aquiles...

```
caminho(X,Z) :- ligacao(X,Z).  
caminho(X,Z) :- caminho(X,Y), ligacao(Y,Z).  
ligacao(a,b).  
ligacao(b,c).
```



arquivos : prolog — Konsole

Arquivo Editar Exibir Favoritos Configurações Ajuda

Welcome to SWI-Prolog (threaded, 64 bits, version 7.6.4)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit <http://www.swi-prolog.org>
For built-in help, use ?- help(Topic). or ?- apropos(Word).

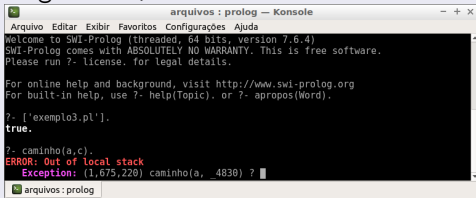
?- ['exemplo2.pl'].
true.

?- caminho(a,c).
true.

?- █

arquivos : prolog

```
caminho(X,Z) :- caminho(X,Y), ligacao(Y,Z).  
caminho(X,Z) :- ligacao(X,Z).  
ligacao(a,b).  
ligacao(b,c).
```



arquivos : prolog — Konsole

Arquivo Editar Exibir Favoritos Configurações Ajuda

Welcome to SWI-Prolog (threaded, 64 bits, version 7.6.4)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit <http://www.swi-prolog.org>
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- ['exemplo3.pl'].
true.

?- caminho(a,c).
ERROR: Out of local stack
Exception: (1,675,220) caminho(a, _4830) ? █

arquivos : prolog

Programação Lógica – Prolog

Outros problemas...

- Prolog não faz o teste de ocorrência

Outros problemas...

- Prolog não faz o teste de ocorrência
 - Quanto unifica uma variável com um termo complexo, não verifica se a variável ocorre dentro do termo ($\{x/f(x)\}$)

Outros problemas...

- Prolog não faz o teste de ocorrência
 - Quanto unifica uma variável com um termo complexo, não verifica se a variável ocorre dentro do termo ($\{x/f(x)\}$)
 - Assim, algumas inferências inconsistentes podem ser feitas

Outros problemas...

- Prolog não faz o teste de ocorrência
 - Quanto unifica uma variável com um termo complexo, não verifica se a variável ocorre dentro do termo ($\{x/f(x)\}$)
 - Assim, algumas inferências inconsistentes podem ser feitas
- Prolog não faz checagem para identificar recursão infinita

Outros problemas...

- Prolog não faz o teste de ocorrência
 - Quanto unifica uma variável com um termo complexo, não verifica se a variável ocorre dentro do termo ($\{x/f(x)\}$)
 - Assim, algumas inferências inconsistentes podem ser feitas
- Prolog não faz checagem para identificar recursão infinita
 - Isso o torna rápido quando são dados os axiomas corretos

Outros problemas...

- Prolog não faz o teste de ocorrência
 - Quanto unifica uma variável com um termo complexo, não verifica se a variável ocorre dentro do termo ($\{x/f(x)\}$)
 - Assim, algumas inferências inconsistentes podem ser feitas
- Prolog não faz checagem para identificar recursão infinita
 - Isso o torna rápido quando são dados os axiomas corretos
 - E incompleto quando são dados os errados

Inferência em LPO

Usos

Inferência em LPO

Usos

- Resolução

Usos

- Resolução
 - Usada por provadores automáticos de teoremas

Usos

- Resolução
 - Usada por provadores automáticos de teoremas
- Forward Chaining:

Usos

- Resolução
 - Usada por provadores automáticos de teoremas
- Forward Chaining:
 - Aplicada a bases dedutivas e sistemas de produção

Usos

- Resolução
 - Usada por provadores automáticos de teoremas
- Forward Chaining:
 - Aplicada a bases dedutivas e sistemas de produção
- Backward Chaining

Usos

- Resolução
 - Usada por provadores automáticos de teoremas
- Forward Chaining:
 - Aplicada a bases dedutivas e sistemas de produção
- Backward Chaining
 - Usada em programação lógica

Inferência em LPO e LProp: Fragilidades

Monotonicidade

- O conjunto de sentenças acarretadas pode apenas aumentar, na medida em que informação é adicionada à base

Monotonicidade

- O conjunto de sentenças acarretadas pode apenas aumentar, na medida em que informação é adicionada à base
- Ou seja, a adição de premissas não podem mudar conclusões antigas

Monotonicidade

- O conjunto de sentenças acarretadas pode apenas aumentar, na medida em que informação é adicionada à base
 - Ou seja, a adição de premissas não podem mudar conclusões antigas
- Isso significa que, se $BC \models \alpha$, então $BC \wedge \beta \models \alpha$, onde β é uma nova asserção na base

Monotonicidade

- O conjunto de sentenças acarretadas pode apenas aumentar, na medida em que informação é adicionada à base
 - Ou seja, a adição de premissas não podem mudar conclusões antigas
- Isso significa que, se $BC \models \alpha$, então $BC \wedge \beta \models \alpha$, onde β é uma nova asserção na base
 - Se antes $BC \models \alpha$, a adição de β a BC não pode mudar isso, e $BC \wedge \beta \models \alpha$

Monotonicidade

- O conjunto de sentenças acarretadas pode apenas aumentar, na medida em que informação é adicionada à base
 - Ou seja, a adição de premissas não podem mudar conclusões antigas
- Isso significa que, se $BC \models \alpha$, então $BC \wedge \beta \models \alpha$, onde β é uma nova asserção na base
 - Se antes $BC \models \alpha$, a adição de β a BC não pode mudar isso, e $BC \wedge \beta \models \alpha$
 - Propriedade chamada de **monotonicidade do acarretamento**

Monotonicidade

- Isso faz com que regras de inferência possam ser aplicadas toda vez que suas premissas forem encontradas na base

Monotonicidade

- Isso faz com que regras de inferência possam ser aplicadas toda vez que suas premissas forem encontradas na base
- Não haverá nada na base para invalidar a conclusão dessa regra

Inferência em LPO e LProp: Fragilidades

Monotonicidade

- Isso faz com que regras de inferência possam ser aplicadas toda vez que suas premissas forem encontradas na base
- Não haverá nada na base para invalidar a conclusão dessa regra
- E qual o problema com isso?

Inferência em LPO e LProp: Fragilidades

Monotonicidade

- Isso faz com que regras de inferência possam ser aplicadas toda vez que suas premissas forem encontradas na base
 - Não haverá nada na base para invalidar a conclusão dessa regra
- E qual o problema com isso?
 - Não captura uma propriedade comum do raciocínio humano
→ a mudança de ideia

Inferência em LPO e LProp: Fragilidades

Monotonicidade

- Isso faz com que regras de inferência possam ser aplicadas toda vez que suas premissas forem encontradas na base
 - Não haverá nada na base para invalidar a conclusão dessa regra
- E qual o problema com isso?
 - Não captura uma propriedade comum do raciocínio humano
→ a mudança de ideia
 - Para esses casos, existe a **lógica não monotônica**

Monotonicidade

- Na lógica não-monotônica, podemos tirar conclusões provisoriamente

Monotonicidade

- Na lógica não-monotônica, podemos tirar conclusões provisoriamente
 - Nos reservando o direito de retratação à luz de nova informação

Monotonicidade

- Na lógica não-monotônica, podemos tirar conclusões provisoriamente
 - Nos reservando o direito de retratação à luz de nova informação
- O conjunto de conclusões apoiadas pela base não necessariamente aumenta

Monotonicidade

- Na lógica não-monotônica, podemos tirar conclusões provisoriamente
 - Nos reservando o direito de retratação à luz de nova informação
- O conjunto de conclusões apoiadas pela base não necessariamente aumenta
 - Pode até reduzir

Inferência em LPO e LProp: Fragilidades

Tempo

- Como lidar com tempo em LPO ou lógica proposicional?

Inferência em LPO e LProp: Fragilidades

Tempo

- Como lidar com tempo em LPO ou lógica proposicional?
- Como fazer uma afirmação valer apenas em um determinado momento?

Tempo

- Como lidar com tempo em LPO ou lógica proposicional?
 - Como fazer uma afirmação valer apenas em um determinado momento?
- Para isso existe a **lógica temporal**

Tempo

- Como lidar com tempo em LPO ou lógica proposicional?
 - Como fazer uma afirmação valer apenas em um determinado momento?
- Para isso existe a **lógica temporal**
 - Assume que fatos valem em instantes particulares, e que esses instantes são ordenados

Inferência em LPO e LProp: Fragilidades

Tempo

- Como lidar com tempo em LPO ou lógica proposicional?
 - Como fazer uma afirmação valer apenas em um determinado momento?
- Para isso existe a **lógica temporal**
 - Assume que fatos valem em instantes particulares, e que esses instantes são ordenados
 - Instantes podem ser pontuais ou intervalos

Inferência em LPO e LProp: Fragilidades

Ordens mais altas

- Em LPO trabalhamos com objetos e suas relações

Inferência em LPO e LProp: Fragilidades

Ordens mais altas

- Em LPO trabalhamos com objetos e suas relações
 - Ex: $\exists x \forall y P(f(x)) \Rightarrow Q(y)$

Inferência em LPO e LProp: Fragilidades

Ordens mais altas

- Em LPO trabalhamos com objetos e suas relações
 - Ex: $\exists x \forall y P(f(x)) \Rightarrow Q(y)$
- E como podemos fazer afirmações sobre propriedades de objetos?

Inferência em LPO e LProp: Fragilidades

Ordens mais altas

- Em LPO trabalhamos com objetos e suas relações
 - Ex: $\exists x \forall y P(f(x)) \Rightarrow Q(y)$
- E como podemos fazer afirmações sobre propriedades de objetos?
 - Ex: Todas as raças de cães vem dos lobos?

Inferência em LPO e LProp: Fragilidades

Ordens mais altas

- Em LPO trabalhamos com objetos e suas relações
 - Ex: $\exists x \forall y P(f(x)) \Rightarrow Q(y)$
- E como podemos fazer afirmações sobre propriedades de objetos?
 - Ex: Todas as raças de cães vem dos lobos?
- Lógica de Segunda Ordem

Inferência em LPO e LProp: Fragilidades

Ordens mais altas

- Em LPO trabalhamos com objetos e suas relações
 - Ex: $\exists x \forall y P(f(x)) \Rightarrow Q(y)$
- E como podemos fazer afirmações sobre propriedades de objetos?
 - Ex: Todas as raças de cães vem dos lobos?
- Lógica de Segunda Ordem
 - Permite expressões como $\forall Q \exists P \exists f \exists x \forall y P(f(x)) \Rightarrow Q(y)$
(quantificador do tipo “para todo conjunto de objetos”)

Inferência em LPO e LProp: Fragilidades

Ordens mais altas

- Em LPO trabalhamos com objetos e suas relações
 - Ex: $\exists x \forall y P(f(x)) \Rightarrow Q(y)$
- E como podemos fazer afirmações sobre propriedades de objetos?
 - Ex: Todas as raças de cães vem dos lobos?
- Lógica de Segunda Ordem
 - Permite expressões como $\forall Q \exists P \exists f \exists x \forall y P(f(x)) \Rightarrow Q(y)$
(quantificador do tipo “para todo conjunto de objetos”)
 - Vê as relações e funções da LPO como objetos em si

Inferência em LPO e LProp: Fragilidades

E muito mais...

Inferência em LPO e LProp: Fragilidades

E muito mais...

- Há ainda as que tratam de incerteza

Inferência em LPO e LProp: Fragilidades

E muito mais...

- Há ainda as que tratam de incerteza
 - Lógica probabilística

Inferência em LPO e LProp: Fragilidades

E muito mais...

- Há ainda as que tratam de incerteza
 - Lógica probabilística
- De inconsistência

Inferência em LPO e LProp: Fragilidades

E muito mais...

- Há ainda as que tratam de incerteza
 - Lógica probabilística
- De inconsistência
 - Lógica paraconsistente

Inferência em LPO e LProp: Fragilidades

E muito mais...

- Há ainda as que tratam de incerteza
 - Lógica probabilística
- De inconsistência
 - Lógica paraconsistente
- De imprecisão

Inferência em LPO e LProp: Fragilidades

E muito mais...

- Há ainda as que tratam de incerteza
 - Lógica probabilística
- De inconsistência
 - Lógica paraconsistente
- De imprecisão
 - Lógica difusa (*Fuzzy*) (veremos mais adiante)

Inferência em LPO e LProp: Fragilidades

E muito mais...

- Há ainda as que tratam de incerteza
 - Lógica probabilística
- De inconsistência
 - Lógica paraconsistente
- De imprecisão
 - Lógica difusa (*Fuzzy*) (veremos mais adiante)
- etc

Considerações Finais

Considerações Finais

- LPO é Completa (Gödel, 1929)

Considerações Finais

- LPO é Completa (Gödel, 1929)
 - Se a base acarreta S , então podemos provar S a partir da base

Considerações Finais

- LPO é Completa (Gödel, 1929)
 - Se a base acarreta S , então podemos provar S a partir da base
- Resolução é um sistema de provas completo para LPO (Robinson, 1965)

Considerações Finais

- LPO é Completa (Gödel, 1929)
 - Se a base acarreta S , então podemos provar S a partir da base
- Resolução é um sistema de provas completo para LPO (Robinson, 1965)
 - Se uma prova existir, podemos encontrá-la

Considerações Finais

- LPO é Completa (Gödel, 1929)
 - Se a base acarreta S , então podemos provar S a partir da base
- Resolução é um sistema de provas completo para LPO (Robinson, 1965)
 - Se uma prova existir, podemos encontrá-la
- E se não existir? O processo pode continuar para sempre

Lógica de Primeira Ordem

Considerações Finais

- LPO é Completa (Gödel, 1929)
 - Se a base acarreta S , então podemos provar S a partir da base
- Resolução é um sistema de provas completo para LPO (Robinson, 1965)
 - Se uma prova existir, podemos encontrá-la
- E se não existir? O processo pode continuar para sempre
 - LPO é semi-decidível: se houver prova, encontramos, se não houver, podemos ficar buscando eternamente

Lógica de Primeira Ordem

E o balde de água fria... Gödel (1931)

E o balde de água fria... Gödel (1931)

- 1º Teorema da Incompletude:

E o balde de água fria... Gödel (1931)

- 1º Teorema da Incompletude:
 - Em qualquer sistema formal consistente F , dentro do qual possa ser feita uma certa quantidade de aritmética, é possível construir uma sentença (chamada **sentença de Gödel**) $G(F)$ tal que:

Lógica de Primeira Ordem

E o balde de água fria... Gödel (1931)

- 1º Teorema da Incompletude:
 - Em qualquer sistema formal consistente F , dentro do qual possa ser feita uma certa quantidade de aritmética, é possível construir uma sentença (chamada **sentença de Gödel**) $G(F)$ tal que:
 - $G(F)$ é uma sentença de F mas não pode ser provada dentro de F

E o balde de água fria... Gödel (1931)

- 1º Teorema da Incompletude:
 - Em qualquer sistema formal consistente F , dentro do qual possa ser feita uma certa quantidade de aritmética, é possível construir uma sentença (chamada **sentença de Gödel**) $G(F)$ tal que:
 - $G(F)$ é uma sentença de F mas não pode ser provada dentro de F
 - Se F for consistente, então $G(F)$ é verdadeira

E o balde de água fria... Gödel (1931)

- 1º Teorema da Incompletude:
 - Em qualquer sistema formal consistente F , dentro do qual possa ser feita uma certa quantidade de aritmética, é possível construir uma sentença (chamada **sentença de Gödel**) $G(F)$ tal que:
 - $G(F)$ é uma sentença de F mas não pode ser provada dentro de F
 - Se F for consistente, então $G(F)$ é verdadeira
 - Ou seja, não há sistema de provas completo e consistente para LPO + aritmética

E o balde de água fria... Gödel (1931)

- 1º Teorema da Incompletude:
 - Em qualquer sistema formal consistente F , dentro do qual possa ser feita uma certa quantidade de aritmética, é possível construir uma sentença (chamada **sentença de Gödel**) $G(F)$ tal que:
 - $G(F)$ é uma sentença de F mas não pode ser provada dentro de F
 - Se F for consistente, então $G(F)$ é verdadeira
 - Ou seja, não há sistema de provas completo e consistente para LPO + aritmética
 - Ou há sentenças que são verdadeiras, mas não prováveis (incompleto), ou que são prováveis, mas não verdadeiras (inconsistente)

E o balde de água fria... Gödel (1931)

- 1º Teorema da Incompletude (cont.):
 - Isso ocorre porque a aritmética nos permite construir sentenças auto-referenciáveis

E o balde de água fria... Gödel (1931)

- 1º Teorema da Incompletude (cont.):
 - Isso ocorre porque a aritmética nos permite construir sentenças auto-referenciáveis
 - P : " P não pode ser provada"

E o balde de água fria... Gödel (1931)

- 1º Teorema da Incompletude (cont.):
 - Isso ocorre porque a aritmética nos permite construir sentenças auto-referenciáveis
 - P : “ P não pode ser provada”
 - Se P for verdadeira, então não podemos prová-la, e o sistema é incompleto

E o balde de água fria... Gödel (1931)

- 1º Teorema da Incompletude (cont.):
 - Isso ocorre porque a aritmética nos permite construir sentenças auto-referenciáveis
 - P : “ P não pode ser provada”
 - Se P for verdadeira, então não podemos prová-la, e o sistema é incompleto
 - Se P for falsa, então P pode ser provada, e acabamos de derivar uma sentença falsa \rightarrow o sistema é inconsistente

E o balde de água fria... Gödel (1931)

- 2º Teorema da Incompletude:

E o balde de água fria... Gödel (1931)

- 2º Teorema da Incompletude:
 - Em qualquer sistema formal consistente F , dentro do qual possa ser feita uma certa quantidade de aritmética, não é possível provar que o sistema em si é consistente

E o balde de água fria... Gödel (1931)

- 2º Teorema da Incompletude:
 - Em qualquer sistema formal consistente F , dentro do qual possa ser feita uma certa quantidade de aritmética, não é possível provar que o sistema em si é consistente
- E, com isso, não podemos provar todos os teoremas da matemática dentro de qualquer sistema de axiomas

E o balde de água fria... Gödel (1931)

- 2º Teorema da Incompletude:
 - Em qualquer sistema formal consistente F , dentro do qual possa ser feita uma certa quantidade de aritmética, não é possível provar que o sistema em si é consistente
- E, com isso, não podemos provar todos os teoremas da matemática dentro de qualquer sistema de axiomas
 - Sequer conseguimos provar que o sistema como um todo é consistente

Referências

- Russell, S.; Norvig P. (2010): Artificial Intelligence: A Modern Approach. Prentice Hall. 3a ed.
 - Slides do livro: aima.eecs.berkeley.edu/slides-pdf/
- Hiž, A. (1957): Inferential Equivalence and Natural Deduction. The Journal of Symbolic Logic, 22(3). pp. 237-240.
- Nilsson, N. J. (1986): Probabilistic Logic. Artificial Intelligence, 28(1). pp. 71-87.
- ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-034Spring-2005/LectureNotes/index.htm
- jmvidal.cse.sc.edu/talks/learningrules/first-orderlogicsdefs.xml

Referências

- www.sciencedirect.com/topics/computer-science/unification-algorithm
- logic.stanford.edu/intrologic/secondary/notes/chapter_12.html
- plato.stanford.edu/entries/logic-nonmonotonic/
- stanford.library.sydney.edu.au/archives/sum2008/entries/logic-nonmonotonic/
- philosophy.fandom.com/wiki/Monotonicity_of_entailment
- plato.stanford.edu/entries/logic-higher-order/
- math.stackexchange.com/questions/1052118/what-are-some-examples-of-third-fourth-or-fifth-order-logic-sentences

Referências

- plato.stanford.edu/entries/logic-paraconsistent/
- www.scientificamerican.com/article/what-is-fuzzy-logic-are-t/
- www.sciencedirect.com/topics/computer-science/fuzzy-logic
- <https://plato.stanford.edu/entries/goedel-incompleteness/>
- <https://www.scientificamerican.com/article/what-is-godels-theorem/>
- https://en.wikipedia.org/wiki/G%C3%B6del%27s_incompleteness_theorems