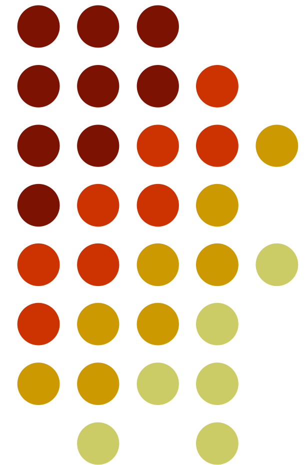
Engenharia do Conhecimento



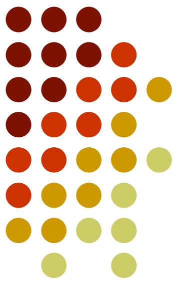
Introdução à Inteligência Artificial

Profa. Josiane

David Poole, Alan Mackworth e Randy Goebel -
"*Computational Intelligence – A logical approach*" - cap. 6

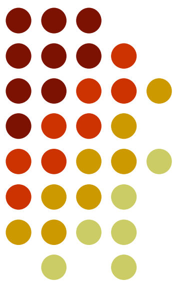
julho/2007

Sistemas Especialistas



- A aquisição de conhecimento é importante neste tipo de sistema
- São Sistemas Baseados em Conhecimento (SBCs)
- A BC é uma representação do conhecimento especialista de um domínio
- Exemplos: sistema de diagnóstico médico, sistema de auxílio a advogados, sistema de projetos de engenharia...

Arquitetura de um Sistema Baseado em Conhecimento (SBC)



Inference Engine

User Interface

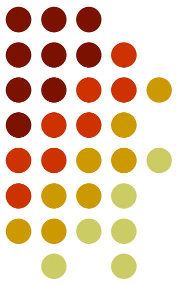
User

Domain Expert

Knowledge Engineer

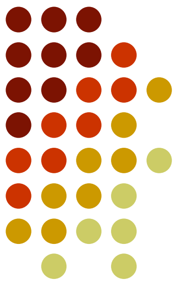
Knowledge Base

Papéis da pessoas envolvidas em um SBC



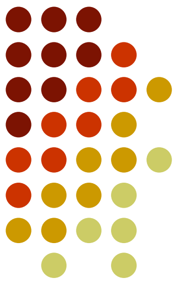
- **Engenheiros de software**
  - Constroem a máquina de inferência e a interface com o usuário
- **Engenheiros de conhecimento**
  - Projetam, constroem e depuram a BC em consulta aos especialistas do domínio
- **Especialistas do domínio**
  - Conhecem o domínio e sabem detalhes de como o sistema especialista funciona
  - Mas tipicamente não têm conhecimento sobre casos particulares
    - Exemplo: sabe sobre doenças, mas não sabe sobre um paciente em particular
  - Têm somente uma visão semântica conhecimento, não tem noção sobre os procedimentos de prova usados pela máquina de inferência

Papéis da pessoas envolvidas em um SBC



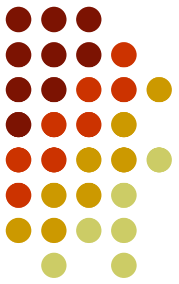
- **Usuários:**
  - Aqueles que tem a necessidade da especialidade
  - Têm informações sobre problemas individuais que podem ser apresentados para o SBC
  - Não conhecem o domínio de especialidade da BC
  - Tipicamente não entendem a estrutura interna do sistema
    - Não sabem qual informação é necessária para o sistema
    - Uma interface simples e natural deve ser fornecida
- Os papéis podem se sobrepor
  - Especialistas pode ser usuários
  - Engenheiros de conhecimento podem ser especialista do domínio

Meta-interpretadores



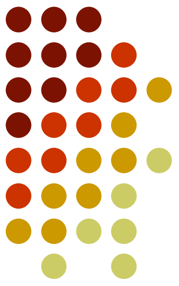
- Para construir um interpretador para uma linguagem, precisamos distinguir:
  - A linguagem base (linguagem objeto)
    - A linguagem do SRR sendo implementado
  - A metalinguagem
    - A linguagem usada para implementar o sistema
  - Podem até mesmo ser as mesmas
    - Inicialmente elas serão as mesmas = datalog

Implementando a linguagem base



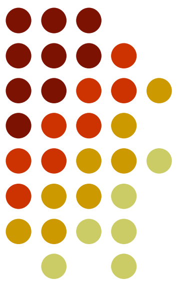
- Precisamos representar construtores do nível base na metalinguagem
- Representamos termos, átomos e corpos de cláusulas, como termos no meta nível
- Representamos cláusulas no nível base como fatos no meta nível
- Variáveis no nível base são representadas como variáveis no meta nível

Representando construções no nível base



- O átomo $p(t_1, t_2, ... , t_n)$ no nível base é representado como o termo $p(t_1, t_2, ... , t_n)$ no meta nível
- O termo *oand*(e_1, e_2) no meta nível denota a conjunção dos corpos de cláusulas e_1 *e* e_2 no nível base
- A constante true no meta nível denota o corpo vazio no nível base
- O átomo *clause*(*h, b*) é verdadeiro se “*h* se *b*” é uma cláusula no nível base da BC

Exemplo de representação



- Cláusulas no nível base

$$connected_to(l_1, w_0).$$

$$connected_to(w_0, w_1) \leftarrow up(s_2).$$

$$lit(L) \leftarrow light(L) \wedge ok(L) \wedge live(L).$$

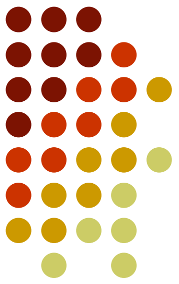
- Podem ser representadas como fatos no meta nível

$$clause(connected_to(l_1, w_0), true).$$

$$clause(connected_to(w_0, w_1), up(s_2)).$$

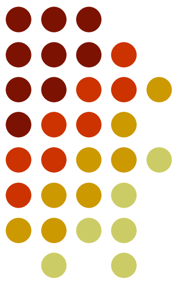
$$clause(lit(L), oand(light(L), oand(ok(L), live(L)))).$$

Melhorando a representação



- Usar a função símbolo “&” ao invés de *oand*
  - Ao invés de escrever *oand*(e_1, e_2), escrevemos e_1 & e_2
- Usar um símbolo de predicado infixado “⇐” no meta nível
  - Ao invés de escrever *clause*(*h*, *b*), escrevemos h ⇐ b
- A cláusula “$h \leftarrow a_1 \wedge a_2 \wedge \ldots \wedge a_n$” no nível base é representada como o átomo $h \Leftarrow a_1 \& a_2 \& \ldots \& a_n$ no meta nível

Exemplo de representação



- Cláusulas no nível base

$$connected_to(l_1, w_0).$$

$$connected_to(w_0, w_1) \leftarrow up(s_2).$$

$$lit(L) \leftarrow light(L) \wedge ok(L) \wedge live(L).$$

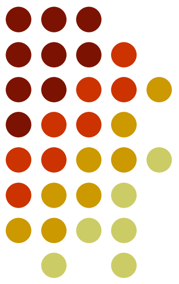
- Podem ser representadas como fatos no meta nível

$$connected_to(l_1, w_0) \Leftarrow true.$$

$$connected_to(w_0, w_1) \Leftarrow up(s_2).$$

$$lit(L) \Leftarrow light(L) \; \& \; ok(L) \; \& \; live(L).$$

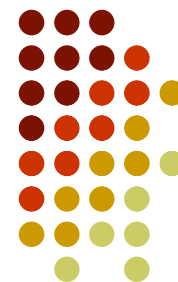
Um meta-interpretador simples



- *Prove*(*G*) é verdadeiro quando o corpo *G* no nível base é uma consequência lógica da base de conhecimento no nível base

$$
\begin{aligned}
&prove(true).\\
&prove((A \mathbin{\&} B)) \leftarrow\\
&\qquad prove(A) \wedge\\
&\qquad prove(B).\\
&prove(H) \leftarrow\\
&\qquad (H \Leftarrow B) \wedge\\
&\qquad prove(B).
\end{aligned}
$$

Exemplo de BC no nível base



$$live(W) \Leftarrow$$

$$connected_to(W, W_1) \;\&$$

$$live(W_1).$$

$$live(outside) \Leftarrow true.$$

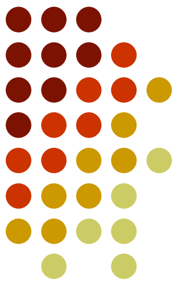
$$connected_to(w_6, w_5) \Leftarrow ok(cb_2).$$

$$connected_to(w_5, outside) \Leftarrow true.$$

$$ok(cb_2) \Leftarrow true.$$

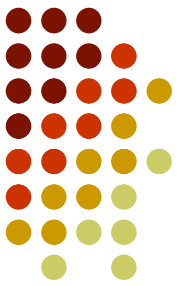
$$?prove(live(w_6)).$$

Expandindo a linguagem base



- Podemos modificar a linguagem base sem modificar a metalinguagem
  - Modificando o meta-interpretador
- Em todos os sistemas práticos, nem todo predicado é definido pelas cláusulas
  - Alguns predicados, não queremos axiomatizar, mas chamar o sistema básico diretamente (procedimentos “embutidos”)
  - Exemplo: *N is E* é verdadeiro se a expressão E for avaliada como o número N
- Podemos expandir a linguagem base para permitir disjunção (;) no corpo da cláusula

Meta-interpretador expandido



$$prove(true).$$

$$prove((A \,\&\, B)) \leftarrow$$
$$\quad prove(A) \wedge prove(B).$$

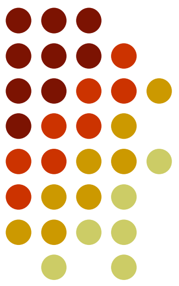
$$prove((A; B)) \leftarrow prove(A).$$

$$prove((A; B)) \leftarrow prove(B).$$

$$prove((N \text{ is } E)) \leftarrow$$
$$\quad N \text{ is } E.$$

$$prove(H) \leftarrow$$
$$\quad (H \Leftarrow B) \wedge prove(B).$$

Busca em profundidade Limitada



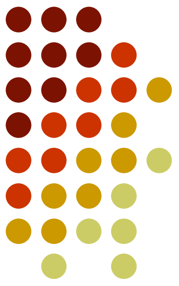
- Adicionar condições reduz o que pode ser provado
- *bprove*(*G*, *D*) é verdade se G pode ser provado com uma árvore de prova de profundidade menor ou igual ao número *D*

$$bprove(true, D).$$

$$bprove((A \,\&\, B), D) \leftarrow$$
$$\quad bprove(A, D) \wedge bprove(B, D).$$

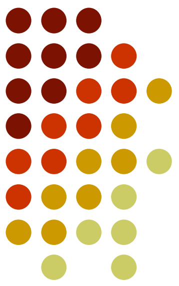
$$bprove(H, D) \leftarrow$$
$$\quad D \geq 0 \wedge D_1 \text{ is } D - 1 \wedge$$
$$\quad (H \Leftarrow B) \wedge bprove(B, D_1).$$

Objetivos adiados



- Alguns objetivos ao invés de serem provados podem ser coletados em uma lista
  - Adiar os subobjetivos com variáveis, esperando que chamadas subsequentes irão associá-las a constantes
  - Adiar as suposições, coletar suposições que são necessárias para provar o objetivo (exemplo)
  - Criar novas regras que deixam de fora (omitem) passos intermediários

Um meta-interpretador que adia



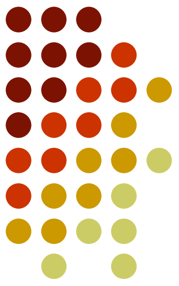
$$dprove(true, D, D).$$

$$dprove((A \,\&\, B), D_1, D_3) \leftarrow$$
$$dprove(A, D_1, D_2) \wedge dprove(B, D_2, D_3).$$

$$dprove(G, D, [G|D]) \leftarrow delay(G).$$

$$dprove(H, D_1, D_2) \leftarrow$$
$$(H \Leftarrow B) \wedge dprove(B, D_1, D_2).$$

Exemplo de BC no nível base



$$live(W) \Leftarrow$$

$$connected_to(W, W_1) \ \&$$

$$live(W_1).$$

$$live(outside) \Leftarrow true.$$

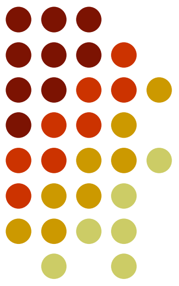
$$connected_to(w_6, w_5) \Leftarrow ok(cb_2).$$

$$connected_to(w_5, outside) \Leftarrow ok(outside_connection).$$

$$delay(ok(X)).$$

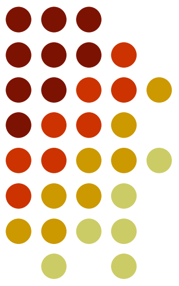
$$?dprove(live(w_6), [\,], D).$$

Perguntando ao usuário



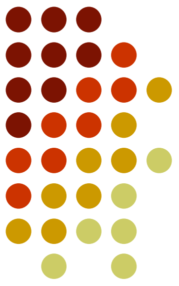
- Como os usuários podem fornecer conhecimento quando:
  - Eles não conhecem o funcionamento interno do sistema
  - Eles não são especialistas no domínio
  - Eles não sabem qual informação é relevante
  - Eles não sabem a sintaxe do sistema
  - Mas eles tem informações essenciais sobre um caso de interesse particular?

Perguntando ao usuário

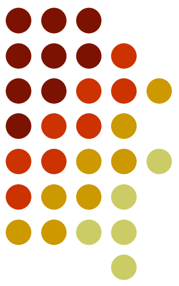


- O sistema pode determinar qual informação é relevante e perguntar ao usuário
- Uma derivação top-down pode determinar qual informação é relevante
- Existem três tipos de objetivos:
  - Objetivos para os quais não é esperado que o usuário saiba a resposta, desta forma o sistema nunca pergunta
  - Objetivos para os quais os usuários devem saber a resposta, e para os quais ele ainda não tenha fornecido uma resposta
  - Objetivos para os quais o usuário já tenha fornecido a resposta

Questões yes/no



- Mais simples forma de pergunta - requerem repostas sim ou não
- O usuário somente é questionado se
  - Existe uma forma de perguntar a questão (*question is askable*)
  - O usuário não respondeu a questão anteriormente
- Quando o usuário já respondeu a questão anteriormente a resposta deve ser recuperada
  - Assumindo que existe uma base de dados externa que armazena respostas



Exemplo sistema elétrico

outside power

cb_1 w_5 s_1 w_1 s_2 w_2 w_3 cb_2 s_3 w_0 w_4 w_6 l_1 l_2 p_2 p_1

circuit breaker

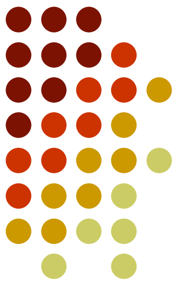
off
switch
on

two-way switch

light

power outlet

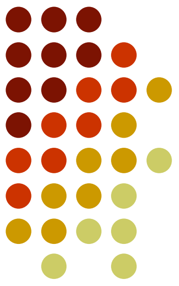
Exemplo sistema elétrico



lit(*L*) <= *ligth*(*L*) & *ok*(*L*) & *live*(*L*).
live(*W*) <= *connected_to*(*W*, W_1) & *live*(W_1).
live(*outside*) <= *true*.
ligth(l_1) <= *true*.
ligth(l_2) <= *true*.
***askable*(*up*(*S*)).**
***askable*(*down*(*S*)).**
connected_to(l_1,w_0) <= *true*.
connected_to(w_0,w_1) <= *up*(s_2).
connected_to(w_0,w_2) <= *down*(s_2).
connected_to(w_1,w_3) <= *up*(s_1).
connected_to(w_2,w_3) <= *down*(s_1).
connected_to(w_4,w_3) <= *up*(s_3).

connected_to(l_2,w_4) <= *true*.
connected_to(p_1,w_3) <= *true*.
connected_to(w_3,w_5) <= *ok*(cb_1).
connected_to(p_2,w_6) <= *true*.
connected_to(w_6,w_5) <= *ok*(cb_2).
connected_to(w_5,*outside*) <= *true*.
ok(l_1) <= *true*.
ok(l_2) <= *true*.
ok(cb_1) <= *true*.
ok(cb_2) <= *true*.

Meta-interpretador que pergunta ao usuário (pseudocódigo)



- *aprove*(*G*) é verdadeiro se *G* é uma consequência lógica da BC no nível base e respostas sim/não são fornecidas pelo usuário

$$aprove(true).$$

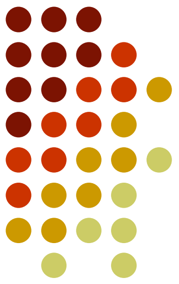
$$aprove((A \,\&\, B)) \leftarrow aprove(A) \wedge aprove(B).$$

$$aprove(H) \leftarrow askable(H) \wedge answered(H, yes).$$

$$aprove(H) \leftarrow$$
$$\quad askable(H) \wedge unanswered(H) \wedge ask(H, Ans) \wedge$$
$$\quad record(answered(H, Ans)) \wedge Ans = yes.$$

$$aprove(H) \leftarrow (H \Leftarrow B) \wedge aprove(B).$$

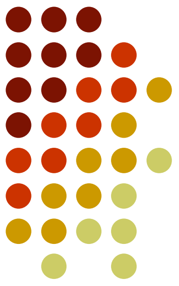
Exemplo de perguntas yes/no



- *?aprove(lit(L)).*

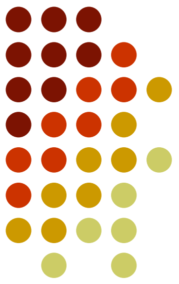
- Diálogo:
  - Sistema: *up*(s_2) é verdadeiro?
  - Usuário: sim.
  - Sistema: *up*(s_1) é verdadeiro?
  - Usuário: não.
  - Sistema: *down*(s_2) é verdadeiro?
  - Usuário: não.
  - Sistema: *up*(s_3) é verdadeiro?
  - Usuário: sim.
  - Resposta: $L = l_2$

Relações Funcionais



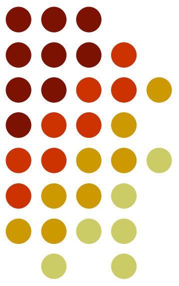
- Não gostaríamos de ter que perguntar *?idade*(*fred*, *0*), *?idade*(*fred*, *1*), *?idade*(*fred*, *2*), ...
- Provavelmente queremos perguntar pela idade de Fred somente uma vez
  - Ter sucesso para perguntas com aquela idade e falhar para outras
- Idade é uma **relação funcional**
- Uma relação *r*(*X*, *Y*) é funcional se, para todo *X* existe um único *Y* para o qual *r*(*X*, *Y*) é verdadeiro

Conseguindo informação do usuário



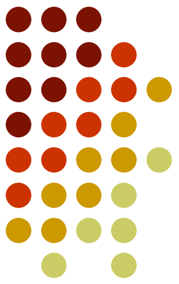
- O usuário pode não conhecer o vocabulário que é esperado pelo engenheiro de conhecimento
- Soluções para o problema:
  - O projetista do sistema fornece um menu de itens do qual o usuário deve escolher o que melhor se encaixa
  - O usuário pode fornecer respostas de forma livre
    - O sistema necessita de um grande dicionário para mapear as respostas para uma forma interna que o sistema entenda

Questões mais gerais



- Exemplo: para o subobjetivo $p(a, X, f(Z))$ o usuário pode ser perguntado por:
  - Para quais X, Z o predicado $p(a, X, f(Z))$ é verdadeiro?
  - Exemplo: para quais Est, Curso o predicado matriculado(Est,Curso) é verdadeiro?
- Usuários devem enumerar todas as instâncias que são verdadeiras?
- Ou devem devem dar as instâncias uma de cada vez conforme o sistema for perguntando por elas? (mais natural)

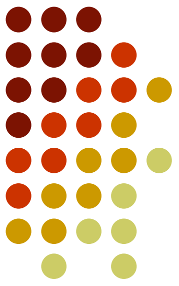
Quando repetir as perguntas?



- Não pergunte por uma questão que é mais específica do que uma pergunta para a qual
  - Uma resposta positiva já foi dada
  - O usuário já respondeu não
  - Exemplo:

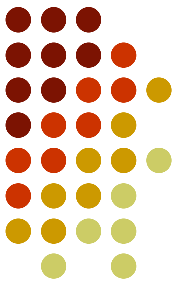
Query	Ask?	Response
$?p(X)$	yes	$p(f(Z))$
$?p(f(c))$	no	
$?p(a)$	yes	yes
$?p(X)$	yes	no
$?p(c)$	no	

Adiando perguntas ao usuário



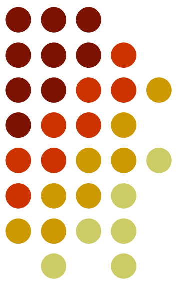
- O sistema deve fazer uma pergunta tão breve quanto ela é encontrada? Ou
- Ele deve adiar o objetivo até que mais variáveis tenham unificado?
- Exemplo: considere a pergunta *?p*(*X*) *& q*(*X*), onde *askable*(*p*(*x*))
  - Se *p*(*X*) tem sucesso para muitas instâncias de *X* e *q*(*X*) tem sucesso para algumas, é melhor adiar a pergunta por *p*(*X*)
  - Se *p*(*X*) tem sucesso para algumas instâncias de *X* e *q*(*X*) tem sucesso para muitas instâncias, então não adie a pergunta

Explicação



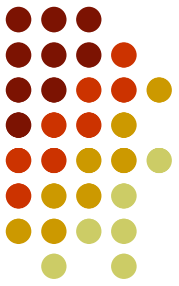
- O sistema deve ser capaz de justificar suas respostas, principalmente quando ele estiver assessorando um humano
- As mesmas características podem ser usadas para explicação e para depurar a BC
- Três mecanismos principais:
  - HOW: pergunta como um objetivo foi derivado
  - WHY: pergunta porque um subobjetivo está sendo provado

Como o sistema prova um objetivo?



- Se g é derivado, deve existir uma instância de uma regra $g \Leftarrow a_1 \& \ldots \& a_k$. Onde cada a_i é derivado
- Se o usuário pergurta HOW g, o sistema pode mostrar a regra
- O usuário pode então perguntar HOW i, para dar a regra que foi usada para provar a_i
- O comando HOW se move para baixo na árvore de prova

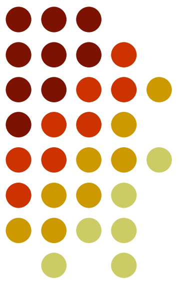
Exemplo sistema elétrico



lit(*L*) <= *ligth*(*L*) & *ok*(*L*) & *live*(*L*).
live(*W*) <= *connected_to*(*W*, W_1) & *live*(W_1).
live(*outside*) <= *true*.
ligth(l_1) <= *true*.
ligth(l_2) <= *true*.
up(s_2).
up(s_3).
down(s_1).
connected_to(l_1,w_0) <= *true*.
connected_to(w_0,w_1) <= *up*(s_2).
connected_to(w_0,w_2) <= *down*(s_2).
connected_to(w_1,w_3) <= *up*(s_1).
connected_to(w_2,w_3) <= *down*(s_1).
connected_to(w_4,w_3) <= *up*(s_3).

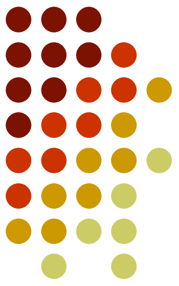
connected_to(l_2,w_4) <= *true*.
connected_to(p_1,w_3) <= *true*.
connected_to(w_3,w_5) <= *ok*(cb_1).
connected_to(p_2,w_6) <= *true*.
connected_to(w_6,w_5) <= *ok*(cb_2).
connected_to(w_5,*outside*) <= *true*.
ok(l_1) <= *true*.
ok(l_2) <= *true*.
ok(cb_1) <= *true*.
ok(cb_2) <= *true*.

HOW (exemplo)



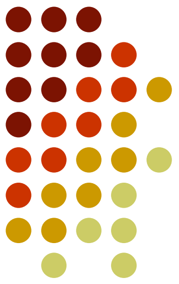
- A resposta para a pergunta ?*lit*(*L*) é $L=l_2$. O usuário pode perguntar HOW
- Sistema: $lit(l_2) <= ligth(l_2)\ \&\ ok(l_2)\ \&\ live(l_2)$.
- Usuário: HOW 3
- Sistema: $live(l_2) <= connected_to(l_2, w_4)\ \&\ live(w_4)$.
- Usuário: HOW 2
- Sistema: $live(w_4) <= connected_to(w_4, w_3)\ \&\ live(w_3)$.
- Usuário: HOW 1
- Sistema: $connected_to(w_4,w_3) <= up(s_3)$.
- Usuário: HOW 1
- Sistema: fato.

Porque o sistema pergunta uma questão?



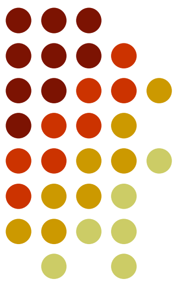
- Saber porque uma questão foi perguntada aumenta a confiança do usuário que sistema está trabalhando adequadamente
- Ajuda o engenheiro de conhecimento a otimizar as questões que são feitas ao usuário
- Uma questão irrelevante pode ser um sintoma de um problema mais grave
- O usuário pode aprender algumas coisas do sistema perguntando porque o sistema está fazendo alguma coisa

Perguntas WHY



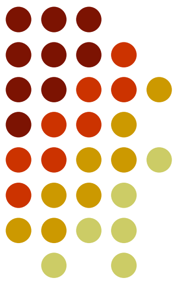
- Quando o sistema pergunta uma questão g, o usuário pode responder com uma pergunta WHY
- O sistema mostra a instância da regra que está sendo tentada para provar (*h*) que precisa de *g*
  - $h \Leftarrow \ldots \;\&\; g \;\&\; \ldots$
- Quando o usuário perguntar porque novamente, o sistema mostra porque *h* está sendo provada
- O comando WHY se move para cima na árvore de prova

WHY (exemplo)



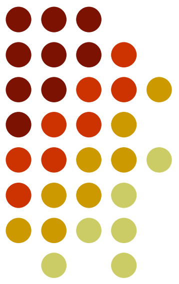
- O usuário pergunta: *?aprove*(*lit*(*L*)).
- O sistema: $up(s_2)$ é verdadeiro?
- O usuário: WHY.
- O sistema: para usar a regra $connected_to(w_0,w_1) <= up(s_2)$.
- O usuário: WHY.
- O sistema: para usar a regra $live(w_0) <= connected_to(w_0, w_1)$ & $live(w_1)$.
- O usuário: WHY.
- O sistema: para usar a regra $live(l_1) <= connected_to(l_1, w_0)$ & $live(w_0)$.
- O usuário: WHY.
- O sistema: para usar a regra $lit(l_1) <= ligth(l_1)$ & $ok(l_1)$ & $live(l_1)$.
- O usuário: WHY.
- O sistema: foi a questão inicial *?aprove*(*lit*(*L*)) perguntada pelo usuário

Depurando a BC



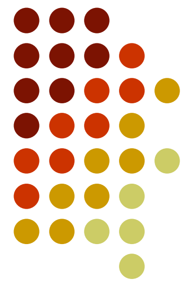
- Existem 4 tipos de problemas não sintáticos que podem aparecer em um sistema baseado em regras:
  - Uma resposta incorreta é produzida
    - Algum átomo considerado falso na interpretação pretendida foi derivado
  - Alguma resposta não foi derivada
    - Alguma prova falhou quando deveria ter tido sucesso
    - Algum átomo verdadeiro não foi derivado
  - O programa entra em um loop infinito
  - O sistema pergunta questões irrelevantes

Depurando respostas incorretas



- Uma resposta incorreta é uma resposta que foi derivada, mas é falsa na interpretação pretendida
- Se g é falsa na interpretação pretendida existe uma prova para g usando $g \Leftarrow a_1 \ \& \ \ldots \ \& \ a_k$.
  - Ou algum a_i é falso => precisamos depurá-lo
  - Ou todos os a_i são verdadeiros => existe um erro na regra

Exemplo sistema elétrico



outside power

cb_1

s_1

w_5

w_1

s_2

w_2

w_3

cb_2

s_3

w_0

w_6

w_4

l_1

l_2

p_2

p_1

circuit breaker

off

switch

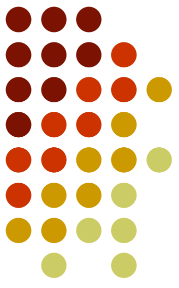
on

two-way switch

light

power outlet

Exemplo sistema elétrico



lit(*L*) <= *ligth*(*L*) & *ok*(*L*) & *live*(*L*).

live(*W*) <= *connected_to*(*W*, W_1) & *live*(W_1).

live(*outside*) <= *true*.

ligth(l_1) <= *true*.

ligth(l_2) <= *true*.

up(s_2).

up(s_3).

down(s_1).

connected_to(l_1,w_0) <= *true*.

connected_to(w_0,w_1) <= *up*(s_2).

connected_to(w_0,w_2) <= *down*(s_2).

connected_to(w_1,w_3) <= up(s_3).

connected_to(w_2,w_3) <= *down*(s_1).

connected_to(w_4,w_3) <= *up*(s_3).

connected_to(l_2,w_4) <= *true*.

connected_to(p_1,w_3) <= *true*.

connected_to(w_3,w_5) <= *ok*(cb_1).

connected_to(p_2,w_6) <= *true*.

connected_to(w_6,w_5) <= *ok*(cb_2).

connected_to(w_5,*outside*) <= *true*.

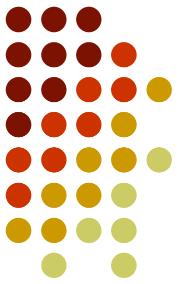
ok(l_1) <= *true*.

ok(l_2) <= *true*.

ok(cb_1) <= *true*.

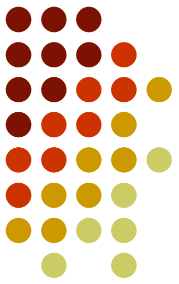
ok(cb_2) <= *true*.

Exemplo respostas incorretas



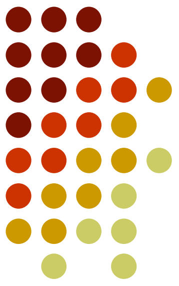
- Suponha que o EC ou o usuário erroneamente colocaram no sistema que w_1 conectado a w_3 depende do estado de s_3 ao invés de s_1
- Lit(l_1) pode ser derivada, quando ela é falsa na interpretação
- *lit*(l_1) <= ***ligth***(***l_1***) & ***ok***(***l_1***) & *live*(l_1). => HOW 3
- *live*(l_1) <= ***connected_to***(***l_1, w_0***) & *live*(w_0). => HOW 2
- *live*(w_0) <= ***connected_to***(***w_0, w_1***) & *live*(w_1). => HOW 2
- *live*(w_1) <= *connected_to*(w_1, w_3) & ***live***(***w_3***). => HOW 1
- *connected_to*(w_1,w_3) <= ***up***(***s_3***).
- A regra está errada.

Depurando respostas faltantes



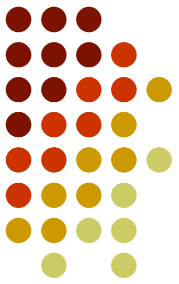
- *g* falha quando deveria ter sucedido
  - Ou existe um átomo na regra que tem sucesso com uma resposta errada => usar HOW para depurá-lo
  - Ou existe um átomo no corpo da regra que falhou quando deveria ter sucesso => usar WHY para depurá-lo
  - Ou existe uma regra faltando para *g*

Exemplo respostas faltantes



- Suponha que s_2 agora está para baixo. A axiomatização que temos falha ao prova lit(l_1), mas deveria ter sucesso:
- *lit*(l_1) <= ***ligth***(***l_1***) & ***ok***(***l_1***) & *live*(l_1). => *live*(l_1) deveria ter sucesso
- *live*(l_1) <= ***connected_to***(***l_1, w_0***) & *live*(w_0). => *live*(w_0) deveria ter sucesso
- *live*(w_0) <= ***connected_to***(***w_0, w_1***) & *live*(w_1). => *live*(w_1) deveria ter falhado mesmo
- *connected_to*(w_0, w_1) teve sucesso e deveria ter falhado
  - Problema *up*(s_2), mas mesmo depois de corrigido *lit*(l_1) continua falhando
- *connected_to*(w_0, w_2) falhou mas deveria ter tido sucesso
  - Problema *down*(s_2), falha mas deveria ter sucesso
  - Não existe nenhuma regra que unifica com *down*(s_2) => erro encontrado.

Depurando loops infinitos



- Não existe uma forma automática para depurar todos os erros, mas muitos deles podem ser detectados
  - Se o subobjetivo é idêntico ao seu ancestral na árvore de prova
  - Se a recursão não está diminuindo a ordem do subobjetivo em cada repetição