RACIOCÍNIO ABDUTIVO EM PROLOG

Silvio do Lago Pereira
Prof. Dr. do Departamento de Tecnologia da Informação – FATEC-SP
slago@ime.usp.br

Resumo

Abdução é uma forma de raciocínio em que uma hipótese é adotada como uma possível explicação para um fato observado, de acordo com leis conhecidas. Em inteligência artificial (IA), a abdução é especialmente útil para tratar problemas que requerem raciocínio de efeitos para causas como, por exemplo, planejamento e diagnóstico. Apesar disto, porém, o Prolog, uma das principais ferramentas para desenvolvimento de aplicações em IA, não é capaz de realizar este tipo de raciocínio. Este artigo mostra como estender a capacidade do sistema Prolog, por meio da implementação de um meta-interpretador, de modo que, além do raciocínio dedutivo nativo, ele também seja capaz de raciocínio abdutivo.

1. Introdução

Abdução, originalmente introduzida por Pierce [1], é uma forma de raciocínio em que uma *hipótese* é adotada como uma possível explicação para um fato observado, de acordo com leis conhecidas. Mais precisamente, se um fato β é observado e sabe-se que $\alpha \rightarrow \beta$, então α pode ser adotada como uma possível explicação para β. Evidentemente, a abdução é uma regra de inferência *fraca*, pois não pode garantir que as explicações abduzidas sejam verdadeiras, mas apenas que sejam *plausíveis*.

Em inteligência artificial (IA) [2], a abdução é especialmente útil para tratar problemas que requerem raciocínio de efeitos para causas [3], como é o caso dos problemas de planejamento, interpretação de linguagem natural, revisão de crenças e diagnóstico.

Por exemplo, a partir da base de conhecimento composta pelas sentenças (1)-(4) a seguir, o raciocínio abdutivo permite inferir as sentenças "*choveu*" e "*o irrigador foi ligado*" como duas explicações plausíveis para o fato "*a grama está escorregadia*".

Se *a grama é molhada* então *ela fica escorregadia*. (1)
Se *chove* então *a rua fica molhada*. (2)
Se *chove* então *a grama fica molhada*. (3)
Se *o irrigador é ligado* então *a grama fica molhada*. (4)

De fato, a abdução é uma forma de raciocínio *não-monotônico*, pois explicações que são consistentes com um determinado estado da base de conhecimento podem se tornar inconsistentes em face de novas informações disponíveis. Por exemplo, se a sentença

A rua não está molhada. (5)

for adicionada à base de conhecimento, então a única explicação plausível para o fato "*a grama está molhada*" passa a ser "*o irrigador foi ligado*".

Outra forma de raciocínio bastante conhecida é a *dedução*, isto é, de $\alpha \rightarrow \beta$ e α conclui-se β. A dedução é uma regra de inferência *forte*, pois garante que a conclusão obtida seja uma consequência lógica das premissas. De fato, esta é a regra de inferência implementada no núcleo do sistema Prolog [4], uma das ferramentas mais usadas em aplicações de inteligência artificial (IA).

Essencialmente, a programação em Prolog consiste em declarar *regras* e *fatos* conhecidos em um determinado contexto de discurso e, em seguida, consultar o sistema para determinar a veracidade de alguns fatos. Por exemplo, em Prolog, a regra "*a grama fica escorregadia se ela está molhada*" pode ser declarada como:

```
grama_escorregadia :- grama_molhada.          (6)
```

Analogamente, a regra "*A grama fica molhada se chove*" pode ser declarada como:

```
grama_molhada :- chove.                       (7)
```

Finalmente, o fato "*chove*" pode ser declarado como:

```
chove.                                        (8)
```

Uma vez que as sentenças (6)-(8) tenham sido fornecidas ao Prolog, este pode ser consultado como a seguir:

```
?- grama_escorregadia.
true

?- not(grama_molhada).
false
```

Porém, sem a sentença (8), o sistema não é capaz de concluir que "*a grama está escorregadia*" (ou de encontrar uma explicação plausível para este fato). Isto acontece porque a busca de uma explicação plausível para um fato observado requer a capacidade de raciocínio abdutivo; mas o sistema Prolog só é capaz de realizar raciocínio dedutivo.

Assim, o objetivo deste artigo é mostrar como estender o mecanismo de inferência nativo do Prolog, de modo que, além do raciocínio dedutivo, ele também seja dotado de raciocínio abdutivo. Para tanto, é definida uma nova linguagem de representação de conhecimento (que permite o uso de predicados *abdutíveis* para representar conhecimento incompleto) e implementado um meta-interpretador capaz de manipular conhecimento expresso nesta nova linguagem.

O restante deste artigo está organizado da seguinte forma: a Seção 2 introduz os fundamentos lógicos da linguagem Prolog; a Seção 3 apresenta uma caracterização formal do raciocínio abdutivo e uma forma de adaptar o mecanismo básico de inferência do Prolog para realizar este tipo de raciocínio; a Seção 4 apresenta os fundamentos, bem como os detalhes de implementação, de um meta-intepretador abdutivo em Prolog e também alguns exemplos de seu uso; e, finalmente, a Seção 5, apresenta as conclusões do trabalho.

2. Fundamentos Lógicos do Prolog

Esta seção apresenta os fundamentos lógicos da linguagem Prolog.

2.1. Programação em lógica

Programação em lógica é um formalismo para programação declarativa que é baseado em lógica de predicados de primeira ordem. A seguir, são apresentadas a terminologia e notação próprias deste formalismo [5].

Uma *constante* ou *predicado* é uma cadeia alfanumérica com inicial minúscula e uma *variável* é uma cadeia alfanumérica com inicial maiúscula. As constantes `true` e `false` denotam, respectivamente, *verdadeiro* e *falso*. Um *termo* é uma constante ou uma variável. Um *átomo* é um predicado seguido de uma tupla de termos. Um *literal* é um átomo (*literal positivo*) ou a negação de um átomo (*literal negativo*). Uma *cláusula* é uma sentença da forma $\varphi_1 \vee \ldots \vee \varphi_n$, em que cada φ_i é um literal. Uma *cláusula de Horn* é uma cláusula com no máximo um literal positivo. Uma *cláusula definida* é uma cláusula de Horn com exatamente um literal positivo. Cláusulas definidas são normalmente escritas como $\alpha \leftarrow \beta_1, \ldots, \beta_n$; neste caso, α é a *cabeça* e $\beta_1, \ldots, \beta_n$ é o *corpo* da cláusula. Uma cláusula da forma $\alpha \leftarrow$ `true` (abreviada por $\alpha \leftarrow$) é denominada *unitária* ou *fato*. Uma cláusula da forma `false` $\leftarrow \beta_1, \ldots, \beta_n$ (abreviada por $\leftarrow \beta_1, \ldots, \beta_n$) é denominada *consulta*. E, finalmente, uma cláusula da forma `false` $\leftarrow$ `true` (abreviada por $\leftarrow$) é denominada *cláusula vazia* e denota uma contradição.

Há dois significados para cláusulas da forma $\alpha \leftarrow \beta_1, \ldots, \beta_n$ (também denominada *regra*):

- *declarativo*: α é verdade se $\beta_1, \ldots, \beta_n$ também são verdade.
- *procedimental*: para solucionar α, solucione $\beta_1, \ldots, \beta_n$.

Entre eles, é o significado procedimental que permite que a lógica formal seja usada como uma linguagem de programação, distinguindo, assim, a lógica de predicados da programação em lógica.

2.2. SLD-Resolução

Um programa lógico é um conjunto finito e não-vazio de cláusulas definidas, através do qual é possível realizar computações combinando-se dois mecanismos básicos: *unificação* e *resolução*. Esta forma de computação é, na verdade, um método específico de prova correto e completo [5].

Formalmente, uma substituição é um mapeamento finito de variáveis em termos, denotado por $\theta = \{X_1 = t_1, \ldots, X_n = t_n\}$, onde X_i é uma variável e t_i é um termo, para $i = 1, \ldots, n$. Substituições são aplicáveis a expressões da linguagem. Se E é uma expressão, então $E\theta$ denota a expressão que se obtém substituindo-se, simultaneamente, cada ocorrência de X_i em E pelo termo t_i, de acordo com θ. A expressão $E\theta$ é uma *instância* de E. Uma cláusula C_1 é uma *variante* de uma cláusula C_2 se existem substituições θ_1 e θ_2 tais que $C_1\theta_1$ é idêntica a C_2 e $C_2\theta_2$ é idêntica a C_1. Essencialmente, uma variante de uma cláusula é essa mesma cláusula com algumas variáveis renomeadas. Dois átomos A_1 e A_2 são *unificáveis* se existe uma substituição θ tal que $A_1\theta$ é idêntico a $A_2\theta$; neste caso, θ é um *unificador* para estes átomos. Um unificador θ_1 é um *unificador mais geral* (*umg*) para A_1 e A_2 se para todo unificador θ_2 de A_1 e A_2 existe uma substituição θ tal que $A_1\theta_1\theta$ é idêntico a $A_1\theta_2$.

Seja Σ um programa lógico. Sejam $C_0 = \leftarrow \alpha_1, \ldots, \alpha_n$ uma consulta e $V_0 = \alpha \leftarrow \beta_1, \ldots, \beta_n$ uma variante de uma das cláusulas de Σ, sem variáveis em comum com C_0. Sem perda de generalidade, suponha que α e α_1 sejam unificáveis por um umg θ. Então, resolvendo-se estas duas cláusulas e obtém-se $C_1 = \leftarrow \beta_1, \ldots, \beta_n, \alpha_2, \ldots, \alpha_n$ como *resolvente*. Iterando-se este passo, ou seja, repetindo o processo com a resolvente C_1 e uma nova variante V_1 de umas das cláusulas do programa Σ, obtém-se uma sequência de resolventes denominada *derivação*.

Uma *SLD-resolução* é uma derivação maximal. Se uma tal derivação termina com a cláusula vazia, temos uma *SLD-refutação* [5].

Por exemplo, a partir do programa lógico Σ_1 definido a seguir:

$\Sigma_1 = \{$*pai*(*adão*,*caim*) $\leftarrow$;
pai(*adão*,*abel*) $\leftarrow$;
pai(*adão*,*seth*) $\leftarrow$;
pai(*seth*,*enos*) $\leftarrow$;
avô(*X*,*Z*) $\leftarrow$ *pai*(*X*,*Y*), *avô*(*Y*,*Z*)$\}$,

pode-se descobrir de quem Adão é avô, de acordo com o conhecimento em Σ_1, obtendo-se uma SLD-refutação para a consulta $\leftarrow$ *avô*(*adão*,*Q*), como ilustrado na árvore de refutação da Figura 1. Neste tipo de árvore, ramos marcados com o símbolo '✘' são aqueles que fracassam em refutar a consulta e ramos marcados com o símbolo '✓' são aqueles que são bem sucedidos. As possíveis respostas para a consulta refutada são todas aquelas resultantes das unificações nos ramos de sucesso. Assim, de acordo com o conhecimento em Σ_1, pode-se afirmar que Adão é avô apenas de Enos.

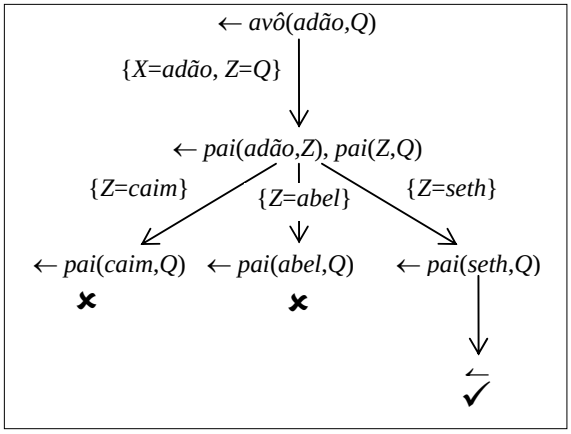

Figura 1 – Árvore de refutação para $\leftarrow$ *avô*(*adão*,*Q*).

2.3. SLDNF-Resolução

Se literais negativos são permitidos no corpo de uma cláusula, a derivação é denominada *SLDNF-resolução* [5]. Nesta forma de derivação, se o primeiro literal na consulta é positivo, procede-se do mesmo modo que na SLD-resolução; caso contrário, se o primeiro literal da

consulta é negativo, digamos $\neg\alpha$, aplicam-se as seguintes regras de derivação:

- $\neg\alpha$ é bem sucedido se e só se α fracassa (falha);
- $\neg\alpha$ fracassa (falha) se e só se α é bem sucedido.

Se $\neg\alpha$ é bem sucedido, ele é removido da consulta e a derivação prossegue; do contrário, a derivação é interrompida e o ramo correspondente é marcado como fracasso. Esta regra de inferência, conhecida como *negação por falha*, é correta e completa [5, 6].

Por exemplo, seja Σ_2 o programa lógico a seguir:

Σ_2 = {*voa*(*X*) ← *ave*(*X*), ¬*pinguim*(*X*);
ave(*fred*) ←;
ave(*bob*) ←;
pinguim(*fred*) ←}

De acordo com Σ_2, apenas *Bob* é capaz de voar. A prova disto é a árvore de SLDNF-refutação construída para a consulta ← *voa*(*Q*), como indicado na Figura 2.

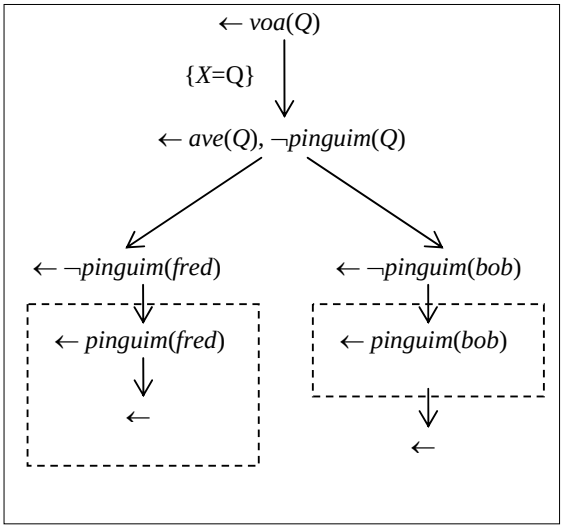

Figura 2 – Árvore de refutação para ← *voa*(*Q*).

2.4. O sistema Prolog

Prolog (*Programming in Logic*) [4] é um sistema de prova automática de teoremas, baseado em SLDNF-resolução, com as seguintes restrições:

- usa as cláusulas do programa na ordem em que elas são escritas, da primeira para a última.
- emprega a estratégia de busca em profundidade para construir a árvore de refutação.

Em Prolog, o programa lógico Σ_1 é codificado como:

```
pai(adão,caim).
pai(adão,abel).
pai(adão,seth).
pai(seth,enos).
avô(X,Z) :- pai(X,Y), pai(Y,Z).
```

Uma vez que este programa tenha sido compilado, pode-se, por exemplo, efetuar a seguinte consulta:

```
?- avô(adão,Q).
Q = enos
```

O programa lógico Σ_2 é codificado como:

```
voa(X) :- ave(X), not(pinguim(X)).
ave(fred).
ave(bob).
pinguim(fred).
```

Com este programa, pode-se, por exemplo, efetuar a seguinte consulta:

```
?- voa(Q).
Q = bob
```

3. Raciocínio Abdutivo em Prolog

Esta seção apresenta os fundamentos do raciocínio abdutivo e discute por que o sistema Prolog não é capaz de utilizá-lo.

3.1. Caracterização de abdução

Seja $\mathcal{A}$ um conjunto de fatos *abdutíveis*, representando explicações plausíveis em um determinado contexto de discurso. Dados um conjunto de sentenças Σ, denominado *base de conhecimento*, e uma sentença γ, denominada *observação*, o raciocínio abdutivo consiste em encontrar um conjunto de sentenças $\Delta \subseteq \mathcal{A}$, denominado *explicação abdutiva* de γ, tal que:

- $\Sigma \cup \Delta$ é consistente
- $\Sigma \cup \Delta \models \gamma$

Esta caracterização é independente da linguagem em que as sentenças são expressas. As condições impostas exigem apenas que as explicações abdutivas de γ sejam consistentes com o estado corrente da base de conhecimento e que esta, completada com os fatos abduzidos, tenha γ como uma de suas consequências lógicas [3, 7].

Critérios adicionais podem ser impostos a fim de restringir as explicações abdutivas. Por exemplo, pode-se impor um critério de *minimalidade* da explicação, ou seja, que não exista $\Delta' \subset \Delta$ tal que $\Sigma \cup \Delta' \models \gamma$. Neste artigo, nenhum critério adicional será imposto.

3.2. Programação em lógica abdutiva

Conforme a Seção 2.2, a SLD-resolução falha quando não é possível resolver uma consulta com uma das cláusulas existentes no programa lógico. Por exemplo, sejam $\mathcal{A}$ = {*chove*, *irrigador_ligado*} e Σ_3 o programa lógico a seguir:

Σ_3 = {*grama_escorregadia* ← *grama_molhada*;
rua_molhada ← *chove*;
grama_molhada ← *chove*;
grama_molhada ← *irrigador_ligado*}

Suponha que observamos o fato *grama_escorregadia* e desejamos encontrar uma explicação plausível para ele, de acordo com as leis estabelecidas em Σ_3. Então, podemos construir uma árvore de refutação para a consulta ←*grama_escorregadia*, usando SLD-resolução. A Figura 3 mostra a árvore construída. Como vemos nesta árvore, a SLD-resolução falha sempre que encontra

fatos que não estão definidos no programa lógico, ou seja, *chove* e *irrigador_ligado*. Porém, se sabemos que estes fatos podem ser abduzidos, então podemos assumir que eles são verdadeiros, por hipótese, e prosseguir com a resolução (vide Figura 4). Ao final do processo, o conjunto de hipóteses formuladas Δ={*chove*←; *irrigador_ligado*←} é justamente o conjunto de explicações plausíveis para o fato inicialmente observado, ou seja:

$\Sigma_3 \cup \Delta \models$ *grama_escorregadia*.

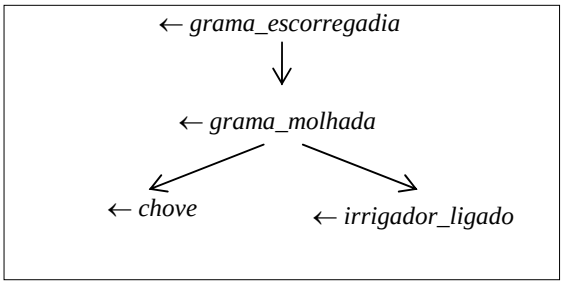

Figura 3 – Árvore de refutação usando SLD-resolução.

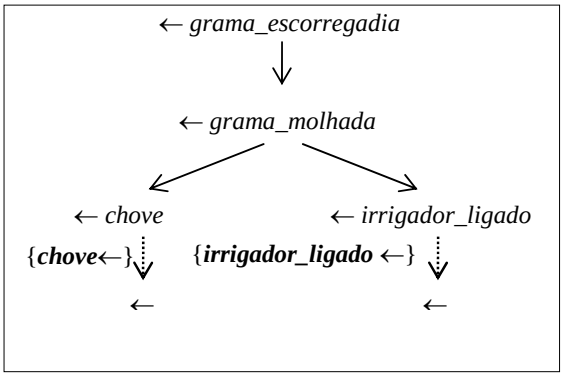

Figura 4 – Árvore de refutação estendida com abdução.

Embora, à primeira vista, o princípio de abdução pareça simples de ser implementado em Prolog, para garantir a consistência do conjunto $\Sigma \cup \Delta$, é preciso tratar das possíveis interações entre abdução e negação por falha [7], ou seja, garantir que um fato já negado não seja abduzido e que um fato já abduzido não seja negado.

4. O Meta-interpretador Abdutivo

Esta seção descreve a nova linguagem proposta para representação de conhecimento envolvendo predicados abdutíveis e apresenta os fundamentos lógicos do meta-interpretador abdutivo desenvolvido em Prolog.

4.1. Representação de conhecimento

Na linguagem proposta neste artigo, as sentenças da base de conhecimento Σ podem ser compostas pelos seguintes elementos básicos:

- *predicados n*-ários denotando propriedades de objetos ou relações entre objetos do contexto de discurso.
- *conectivos lógicos* denotando abdução (`@`), negação (`~`), conjunção (`&`) e implicação (`=>`).
- *valores-verdade* denotando verdade (`true`) e falso (`false`).

As sentenças podem ser dos seguintes tipos:

- *fato*: um fato é uma sentença da forma

  `true => p`,

  estabelecendo que o átomo `p` é verdade.
- *regra*: uma regra é uma sentença da forma

  p_1 `&` … `&` p_n `=> p`,

  estabelecendo que o átomo `p` é verdade se a conjunção de literais p_1`&`…`&`p_n também o for.
- *restrição*: uma restrição é uma sentença da forma

  p_1 `&` … `&` p_n `=> false`,

  estabelecendo que a conjunção de literais p_1`&`…`&`p_n é falsa.

Ademais, predicados abdutíveis devem ser prefixados pelo conectivo `@`, quando aparecem no antecedente de uma implicação. Um predicado abdutivo ocorrendo na cabeça de uma regra ou no corpo de uma restrição não deve ter este prefixo.

Por exemplo, as sentenças:

```
true => pai(adão,caim)
true => pai(seth,enos)
true => igual(X,X)
```

estabelecem que "*Adão é pai de Caim*", "*Seth é pai de Enos*" e "*X é igual a X*" são fatos. As sentenças:

```
~igual(X,Y) => dif(X,Y)
@pai(X,Y) & @pai(Y,Z) => avô(X,Z)
```

estabelecem que "*se X não é igual a Y, então X é diferente de Y*" e "*se X é pai de Y e Y é pai de Z, então X é avô de Z*" são regras. Além disto, declara que fatos desconhecidos sobre a relação `pai` podem ser abduzidos. E, finalmente, as sentenças:

```
pai(X,A) & pai(Y,A) & dif(X,Y) => false
pai(X,X) => false
```

estabelecem que "*uma pessoa não pode ter dois pais*" e "*uma pessoa não pode ser pai de si mesma*" são restrições. Estas restrições são importantes para impedir que fatos incoerentes sobre a relação `pai` sejam abduzidos.

A seguir, apresentamos alguns exemplos que darão uma melhor intuição sobre a semântica desta linguagem.

4.2. Exemplos de aplicação

O meta-interpretador proposto neste artigo é baseado em dois predicados principais: `kb(N,K)`, denota uma base de conhecimento `K` identificada pelo número `N`, e `sat(N,S)`, denota que a base de conhecimento identificada pelo número `N` satisfaz a sentença `S`. Se `sat(N,S)` é verdade, o sistema Prolog apresenta as hipóteses que são necessárias para esta conclusão; caso contrário, ele responde `false`.

Por exemplo, considere a base de conhecimento declarada a seguir:

```
kb(1,[grama_molhada => grama_escorregadia,
      @chuva => rua_molhada,
      @chuva => grama_molhada,
      @irrigador_ligado => grama_molhada]).
```

Uma vez que esta declaração seja comunicada ao sistema Prolog, pode-se realizar a seguinte consulta:

```
?- sat(1,grama_escorregadia).
Hipótese
  true => chuva
Hipótese
  true => irrigador_ligado
```

Como pode ser observado, o sistema Prolog estendido com o meta-interpretador apresenta duas hipóteses como explicação para `grama_escorregadia`, em vista do conhecimento na base 1. No caso da hipótese `chuva`, o raciocínio é o seguinte: assumindo que `chuva` é verdade, temos que `rua_molhada` é verdade; mas, se `rua_molhada` é verdade, conclui-se que `grama_escorregadia` também deve ser. No caso da hipótese `irrigador_ligado`, o raciocínio é análogo.

Considere agora a base de conhecimento a seguir, à qual foi adicionada uma sentença estabelecendo que a "*rua não está molhada*":

```
kb(2,[grama_molhada => grama_escorregadia,
      @chuva => rua_molhada,
      @chuva => grama_molhada,
      @irrigador_ligado => grama_molhada,
      rua_molhada => false]).
```

Então, repetindo a consulta anterior, obtém-se:

```
?- sat(2,grama_escorregadia).
Hipótese
  true => irrigador_ligado
```

De fato, com a nova restrição imposta, o sistema não pode mais inferir que `chuva` é uma explicação plausível para o fato `grama_escorregadia`.

O próximo exemplo ilustra um caso em que as sentenças da base de conhecimento possuem variáveis:

```
kb(3,[true => pai(adão,caim),
      true => pai(adão,abel),
      true => pai(seth,enos),
      true => igual(X,X),
      ~igual(X,Y) => dif(X,Y),
      @pai(X,Y) & @pai(Y,Z) => avô(X,Z),
      @pai(P,X) & @pai(P,Y) &
      dif(X,Y) => irmão(X,Y),
      pai(X,A) & pai(Y,A) &
      dif(X,Y) => false,
      pai(X,X) => false]).
```

Por exemplo, para encontrar uma explicação para o fato "*Adão é avô de Enos e Enos é irmão de Ló*", pode-se realizar a seguinte consulta:

```
?- sat(3, avô(adão,enos) & irmão(enos,ló)).
Hipótese
  true => pai(adão,seth)
  true => pai(seth,ló)
  igual(enos,ló) => false
```

A explicação encontrada pelo sistema, em vista do conhecimento disponível na base 3, é a seguinte: como Seth é pai de Enos, para Adão ser avô de Enos é necessário que Adão seja pai de Seth; além disto, para que Enos seja irmão de Ló, é necessário que eles sejam filhos do mesmo pai; portanto, Seth também deve ser pai de Ló. A restrição de que Enos e Ló não sejam iguais (isto é, não representem a mesma pessoa como, por exemplo, nome e apelido) é necessária porque uma pessoa não pode ser irmã de si mesma.

Um exemplo de aplicação em diagnóstico é:

```
kb(4,[true => conectada(lâmpada1,bateria),
      true => conectada(lâmpada2,bateria),
      @queimada(X) => apagada(X),
      conectada(X,Y) &
      @descarregada(Y) => apagada(X),
      ~queimada(X) & conectada(X,Y) &
      ~descarregada(Y) & apagada(X)=>false]).
```

Esta base de conhecimento estabelece que:

- *as lâmpadas 1 e 2 estão conectadas à bateria*;
- *uma lâmpada queimada fica apagada*;
- *uma lâmpada conectada a uma bateria descarregada fica apagada*;
- *uma lâmpada não queimada, conectada a uma bateria que não está descarregada, não pode estar apagada.*

Suponha que a lâmpada 1 esteja apagada e que uma explicação plausível para este fato seja desejada. Esta explicação pode ser obtida com a seguinte consulta:

```
?- sat(4, apagada(lâmpada1) ).
Hipótese
  true => descarregada(bateria)
Hipótese
  true => queimada(lâmpada1)
```

Suponha agora que a lâmpada 1 esteja apagada mas a lâmpada 2 não. Uma explicação para estes fatos é obtida com a consulta:

```
?- sat(4,apagada(lâmpada1) &
         ~apagada(lâmpada2) ).
Hipótese
  true => queimada(lâmpada1)
  apagada(lâmpada2) => false
```

Como era esperado, com a observação de que a lâmpada 2 está acesa, a única explicação plausível para a lâmpada 1 estar apagada é que ela esteja queimada.

4.3. Implementação em Prolog

A implementação do meta-interpretador abdutivo, na Figura 5, consiste dos seguintes predicados em Prolog:

- `sat(N,S)`: este predicado recupera a definição da base de conhecimento `K` de número `N`; encontra o conjunto de hipóteses que explicam a sentença `S`, em vista do conhecimento em `K` e, em seguida, mostra cada uma destas hipóteses.
- `sat(S,K1,K2)`: este predicado enumera todas as possíveis formas de provar uma sentença `S`, em função de sua estrutura, a partir da base de conhecimento `K1`. Para tanto, pode ser necessário que alguns fatos derivados de negação por falha ou abdução sejam adicionados à base de conhecimento `K1`, resultando na nova base `K2`. A explicação abdutiva de `S` é dada por Δ = `K2-K1`.

```
% ablog.pl – meta-interpretador abdutivo
:- op(950,fx,user:('~')).
:- op(951,fx,user:('@')).
:- op(952,xfy,user:('&')).
:- op(953,xfx,user:('=>')).
% predicado principal
sat(N,S) :-
      kb(N,K),
      setof(H,K^K1^K2^(sat(S,K,K1),
      subtract(K1,K,K2),sort(K2,H)),Hs),
      forall(member(H,Hs),
            (format('~nHipótese~n'),
             mostra(H))), !.
% regras de prova
sat(S,K1,K2) :- S=(P&Q),
    sat(P,K1,K),sat(Q,K,K2).
sat(S,K1,K2) :- S=(P&Q),
    sat(Q,K1,K), sat(P,K,K2).
sat(S,K1,K2) :- S=(~P), neg(P,K1,K2).
   sat(S,K1,K2) :- S=(@P), abd(P,K1,K2).
sat(S,K1,K2) :- res(S,K1,K2).
% regras de negação por falha
neg(S,K,K) :-
   member(S=>false,K).
neg(S,K,[(S=>false)|K]) :-
   ground(S),
   not(res(S,K,K)).
% regras de abdução
abd(S,K,K) :-
   member(true=>S,K).
abd(S,K,[(true=>S)|K]) :-
   ground(S),
   not(res(S,K,K)),
   not(res(false,[true=>S|K],_)).
% regras de resolução
res(true,K,K).
res(S,K1,K2) :-
   copy_term(K1,K),
   member(P=>S,K),
   sat(P,K1,K2).
```

Figura 5 – Meta-interpretador abdutivo em Prolog.

- `neg(S,K1,K2)`: este predicado prova a negação da sentença `S`, a partir do conhecimento em `K1`, usando *negação por falha*. Se uma sentença da forma `S=>false` pertence a `K1`, então `K2` = `K1`; caso contrário, se não é possível provar `S` a partir de `K1` (possivelmente estendida com fatos negados ou abduzidos anteriormente), então `K2` = `K1`$\cup$`{S=>false}`.
- `abd(S,K1,K2)`: este predicado prova a sentença `S`, a partir do conhecimento em `K1`, usando *abdução*. Se uma sentença da forma `true=>S` pertence a `K1`, então `K2` = `K1`; caso contrário, se não é possível provar `S` a partir de `K1`, nem a hipótese de `S` ser verdadeira contradiz o conhecimento em `K1`, então `K2` = `K1`$\cup${`true=>S`}.
- `res(S,K1,K2)`: este predicado prova a sentença `S` a partir do conhecimento em `K1`, usando *resolução*. Para tanto, ele encontra uma sentença da forma `P=>S` e tenta satisfazer a condição `P`. O conjunto Δ = `K2-K1` denota os fatos negados ou abduzidos a fim de satisfazer `P`.
- `mostra(F)`: este predicado mostra fatos negados ou abduzidos durante a prova de uma sentença.

5. Conclusões

Este artigo ressalta a importância da abdução em IA para tratar problemas que requerem raciocínio de efeitos para causas (por exemplo, diagnóstico) e que, apesar disto, o sistema Prolog, uma das principais ferramentas para desenvolvimento de aplicações em IA, não é capaz de realizar este tipo de raciocínio.

Como solução, propõe-se uma linguagem para representação de conhecimento envolvendo predicados abdutíveis (que podem ser assumidos como hipóteses) e um meta-interpretador para usar este tipo de conhecimento.

Para demonstrar que este meta-interpretador estende o Prolog com a capacidade de raciocínio abdutivo, uma série de problemas, cujas soluções dependem deste tipo de raciocínio, foram apresentados e solucionados.

Como trabalhos futuros, pretende-se apresentar uma prova formal das propriedades lógicas do meta-interpretador proposto e testar sua funcionalidade em aplicações práticas mais complexas.

Agradecimentos

Ao CNPq, pela bolsa de produtividade em pesquisa concedida, conforme o processo 304322/2009-1.

Referências Bibliográficas

[1] C. S. Pierce, **Collected Papers of Charles Sanders Pierce**, vol. 2, Harvard University Press, 1931-1958.

[2] S. Russell & P. Norvig, **Artificial Intelligence – A Modern Approach**, 2nd edition, Prentice-Hall, 2004.

[3] P. T. Cox & T. Pietrzykowsky, **Causes for Effects: their Computation and Applications**, 8th Int. Conf. on Automated Deduction, p. 608-621, 1986.

[4] L. Sterling & E. Shapiro, **The Art of Prolog**, MIT Press, 1986.

[5] K. R. Apt, **Logic Programming**, In Handbook of Theoretical Computer Science, J. van Leeuven, editor, Elsevier Science Publishers, p. 493-573, 1990.

[6] K. Clark, **Negation as Failure**, In Herve Gallaire & Jack Minker, editors, Logic and Data Bases, p. 293-332, Plenun Press, 1978.

[7] M. Denecker & D. Schreye, **SLDNFA: An Abductive Procedure for Normal Abductive Programs**, International Joint Conference and Symposium on Logic Programming, K. R. Apt, p. 686-700, 1992.