Instituto de Ciências Matemáticas e de Computação

ISSN - 0103-2569

O Sistema de Programação Lógica Indutiva Aleph - Características e Funcionamento

Mariza Ferro
Fabiana Esteves de Paiva
Maria Carolina Monard

Nº 300

RELATÓRIOS TÉCNICOS DO ICMC

São Carlos
Jun./2007

O Sistema de Programação Lógica Indutiva Aleph Características e Funcionamento

Mariza Ferro
Fabiana Esteves de Paiva
Maria Carolina Monard

Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação
Departamento de Ciências de Computação e Estatística
Laboratório de Inteligência Computacional
Caixa Postal 668, 13560-970 - São Carlos, SP, Brasil
e-mail: {mariza, fapaiva, mcmonard}@icmc.usp.br

Resumo

Mineração de Dados Relacional (MDR) é um processo de descoberta de novos padrões e conhecimento útil que envolve múltiplas tabelas (relações) de uma base de dados relacional. A maioria das abordagens de MDR é baseada em Programação Lógica Indutiva (PLI). PLI é definida como a intersecção de Aprendizado de Máquina indutivo e programação lógica e tem se expandido para cobrir todas as tarefas de MDR. O sistema de PLI Aleph, foco de estudo deste trabalho, possui, além das características comuns aos sistemas de PLI, a capacidade de emular funcionalidades de vários outros sistemas de PLI, tais como, regras de classificação relacionais, árvores de decisão relacionais, regras de associação relacionais, entre outras. Neste trabalho são descritas maneiras de como emular diversos sistemas de PLI utilizando Aleph, descrevendo o ajuste de parâmetros necessário e a escolha dos diversos comandos disponíveis.

Palavras-Chave: Aprendizado de Máquina, Programação Lógica Indutiva, Aleph.

Junho 2007

Sumário

1 Introdução **2**

2 Programação Lógica Indutiva e Aleph **5**

2.1 Características Básicas do Sistema Aleph 6

2.1.1 Como Iniciar o Sistema 6

2.1.2 Sobre o Sistema 7

2.1.3 Declarações de Modo 8

2.1.4 Tipos 10

2.1.5 Declaração dos *Determinations* 11

2.1.6 Exemplos Positivos e Negativos 11

2.1.7 Construção da Cláusula *Bottom* 12

2.1.8 Parâmetros 16

2.2 Exemplo de Execução 16

3 Uso Avançado do Aleph **24**

3.1 Comparação entre a Construção de Teorias induce/0 e induce_cover/0 25

3.2 Indução de Árvores de Decisão Relacionais 29

3.3 Regras de Associação Relacional 35

3.4 Métodos de Busca Estocástica 42

3.5 Aprendizado Abdutivo 44

3.6 Aprendizado de *Features* 46

3.7 Definições do Usuário 47

3.7.1 Restrições de Integridade 47

3.7.2 Refinamento 48

3.7.3 Poda 48

3.7.4 Custo 49

4 Conclusão **50**

Referências **51**

Lista de Figuras

1 Substituições dos argumentos dos predicados a partir da unificação do exemplo com a cabeça. 13

2 Substituições dos argumentos dos predicados a partir das unificações com fatos do conhecimento do domínio. 14
3 Relação entre os tipos de variáveis dos argumentos de cada predicado. . 15
4 Trens que viajam para Leste e Oeste (Michalski et al., 1989) 17
5 Árvore de decisão relacional para predição da variável classe A do predicado alvo *manutenção(M,A)* 29
6 Exemplo de uma regra de associação 36

Lista de Tabelas

1 Arquivos `trens.f` e `trens.n` 17
2 Conjunto de exemplos 26

Lista de Abreviaturas

AM Aprendizado de Máquina

IA Inteligência Artificial

LABIC Laboratório de Inteligência Computacional

LPO Linguagem de Primeira Ordem

PLI Programação Lógica Indutiva

MD Mineração de Dados

MDR Mineração de Dados Relacional

1 Introdução

Mineração de Dados (MD) é um processo cujo principal objetivo é extrair padrões de bases de dados. As principais etapas do processo de Mineração de Dados (MD) são: o pré-processamento de dados, na qual procura-se melhorar a qualidade dos dados; a etapa posterior de extração de padrões, na qual são freqüentemente utilizados algoritmos de aprendizado e a etapa de pós-processamento desses padrões (conhecimento) na qual o objetivo é selecionar ou melhorar os padrões encontrados. A maioria das abordagens de MD assume que os dados estão em uma única tabela. Esse tipo de dados pode ser manipulado pelos algoritmos de Aprendizado de Máquina (AM) proposicional que utilizam uma linguagem de representação proposicional para descrever exemplos, tais como, uma tabela atributo-valor. Por outro lado, as abordagens de Mineração de Dados Relacional (MDR), muitas das quais são baseadas em Programação Lógica Indutiva (PLI), realizam a busca por padrões que envolvem múltiplas tabelas (relações) de uma base de dados relacional. Esses métodos de Mineração de Dados Relacional (MDR) utilizam uma linguagem de representação relacional baseada em lógica de primeira ordem (Džeroski, 2003).

Assim como muitos dos algoritmos utilizados na etapa de extração de padrões no processo de MD vêm da área de Aprendizado de Máquina (AM), muitos dos algoritmos de MDR vêm da área de Programação Lógica Indutiva (PLI), que consiste da intersecção do AM indutivo e da programação lógica. O objetivo da PLI é aprender programas lógicos a partir de exemplos e conhecimento do domínio (Lavrač & Džeroski, 2001). Nos últimos anos, a PLI tem se expandido para cobrir todas as tarefas de MDR. Os tipos mais comuns de técnicas utilizadas em AM proposicional para descrever os padrões encontrados têm sido estendidos para suas versões relacionais tais como, regras de classificação relacionais, árvores de decisão relacionais, regras de associação relacionais, entre outros, descritos brevemente a seguir.

A tarefa de indução de regras relacionais é a tarefa mais freqüentemente empregada em PLI. Essa é a tarefa do aprendizado de definições lógicas de relações na qual tuplas que pertencem ou não pertencem à relação alvo são dadas como exemplos de treinamento. A partir dos exemplos de treinamento, PLI induz um programa lógico (definição do predicado) que define a relação alvo em termos de outras relações que são dadas como conhecimento de fundo. Alguns exemplos de sistemas de PLI que realizam indução de regras são o Progol (Muggleton, 1995) e o FOIL (Quinlan, 1990).

Nas árvores de decisão proposicionais, nós internos são testes e os nós folha são as

predições para os valores da classe. As árvores de decisão relacionais possuem uma estrutura muito parecida com as proposicionais. A maior diferença para as árvores relacionais está nos testes que aparecem nos nós internos. No caso relacional, os testes são consultas lógicas. O sistema de PLI Tilde (Blockeel, 1998) faz indução de árvores de decisão relacionais.

As regras de associação caracterizam o quanto a presença de um conjunto de itens de uma base de dados implica a presença de algum outro conjunto distinto de itens nos mesmos registros. No caso das bases de dados relacionais, regras de associação relacionais poderão ser descobertas. O sistema WARMR (Dehaspe & De Raedt, 1997) realiza indução de regras de associação relacionais.

O sistema Aleph, ***A*** ***L****earning* ***E****ngine for* ***P****roposing* ***H****ypotheses* (Srinivasan, 2000), é um sistema de PLI escrito em Prolog, o qual foi inicialmente conhecido como P-Progol. O sistema Aleph, foco deste trabalho, foi desenvolvido com o intuito de ser um protótipo para explorar idéias de PLI. O Aleph possui uma poderosa linguagem de representação que permite representar expressões complexas e incorporar novo conhecimento do domínio com facilidade. Além dessas características comuns aos sistemas de PLI, o Aleph possui outras características muito interessantes, tais como: permitir escolher qual a ordem de geração das regras (geral para o específico ou específico para o geral); mudar a função de avaliação (entropia, precisão, Laplace entre outras) e a ordem de busca (*Hill-Climbing*, *Branch-and-Bound*, entre outras). Utilizando os parâmetros disponíveis no Aleph para modificação de funções de busca, funções de avaliação, além de diversos parâmetros específicos para determinadas tarefas, é possível emular funcionalidades de vários outros sistemas de PLI, tais como: Progol (Muggleton, 1995), FOIL (Quinlan, 1990), FORS (Karalič & Bratko, 1997), Indlog (Camacho, 1994), MIDOS (Wrobel, 1997), Tilde (Blockeel, 1998) e WARMR (Dehaspe & De Raedt, 1997). Devido a essa versatilidade do Aleph, foi decidido utilizá-lo em nossas pesquisas relacionadas a aprendizado relacional.

Porém, por falta de documentação adequada sobre o Aleph, a maneira de como emular esses outros sistemas não está o suficientemente clara. É necessário saber como ajustar adequadamente os parâmetros, escolher corretamente os diversos comandos disponíveis para se construir uma teoria e, até mesmo, saber quais dados podem ser utilizados para cada tarefa. Todas essas dúvidas na utilização do Aleph motivaram a realização deste trabalho, que tem como objetivo entender a ação dos comandos e parâmentros mais importantes do Aleph, bem como documentar o funcionamento dos mesmos.

Este trabalho está organizado da seguinte forma: na Seção 2 são apresentados alguns

conceitos sobre PLI, características básicas e de funcionamento do sistema de PLI Aleph, além de um exemplo de execução utilizando Aleph; na Seção 3 são apresentadas as formas disponíveis para se construir uma teoria, alterar funções de busca e avaliação, e como emular as funcionalidades do Aleph para expressar os padrões utilizando como linuagens de descrição árvores de decisão e regras de associação relacionais, além dos métodos de busca estocásticos. Descreve-se ainda como o usuário pode interferir no processo de aprendizado; por fim, na Seção 4 são apresentadas algumas considerações finais sobre este trabalho.

2 Programação Lógica Indutiva e Aleph

PLI é uma área de pesquisa em AM definida como a intersecção de AM indutivo e programação lógica (Lavrač & Flach, 2001). Em programação lógica, o conceito a ser aprendido é um programa lógico, *i.é.*, um conjunto de cláusulas de Horn. Da lógica computacional, PLI herda seu formalismo representacional, várias técnicas bem estabelecidas e uma profunda base teórica. Do AM indutivo, herda uma abordagem experimental e orientação para aplicações práticas, tais como o desenvolvimento de ferramentas e técnicas para induzir hipóteses a partir de exemplos e sintetizar novos conhecimentos a partir de experiências (Muggleton & Raedt, 1994).

A habilidade dos seres humanos de fazer generalizações a partir de alguns fatos, ou descobrir padrões em coleções de observações aparentemente caóticas, é atingida por meio do aprendizado por indução. A indução é a forma de inferência lógica que permite obter conclusões gerais a partir de um conjunto particular de exemplos, ou casos observados. É caracterizada como o raciocínio que parte do específico para o geral, da parte para o todo. No aprendizado por indução, o aprendiz adquire um conceito fazendo inferências indutivas sobre os fatos apresentados. A indução pode ser vista como o inverso da dedução. Hipóteses geradas pela inferência indutiva podem ou não preservar a verdade, ao contrário da dedução na qual, se todas as premissas forem verdadeiras, a conclusão é verdadeira. Há uma segunda diferença entre as hipóteses dedutivas e indutivas. Na dedução, toda a informação do conteúdo factual da conclusão já está, pelo menos implicitamente, nas premissas. Em contraste, na indução a conclusão contém informação que excede a das premissas, como mostrado a seguir.

Exemplo de hipótese dedutiva

Todo mamífero tem coração.
Todos os felinos são mamíferos.
$\longrightarrow$ *Todos os felinos têm coração.*

Exemplo de hipótese indutiva

Todos os felinos que foram observados tinham coração.
$\longrightarrow$ *Todos os felinos têm coração.*

PLI pode ser vista como o estudo de métodos de aprendizado para dados e regras que são representados como predicados lógicos de primeira ordem. Predicados lógicos

permitem variáveis quantificadas e relações, e podem representar conceitos que não podem ser expressos usando uma linguagem de descrição proposicional. Uma base de dados relacional pode ser facilmente traduzida em lógica de primeira ordem e ser usada como fonte de dados para PLI.

Como exemplo, considere as seguintes regras escritas na sintaxe Prolog, as quais definem a relação `tio/2`.

```
tio(X,Y):- irmao(X,Z), progenitor(Z,Y).
tio(X,Y):- marido(X,Z),irma(Z,W), progenitor(W,Y).
```

O objetivo da PLI é inferir regras, como as apresentadas acima, dada uma base de dados com fatos e definições lógicas de outras relações. Por exemplo, um sistema de PLI pode aprender as regras para `tio/2`, chamado de predicado alvo, dado um conjunto de exemplos positivos e negativos das relações `tio/2` e um conjunto de fatos para as relações `progenitor/2`, `irmao/2`, `irma/2` e `marido/2`, chamadas de predicados de conhecimento de fundo para os membros de uma família (Mooney et al., 2002).

Em termos um pouco mais formais, dado um programa lógico B que modela o conhecimento de fundo e um conjunto de termos *ground* G que representa os exemplos de treinamento, PLI constrói um programa lógico H, tal que $B \wedge H \models G$. Se o operador dedutivo usado for a resolução, então o operador indutivo necessário para resolver a equação é o operador de resolução inversa (Muggleton, 1995).

Atualmente existem diversos sistemas de PLI, entre eles o sistema Aleph (Srinivasan, 2000). Esse sistema de PLI possui uma poderosa linguagem de representação além de diversos recursos que lhe permite simular funcionalidades de vários outros sistemas de PLI existentes, como mencionado anteriormente. O sistema Aleph é o foco principal deste trabalho e será apresentado com maiores detalhes nas próximas seções.

2.1 Características Básicas do Sistema Aleph

Nesta seção são explicados em maiores detalhes várias características e comandos utilizados para a execução do sistema Aleph.

2.1.1 Como Iniciar o Sistema

Aleph é um sistema escrito em Prolog que deve ser utilizado com o compilador Yap Prolog. Ainda que o compilador SWI Prolog poderia ser utilizado, experimentos reali-

zados mostraram que sua utilização prejudica algumas das funcionalidades do sistema.

Para carregar o Aleph, deve-se inicialmente carregar o sistema Yap e, em seguida, chamá-lo através do comando `[aleph].`, como mostra o exemplo a seguir:

```
fapaiva@penelope:~$ yap
% Restoring file /usr/local/lib/Yap/startup
YAP version Yap-5.1.1
   ?- [aleph].
 % consulting /home/fapaiva/aleph.pl...


A L E P H
Version 5
Last modified: Sun Jun  4 10:51:31 UTC 2006

Manual: http://www.comlab.ox.ac.uk/oucl/groups/machlearn/Aleph/index.html

 % consulted /home/fapaiva/aleph.pl in module user, 140 msec 1138688 bytes
yes
   ?-
```

2.1.2 Sobre o Sistema

Aleph é um sistema de PLI que constrói incrementalmente as cláusulas de uma hipótese, iterativamente, por meio do seguinte algoritmo básico:

1. Seleciona um exemplo positivo do conjunto de exemplos de treinamento para ser generalizado. Se não existirem mais exemplos, pára;

2. Saturação: constrói a cláusula mais específica, não *ground*, que implique o exemplo selecionado no passo anterior. Isso é feito por repetidas aplicações do operador de resolução inversa sobre o exemplo, até que todos seus termos *ground* tenham sido substituídos por variáveis, as quais são distribuídas pelo corpo da cláusula, até que o exemplo positivo *ground* original, e somente ele, seja coberto pela cláusula *bottom*. A cláusula minimamente generalizada é chamada de cláusula *bottom*;

3. Redução: realiza uma busca por uma "boa" cláusula entre a cláusula maximamente geral (*maximally general*), a cláusula vazia e a cláusula *bottom* maximamente específica (*maximally specific*). O espaço entre a cláusula vazia e a cláusula

bottom é parcialmente ordenado pela subsunção-θ e a busca prossegue pelo reticulado definido por essa ordenação parcial. A "boa qualidade" de cada cláusula encontrada ao longo do caminho de busca é medida utilizando uma função de avaliação;

4. *Cover Removal*: adiciona a cláusula de melhor qualidade, chamada de *best clause*, à teoria e remove todos os exemplos positivos cobertos por ela da base de exemplos;

5. Repete até que todos os exemplos positivos sejam cobertos.

O Aleph utiliza a informação contida nos seguintes três arquivos para construir uma teoria:

- `file.b`: contém a informação sobre o conhecimento do domínio (intensional e extensional), as restrições de busca e linguagem (declarações de modo, determinations), restrições de tipos e parâmetros;
- `file.f`: contém os exemplos positivos (somente fatos *ground*);
- `file.n`: contém os exemplos negativos (somente fatos *ground*). Pode não existir esse arquivo, já que Aleph pode realizar aprendizado utilizando somente exemplos positivos.

2.1.3 Declarações de Modo

As declarações de modo no arquivo `file.b` descrevem as relações (predicados) entre objetos de dados e tipos desses dados. Essas declarações permitem também informar ao Aleph se a relação pode ser utilizada na cabeça (declarações `modeh`) ou no corpo (declarações `modeb`) das regras geradas. Na declaração de modo também são descritos o tipo dos argumentos e o número de instanciações de cada predicado. As declarações têm o seguinte formato:

```
mode(Numero_Chamadas,Modo).
```

O `Numero_Chamadas`, ou como é mais conhecido, o *recall*, determina o limite do número de instanciações alternativas de um predicado. Uma instanciação do predicado é uma substituição de tipos declarados no arquivo `.b` para cada variável ou constante. O *recall* pode ser qualquer número positivo $n \geq 1$ ou '*'.

Se é conhecido que há somente um certo número de instanciações possíveis para um determinado predicado, é possível informar isso pelo *recall*. Por exemplo, para uma declaração do predicado `progenitor_de(Pai,Filho)` poderia ser dado um *recall* igual a 2, pois todas as pessoas terão no máximo dois progenitores, enquanto que o *recall* seria 4 para uma declaração do predicado `avo(Avo,Neto)`. O *recall* '*' é utilizado quando não há limite para o número de instanciações de um predicado. Por exemplo, em princípio, não há limite para o número de ancestrais que uma pessoa pode ter.

A segunda parte da declaração do `mode/2`, denominada de `Modo`, indica o formato do predicado que será utilizado e tem o seguinte formato:

```
predicado(Tipo_Argumento1,Tipo_Argumento2,...,Tipo_Argumenton).
```

Para o aprendizado da relação `sogra_de(Sogra,Genro)` com conhecimento do domínio descrito, por exemplo, pelas relações `progenitor_de(Mae,Filha)` e `esposa_de(Esposa,Marido)` as declarações de modo podem ser:

```
:- modeh(1,sogra_de(+mulher,+homem)).
:- modeb(*,progenitor_de(+mulher,-mulher)).
:- modeb(1,esposa_de(+mulher,+homem)).
```

A declaração `modeh/2` indica o predicado que irá compor a cabeça das regras. O valor de *RecallNumber* é 1 para esse predicado, pois o predicado pode ter uma única resposta (sim ou não), dados os dois argumentos de entrada. No exemplo, `modeh/2` informa que a cabeça das regras deve ser `sogra_de(Sogra,Genro)`, onde `Sogra` é do tipo `mulher` e `Genro` é do tipo `homem`. As declarações `modeb/2` indicam que as regras geradas podem ter no corpo os predicados `progenitor_de(Mae,Filha)` e `esposa_de(Esposa,Marido)`, nos quais `Mae`, `Filha` e `Esposa` são do tipo `mulher` e `Marido` é do tipo `homem`. O valor de *RecallNumber* para o predicado `progenitor_de(Mae,Filha)` é igual a '*', pois não se sabe quantas filhas uma mãe pode ter. Para o predicado `esposa_de(Esposa,Marido)` esse parâmetro tem valor '1', pois apenas uma das filhas será a esposa.

Os tipos dos argumentos dos predicados podem ser estruturados ou simples. Os tipos estruturados são da forma $f(..)$, no qual f é um símbolo de função. Um exemplo de argumento estruturado é:

```
:- modeh(1,member(+number,[+number|+lista])).
```

Os tipos simples de argumentos podem ser: +, - ou #. O símbolo '+' indica que o argumento do predicado é uma variável de entrada. O símbolo '-' indica uma variável de saída e o símbolo # indica que esse argumento é uma constante. A utilização desses tipos de variáveis devem seguir as regras listadas a seguir:

- Variável de Entrada (+): Qualquer variável de entrada do tipo `T` em um literal do corpo B_i (`modeb`) deve aparecer ou como uma variável de saída do tipo `T` em um literal do corpo antes de B_i, ou como uma variável de entrada do tipo `T` na cabeça, como mostra o exemplo a seguir:

  ```
  :- modeh(1, sogra_de(+mulher,+homem)).
  :- modeb(*, progenitor_de(+mulher,-mulher)).
  :- modeb(1, esposa_de(+mulher,+homem)).
  ```

  A variável de entrada do tipo `homem` do predicado `esposa_de/2` vem da variável de mesmo tipo da cabeça. A variável de entrada do tipo `mulher` do predicado `esposa_de/2` vem ou da variável de saída de mesmo tipo do predicado `progenitor_de/2` ou da variável de entrada de mesmo tipo da cabeça.

- Variável de Saída (-): Qualquer variável de saída do tipo `T` na cabeça (`modeh`) deve aparecer como uma variável de saída do tipo `T` em um literal do corpo B_i. Qualquer literal que possua variável de saída deve ter, pelo menos, uma variável de entrada. Como mostra o exemplo a serguir:

  ```
  :- modeh(1, sogra_de(+mulher,-homem)).
  :- modeb(1, progenitor_de(+mulher,-mulher)).
  :- modeb(1, esposa_de(+mulher,-homem)).
  ```

  A variável de saída do tipo `homem` da cabeça vem da variável de saída de mesmo tipo do predicado `esposa_de/2`.

- Constante: Qualquer argumento denotado por #`T` só pode ser substituído por átomos *ground*.

2.1.4 Tipos

Os tipos devem ser especificados para cada argumento dos predicados utilizados na construção de uma hipótese. Para o Aleph, os tipos são só nomes e a declaração de tipos nada mais é que um conjunto de fatos. Aleph reconhece argumentos de tipos diferentes e os trata distintamente. Por exemplo, a descrição de objetos dos tipos `homem` e `mulher` poderia ser

```
mulher(jane).
mulher(sueli).
homem(henrique).
homem(junior).
    ...
```

2.1.5 Declaração dos *Determinations*

O Aleph utiliza `determination/2` para declarar os predicados que podem ser usados para construir uma hipótese. Essa declaração tem o formato:

```
determination(Pred_Alvo/Aridadea,Pred_Corpo/Aridadec).
```

O primeiro argumento é o nome do predicado alvo e sua aridade, isto é, o predicado que irá aparecer na cabeça da regra induzida. O segundo argumento consiste do nome e aridade de um predicado que pode aparecer no corpo da cláusula. Por exemplo, para o aprendizado da relação `sogra_de(Sogra,Genro)`, uma declaração seria:

```
:-determination(sogra_de/2,progenitor_de/2).
:-determination(sogra_de/2,esposa_de/2).
```

Para que um predicado seja utilizado na construção de uma hipótese, é necessário que ele tenha declarações tanto de `modeb/2` quanto de `determination/2`. Caso uma das declarações esteja ausente, o predicado não será utilizado na construção da hipótese.

2.1.6 Exemplos Positivos e Negativos

Exemplos positivos e negativos do conceito a ser aprendido devem ser apresentados na forma extensional (fatos) nos arquivos com extensão `.f` e `.n` respectivamente. Por exemplo, para o aprendizado do conceito `sogra_de(Sogra,Genro)` poderíamos ter a declaração dos seguintes exemplos positivos no arquivo com extensão `.f`:

```
sogra_de(jane,henrique).
sogra_de(sueli,joao).
```

E os seguintes exemplos negativos no arquivo com extensão `.n`:

```
sogra_de(jane,joao).
sogra_de(sueli,henrique).
```

Vale observar que Aleph pode realizar aprendizado somente com exemplos positivos, nesse caso o arquivo `.n` não é utilizado.

2.1.7 Construção da Cláusula *Bottom*

Como já mencionado anteriormente, a cláusula *bottom*, construída no passo de saturação, é a cláusula mais específica que implique o exemplo selecionado. Ela é importante, pois é a partir dela que o Aleph constrói as possíveis hipóteses para expressar o conceito a ser aprendido. Sua corretude depende do parâmetro *recall* e das relações entre os tipos dos argumentos dos predicados presentes nas declarações de modo. Para ilustrar a evolução do passo de saturação, bem como a finalidade dos parâmetros utilizados nesse passo, considere o seguinte conteúdo do arquivo `.b`

```
:- modeh(1, sogra_de(+mulher,-homem)).
:- modeb(*, progenitor_de(+mulher,-mulher)).
:- modeb(1, esposa_de(+mulher,-homem)).

:- determination(sogra_de/2,progenitor_de/2).
:- determination(sogra_de/2,esposa_de/2).

%declaração dos tipos
homem(pai1).
homem(marido1).
homem(marido2).
mulher(mae1).
mulher(filha11).
mulher(filha12).
mulher(neta11).

%conhecimento de fundo
progenitor_de(mae1, filha11).
progenitor_de(pai1, filha11).
progenitor_de(mae1, filha12).
progenitor_de(pai1, filha12).
progenitor_de(filha11, neta11).
esposa_de(mae1, pai1).
esposa_de(filha11, marido1).
esposa_de(filha12, marido2).
```

o seguinte conteúdo do arquivo de exemplos positivos (`.f`)

```
sogra_de(mae1, marido1).
sogra_de(mae1, marido2).
```

e o seguinte conteúdo do arquivo de exemplos negativos (`.n`).

```
sogra_de(mae1, pai1).
sogra_de(filha11, marido2).
sogra_de(filha11, marido1).
```

A saturação do exemplo positivo `sogra_de(mae1, marido1)` segue os seguintes passos:

1. Substituição dos argumentos de entrada da cabeça pelos valores correspondentes do exemplo positivo a ser saturado. Os argumentos de saída da cabeça são substituídos por variáveis. As constantes são sempre substituídas por átomos *ground*. No exemplo `sogra_de(mae1, marido1)` a variável do tipo `mulher` é substituída por `mae1` e a variável de saída do tipo `homem` é substituída por uma variável de nome `A`. O resultado da substituição é: `sogra_de(mae1, A)`.

2. São escolhidos os predicados do corpo que tenham argumentos de entrada do mesmo tipo que os argumentos de entrada da cabeça. Os argumentos de entrada desses predicados são substituídos pelos mesmos valores cujos argumentos de entrada da cabeça foram substituídos no passo acima. Os argumentos de saída desses predicados são substituídos por variáveis — Figura 1. Então, procura-se por fatos no conhecimento do domínio que unifiquem com esses predicados. Essa busca por fatos pode ocorrer de uma a N vezes, dependendo do valor do parâmetro *recall*, para cada argumento de entrada. Todos os fatos no conhecimento do domínio que unificam com o predicado informam quais valores as variáveis desse predicado podem assumir. Por exemplo, a variável `B` do tipo `mulher` do predicado `progenitor_de/2` pode ser substiuída pelos valores `filha11` e `filha12`. E a variável `C` do tipo `homem` do predicado `esposa_de/2` pode ser substituída pelo valor `pai1` — Figura 2.

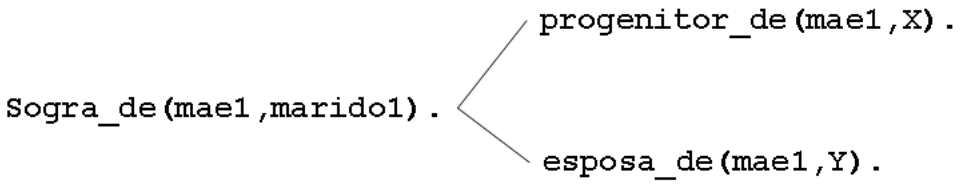

Figura 1: Substituições dos argumentos dos predicados a partir da unificação do exemplo com a cabeça.

3. Os valores encontrados para as variáveis de saída são utilizados para a busca por fatos de predicados do corpo que possuem variáveis de entrada do mesmo tipo. Por exemplo:

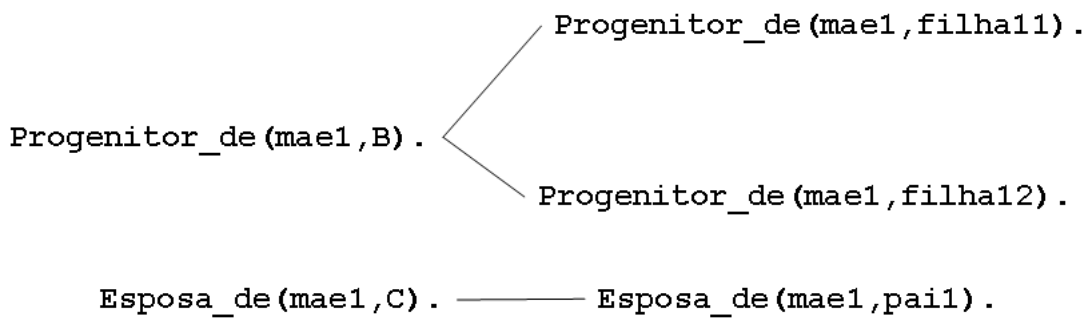

Figura 2: Substituições dos argumentos dos predicados a partir das unificações com fatos do conhecimento do domínio.

`esposa_de(filha11, C).` ⟶ `esposa_de(filha11, marido1).`

`esposa_de(filha12, C).` ⟶ `esposa_de(filha12, marido2).`

Nesse caso, a variável `C` do tipo `homem` só pode assumir o valor `marido1`, pois a substituição pelo valor `marido2` não implica o exemplo `sogra_de(mae1, marido1)`. Esse é um caso da regra da variável de saída em que uma variável de saída da cabeça deve vir de uma variável de saída do mesmo tipo do corpo.

4. Ao final, são adicionados à cláusula *bottom* os predicados encontrados cujos valores combinam com os valores do exemplo saturado (condição para que o exemplo seja coberto) e todos os valores encontrados para os argumentos dos predicados são substituídos por variáveis. Os argumentos substituídos nos passos anteriores por valores idênticos, são agora substituídos por variáveis de mesmo nome. Por exemplo:

```
sogra_de(A,B):-
   progenitor_de(A,C),progenitor_de(A,D),
   progenitor_de(A,E),esposa(C,B).
```

De acordo com as regras listadas na Seção 2.1.3 na página 8, para cada tipo de variável (entrada, saída ou constante), a relação entre os tipos de variáveis dos argumentos de cada predicado se dá como apresentado na Figura 3.

O sistema Aleph procura pelos fatos no conhecimento do domínio na ordem em que aparecem no arquivo `.b`. Nesse caso, se o valor do parâmetro *recall* fosse 1 para o predicado `progenitor_de/2`, o resultado da cláusula *bottom* ainda estaria correto, pois o primeiro fato encontrado no conhecimento do domínio seria o da `filha11` que é

```
:- modeh(1,sogra_de(+mulher,-homem)).
:- modeb(*, progenitor_de(+mulher,-mulher)).
:- modeb(1, esposa_de(+mulher,-homem)).
```

Figura 3: Relação entre os tipos de variáveis dos argumentos de cada predicado.

casada com o `marido1`. Porém, trocando os fatos `progenitor_de(mae1, filha11)` e `progenitor_de(mae1, filha12)` de posição no arquivo `.b` (veja abaixo a alteração no arquivo `.b`), o primeiro fato encontrado no conhecimento do domínio seria `progenitor_de(mae1, filha12)`, que não é casada com `marido1`. Não encontrando a filha correta, o sistema não encontraria um fato para `esposa_de/2` que casasse com o exemplo, e assim, a cláusula *bottom* gerada seria: `sogra_de(A,B)`. Portanto, a escolha do valor para o parâmetro *recall* pode ser a causa de uma cláusula *bottom* gerada incorretamente.

```
:- modeh(1, sogra_de(+mulher,-homem)).
:- modeb(1, progenitor_de(+mulher,-mulher)).
:- modeb(1, esposa_de(+mulher,-homem)).

:- determination(sogra_de/2,progenitor_de/2).
:- determination(sogra_de/2,esposa_de/2).

%declaração dos tipos
homem(pai1).
homem(marido1).
homem(marido2).
mulher(mae1).
mulher(filha11).
mulher(filha12).
mulher(neta11).

%conhecimento de fundo

%progenitor_de(mae1, filha12) aparece antes de progenitor_de(mae1, filha11).
%Isso implica que progenitor_de(mae1, filha12) será o primeiro e único fato
%(único, pois Recall é 1) encontrado para o predicado progenitor_de/2.
progenitor_de(mae1, filha12).
progenitor_de(pai1, filha11).
progenitor_de(mae1, filha11).
progenitor_de(pai1, filha12).
progenitor_de(filha11, neta11).
```

```
esposa_de(mae1, pai1).
esposa_de(filha11, marido1).
esposa_de(filha12, marido2).
```

2.1.8 Parâmetros

O Aleph permite ao usuário determinar uma variedade de restrições no espaço das possíveis hipóteses a serem aprendidas, bem como na busca realizada nesse espaço de possíveis hipóteses, pela definição de novos valores dos parâmetros disponíveis. O predicado `set/2` permite ao usuário definir o valor do parâmetro `Parametro`

```
set(Parametro,Valor).
```

O valor corrente de um determinado parâmetro pode ser obtido usando

```
setting(Parametro,Valor).
```

enquanto que

```
noset(Parametro).
```

retorna o valor desse parâmetro para seu valor padrão. O uso de diversos parâmetros no Aleph será exemplificado ao longo deste trabalho.

2.2 Exemplo de Execução

Nesta seção é apresentado um exemplo de como utilizar o Aleph, ou seja, qual a seqüência básica de comandos utilizados para se construir uma teoria e como o Aleph irá responder a esses comandos. Para ilustrar, será utilizado o exemplo dos trens de Michalski (Michalski et al., 1989), no qual, dada uma seqüência de trens — Figura 4 — cada trem tem um número de vagões e cada vagão pode ter diferentes propriedades, indicadas pelo desenho de uma figura com um determinado formato no seu interior. Também sabe-se que o trem está viajando para o leste (East) ou para o oeste (West). O problema é encontrar uma regra capaz de predizer, dadas as propriedades dos seus vagões, se o trem está viajando para leste; esse trem é chamado de `eastbound`.

A Tabela 1 mostra a declaração dos exemplos positivos e negativos nos arquivos correspondentes `trens.f` e `trens.n`. O próximo passo é definir o arquivo `trens.b` com

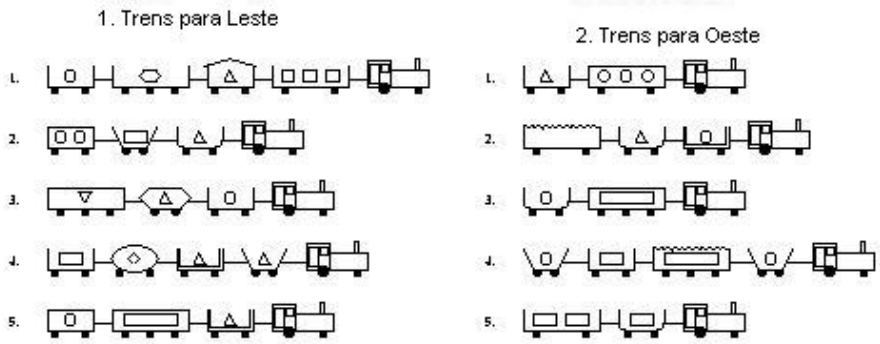

Figura 4: Trens que viajam para Leste e Oeste (Michalski et al., 1989)

o conhecimento de fundo e as diretivas para a execução do Aleph. A primeira diretiva `set/2` indica que as regras que constituem a hipótese devem cobrir, no mínimo, dois exemplos positivos. Exceto essa definição de `minpos`, cujo valor padrão é 1, Aleph foi executado com os parâmetros padrão. O conceito a ser aprendido é denominado `eastbound/1`, portanto deve ser declarado com `modeh/2`, e os predicados que podem aparecer na regra devem ser declarados com `modeb/2`, utilizando os tipos de dados dos argumentos desses predicados, definidos mais a frente, bem como as instanciações desses argumentos, como mostrado a seguir.

Ex.(+)	`trens.f`	Ex.(-)	`trens.n`
1	`eastbound(east1).`	6	`eastbound(west6).`
2	`eastbound(east2).`	7	`eastbound(west7).`
3	`eastbound(east3).`	8	`eastbound(west8).`
4	`eastbound(east4).`	9	`eastbound(west9).`
5	`eastbound(east5).`	10	`eastbound(west10).`

Tabela 1: Arquivos `trens.f` e `trens.n`

```
:- set(minpos,2).
:- modeh(1,eastbound(+train)).
:- modeb(1,short(+car)).
:- modeb(1,closed(+car)).
:- modeb(1,long(+car)).
:- modeb(1,open_car(+car)).
:- modeb(1,double(+car)).
:- modeb(1,jagged(+car)).
:- modeb(1,shape(+car,#shape)).
:- modeb(1,load(+car,#shape,#int)).
:- modeb(1,wheels(+car,#int)).
:- modeb(*,has_car(+train,-car)).
```

Também devem ser declarados os `determination/2`, especificando os predicados que

podem aparecer na cabeça e no corpo das regras com essa cabeça, como mostrado a seguir.

```
:- determination(eastbound/1,short/1).
:- determination(eastbound/1,closed/1).
:- determination(eastbound/1,long/1).
:- determination(eastbound/1,open_car/1).
:- determination(eastbound/1,double/1).
:- determination(eastbound/1,jagged/1).
:- determination(eastbound/1,shape/2).
:- determination(eastbound/1,wheels/2).
:- determination(eastbound/1,has_car/2).
:- determination(eastbound/1,load/3).
```

Para definir o tipo de dados `car/1` enumerando os possíveis valores que pode assumir, esses valores foram definidos considerando primeiro o trem e segundo o vagão. Por exemplo `car(car_14)` refere-se ao valor do tipo `car` para o trem 1 vagão 4. Também são usados `shape/1` que define os possíveis formatos das figuras relacionadas às propriedades dos vagões, bem como `train/1` que define os trens.

```
car(car_11).  car(car_12).  car(car_13).  car(car_14).
car(car_21).  car(car_22).  car(car_23).
car(car_31).  car(car_32).  car(car_33).
car(car_41).  car(car_42).  car(car_43).  car(car_44).
car(car_51).  car(car_52).  car(car_53).
car(car_61).  car(car_62).
car(car_71).  car(car_72).  car(car_73).
car(car_81).  car(car_82).
car(car_91).  car(car_92).  car(car_93).  car(car_94).
car(car_101).  car(car_102).

shape(elipse).  shape(hexagon).  shape(rectangle).  shape(u_shaped).
shape(triangle). shape(circle). shape(nil).

train(east1).  train(east2).  train(east3).  train(east4).  train(east5).
train(west6).  train(west7).  train(west8).  train(west9).  train(west10).
```

O próximo passo consiste em definir cada um dos 10 trens. Por exemplo, o trem 1 é definido como

```
short(car_12).            % vagão 2 é curto
```

```
closed(car_12).            % vagão 2 é fechado
long(car_11).              % vagão 1 é longo
long(car_13).              % vagão 3 é longo
short(car_14).             % vagão 4 é longo
open_car(car_11).          % vagão 1 é aberto
open_car(car_13).          % vagão 3 é aberto
open_car(car_14).          % vagão 4 é aberto
shape(car_11,rectangle).   % vagão 1 tem forma rectangular
shape(car_12,rectangle).   % vagão 2 tem forma rectangular
shape(car_13,rectangle).   % vagão 3 tem forma rectangular
shape(car_14,rectangle).   % vagão 4 tem forma rectangular
load(car_11,rectangle,3).  % vagão 1 carrega 3 retângulos
load(car_12,triangle,1).   % vagão 1 carrega 1 triângulo
load(car_13,hexagon,1).    % vagão 1 carrega 1 hexágono
load(car_14,circle,1).     % vagão 1 carrega 1 círculo
wheels(car_11,2).          % vagão 1 tem 2 rodas
wheels(car_12,2).          % vagão 1 tem 2 rodas
wheels(car_13,3).          % vagão 1 tem 3 rodas
wheels(car_14,2).          % vagão 1 tem 2 rodas
has_car(east1,car_11).     % o vagão 1 é do trem east1
has_car(east1,car_12).     % o vagão 2 é do trem east1
has_car(east1,car_13).     % o vagão 3 é do trem east1
has_car(east1,car_14).     % o vagão 4 é do trem east1
```

Os outros 9 (nove) trens devem ser descritos de maneira semelhante no arquivo `trens.b`. Considerando que todos os trens estão descritos nesse arquivo, os três arquivos são lidos executando `read_all/1` como mostrado a seguir.

```
?- read_all(train).
```

Com esse comando também é possível informar o caminho onde Aleph deve consultar os arquivos. Por exemplo:

```
?- read_all('/home/mariza/examples-aleph/trains/train').
```

Após consultar os arquivos, Aleph está pronto para induzir uma teoria. Para que o Aleph execute seu algoritmo básico, utiliza-se o comando `induce/0`:

```
?- induce.
```

Seguindo os passos do algoritmo básico de funcionamento o primeiro passo é a seleção de um exemplo para ser generalizado, gerando a seguinte execução:

```
[select example] [1]
[sat] [1]
[eastbound(east1)]
```

`[select example]` indica o número do exemplo selecionado para ser generalizado. `[sat]` indica o número do exemplo que será utilizado no passo de saturação e na terceira linha é apresentado o exemplo selecionado; neste exemplo `[eastbound(east1)]`.

O segundo passo é a construção da cláusula mais específica que implica o exemplo selecionado no passo anterior e que esteja de acordo com as restrições da linguagem. Normalmente é uma cláusula com muitos literais, a qual, como já mencionado, é chamada de cláusula *bottom* e o passo é chamado de saturação:

```
[bottom clause] eastbound(A) :- has_car(A, B), has_car(A, C),
has_car(A, D), has_car(A, E),short(B), short(D), closed(D),
long(C),long(E), open_car(B), open_car(C), open_car(E), shape(B,
rectangle), shape(C, rectangle), shape(D, rectangle), shape(E,
rectangle),wheels(B, 2), wheels(C, 3), wheels(D, 2), wheels(E, 2),
load(B, circle, 1), load(C, hexagon, 1), load(D, triangle, 1),
load(E, rectangle, 3).
[literals] [25]
[saturation time] [0.0200288]}
```

`[literals]` indica o número de literais presentes na cláusula *bottom* e `[saturation time]` o tempo gasto para a construção da cláusula *bottom*.

Para generalizar apenas um único exemplo, utiliza-se o comando `sat(V)`, onde `V` é o número do exemplo que se deseja generalizar. Esse comando apenas encontrará a cláusula *bottom* para o exemplo escolhido. Caso se deseje encontrar a *best clause* gerada por essa cláusula, utiliza-se o comando `reduce`. Depois da redução, pode-se ainda utilizar os comandos `covers` e `coversn` que mostram, respectivamente, os exemplos positivos e os exemplos negativos cobertos pela *best clause*.

O terceiro passo é a busca, implementada por um algoritmo *Branch-and-Bound*. Nesse passo encontra-se uma cláusula mais geral que a cláusula *bottom*. Isso é feito buscando-se por algum subconjunto de literais da cláusula *bottom* que tenha melhor *score*, ou seja, cubra o maior número de exemplos positivos e o menor número de negativos. Como mencionado, esse é o passo chamado de redução:

```
[reduce]
[best label so far] [[1, 0, 2, 1]/0]
```

```
eastbound(A).
[5/5]
eastbound(A) :-
   has_car(A, B).
[5/5]
 eastbound(A) :-
   has_car(A, B).
[5/5]
 eastbound(A) :-
   has_car(A, B).
   ...

[-------------------------------------]
[found clause]
 eastbound(A) :-
   has_car(A, B), short(B), closed(B).
[pos-neg] [5]
[clause label] [[5, 0, 4, 5]]
[clauses constructed] [100]
[-------------------------------------]
[clauses constructed] [100]
[search time] [0.0801152]
[best clause] eastbound(A) :-
   has_car(A, B), short(B), closed(B).
[pos-neg] [5]
[atoms left] [0]
}
```

No passo de redução, como já mencionado, busca-se a cláusula com melhor *score* e a cada cláusula construída nessa busca é apresentado o número de exemplos positivos e negativos cobertos pela cláusula. Na execução acima `[5/5]` indica que a cláusula cobre 5 exemplos positivos e 5 negativos.

Em `[foundclause]` é apresentada uma cláusula com um bom *score*. `[pos-neg]` apresenta a cobertura da cláusula se a função de avaliação é cobertura (`coverage`). Por exemplo, se a função de avaliação utilizada é entropia, então será apresentado o nome da função [`entropy`] e o valor para a cláusula.

`[clause label]` é apresentada como `[[P,N,L,Val]]`, onde `P` e `N` representam, respectivamente, o número de exemplos positivos e negativos cobertos pela cláusula; `L` indica o número de literais da cláusula sendo que o valor de `L` é menor ou igual ao valor do parâmetro `clauselength`; `Val` indica o *score* da cláusula.

`[clauses constructed]` indica o número de nós de busca explorados para encontrar a cláusula. `[best clause]` apresenta a melhor cláusula encontrada, entre todas as `[found clause]` encontradas na busca, e que será adicionada à teoria final seguida da sua cobertura (`[pos-neg]`) e `[atoms left]` indica o número de exemplos positivos que não foram cobertos pela cláusula.

O quarto passo é a remoção dos exemplos redundantes. A cláusula com melhor *score* é adicionada à teoria e todos os exemplos positivos cobertos por essa cláusula são removidos. Esse passo é chamado de *cover removal* e o resultado final será a teoria encontrada. Também é apresentada a matriz de confusão utilizando o próprio conjunto de treinamento:

```
[theory]

[Rule 1] [Pos cover = 5 Neg cover = 0]
eastbound(A) :-
   has_car(A, B), short(B), closed(B).

[Training set performance]
          Actual
        +          -
     +  5          0          5
Pred
     -  0          5          5
        5          5         10

Accuracy = 1
[Training set summary] [[5, 0, 0, 5]]
[time taken] [0.110158]
[total clauses constructed] [100]
```

Para a teoria encontrada são apresentadas as regras e, para cada regra, são apresentados os valores de `Pos cover` e `Neg cover` que indicam, respectivamente, o número de exemplos positivos e negativos cobertos pela regra. `[Training set performance]` apresenta a matriz de confusão na seguinte forma:

```
[Training set performance]
          Actual
        +          -
     +  TP         FP      PP
Pred
     -  FN         TN      PN
        P          N       Total
```

onde `TP` e `FP` representam, respectivamente, os verdadeiros positivos e falsos positivos, e `TN` e `FN` os verdadeiros negativos e falsos negativos; `P` indica o número total de exemplos positivos (`P=TP+FN`) e `N` o número total de exemplos negativos (`N=FP+TN`); `PP` representa o número de exemplos classificados como positivos (`PP=TP+FP`) e `PN` o número de exemplos classificados como negativos (`PN=FN+TN`); `Total` indica o número total de exemplos (`Total=PP+PN`); `[Training set summary]` representa a matriz de confusão resumida `[[TP,FP,FN,TN]]`; `[time taken]`, o tempo total para construir a teoria e `[total clauses constructed]`, o número de cláusulas construídas até encontrar a teoria final.

Pode-se armazenar a teoria encontrada em um arquivo, alterando o parâmetro `set(record, V)`, cujo valor padrão é `V = false`. Se `V = true`, deve-se fornecer o caminho do arquivo através do parâmetro `:-set(recordfile,V)`, onde `V` é o caminho do arquivo.

3 Uso Avançado do Aleph

Aleph permite construir uma teoria por meio do algoritmo básico, descrito na Seção 2 na página 5. Como dito anteriormente, o algoritmo básico é executado por meio do comando `induce/0`. Além do comando básico, Aleph possui várias outras formas de construir uma teoria. A seguir são apresentados os comandos disponíveis para induzir uma teoria.

- `induce/0` - Constrói uma cláusula por vez. A cada *best clause* encontrada, todos os exemplos positivos cobertos por ela são removidos (*greedy cover removal*);
- `induce_cover/0` - Similar ao `induce/0`, a diferença é que os exemplos positivos cobertos por cada *best clause* não são removidos (sem *greedy cover removal*). A teoria construída por `induce/0` e `induce_cover/0` é dependente da ordem em que os exemplos positivos são apresentados. Uma comparação entre esses dois comandos é apresentada na Seção 3.1;
- `induce_max/0` - Constrói uma cláusula por vez, e assim como o `induce_cover/0` não remove os exemplos positivos cobertos por cada *best clause*. A diferença é que os passos de saturação e redução são realizados para todos os exemplos, ou seja, esses passos continuam para todos os exemplos, mesmo que já se tenha encontrado cláusulas que cobrem todos os exemplos positivos. Devido a essa característica, a ordem dos exemplos positivos não afeta a teoria final. Para tornar a busca mais eficiente, memoriza-se a cobertura da *best clause* encontrada para cada exemplo e, assim, Aleph constrói um conjunto máximo de cobertura como solução;
- `induce_theory/0` - Não constrói uma cláusula por vez como nos casos anteriores. Este comando somente é utilizado quando a função de busca é `rls`, ou seja, para os métodos de busca aleatórios. Detalhes de como utilizar esses métodos e como eles funcionam encontram-se na Seção 3.4 na página 42;
- `induce_tree/0` - Comando utilizado para a construção de árvores de decisão relacionais. Detalhes estão descritos na Seção 3.2 na página 29;
- `induce_constraints/0` - Comando utilizado para realizar a indução de restrições. Esse comando emula o sistema Claudien (Dehaspe et al., 1996), encontrando todas as restrições baseado no conhecimento de fundo. Maiores detalhes podem ser encontrados na Seção 3.3 na página 35;

- `induce_incremental/0` - Constrói a teoria com uma forma de aprendizado incremental, ao contrário dos outros comandos disponíveis que realizam aprendizado em *batch*, no qual todos os exemplos e o conhecimento de fundo devem ser fornecidos antes que o aprendizado comece.

- `induce_modes/0` - Faz a extração de modos e tipos automaticamente a partir do conhecimento do domínio e declarações dos `determinations` fornecidos.

Aleph também possui várias estratégias de busca que podem ser utilizadas, tais como, `ar` que implementa uma forma simplificada de regras de associação — Seção 3.3 — , `rls` e `scs` para serem utilizadas pelos métodos de busca aleatória — Seção 3.4. Este parâmetro é ajustado utilizando `set(search,Strategy)`. Alguns dos métodos de busca disponíveis são apresentados ao longo deste trabalho. Detalhes de outras estratégias podem ser encontradas em (Srinivasan, 2000).

Além da forma como as teorias podem ser construídas e das estratégias de busca, Aleph permite modificar a função de avaliação utilizada. Este parâmetro é ajustado utilizando `set(evalfn,V)`. `V` pode ser `coverage`, `compression`, `posonly`, `pbayes`, `accuracy`, `laplace`, `auto_m`, `mestimate`, `entropy`, `gini`, `sd`, `wracc` ou `user` (padrão é `coverage`). Maiores detalhes sobre as funções de avaliação podem ser encontrados em (Srinivasan, 2000).

Dessa maneira, por meio dos diversos parâmetros que Aleph tem disponível, além da característica de permitir que se altere as funções de busca, funções de avaliação e dos diferentes comandos para construir as teorias, é possível emular alguns sistemas de PLI, construindo diferentes teorias, tais como, regras de associação relacionais, árvores de decisão relacionais e indução de restrições, entre outras.

3.1 Comparação entre a Construção de Teorias induce/0 e induce_cover/0

A diferença entre os comandos `induce/0` e `induce_cover/0` pode ser crucial para o aprendizado do conceito. Considere o conjunto de exemplos na Tabela 2. Claramente, uma possível hipótese correspondente a esses exemplos seria:

$$\text{Se At-1} \leq 4 \text{ então classe} = +.$$

Considere o seguinte arquivo de conhecimento do domínio `.b`:

Ex.	At-1	Classe
1	2	+
2	3	+
3	4	+
4	6	-
5	7	-
6	8	-

Tabela 2: Conjunto de exemplos

```
:-modeh(1,class(+att_id,+class)).
:-modeb(*,peso(+att_id,-peso)).
:-modeb(*,menor_igual(+peso,#peso)).

:-determination(class/2,peso/2).
:-determination(class/2,menor_igual/2).

class(pos).
class(neg).

menor_igual(X,Y):-
var(Y), !,
X = Y.

menor_igual(X,Y):-
number(X), number(Y),
X =< Y.

peso(1,2).
peso(2,3).
peso(3,4).
peso(4,6).
peso(5,7).
peso(6,8).
peso(7,4).
```

o arquivo de exemplos positivos `.f`

```
class(1,pos).
class(2,pos).
class(3,pos).
```

e o arquivo de exemplos negativos `.n`.

```
class(4,neg).
```

```
class(5,neg).
class(6,neg).
```

Executando o comando básico `induce/0`, o conceito não é aprendido. Isso acontece porque esse comando remove os exemplos positivos cobertos por cada *best clause* encontrada. Analisando a execução, é possível observar que Aleph constrói, para o primeiro exemplo positivo, a seguinte cláusula *bottom*:

```
class(A,B) :-
   peso(A,C), menor_igual(C,2).
```

O processo de redução dessa cláusula não encontrará uma regra que cubra mais exemplos positivos, pois como pode ser observado na Tabela 2, todos os outros têm peso maior que 2. E assim, a *best clause* encontrada pelo sistema (que cobre o exemplo e tem o menor número de literais) é `class(1,pos)`, ou seja, o próprio exemplo. Como ainda faltam dois outros exemplos a serem cobertos, satura-se o segundo exemplo, cuja cláusula *bottom* encontrada é:

```
class(A,B) :-
   peso(A,C), menor_igual(C,3).
```

Com essa cláusula, o sistema poderia encontrar uma regra que cobrisse também o primeiro exemplo positivo `class(1,pos)`, que tem peso menor que 3. Porém, esse exemplo foi removido da base de exemplos positivos, pois o sistema já encontrou uma *best clause* para cobri-lo. Novamente, a *best clause* encontrada pelo sistema cobre apenas o exemplo atual `class(1,pos)`. O mesmo acontece com o próximo exemplo `class(3,pos)`, que poderia cobrir os outros 2 exemplos se o sistema não os tivesse removido da base de exemplos positivos. Como consequência, a teoria encontrada pelo sistema consiste simplesmente dos exemplos positivos.

```
[theory]

[Rule 1] [Pos cover = 1 Neg cover = 0]
class(1,pos).

[Rule 2] [Pos cover = 1 Neg cover = 0]
class(2,pos).

[Rule 3] [Pos cover = 1 Neg cover = 0]
```

```
class(3,pos).

[Training set performance]
         Actual
       +        -
     + 3        0        3
Pred
     - 0        3        3

       3        3        6
```

Utilizando o comando `induce/0`, o sistema induziria uma teoria mais geral apenas se o terceiro exemplo `class(3,pos)` estivesse em primeiro lugar no arquivo `.f`. Dessa forma, o sistema ainda não teria removido os exemplos positivos com peso menor ou igual a 4. Em contrapartida, executando o comando `induce_cover/0` obtém-se uma teoria mais geral.

```
[theory]

[Rule 1] [Pos cover = 1 Neg cover = 0]
class(1,pos).

[Rule 2] [Pos cover = 2 Neg cover = 0]
class(A,B) :-
   peso(A,C), menor_igual(C,3).

[Rule 3] [Pos cover = 3 Neg cover = 0]
class(A,B) :-
   peso(A,C), menor_igual(C,4).

[Training set performance]
         Actual
       +        -
     + 3        0        3
Pred
     - 0        3        3

       3        3        6
```

Apesar de parecer mais vantajosa, a execução do Aleph com o comando `induce_cover/0` é mais lenta. Além disso, produz cláusulas redundantes, ou seja, que cobrem os mesmos exemplos. Essa característica torna necessária a utilização de um programa de separação para encontrar algum subconjunto dessas cláusulas que minimize essa redundância,

como por exemplo o sistema *T-Reduce*. Uma cópia desse sistema está disponível em: http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/misc/treduce.pl

3.2 Indução de Árvores de Decisão Relacionais

O sistema Aleph é capaz de fazer a indução de árvores de decisão relacionais por meio da emulação das funcionalidades do sistema de PLI Tilde (Blockeel, 1998). O Tilde, versão relacional do $\mathcal{C}4.5$ Quinlan (1988), é um dos mais famosos algoritmos para indução de árvores relacionais.

Um exemplo de uma árvore relacional é apresentado na Figura 5. Essa árvore prediz a ação de manutenção A, que deve ser realizada na máquina M (*manutenção(M,A)*), baseada nas partes que a máquina possui (*temparte(M,X)*), sua condição (*usado(X)*) e a facilidade de substituição (*insubstituivel(X)*). O predicado alvo é *manutenção(M,A)*, a variável classe é A e o conhecimento de fundo consiste dos predicados *temparte(M,X)*, *usado(X)* e *insubstituivel(X)*.

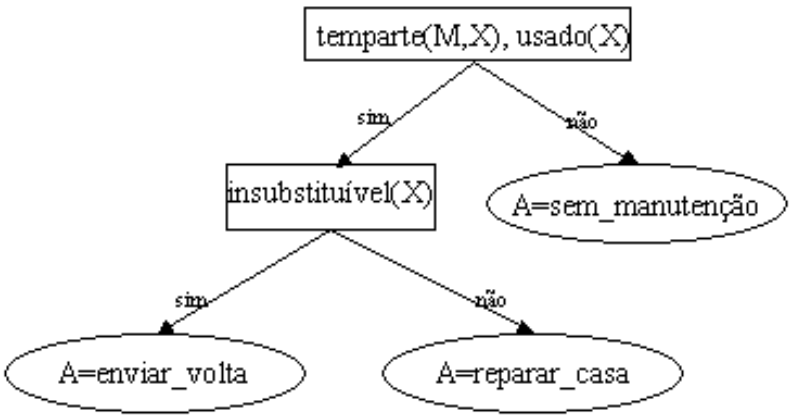

Figura 5: Árvore de decisão relacional para predição da variável classe A do predicado alvo *manutenção(M,A)*

Árvores de decisão relacionais possuem uma estrutura muito parecida com as árvores de decisão proposicionais. Os nós internos contêm testes e os nós folha as predições para os valores da classe. Caso a variável classe seja discreta ou contínua, a árvore será, respectivamente, para problemas de classificação ou regressão. A árvore da Figura 5 é uma árvore de classificação relacional. A maior diferença entre as árvores relacionais e proposicionais está nos testes que aparecem nos nós internos. No caso relacional os testes são consultas lógicas (*queries*) (Lavrač & Džeroski, 2001).

As árvores relacionais são árvores binárias. Cada nó interno tem um ramo esquerdo (sim), ou seja, uma resposta/substituição que torne essa consulta verdadeira, e um ramo direito (não) caso contrário. É importante notar que as variáveis podem ser compartilhadas entre os nós. Uma variável introduzida em um nó pode ser referenciada em uma sub árvore esquerda daquele nó. Por exemplo, na árvore da Figura 5, a variável X em *insubstituivel(X)* refere-se a uma parte X da máquina M, introduzida no teste *temparte(M,X)*, *usado(X)* no nó raíz. O teste a ser executado em um nó é a conjunção dos literais do próprio nó com os literais no caminho da raíz até o nó em questão. No exemplo da Figura 5, os testes que levam ao nó folha *A=enviar_volta* são: *temparte(M,X)*, *usado(X)*, *insubstituivel(X)*. Em outras palavras, a máquina precisa ser enviada de volta para manufatura somente se tem alguma parte dela que seja usada e é insubstituivel (Džeroski, 2003).

Assim, uma árvore de decisão relacional não pode ser transformada em um programa lógico que consiste de uma cláusula por folha, ao contrário da árvore de decisão proposicional, na qual a transformação de uma regra de decisão por folha é possível. Árvores relacionais podem ser facilmente transformadas em listas de decisão de primeira ordem, ou seja, conjuntos de cláusulas ordenadas (as cláusulas de programas lógicos são não-ordenadas). Quando se está aplicando uma lista de decisão a um exemplo, sempre utiliza-se a primeira cláusula que se aplica e é produzida uma resposta. Quando aplicando um programa lógico, todas as cláusulas aplicáveis são usadas e um conjunto de respostas pode ser produzido.

Uma lista de decisão é produzida utilizando busca em profundidade *depth-first* para visitar os nós da árvore relacional, *i.é.*, descendo primeiro pelo ramo da esquerda. Cada folha representa uma predição e todas as condições ao longo do ramo esquerdo levam àquela folha. Uma lista de decisão obtida da árvore na Figura 5 é apresentada a seguir:

manutenção(M,X)← *temparte(M,X)*,*usado(X)*, *insubstituivel(X)*, !, *A=enviar_volta.*

manutenção(M,X)←*temparte(M,X)*,*usado(X)*,!, *A=reparar_casa.*

manutenção(M,X)← *A=sem_manutenção.*

ordem para assegurar que somente a primeira cláusula aplicável foi utilizada. Para a primeira cláusula apresentada na lista de decisão, as condições em ambos os nós internos são válidas, ou seja, todos os ramos que levam à folha *enviar_volta* estão na esquerda (resposta sim). Na segunda cláusula, como a primeira não foi verdadeira, para alcançar o nó folha *reparar_casa*, segue-se pelo ramo esquerdo (sim), o que repre-

senta que a condição do nó *temparte(M,X)* é válida; em seguida esse ramo leva ao nó *insubstituivel(X)*, no qual o ramo de saída que vai levar à folha é o direito (não), o que representa que a condição do nó *insubstituivel(X)* não é válida. Assim, para a segunda cláusula, somente a condição da raíz é válida.

Observe que o algoritmo básico do sistema realiza a classificação de apenas uma classe. Mesmo que se tenha um problema com mais de uma classe, apenas uma delas será positiva, e todas as outras serão negativas. A construção de uma teoria baseada em árvores, porém, classificará todas elas, sendo mais vantajosa para resolver problemas multiclasses.

Aleph realiza a construção de uma teoria baseada em árvores relacionais utilizando o comando:

```
induce_tree.
```

e baseado em uma abordagem de particionamento recursivo para construir a árvore, que pode ser de 3 tipos: classificação, regressão ou *class probability*. O procedimento básico constrói uma árvore para predizer o argumento de saída do predicado alvo. Dessa forma, a cabeça da cláusula deve ter apenas um único argumento do tipo saída. Além disso, esse argumento não deve estar presente na declaração de tipos, pois os valores que pode assumir são especificados com o parâmetro `:-set(classes, list)`, explicado com mais detalhes adiante.

A seguir é apresentado um exemplo no qual, dados nomes de animais, deseja-se predizer se são mamíferos ou não. São apresentados alguns exemplos e a declaração do `modeh/2`.

```
%Exemplos
class(cachorro,mamifero).
class(gato, mamifero).
class(tartaruga,n_mamifero).

:- modeh(1,class(+animal,-class)).
```

É importante observar que todos os exemplos são colocados no arquivo `.f`, e não é necessário declarar `set(evalfn,posonly)`. No exemplo descrito, tanto os exemplos da classe `mamifero` como os da classe `n_mamifero` são colocados no mesmo arquivo.

Para problemas de classificação, é realizada a poda no conjunto final de cláusulas da árvore e a técnica utilizada é o método *reduced-error* proposto por Quinlan (1988).

Para os problemas de regressão, a poda é feita utilizando uma estimativa pessimista (*pessimistic estimate*) e para *class probability* a poda não é utilizada.

No processo de aprendizado com árvores relacionais os caminhos da raiz até as folhas constituem cláusulas, assim, em todos os casos o resultado é apresentado como uma seqüência de regras. Para que o Aleph possa induzir a árvore relacional, alguns parâmetros precisam ser ajustados:

`set(tree_type,+V)` Este parâmetro ajusta o tipo de árvore que será utilizado no aprendizado. `V` é `classification`, `class_probability` ou `regression`;

`set(classes,+V)` `V` é uma lista de classes a ser pretida pela árvore. É usada somente para classificação (`classification`) ou *class probability*(`class_probability`).

Exemplo: `set(classes,[mamifero,n_mamifero]).`;

`set(evalfn,+V)` Especifica qual a função de avaliação que será usada na busca. Para árvores de classificação e *class probability* esse parâmetro deve ser ajustado para `entropy` ou `gini`. No caso de árvores de regressão, `V` é ajustado automaticamente por Aleph para `sd` (*standard deviation*);

Além desses parâmetros obrigatórios, há outros parâmetros relevantes, tais como:

`set(prune_tree,+V)` `V` é `auto` ou `false` (padrão é `false`). Determina se as regras construídas utilizando o aprendizado de árvore devem ser podadas utilizando poda pessimista;

`set(mingain,+V)` `V` é um número real (padrão é 0.5). Especifica o ganho mínimo esperado para dividir uma folha durante o processo de construção da árvore;

`set(confidence,+V)` `V` é um número real no intervalo de 0.0 a 1.0 (padrão é 0.95). Determina a confiança para poda de regras no aprendizado com árvores. Se ajustado para 0.25 é idêntico ao utilizado pelo $\mathcal{C}4.5$;

`set(lookahead,+V)` `V` é um número inteiro (padrão é 1). Ajusta um valor para o operador automático de refinamento, permitindo que sejam adicionados mais que um literal por vez. Quando induzindo árvores, o parâmetro `lookahead` é utilizado para avaliar a qualidade de uma divisão de um nó na busca, considerando não só a qualidade da cláusula no nó filho, mas também a qualidade dos nós seguintes. Em outras palavras, ajusta a profundidade analisada na busca antes de realizar o refinamento;

`set(minpos,+V)` `V` é um inteiro positivo (padrão é 1). Para árvores, esse parâmetro especifica o número mínimo de exemplos que deve haver em uma folha para que ela seja dividida.

Além desses parâmetros que são ajustados pelo usuário, o parâmetro `refine` é ajustado automaticamente pelo sistema para `auto`. Além disso, o Aleph versão 5, com data de modificação do dia 4 de junho de 2006, requer a utilização do parâmetro `set(dependent,+V)`, onde `V` é o inteiro positivo que representa o argumento da variável dependente nos exemplos, ou seja, `V` é o número do argumento do predicado alvo que representa a classe. Por exemplo, se o argumento que representa a classe é o primeiro, `V` = 1. Note que a especificação desse parâmetro fornece mais liberdade para a elaboração do predicado alvo, pois pode-se indicar qualquer argumento para representar a classe, não necessariamente um argumento de saída.

A seguir é apresentado um exemplo de como utilizar o Aleph para indução de árvores de decisão relacionais. Utilizando o arquivo `animals.b`, apresentado a seguir, no qual estão definidos os parâmetros, as declarações de modos, determinations, tipos e algumas declarações de conhecimento de fundo:

```
% Especificando o tipo da árvore
:- set(tree_type,classification).
:- set(classes,[mammal,nmammal]).
:- set(minpos,2).
:- set(prune_tree,auto).
:- set(confidence,0.25).
:- set(evalfn,entropy).
:- set(lookahead,2).

:- modeh(1,class(+animal,-class)).
:- modeb(1,has_gills(+animal)).
:- modeb(1,has_covering(+animal,#covering)).
:- modeb(1,has_legs(+animal,#nat)).
:- modeb(1,homeothermic(+animal)).
:- modeb(1,has_eggs(+animal)).
:- modeb(1,not(has_gills(+animal))).
:- modeb(1,nhas_gills(+animal)).
:- modeb(*,habitat(+animal,#habitat)).
:- modeb(1,has_milk(+animal)).

:-determination(class/2,has_gills/1).
:-determination(class/2,has_covering/2).
:-determination(class/2,has_legs/2).
```

```
:-determination(class/2,momeotermic/1).
:-determination(class/2,has_egss/1).
:-determination(class/2,nhas_gills/1).
:-determination(class/2,habitat/2).
:-determination(class/2,has_milk/1).

% Declaração dos Tipos
animal(dog).  animal(dolphin).  animal(platypus).  animal(bat).
animal(trout).  animal(herring).  animal(shark). animal(eel).
animal(lizard).  animal(crocodile).  animal(t_rex).  animal(turtle).
animal(snake).  animal(eagle).  animal(ostrich).  animal(penguin).
animal(cat). animal(dragon).  animal(girl).  animal(boy).
class(mammal).  class(fish).  class(reptile).  class(bird).
covering(hair).  covering(none).  covering(scales).  covering(feathers).
habitat(land).  habitat(water).  habitat(air).  habitat(caves).

%Conhecimento de Fundo
has_covering(dog,hair). has_covering(dolphin,none).
has_covering(platypus,hair). has_covering(bat,hair).

has_legs(dog,4). has_legs(dolphin,0). has_legs(platypus,2).
has_milk(dog). has_milk(cat). has_milk(dolphin).

homeothermic(dog). homeothermic(cat). homeothermic(dolphin).
homeothermic(platypus).

habitat(dog,land). habitat(dolphin,water).
habitat(platypus,water). habitat(bat,air).

has_eggs(platypus). has_eggs(trout). has_eggs(herring).
has_eggs(shark).

has_gills(trout). has_gills(herring). has_gills(shark).
has_gills(eel).

nhas_gills(X) :- animal(X), not(has_gills(X)).
```

Note que todos os tipos foram especificados no arquivo `.b`. No aprendizado de árvores relacionais, é necessário que se especifique todos os tipos declarados nos argumentos dos predicados. Diferentemente do aprendizado básico do sistema, o aprendizado de árvores ignora predicados que possuem algum de seus tipos (que podem ser de entrada, saída ou constantes) não especificados.

Os exemplos no arquivo `animal.f`, são apresentado a seguir.

```
class(bat,mammal).
class(dog,mammal).
class(platypus,mammal).
class(eagle,nmammal).
class(eagle,nmammal).
class(ostrich,nmammal).
class(shark,nmammal).
class(crocodile,nmammal).
class(shark,nmammal).
```

O resultado final da execução do Aleph é:

```
[theory]

[Rule 1] [Pos cover = 6 Neg cover = 0]
class(A,B) :-
   has_covering(A,hair), B=mammal.

[Rule 2] [Pos cover = 14 Neg cover = 0]
class(A,B) :-
   not has_covering(A,hair), B=nmammal.

[Training set performance]
          Actual
       +        -
     + 20        0        20
Pred
     - 0        0        0

       20        0        20

Accuracy = 1.0
[Training set summary] [[20,0,0,0]]
```

É importante saber que todos os experimentos realizados para a construção de árvores de decisão relacionais utilizando Yap-5.1.1 e Aleph versão 5, com data de modificação de 6 de novembro de 2005, não geraram teoria.

3.3 Regras de Associação Relacional

Regras de associação constituem uma das técnicas mais utilizadas em mineração de dados quando não se conhece o predicado alvo do problema. As regras de associação

caracterizam o quanto a presença de um conjunto de itens nos registros (exemplos) de uma base de dados implica a presença de algum outro conjunto distinto de itens nos mesmos registros (Dehaspe & Toivonen, 2001). Um exemplo típico de utilização de regras de associação proposicionais é apresentado na Figura 6.

Figura 6: Exemplo de uma regra de associação

O formato de uma regra de associação é:

$$\text{LHS} \Longrightarrow \text{RHS [suporte, confiança]}$$

onde LHS (Left Hand Side) representa o lado esquerdo da regra e RHS (Right Hand Side) o lado direito, sendo que LHS e RHS são definidos por conjuntos disjuntos de itens. Existem duas medidas que estão sempre associadas a uma regra de associação: o suporte e a confiança. O suporte quantifica a incidência da regra no conjunto de dados e representa a freqüência com que LHS e RHS ocorrem juntos no conjunto de dados. A confiança indica a freqüência com que LHS e RHS ocorrem juntos em relação ao número total de registros nos quais LHS ocorre, ou seja, representa a probabilidade de ocorrência de RHS dado que LHS ocorre.

Nos casos em que a base de dados utilizada for relacional, regras de associação relacional poderão ser descobertas. Nesse caso, consultas *Datalog* podem ser vistas como a versão relacional dos itemsets. Um exemplo de consulta é:

```
? - pessoa(X), pais(X,Y), tem_animal_estimação(Y,Z).
```

Considere o itemset {pessoa,pais,filho,animal_estimação} para regras de associação proposicionais. Uma interpretação desse padrão é que pessoa, pais, filho e animal_estimação

ocorrem juntos. Este é o mesmo significado da consulta anterior, entretanto, as variáveis `X`, `Y` e `Z` adicionam informação extra: pessoa e pais são os mesmos, pais e filhos pertencem à mesma família e os animais de estimação pertencem aos filhos. Isso ilustra o fato de que as consultas são mais expressivas que os itemsets.

A tarefa de descobrir padrões relacionais que estejam associados entre si foi endereçada ao sistema relacional WARMR (Dehaspe & De Raedt, 1997). O WARMR não gera itemsets, apenas padrões freqüentes. As regras geradas a partir de consultas freqüentes são chamadas de consultas estendidas. O WARMR recebe como entrada uma base de dados, uma freqüência mínima e um *bias* de linguagem L. O *bias* de linguagem especifica um átomo chave e modos de entrada e saída para os predicados ou relações.

Para a descoberta de padrões freqüentes é necessária a noção de freqüência. Considerando as consultas como padrões e considerando que as consultas podem ter variáveis, não é muito óbvia qual a freqüênia de uma consulta. Isso é resolvido especificando-se um parâmetro adicional para a tarefa de descoberta de padrões, chamado de chave. A chave é um átomo que deve estar presente em todas as consultas durante o processo de descoberta, pois ele determina o que contar. Se `pessoa(X)` é a chave, contam-se pessoas; se `pais(X,Y)` é a chave, contam-se pares(pais,filhos). Assim, utilizando o parâmetro chave e fazendo essa contagem encontra-se o suporte e a confiança das regras de associação relacional. Esse parâmetro é importante uma vez que se torna possível mudar o foco da aplicação sem que as entradas e, em particular, a base de dados, sejam modificadas (Džeroski, 2003).

Para se obter regras de associação relacionais utilizando o Aleph, pode-se utilizar uma das duas abordagens existentes. Uma das abordagens é utilizar a busca `ar` (`set(search,ar)`), que é uma emulação do WARMR. Entretando, vale notar que o Aleph não faz o mesmo que o WARMR. O que o Aleph faz é rotular como *good clause* todas as regras encontradas, durante o passo de redução e que cobrem um número maior ou igual a uma certa fração dos exemplos positivos. Essa fração é definida pelo parâmetro `set(pos_fraction,V)` que deve ter seu valor ajustado entre 0.0 e 1.0.

A seguir é apresentado um exemplo de como utilizar o Aleph para emular o WARMR. No arquivo `customer.b`, mostrado a seguir, é definido o *bias* de linguagem, o conhecimento de fundo e os tipos, entre outros. A definição do `modeh/2` pode ser vista como a definição do parâmetro chave utilizada pelo WARMR, e todos os argumentos do predicado alvo devem ser do tipo de entrada.

```
:- set(search,ar).
:- set(pos_fraction,0.5).
```

```
:- modeh(*,customer(+person)).
:- modeb(1,parent(+person,-person)).
:- modeb(1,parent(-person,+person)).
:- modeb(*,buys(+person,#product)).
:- modeb(*,nbuys(+person,#product)).
:- modeb(1,not(parent(+person,+person))).

:- determination(customer/1,person/1).
:- determination(customer/1,parent/2).
:- determination(customer/1,buys/2).
:- determination(customer/1,(not)/1).
:- determination(customer/1,nbuys/2).

nbuys(A,B):- person(A), product(B),
             not(buys(A,B)).

person(allen). person(bill). person(carol).
person(diana). person(zoe).
product(wine). product(cola). product(pizza).
parent(allen,bill). parent(allen,carol).
parent(bill,zoe). parent(carol,diana).
buys(allen,wine). buys(bill,cola).
buys(bill,pizza). buys(diana,pizza).
```

No caso dessa abordagem, não são utilizados exemplos negativos, apenas exemplos positivos. Um exemplo desse arquivo é apresentado a seguir no arquivo `customer.f`.

```
customer(allen).
customer(bill).
customer(carol).
customer(diana).
```

Os arquivos são lidos utilizando `read_all(costumer)` e em seguida a indução pode ser ativada utilizando o comando `induce/0`. Ao final são apresentadas as cláusulas rotuladas como `[good clause]`, a teoria e a matriz de confusão. Perceba que todas as cláusulas rotuladas como `[good clause]` cobrem um número maior ou igual que o valor atribuído ao parâmetro `pos_fraction` multiplicado pela quantidade de exemplos positivos, ou seja, 2 ou mais exemplos.

```
[good clauses]
customer(A).
```

```
[pos-neg] [4]
customer(A) :-
   parent(A,B).
[pos-neg] [3]
customer(A) :-
   parent(B,A).
[pos-neg] [3]
customer(A) :-
   nbuys(A,wine).
[pos-neg] [3]
customer(A) :-
   nbuys(A,pizza).
[pos-neg] [2]
customer(A) :-
   parent(A,B), buys(B,pizza).
[pos-neg] [2]
customer(A) :-
   parent(A,B), nbuys(B,wine).
[pos-neg] [3]

...

[atoms left] [0]
[positive examples left] [0]
[estimated time to finish (secs)] [0.0]

[theory]

[Rule 1] [Pos cover = 4 Neg cover = 0]
customer(A).

[Training set performance]
         Actual
       +        -
     + 4        0        4
Pred
     - 0        0        0

       4        0        4

Accuracy = 1.0
[Training set summary] [[4,0,0,0]]
[time taken] [0.024]
[total clauses constructed] [63]
```

É importante observar que, quando o objetivo é encontrar regras de associação relacionais, o resultado final (`[theory]`) não é o importante, mas sim, todas as associações que o Aleph conseguiu encontrar ao longo da busca, ou seja, as cláusulas `[good clause]` que contêm associações verdadeiras de acordo com os exemplos positivos.

A outra abordagem que pode ser utilizada com o Aleph, consiste em encontrar as associações não verdadeiras contidas na base de dados. Para isso, utiliza-se o comando `induce_constraints/0`. Esse comando emula o sistema Claudien (Dehaspe et al., 1996), encontrando todas as restrições no conhecimento de fundo. As restrições encontradas são da forma `false:-body`. Uma vez que `body` inclui a negação de literais, é possível se obter regras com diferentes símbolos predicados na cabeça.

O exemplo a seguir demonstra o aprendizado de restrições. Note que, para que o Aleph encontre as restrições, o usuário deve fazer declarações de modo para `false` para a cabeça, e `not` para os predicados do corpo que definem o conceito. O predicado do corpo que determina o conceito deve ter apenas um argumento de saída. Além disso, como as restrições são baseadas no conhecimento de fundo, exemplos positivos e negativos não são utilizados. Os valores apresentados por `[pos/neg]` indicam que o Aleph considera os dados do conhecimento do domínio como exemplos negativos (já que se deseja negar aquele conhecimento), e não há exemplos positivos.

```
:- modeh(1,false). %declaração de modo para false

:- modeb(*,humano(-pessoa)). %conceito humano, com apenas uma argumento de saída

:- modeb(1, homem(+pessoa)). %humano é definido pelos predicados homem e mulher
:- modeb(1, mulher(+pessoa)).
:- modeb(1, not(homem(+pessoa))). %declarações de modo para not
:- modeb(1, not(mulher(+pessoa))).

:- determination(false/0,human/1).
:- determination(false/0,male/1).
:- determination(false/0,female/1).
:- determination(false/0,(not)/1).

male('Fred'). %conhecimento de fundo
female('Wilma').
human('Fred').
human('Wilma').
```

As restrições que Aleph encontra são:

```
[constraints]
false :-
   human(A), male(A), female(A).
[pos-neg] [0]
false :-
   human(A), female(A), male(A).
[pos-neg] [0]
false :-
   human(A), not male(A), not female(A).
[pos-neg] [0]
false :-
   human(A), not female(A), not male(A).
[pos-neg] [0]
[total clauses constructed] [27]
```

A primeira e a segunda regras são equivalentes e indicam que não existem humanos que sejam homem e mulher ao mesmo tempo. A terceira e a quarta regras são também equivalentes e indicam que não existem humanos que não sejam nem homens nem mulheres. Encontradas as restrições, o usuário pode, então, remover as restrições redundantes e transformar restrições (transladando para a cabeça os predicados com `not`). Nesse caso, obtém-se as seguintes regras equivalentes às anteriores:

```
%não existe humano que seja homem e mulher ao mesmo tempo
false:- humano(A),homem(A),mulher(A).
%um homem ou uma mulher é um humano
homem(A);mulher(A):- humano(A).
```

Abaixo é apresentado outro exemplo, em que se deseja encontrar restrições para o conceito `ruivo`.

```
:- modeh(1,false).

:- modeb(1,ruivo(-person)).

:- modeb(1,cor_cabelo(+person, #cor)).
:- modeb(1,not(cor_cabelo(+person, #cor))).

:- determination(false/0,ruivo/1).
:- determination(false/0,cor_cabelo/2).
:- determination(false/0,(not)/1).
```

```
person(fred).
person(wilma).

cor(vermelho).
cor(amarelo).
cor(castanho).

ruivo(fred).
ruivo(wilma).

cor_cabelo(fred, vermelho).
cor_cabelo(wilma, vermelho).
```

Algumas das restrições encontradas são:

```
[constraints]
false :-
   ruivo(A), cor_cabelo(A,amarelo).%ruivos não têm cabelo amarelo
[pos-neg] [0]
false :-
   ruivo(A), cor_cabelo(A,castanho).%ruivos não têm cabelo castanho
[pos-neg] [0]
false :-
   ruivo(A), not cor_cabelo(A,vermelho).%ruivos não devem não ter cabelo vermelho
```

O resultado das duas abordagens são associações encontradas na bases de dados, verdadeiras, ou falsas no caso da segunda abordagem apresentada. Os resultados dessas duas abordagens podem ser combinados para encontrar associações interessantes na base de dados.

3.4 Métodos de Busca Estocástica

A forma mais simples de busca aleatória é: dada uma amostra de N elementos (cláusulas ou teorias) do espaço de busca, pontue (*score*) esses elementos e retorne o melhor. Um estudo sobre busca aleatória em PLI pode ser encontrado em (Srinivasan, 1999).

Para realizar a seleção estocástica de cláusulas, deve-se ajustar o seguinte parâmetro: `set(search,scs)`. O número N é ajustado pelo valor associado a `scs_sample/2` ou calculado automaticamente por:

```
set(scs_prob,P)
set(scs_percentile,K)
```

Assim, o tamanho da amostra é calculado por:

$$\frac{log(1-P)}{log\left(1-\frac{K}{100}\right)}$$

Essa fórmula calcula o número de cláusulas que devem ser amostradas, com probabilidade de no mínimo P, em pelo menos uma cláusula nos K-porcento de cláusulas. Isso é especificado ajustando o `set/2 de scs_type` para `blind` ou `informed`. Se `blind`, então as cláusulas são selecionadas uniformemente no espaço de todas as possíveis hipóteses. Se `informed`, então são dirigidas para uma distribuição específica para o tamanho das cláusulas. O tamanho das cláusulas, i.é, o número de sentenças no corpo da cláusula, pode ser pré-especificado pelo parâmetro `clauselength_distribuition` ou, quando esse tamanho não é declarado, então ele é obtido automaticamente por um esquema similar ao Monte-Carlo.

A cláusula inicial aleatoriamente selecionada é freqüentemente construída como segue:

1. Um exemplo é selecionado. O exemplo pode ser selecionado automaticamente pelo sistema ou pelo usuário usando o comando `sat`;
2. A cláusula *bottom* é construída para o exemplo selecionado;
3. Uma cláusula legal é aleatoriamente dirigida para a cláusula *bottom*.

O sistema pode ser ajustado para não construir a cláusula *bottom* ajustando-se o parâmetro `construct_bottom` para `false`; nesse caso a cláusula selecionada pode ser o resultado de uma distribuição sobre o tamanho das cláusulas definido pelo parâmetro // `clauselength_distribuition`.

O método estocástico de busca local (*randomised local search*) usa os algoritmos GSAT (Éric Grégoire & Mazure, 1998), WSAT (Selman et al., 1994), RRR (Zelezny et al., 2002) ou Annealing (Kirkpatrick, Gelatt, & Vecchi, Kirkpatrick et al.) para busca em PLI. Para utilizar esse método, a função de busca é ajustada para: `set(search,rls)` e o tipo de busca aleatória especificado pelo parâmetro `rls_type`, que deve ser ajustado para `gsat`, `wsat`, `rrr` ou `anneal` respectivamente. Quando a busca é `rls` utiliza-se o comando `induce_theory/0`.

Os algoritmos GSAT, WSAT e Annealing utilizam múltiplos reinícios, e cada um desses reinícios serve como ponto inicial para movimentos locais no espaço de busca. O limite do número de reinícios é especificado pelo parâmetro `tries`, ajustado com um valor inteiro positivo, e o número de movimentos por `moves`, ajustado com valor ≥ 0. A implementação do algoritmo WSAT requer ainda um *random-walk probability*, o qual é especificado pelo parâmetro `walk` com um valor entre 0 e 1. Se `walk` for igual a 0 o WSAT será equivalente ao algoritmo GSAT.

O algoritmo de busca *Randomised Rapid Restarts* (RRR) é descrito em (Zelezny et al., 2002). RRR pode utilizar dois diferentes critérios de parada:

1. O número mínimo de exemplos positivos que devem ser encontrados por uma cláusula (`minpos`) e se a utilidade aceitável é alcançada (`minscore`);

2. A precisão mínima é alcançada (`minacc`). Nesse caso a função de avaliação deve ser precisão (`set(evalfn,accuracy)`).

O algoritmo annealing é restringido a usar uma temperatura fixa; a temperatura é ajustada pelo parâmetro `temperature` com valor real > 0.

3.5 Aprendizado Abdutivo

O aprendizado abdutivo pode ser utilizado para completar definições do conhecimento de fundo, assumindo que o conhecimento de fundo está correto, porém incompleto. Esse aprendizado está incorporado no comando `induce/0`, bastando apenas alterar o parâmetro `abduce` para `true` (padrão é `false`) para que seja ativado. Para declarar candidatos à abdução, utiliza-se o comando `abducible(N/A)`, onde `N` é o nome do predicado e `A` é sua aridade.

Considere o seguinte conhecimento de fundo (arquivo `.b`):

```
:-modeh(1, grandfather(+person, -person)).

:-modeb(*, father(+person, -person)).
:-modeb(*, mother(+person, -person)).
:-modeb(*, parent(+person, -person)).

:-determination(grandfather/2, father/2).
```

```
:-determination(grandfather/2, mother/2).
:-determination(grandfather/2, parent/2).

:-set(abduce, true).
:-abducible(parent/2).

person(jane).
person(robert).
person(peter).
person(fred).

father(fred, jane).

mother(jane, robert).
mother(jane, peter).

grandfather(X,Y):-father(X,Z), parent(Z,Y).
parent(X,Y):-father(X,Y).
```

e os seguintes exemplos positivos (arquivo `.f`):

```
grandfather(fred,robert).
grandfather(fred,peter).
```

e negativos (arquivo `.n`):

```
grandfather(robert,peter).
grandfather(jane,peter).
```

Primeiramente, o sistema tenta encontrar outra cláusula para `grandfather/2`, resultando em:

```
grandfather(A,B) :-
   father(A,C), mother(C,B).
```

Então, o sistema encontra cláusulas que definam os predicados declarados como `abducible`. Essas cláusulas têm mais utilidade que a cláusula encontrada acima, pois elas completam o conhecimento do domínio. A cláusula encontrada para o predicado `parent/2` foi:

```
parent(A,B) :-
   mother(A,B).
```

Para melhorar a eficiência, pode-se limitar o número máximo de átomos *ground* em qualquer abdução através do comando `:-set(max_abducibles, +V)`, onde `V` é um inteiro (padrão é 2).

3.6 Aprendizado de *Features*

O aprendizado de *features* pode ser executado através do comando `induce_features/0`. Seu procedimento é muito parecido com o `induce_cover/0`, pois também não remove os exemplos positivos a cada *best clause* encontrada. Porém, cada *good clause* encontrada é adicionada como uma nova *feature*. Esse processo continua até que as regras encontradas cubram todos os exemplos positivos ou até que a quantidade de *features* encontradas ultrapasse um certo limite. Esse limite é definido pelo parâmetro `set(max_features,V)`, onde `V` é um inteiro positivo (padrão é infinito).

Cada feature encontrada contém duas informações: um número identificador da *feature* e uma cláusula (`Head:-Body`). Uma *feature* é verdadeira para um exemplo, se e somente se, o exemplo unifica com a cabeça e o corpo é verdadeiro. Para problemas de classificação, o usuário deve especificar a variável dependente. Isso é feito através do parâmetro `set(dependent,V)`, onde `V` é um inteiro positivo que indica o argumento da variável dependente nos exemplos. As *features* têm a seguinte forma:

```
feature(ID,(Head:-Body)).
```

A rotulação de cláusulas como *good clause* depende de vários parâmetros do sistema. Abaixo são apresentados alguns parâmetros relevantes nesse processo:

`:- set(clauselength,V)`: `V` é um inteiro positivo (padrão é 4). Limita o número máximo de literais de uma cláusula aceitável;

`:- set(minacc,V)`: `V` é um real entre 0.0 e 1.0 (padrão é 0.0). Limita a mínima precisão de uma cláusula aceitável. A precisão de uma cláusula tem o mesmo significado que precisão, ou seja, `p/(p+n)`, onde `p` é o número de exemplos positivos cobertos pela cláusula (os verdadeiros positivos) e `n` é o número de exemplos negativos cobertos pela cláusula (os falsos positivos);

`:- set(minscore,V)`: `V` é um real (padrão é infinito). Limita a mínima utilidade de uma cláusula aceitável. A utilidade da cláusula é definida pela função de avaliação adotada. Detalhes sobre como definir a função de avaliação encontram-se na Seção 3 na página 24;

`:- set(minpos,V)`: `V` é um inteiro positivo (padrão é 1). Limita o número mínimo de exemplos positivos cobertos por uma cláusula. Esse parâmetro é útil para se evitar que o Aleph adicione à teoria cláusulas com um único literal;

`:- set(noise,V)`: `V` é um inteiro positivo, incluindo zero (padrão é 0). Limita o número máximo de exemplos negativos que podem ser cobertos por uma cláusula aceitável;

`:- set(nodes,V)`: `V` é um inteiro positivo (padrão é 5000). Limita o número máximo de nós a serem explorados durante a busca por uma cláusula aceitável.

`:- set(explore,V)`: `V` pode ser `true` ou `false` (padrão é `false`). Se verdadeiro, força o sistema a continuar a busca até que todos os elementos restantes da busca sejam piores que o melhor elemento atual. Quando é falso, a busca pára quando é certo que todos os elementos restantes da busca não são melhores que o melhor elemento atual. (Observe que com *false*, o critério é mais suave).

É importante observar que as cláusulas que não respeitam os limites definidos por um parâmetro não são adicionadas à teoria. No caso do aprendizado de *features*, essas cláusulas não são adicionadas como uma nova *feature*.

3.7 Definições do Usuário

O Aleph permite que o usuário defina restrições de integridade, de refinamento, de poda e de custo, descritas a seguir.

3.7.1 Restrições de Integridade

O sistema Aleph permite que o usuário defina restrições de integridade. Dessa forma, todas as hipóteses que violarem as restrições serão consideradas inconsistentes e serão recusadas. Note que essas restrições funcionam como os exemplos negativos, sendo úteis para encontrar hipóteses interessantes quando se tem apenas exemplos positivos.

A declaração das restrições se dá no arquivo `.b` e tem a seguinte forma:

```
false:-
        Body.
```

onde `Body`, o corpo da restrição, é o conjunto de literais que especificam as condições que não devem ser violadas. Nesse corpo, geralmente se utiliza o comando `hypothesis/3` para obter a cláusula em construção no momento. Sua estrutura é a seguinte:

```
hypothesis(-Head,-Body,-Label)
```

onde `Head` e `Body` são, respectivamente, a cabeça e o corpo da cláusula sendo construída, e `Label` é a lista `[P,N,L]`, sendo `P` a quantidade de exemplos positivos cobertos pela cláusula em construção, `N` a quantidade de exemplos negativos cobertos pela cláusula em construção e `L` a quantidade de literais. Note que uma restrição de integridade não impede que o refinamento de uma cláusula que viola uma ou mais restrições seja inaceitável.

3.7.2 Refinamento

O sistema Aleph permite que o usuário defina como se dará o refinamento durante a etapa de busca através do comando `refine/2`. O comando é declarado no arquivo `.b` e tem a seguinte forma:

```
refine(Clause1,Clause2):-
        Body.
```

onde a cláusula `Clause1` será refinada para a cláusula `Clause2`, caso o corpo `Body` seja verdadeiro, ou seja, `Clause2` será a cláusula mais específica. O conjunto de refinamentos para qualquer cláusula é obtido através de *backtracking*.

3.7.3 Poda

Quando se sabe que certas cláusulas (e seus refinamentos) não devem aparecer no conjunto de hipóteses, o usuário pode definir condições de poda para elas. Essa é uma maneira de fornecer informações sobre o espaço de hipóteses, aumentando a eficiência do sistema. As definições devem ser declaradas no arquivo `.b` e são da seguinte forma:

```
prune((ClauseHead:-ClauseBody)):-
        Body.
```

significando que, se a condição `Body` for satisfeita, a cláusula `ClauseHead:-ClauseBody` será podada.

3.7.4 Custo

Aleph permite que o usuário especifique custo para as cláusulas. A especificação dos custos deve ser declarada no arquivo `.b`, e tem a seguinte forma:

```
cost(Clause,ClauseLabel,Cost):-
        Body.
```

onde `ClauseLabel` é a lista `[P,N,L]`, onde `P` é o número de exemplos positivos cobertos pela cláusula, `N` é o número de exemplos negativos cobertos pela cláusula e `L` é o número de literais da cláusula. Quando o usuário especifica custo, Aleph deixa a cargo do usuário a especificação de estratégias de poda.

4 Conclusão

Os sistemas de Programação Lógica Indutiva vêm sendo cada vez mais utilizados como método de extração de conhecimento para Mineração de Dados Relacional. Os sistemas mais utilizados são os que induzem regras de classificação relacionais. No entanto, atualmente já existem sistemas se PLI para indução de árvores de decisão relacionais, regras de associação relacionais e indução de restrições, entre outros.

Neste trabalho foram estudadas características do sistema de PLI Aleph mostrando como ele funciona e como utilizá-lo. Aleph é um sistema de PLI que foi desenvolvido com o intuito de ser um protótipo para explorar idéias. Aleph possui características importantes para o desenvolvimento de pesquisas em PLI, pois permite ao usuário modificar a ordem de geração das regras, funções de busca e avaliação de acordo as necessidades da aplicação que o usuário deseja desenvolver. Além da indução de regras de classificação relacionais, Aleph permite ainda a indução de árvores de decisão relacionais, disponibiliza métodos de busca estocásticos, induz restrições e uma forma simplificada de regras de associação relacionais. Essas abordagens podem ser utilizadas por meio do ajuste de parâmetros específicos para cada uma delas. Neste trabalho foram apresentadas cada uma dessas abordagens exemplificando como e quais parâmetros ajustar e apresentando o resultado das execuções. Aleph disponibiliza ainda outros recursos que não foram abordados neste trabalho mas que podem ser encontrados no manual do Aleph (Srinivasan, 2000).

Referências

Blockeel, H. (1998). Tilde - Top Down Induction of Logical Decision Trees. *Technical Report*, K.U.Leuven Dept. of Computer Science. `http://www.cs.kuleuven.ac.be/~ml/Tilde/`. Citado nas páginas 3 and 29.

Camacho, R. (1994). Learning Stage Transition Rules with Indlog. Em S. Wrobel (ed.), *ILP94*, Volume 237 of *GMD-Studien*, pp. 273–290. Gesellschaft für Mathematik und Datenverarbeitung MBH. Citado na página 3.

Dehaspe, L. & L. De Raedt (1997). Mining Association Rules in Multiple Relations. Em S. Džeroski & N. Lavrač (eds.), *ILP97*, pp. 125–132. Springer-Verlag. Citado nas páginas 3 and 37.

Dehaspe, L., L. D. Raedt, & W. Laer (1996). CLAUDIEN : The CLAUsal DIscovery ENgine User's Guide - version 3 .0. `http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW239.ps.gz`. Citado nas páginas 24 and 40.

Dehaspe, L. & H. Toivonen (2001). *Discovery of Relational Association Rules* (1 ed.), Capítulo 8, pp. 189–212. Volume 1 of Lavrač & Džeroski (2001). Citado na página 36.

Džeroski, S. (2003). Multi-Relational Data Mining: An Introduction. *SIGKDD Explorations 5*(1), 1–16. Citado nas páginas 2, 30, and 37.

Karalič, A. & I. Bratko (1997). First-Order Regression. *Machine learning 26*(2-3), 147–176. Citado na página 3.

Kirkpatrick, S., C. D. Gelatt, & M. P. Vecchi. Optimization by Simulated Annealing. *Science*. Citado na página 43.

Lavrač, N. & S. Džeroski (2001). *Relational Data Mining*. Berlin: Springer-Verlag. Citado nas páginas 2, 29, and 51.

Lavrač, N. & P. A. Flach (2001). An extended Transformation Approach to Inductive Logic Programming. *ACM Transactions on Computational Logic (TOCL) 2*(4), 458–494. Citado na página 5.

Michalski, R. S., K. Kaufman, & A. Schultz (1989). Emerald i: An integrated system of machine learning and discovery programs for education and research: User's guide. *Technical Report*, George Mason University, Fairfax. `citeseer.lcs.mit.edu/168008.html`. Citado nas páginas 1, 16, and 17.

Mooney, R., P. Melville, L. Tang, J. Shavlik, I. Dutra, D. Page, & V. S. Costa (2002). Relational Data Mining with Inductive Logic Programming for Link Discovery. Em *Proceedings of the National Science Foundation Workshop on Next Generation Data Mining*, Baltimore. AAAI/MIT Press. Citado na página 6.

Muggleton, S. (1995). Inverse Entailment and Progol. *New Generation Computing 13*, 245–286. Citado nas páginas 2, 3, and 6.

Muggleton, S. & L. D. Raedt (1994). Inductive Logic Programming. *Journal of Logic Programming 19–20*, 629–679. Citado na página 5.

Quinlan, J. R. (1988). *C4.5 Programs for Machine Learning*. CA: Morgan Kaufmann. Citado nas páginas 29 and 31.

Quinlan, J. R. (1990). Learning Logical Definitions from Relations. *Machine Learning 5*(3), 239–266. Citado nas páginas 2 and 3.

Éric Grégoire & B. Mazure (1998). Logically-complete local search for propositional nonmonotonic knowledge bases. Em *Proc. of the 7th Int. Workshop on Nonmonotonic Reasoning*, Trente, pp. 37–45. Citado na página 43.

Selman, B., H. A. Kautz, & B. Cohen (1994). Noise strategies for improving local search. Em *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)*, Seattle, pp. 337–343. Citado na página 43.

Srinivasan, A. (1999). A study of two sampling methods for analysing large datasets with ILP. *Data Mining and Knowledge Discovery 3*(1), 95–123. Citado na página 42.

Srinivasan, A. (2000). The Aleph Manual. *Technical Report*, Oxford University. `http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/aleph_toc.html`. Citado nas páginas 3, 6, 25, and 50.

Wrobel, S. (1997). An Algorithm for Multi-Relational Discovery of Subgroups. Em J. Komorowski & J. Zytkow (eds.), *Proceedings of the First European Symposium on Principles of Data Mining and Knowledge Discovery (PKDD-97)*, Berlin, pp. 78–87. Springer-Verlag. Citado na página 3.

Zelezny, F., A. Srinivasan, & D. Page (2002). Lattice-Search Runtime Distributions May Be Heavy-Tailed. Em *Proceedings of the The Twelfth International Conference on Inductive Logic Programming*. Springer Verlag. Citado nas páginas 43 and 44.