UTILIZANDO A CLÁUSULA MAIS ESPECÍFICA E DECLARAÇÃO DE MODOS NA REVISÃO DE TEORIAS DE PRIMEIRA-ORDEM A PARTIR DE EXEMPLOS

Ana Luísa de Cerqueira Leite Duboc

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Gerson Zaverucha, Ph.D.

Prof. Mário Roberto Folhadela Benevides, Ph.D.

Prof. Sheila Regina Murgel Veloso, D.Sc.

Prof. Renata Wassermann, Ph.D.

RIO DE JANEIRO, RJ - BRASIL

MARÇO DE 2008

DUBOC, ANA LUÍSA DE CERQUEIRA LEITE

Utilizando a cláusula mais específica e declaração de modos na revisão de teorias de primeira-ordem a partir de exemplos [Rio de Janeiro] 2008

XIII, 96p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2008)

Dissertação - Universidade Federal do Rio de Janeiro, COPPE

1. Revisão de Teoria
2. Cláusula mais específica
3. ILP
4. Aprendizado de Máquina
5. Lógica de Primeira Ordem

I. COPPE/UFRJ II. Título(série)

Dedicatória

Aos meus pais, sem os quais nada disso seria possível.

"Toda história tem seu fim, mas na vida cada final é um novo começo"

Agradecimentos

A Deus, por estar sempre comigo, por me dar forças para continuar quando pensei em desistir, e por me mostrar que sonhos podem se tornar reais, mesmo que seja difícil realizá-los.

Aos meus pais, Maria Lúcia e Gilson, por sempre me apoiarem e me darem o suporte necessário para que eu chegasse até aqui, por acreditarem em mim, por não duvidarem da minha capacidade, e, o mais importante, pelo amor que me dedicam a cada dia.

À minha irmã, Letícia, pela paciência enorme que teve comigo, principalmente quando nada funcionava no meu computador, e por me incentivar a seguir em frente, querendo sempre o melhor pra mim.

Ao meu orientador, Professor Doutor Gerson Zaverucha, pelos conhecimentos transmitidos, por me guiar na elaboração deste trabalho, por acreditar em mim até quando eu mesma não acreditava mais.

À minha mais nova amiga, Aline Paes, por me ajudar a recomeçar no mestrado, pelos ensinamentos, pela paciência, por tirar minhas dúvidas, pelos conselhos, e por me apoiar sempre.

Aos meus amigos Carina e Alexandre Silva, por sempre me ouvirem, me apoiarem e acreditarem em mim.

Ao meu amigo e professor João Carlos Pereira da Silva, pelas palavras de amizade, por ser meu psicólogo e amigo nos momentos em que mais precisei.

Ao Eric, Rafael e Guilherme, alunos da iniciação científica, pela contribuição feita a este trabalho.

Aos amigos do mestrado e ao restante dos meus amigos e da minha família, por todo apoio, carinho e compreensão.

Ao CNPQ pelo suporte financeiro.

Obrigada a todos que de alguma forma contribuíram para que este trabalho fosse concluído, me apoiando, incentivando e acreditando em mim.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

UTILIZANDO A CLÁUSULA MAIS ESPECÍFICA E DECLARAÇÃO DE MODOS NA REVISÃO DE TEORIAS DE PRIMEIRA ORDEM A PARTIR DE EXEMPLOS

Ana Luísa de Cerqueira Leite Duboc

Março/2008

Orientador: Gerson Zaverucha

Programa: Engenharia de Sistemas e Computação

Sistemas de revisão de teorias são desenvolvidos para melhorar a acurácia de uma teoria inicial dada, tornando-as mais acuradas e compreensíveis do que as geradas por métodos puramente indutivos. Estes sistemas partem da teoria inicial fornecida e a modifica nos pontos onde há uma falha na classificação dos exemplos. Uma das modificações possíveis é a adição de antecedentes à cláusula sendo revisada. Um conhecido sistema de revisão de teorias é o FORTE, cuja operação de adição de antecedentes é baseada no FOIL, e considera todos os literais da base de conhecimento como possíveis antecedentes a serem adicionados na cláusula. Tal fato acarreta em um espaço de busca muito grande, o que domina o custo do processo de revisão. O estado da arte dos algoritmos de aprendizado em ILP restringem a busca de antecedentes aos literais relevantes para um determinado exemplo positivo. Tais literais compõem a cláusula mais específica, gerada por uma busca direcionada a modos. Visando melhorar a eficiência da operação de adição de antecedentes no FORTE, propomos a utilização da cláusula mais específica como espaço de busca de antecedentes, e a declaração de modos para validar os literais da cláusula mais específica sendo adicionados à cláusula revisada. Resultados experimentais mostram que o processo de revisão passou a ser três ordens de magnitude mais rápido do que o que é feito no FORTE, e teorias mais compreensíveis foram geradas sem prejuízo da acurácia. A abordagem proposta também melhora significativamente a acurácia de teorias geradas a partir do sistema Aleph, o mais utilizado sistema de ILP.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

USING THE MOST ESPECIFIC CLAUSE AND MODES DECLARATION ON FIRST-ORDER REVISION OF THEORIES FROM EXAMPLES

Ana Luísa de Cerqueira Leite Duboc

March/2008

Advisor: Gerson Zaverucha

Department: Systems Engineering and Computer Science

Theory revision systems are designed to improve the accuracy of an initial theory, producing more accurate and comprehensible theories than purely inductive methods. Such systems start from an initial theory and modify it on points where some example is misclassified. One of the possible modifications is the addition of antecedents to the clause been revised. A known theory revision system is FORTE. The adding antecedents operation of FORTE is based on FOIL, and therefore considere all the literals of background knowledge as possible antecedents to be added to the the clause. This fact implies in a huge search space which dominates the cost of the revision process. The state of the art of learning algorithms in ILP restrict the search of antecedents to literals that are relevant to a positive example. These literals form the most specif clause, which is generated through a mode directed search. Aiming to impove the efficiency of the antecedent adding operation in FORTE, we propose the use of the most specif clause as antecedent search space. Besides that, we also propose the use of modes to define which literals of the most specif clause can effectively be added to the clause been revised. Experimental results show that the runtime of the revision process is on average three orders of magnitude faster than the one that is done on FORTE, and generate more comprehensible theories without decreasing the accuracy. The proposed approach also significantly improves predictive accuracy over theories generated by Aleph system, the most used system in ILP.

Sumário

Lista de Abreviaturas

BC *Bottom Clause*

C Cláusula a ser especializada

E Conjunto de exemplos fornecido ao sistema de revisão

H Teoria de primeira-ordem

ILP *Inductive Logic Programming*

BK Conhecimento preliminar

Lista de Algoritmos

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A lógica de primeira-ordem é um sistema robusto de representação de conhecimento, pois consegue representar situações complexas envolvendo vários objetos e relações entre eles. Dentro da área de Programação em Lógica Indutiva(ILP) (MUGGLETON, 1991), (MUGGLETON, 1992a), (LAVRAC, DZEROSKI, 1994), (MUGGLETON, De RAEDT, 1994), (MUGGLETON, 1999), definida como a interseção entre o aprendizado indutivo e a programação lógica, algoritmos de aprendizado têm sido desenvolvidos para aprender teorias em lógica de primeira-ordem. A partir de uma teoria inicial assumida como correta e que portanto não pode ser modifcada, o que chamamos de conhecimento preliminar, e de um conjunto de exemplos positivos e negativos, sistemas ILP acham um conjunto de novas cláusulas que implique logicamente os exemplos positivos e não implique os exemplos negativos.

Enquanto o aprendizado indutivo lida com aprender teorias a partir do zero, já que não considera teorias iniciais dadas que fazem parte do conhecimento preliminar, a área de ILP também engloba a tarefa de revisar teorias, onde, além do conhecimento preliminar, partimos de uma teoria inicial que pode esta incorreta ou imcompleta, e portanto pode ser modificada. Portanto o objetivo é achar uma teoria revisada que passe a implicar logicamente os exemplos positivos e não implique logicamente os exemplos negativos.

A única diferença em relação aos sistemas ILP é que uma teoria inicial incorreta e modificável é dada. Os sistemas de revisão de teorias tem como objetivo gerar teorias revisadas mais acuradas do que a inicial.

Sistemas de revisão de teorias partem da teoria inicial fornecida e a modifica

nos pontos onde é verificada uma classificação incorreta dos exemplos. A melhor modificação entre todas as revisões propostas é escolhida. As modificações são feitas utilizando-se operadores de revisão, que se dividem em operadores de especialização e generalização. Pontos na teoria onde a prova de instâncias positivas falha são lugares onde a teoria precisa ser generalizada, e cláusulas usadas em provas de instâncias negativas são pontos onde a teoria precisa ser especializada. Uma das modificações usadas é a adição de antecedentes, que adiciona um antecedente em uma cláusula incorreta para excluir exemplos negativos que estão sendo provados, não permitindo ao máximo possível que exemplos positivos deixem de ser provados.

Existem várias abordagens de revisão de teorias de primeira-ordem(WROBEL, 1996) tais como FORTE(RICHARDS, MOONEY, 1995), AUDREY(WOGULIS, 1991), CLINT(RAEDT, BRUYNOOGHE, 1992), MIS(SHAPIRO, 1983), RUTH(ADé, et al., 1994) e KRT(WROBEL, 1994). De acordo com (WROBEL, 1996), os sistemas GOLEM(MUGGLETON, FENG, 1992) e CIGOL(MUGGLETON, BUNTINE, 1988), apesar de serem sistemas de aprendizado de teorias, podem ser considerados como sistemas de revisões mesmo não aceitando teoria inicial, pois estes sistemas conseguem aprender teorias envolvendo múltiplos predicados. O sistema de revisão de teorias FORTE difere da maioria dos sistemas citados pois faz revisão automática ao invés de interativa com o usuário e utiliza operadores de generalização e especialização como espaços de busca, entre eles o operador de adição de antecedentes.

A Maioria dos sistemas ILP, tal como o Aleph/Progol (SRINIVASAN, 2001), (MUGGLETON, 1995), utilizam o algoritmo de cobertura, onde hipóteses são construídas iterativamente cláusula por cláusula. O algoritmo de cobertura é guloso no sentido de que cada iteração adiciona a melhor cláusula de acordo com algum critério de avaliação local, o que leva à geração de cláusulas localmente ótimas. Além disso, apenas um predicado é aprendido, o que resulta em muitas cláusulas com o mesmo predicado na cabeça e desnecessariamente grandes(BRATKO, 1999). Já o aprendizado de teorias feito pelo FORTE realiza um aprendizado global, considerando cláusulas previamente criadas quando se aprende uma nova cláusula, não usando portanto o algoritmo de cobertura e consequentemente gerando teorias menores.

No FORTE, a busca de antecedentes a serem adicionados pode ser feita através

de dois algoritmos: *hill climbing* e *relational pathfinding*. O *hill climbing* adiciona um antecedente de cada vez, procurando pelo que proporciona melhor desempenho à teoria. O *relational pathfinding*(RICHARDS, MOONEY, 1992) adiciona vários antecedentes de uma só vez, tendo como idéia principal a busca por um caminho num domínio relacional. Ao contrário do *hill climbing*, o *relational pathfinding* é um método de adição de antecedentes desenvolvido para escapar do máximo local. Em ambos os algoritmos a busca por tais antecedentes é uma busca *top-down* baseada no FOIL(QUINLAN, 1990), e o espaço de busca de literais é muito grande pois considera toda a base de conhecimento, sendo que alguns destes literais podem não cobrir nem ao menos um exemplo positivo, o que domina o custo do processo de revisão, pois avaliar cada literal é uma operação muito custosa. A base de conhecimento inclui, entre outras coisas, a definição dos predicados e os tipos de seus argumentos.

O estado da arte dos algoritmos de aprendizado em ILP restringem a busca de antecedentes aos literais relevantes para um determinado exemplo positivo. Tais literais compõem a cláusula mais específica, também chamada de *bottom clause* (MUGGLETON, 1995), que é gerada através de uma busca direcionada a modos. Um sistema conhecido que utiliza a *bottom clause* como espaço de busca para aprendizado de teorias é o Aleph/Progol (SRINIVASAN, 2001), (MUGGLETON, 1995). O Aleph utiliza o algoritmo *hill climbing* para buscar antecedentes, e recentemente foi incluído no Aleph a utilização do algoritmo *relational pathfinding* (ONG, et al., 2005), tendo como objetivo estender o algoritmo *pahtfinding* para fazer uso de declaração de modos, aplicando esta extensão à *bottom clauses*, e passando a buscar caminhos relacionais na cláusula mais específica como forma de restringir o espaço de busca.

Visando melhorar a eficiência da operação de adição de antecedentes no sistema FORTE, propomos a utilização da cláusula mais específica como espaço de busca de antecedentes, tanto na abordagem *hill climbing* quanto no algoritmo *relational pathfinding*. Além disso, propomos também a utilização de declaração de modos para definir quais literais da cláusula mais específica poderão efetivamente ser adicionados à cláusula sendo revisada, pois através dos modos podemos definir se um predicado deve estar na cabeça ou no corpo de uma cláusula, podemos definir

o número máximo de instanciações de um predicado na cláusula e também podemos restringir os argumentos dos predicados a serem de entrada, saída ou constante. Esperamos com isso restringir a busca aos literais mais relevantes, e consequentemente reduzir o tempo de busca e revisão de teorias, mantendo a geração de teorias mais acuradas.

Além disso, sabemos que existem algoritmos de aprendizado de teorias, tal como o Hyper(BRATKO, 1999), que ao contrário do Aleph fazem aprendizado de múltiplos predicados simultaneamente, e que conseguem obter teorias menores e mais acuradas, porém em um tempo muito demorado. Portanto, como um resultado complementar, queremos mostrar que aprender teorias usando técnicas de revisão juntamente com a utilização da cláusula mais específica, é melhor do que o algoritmo de cobertura do Aleph, e que além de conseguir teorias mais acuradas e menores, também consegue um tempo compatível ao do Aleph.

A presente dissertação está organizada da seguinte forma: no capítulo 2 serão descritos conceitos básicos relacionados com a nossa pesquisa. No final do capítulo 2 mostramos alguns trabalhos relacionados a esta dissertação, discutindo outros motivos pelos quais o FORTE foi escolhido como sistema a ser usado na nossa proposta. No capítulo 3 introduzimos a nossa proposta de utilização de modos e da cláusula mais específica para revisar teorias de primeira-ordem a partir de exemplos. Os resultados experimentais são descritos no capítulo 4, e finalmente no capítulo 5 apresentamos nossas conclusões e trabalhos futuros.

Capítulo 2

Conceitos Preliminares

Neste capítulo apresentaremos as técnicas nas quais baseamos nossa pesquisa e que serão importantes para o entendimento do restante da dissertação. Os leitores já familiarizados com algum desses assuntos podem encaminhar-se para as outras seções.

Este trabalho compreende temas como sistemas de representação em lógica de primeira-ordem, aprendizado de teorias utilizando-se a cláusula mais específica e sistemas de revisão de teorias. Portanto, em um primeiro momento serão apresentados conceitos relacionados à lógica e então na seção 2.1 é definida a terminologia utilizada em lógica de primeira-ordem (CASANOVA, et al., 1987), (LLOYD, 1987), (ZAVERUCHA, 2008), na seção 2.2 definimos Programação em Lógica Indutiva(ILP) (MUGGLETON, 1991), (MUGGLETON, 1992a), (LAVRAC, DZEROSKI, 1994), (MUGGLETON, De RAEDT, 1994), (MUGGLETON, 1999), na seção 2.3 descrevemos o PROGOL(MUGGLETON, 1995), (M. FERRO, 2007) que introduz a geração da cláusula mais específica, e na seção 2.4 descrevemos revisão de teorias de primeira-ordem (WROBEL, 1996), dando ênfase ao sistema FORTE(RICHARDS, MOONEY, 1995). Por fim, na seção 2.5 citamos alguns sistemas de revisão de teorias e suas diferenças em relação ao FORTE.

2.1 Lógica de primeira-ordem

A lógica de primeira-ordem, conhecida também como cálculo de predicados, é um sistema lógico que estende a lógica proposicional.

A lógica de primeira-ordem é capaz de representar relações entre objetos, con-

cluir particularizações de uma propriedade geral dos indivíduos de um universo de discurso, assim como derivar generalizações a partir de fatos que valem para um indivíduo arbitrário do universo de discurso. Para ter tal poder de expressão, a linguagem de primeira-ordem usa símbolos mais sofisticados do que os da linguagem proposicional. Para expressar propriedades que valem para todos os indivíduos ou apenas para alguns, são utilizados os quantificadores $\forall$(universal) e $\exists$(existencial), respectivamente.

Daremos a seguir alguns conceitos e terminologias referentes à lógica de primeira-ordem.

Um *alfabeto de primeira-ordem* consiste de:

a) Símbolos lógicos:

a1) Pontuação: "(",",",",")"

a2) Conectivos: $\neg$(negação),$\wedge$(conjunção), $\vee$(disjunção), $\leftarrow$(implicação) e $\leftrightarrow$(bi-condicional)

a3) Quantificadores: $\forall$(universal) e $\exists$(existencial)

a4) Variáveis, que se iniciam com letras maiúsculas. Exemplos: A, Var

a5) Símbolo de igualdade (opcional): $=$

b) Símbolos não-lógicos: funções(que se iniciam por letras minúsculas), constantes(símbolos funcionais de aridade 0) e predicados.

O conjunto de *termos de primeira-ordem* é o conjunto formado pelas variáveis, constantes, e funções aplicadas a termos, como por exemplo $f(t_1, ..., t_n)$, onde $t_1, ..., t_n$ são termos. Se $t_1, ..., t_n$ são termos e P é um símbolo predicativo n-ário, então $P(t_1, ..., t_n)$ é chamado de fórmula atômica ou átomo. Exemplo: *tio(joão,X).* Um átomo é um literal positivo, e a negação de um átomo é um literal negativo.

Se A e B são fórmulas, então ($\neg$A), (A$\wedge$B), (A$\vee$B), (A$\leftarrow$B) e (A$\leftrightarrow$B) são fórmulas. Se B é uma fórmula e X é uma variável, então $\forall_X$(B) e $\exists_X$(B) também são fórmulas.

Uma *cláusula* é uma disjunção de literais precedida por um prefixo de quantificadores universais, um para cada variável que aparece na disjunção, na seguinte forma:

$$\forall X_1 \forall X_2 ... \forall X_s (L_1 \vee L_2 \vee ... \vee L_m)$$

onde cada L_i é um literal e $X_1, X_2, ..., X_s$ são todas as variáveis ocorrendo em $(L_1 \vee L_2 \vee ... \vee L_m)$. O conjunto $\{A_1, A_2, ..., A_h, \neg B_1, \neg B_2, ..., \neg B_b\}$, onde A_i e B_i são átomos, indica a cláusula

$$\forall A_1 \forall A_2 ... \forall A_h \forall B_1 \forall B_2 ... \forall B_b (A_1 \vee ... A_h \vee \neg B_1 \vee ... \vee \neg B_b)$$

que é equivalentemente representado por $\{A_1 \vee ... \vee A_h \leftarrow B_1 \wedge ... \wedge B_b\}$, e pode ser escrita como $A_1, ..., A_h \leftarrow B_1, ..., B_b$, onde $A_1, ..., A_h$ é a cabeça e $B_1, ..., B_b$ é o corpo da cláusula. Normalmente em ILP é usado :- ao invés de $\leftarrow$. Uma *teoria* é um conjunto de cláusulas, representando a conjunção dessas cláusulas.

Uma *cláusula de Horn* é uma cláusula que contém no máximo um literal positivo, ou seja, um literal na cabeça. Cláusulas com exatamente um literal na cabeça são chamadas de *cláusulas definidas*. Um *fato* é uma cláusula definida com o corpo vazio, representado por $A \leftarrow$ e denotado por A. Um conjunto de cláusulas definidas é chamado de *programa lógico definido*. A figura 2.1 mostra um exemplo de um programa lógico definido.

```
pais(jorge,julio).        pais(jorge,lucia).        genero(jorge,masculino).
genero(julio,masculino).  genero(lucia,feminino).

pai(X,Y) :- pais(X,Y), genero(X, masculino).
mae(X,Y) :- pais(X,Y), genero(X,feminino).
irmao(X,Y) :- irmaos(X,Y), genero(X,masculino).
irma(X,Y) :- irmaos(X,Y), genero(X,feminino).
irmaos(X,Y) :- pais(Z,X), pais(Z,Y), X \= Y.
```

Figura 2.1: Exemplo de um programa lógico definido

Uma *substituição* $\theta = \{V_1/t1, ..., V_n/t_n\}$ é uma associação de termos t_i para variáveis V_i. Aplicar uma substituição θ para um termo, átomo ou cláusula F produz o termo, átomo ou cláusula instanciado $F\theta$, onde todas as ocorrências de variáveis V_i são simultaneamente substituídas pelo termo t_i. Um termo, átomo ou cláusula é básica quando não existe nenhuma variável ocorrendo nele. Uma cláusula é livre de funções se ela não contém símbolos funcionais. Seja $\sum = \{E_1, ..., E_n\}$ um

conjunto de fórmulas e β uma substituição. β é um unificador de $\sum$ se, e somente se, $E_1\beta = ... = E_n\beta$.

Uma cláusula C $\theta - subsume(\preceq)$ a cláusula D se existe uma substituição θ tal que $C\theta \subseteq D$. Por exemplo, a cláusula $C : f(A,B) \leftarrow p(B,G), q(G,A)$ $\theta - subsume$ a cláusula $D : f(a,b) \leftarrow p(b,g), q(g,a), t(a,d)$ através da substituição $\{A/a, B/b, G/g\}$. Portanto, $C \preceq D$.

Interpretação (ou atribuição de valor-verdade) é um processo que mapeia uma sentença em alguma declaração em relação a um determinado domínio. Se a interpretação der o valor verdadeiro para uma sentença então ela satisfaz esta sentença e tal interpretação é chamada um *modelo* da sentença. A noção de modelo é aplicada extensionalmente para uma teoria clausal: uma interpretação é um modelo para um conjunto se ela é um modelo para cada um dos membros do conjunto. Uma teoria DB implica logicamente uma cláusula G ($DB \models G$), ou seja, G é *consequência lógica* de DB, se e somente se todo modelo de DB também for modelo de G.

Um *sistema dedutivo SD = (L, AX, R)* é um sistema formado por uma linguagem L, um conjunto de axiomas lógicos AX e um conjunto de regras de inferência R. Uma fórmula G é um teorema (ou G tem uma *dedução* ou *prova* a partir) de um conjunto de fórmulas DB, no sistema dedutivo SD (denotado por $DB \vdash G$), se e somente se, existir uma sequência finita de fórmulas $D_1, ..., D_n$ tal que:

- a) $D_n = G$
- b) Para todo $i \in [1, n]$, uma das condições abaixo é satisfeita:
  - D_i é uma instância de um axioma lógico de AX
  - $D_i \in DB$
  - existem $j, k < i$ tais que D_i pode ser obtido pela aplicação de uma das regras de inferência de R.

2.2 Programação em Lógica Indutiva

Tendo sido definida a terminologia utilizada em lógica de primeira-ordem e em programação lógica, podemos definir brevemente o que vem a ser Programação em Lógica Indutiva (ILP – *Inductive Logic Programming*). Para mais informações sobre o assunto veja (MUGGLETON, 1991), (MUGGLETON, 1992a), (LAVRAC, DZEROSKI, 1994), (MUGGLETON, De RAEDT, 1994), (MUGGLETON, 1999).

Programação em Lógica Indutiva(ILP) é uma área da inteligência artificial que investiga a construção indutiva de teorias de cláusulas de Horn de primeira-ordem a partir de exemplos e de um conhecimento preliminar. ILP é definido como sendo a interseção entre aprendizado indutivo e programação em lógica. Do aprendizado indutivo ILP herda o seu objetivo: construção de ferramentas e técnicas para induzir hipóteses a partir de observações (exemplos) e sintetizar novo conhecimento a partir da experiência. Da Programação em Lógica(LLOYD, 1987), ILP herda o formalismo representacional, como a representação de teorias, hipóteses, exemplos e conhecimento preliminar, técnicas bem estabelecidas e uma profunda base teórica.

ILP 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 ILP. 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 de ILP 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 ILP 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 preliminar para os membros de uma família(MOONEY, et al., 2002).

A hipótese fundamental do aprendizado indutivo é que qualquer hipótese descoberta que aproxima bem uma determinada função alvo usando um conjunto suficientemente grande de exemplos de treinamento, também aproximará bem a função alvo sobre outros exemplos não observados (generalização). A partir dos exemplos de treinamento, o sistema de ILP induz um programa lógico correspondente à visão que define a relação alvo, em termos de outras relações que são dadas como conhecimento preliminar. Em sistemas ILP, os exemplos de treinamento, o conhecimento preliminar e as hipóteses induzidas são todos expressos na forma de programas lógicos, sendo que os exemplos de treinamento são representados como fatos da relação alvo a ser aprendida. O conjunto de conhecimento preliminar é definido como extensional se ele é representado como um conjunto de literais básicos, tais como $pai(pedro, ana)$, ou definido como intensional, se é representado como uma teoria de cláusulas de Horn. Por exemplo, $avo(X, Y) \leftarrow pai(X, Z), pai(Z, Y)$ e um conjunto de fatos básicos definindo a relação *pai* é considerado um conhecimento preliminar intensional. Podemos definir a tarefa básica de ILP da seguinte forma (LAVRAC, DZEROSKI, 1994):

Definição 2.1. *Dados:*

- *Um conhecimento preliminar invariante* BK
- *Um conjunto de exemplos positivos* E_+ *e negativos* E_-

Achar:

- *Uma teoria* H *que, junto com o conhecimento preliminar* BK*, implique logicamente todos os exemplos positivos (completo),* $BK \cup H \models E^+$ *e nenhum dos exemplos negativos (consistência),* $\forall e^- \in E^- : BK \cup H \not\models e^-$*, e que esteja de acordo com o critério de qualidade, tal como o de minimalidade do tamanho da teoria.*

Para formalizar esta definição vamos definir o que vem a ser a relação de cobertura em ILP: dados BK, H e E definidos como anteriormente, a hipótese H cobre um exemplo $e \in E$ em relação a BK se $BK \wedge H \models e$

As definições acima requerem que a relação alvo c e a teoria H concordem em todos os exemplos de E. Porém não há garantias de que H irá corresponder a c em exemplos não vistos. A acurácia de classificar exemplos não vistos é o principal critério de sucesso do sistema de aprendizado.

Definição 2.2. *Dada uma teoria de primeira-ordem H, a acurácia de classificação de H é medida como sendo a porcentagem de exemplos corretamente classificados pela teoria.*

A transparência de H denota o quanto ela é compreensível por humanos. Uma medida utilizada é o tamanho da hipótese, expressa pelo total de condições no conjunto de regras que formam H.

Sistemas ILP podem aprender um único conceito ou múltiplos conceitos (predicados). O aprendiz pode tentar aprender um conceito do zero ou pode aceitar uma teoria inicial que pode ser revisada no processo de aprendizado.

2.3 PROGOL e a construção da *bottom clause*

Em ILP, a busca pela hipótese exige uma heurística de busca no espaço de hipóteses. Geralmente existem duas abordagens: top-down e bottom-up. As duas abordagens, top-down e bottom-up, podem ser vistas como um tipo de algoritmo de cobertura (*covering*). Porém elas diferem na forma como a cláusula é construída. Na abordagem top-down(QUINLAN, 1990), a cláusula é construída do mais geral para o específico, sendo que a busca geralmente começa da cláusula mais geral e sucessivamente a especializa com predicados do conhecimento preliminar, de acordo com alguma heurística de busca. Na abordagem bottom-up(MUGGLETON, 1992a), a busca começa com a hipótese mais específica, com o conjunto de exemplos, e constrói as cláusulas do específico para o geral através da generalização de cláusulas mais específicas. A Abordagem top-down mais popular em ILP é o FOIL(QUINLAN, 1990), enquanto que a abordagem bottom-up mais popular em

ILP é o GOLEM(MUGGLETON, FENG, 1992). PROGOL/Aleph(SRINIVASAN, 2001),(MUGGLETON, 1995) é uma abordagem que combina top-down e bottom-up. Ele combina o FOIL com implicação inversa(*inverse entailment*), originada a partir da abordagem bottom-up. Para mais informações sobre implicação inversa consulte (MUGGLETON, 1992a).

O FOIL, definido em Algoritmo 2.1, aprende a definição de um conceito alvo, na forma de cláusulas Horn de primeira-ordem livres de função, dados os predicados do conhecimento preliminar, que é definido extensionalmente. FOIL contém um loop externo onde, em cada iteração, acha uma cláusula que implique logicamente uma parte dos exemplos positivos e é consistente com os exemplos negativos. O loop termina quando o conjunto de cláusulas gerado implica logicamente todos os exemplos positivos. O loop interno constrói uma única cláusula, começando com a hipótese mais geral e adicionando literais a ela até que ela deixe de provar exemplos negativos. Os literais são ranqueados usando a métrica de ganho de informação e o literal que maximiza o ganho é escolhido.

Abordagens bottom-up são todas baseadas em generalizações de exemplos positivos através de operadores de generalização. O CIGOL(MUGGLETON, BUNTINE, 1988) é uma destas abordagens e usa resolução inversa.

2.3.1 Algoritmo PROGOL

PROGOL/Aleph é um sistema ILP que usa um simples algoritmo de cobertura, como o do FOIL, no qual cada iteração realiza os seguintes passos:

1. Seleciona aleatoriamente um exemplo positivo para ser generalizado. Este exemplo deve pertencer ao conjunto de exemplos de treinamento positivos que ainda não foram cobertos. Se não existirem mais exemplos, pára;

2. Saturação: constrói a cláusula mais específica, variabilizada, que implique o exemplo selecionado no passo anterior. A cláusula minimamente generalizada é chamada de *bottom clause*.

3. Redução: realiza uma busca por uma "boa" cláusula entre a cláusula maximamente geral, a cláusula de corpo vazio e a *bottom clause* maximamente específica. A "boa qualidade"de cada cláusula encontrada ao longo do caminho de

Algoritmo 2.1 Algoritmo FOIL, proposto em (QUINLAN, 1990)

Entrada:
$R(V_1, V_2, ..., V_k)$: predicado alvo
E^+: Exemplos positivos
E^-: Exemplos negativos
Saída:
H: O conjunto de cláusulas aprendidas
$H := \emptyset$
Positivos_para_cobrir $:= E^+$
Enquanto *Positivos_para_cobrir* não é vazio **faça**
/* Procure por uma cláusula consistente que cubra um largo subconjunto de *Positivos_para_cobrir* */
$C := R(V_1, V_2, ..., V_k) \leftarrow$
$T :=$ *Positivos_para_cobrir* $\cup E^-$
Enquanto T contém tuplas negativas **faça**
Ache o literal L que maximiza *ganho*(L) para adicionar à cláusula C
Forme um novo conjunto T' extendendo cada tupla t em T que satisfaz L com suas novas variáveis
Substitua T por T'
Fim Enquanto
Adicione C a H
Remova de *Positivos_para_cobrir* os exemplos cobertos por C
Fim Enquanto
Retorne H

busca é medida utilizando uma função de avaliação;

4. *Cover Removal*: adiciona a cláusula de melhor qualidade 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 PROGOL/Aleph, assim como a maioria dos sistemas de aprendizado de teorias, aprende um conceito por vez. A seguir explicaremos o que vem a ser Modos e *Determinations*, que são conceitos necessários para se entender como que a *bottom clause* é construída no PROGOL/Aleph.

2.3.2 Declaração de Modos

Já que a *bottom clause* pode ter um número infinito de literais no corpo, PROGOL/Aleph usa uma técnica chamada "declaração de modos" juntamente com o limite de profundidade de variável para restringir o tamanho da *bottom clause*. O limite de profundidade de uma váriável em uma determinada cláusula é a profun-

didade máxima que a árvore de construção da *bottom clause* pode atingir, onde em cada nível da árvore novas variáveis são criadas a partir das variáveis do nível anterior. Este conceito de profundidade será melhor explicado no exemplo de construção da *bottom clause* dado mais adiante.

As declarações de modos descrevem as relações(predicados) entre objetos de dados e tipos desses dados. Estas declarações permitem também informar se a relação pode ser utilizada na cabeça (declarações *modeh*) ou no corpo (declarações *modeb*) das regras geradas. Na declaração de modos também são descritos o tipo dos argumentos e o número máximo de instanciações de cada predicado. As declarações tem o formato $mode(Numero\_Chamadas, Modo)$. O $Numero\_Chamadas$, ou como é mais conhecido, o $recall\_number$, determina o limite do número de instanciações alternativas de um predicado. O $recall\_number$ 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 isto pelo $recall\_number$. Por exemplo, para uma declaração do predicado $progenitor\_de(Pai, Filho)$ poderia ser dado um $recall\_number$ igual a 2, pois todas as pessoas terão no máximo dois progenitores. O $recall\_number$ '*' 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 de declaração de *mode/2*, denominada de *Modo*, indica o formato do predicado que será utilizado da seguinte forma:

$$predicado(TipoArgumento\_1, TipoArgumento\_2, ..., TipoArgumento\_n).$$

Para o aprendizado da relação $sogra\_de(Sogra, Genro)$ com conhecimento preliminar descrito, por exemplo, pelas relações $progenitor\_de(Mae, Filha)$ e $esposa\_de(Esposa, Marido)$ as declarações de modo podem ser:

$$:-modeh(1, sogra\_de(+mulher, +homem)).$$
$$:-modeb(*, progenitor\_de(+mulher, -mulher)).$$
$$:-modeb(1, esposa\_de(+mulher, +homem)).$$

onde os símbolos + e − serão explicados a seguir. A declaração *modeh/2* indica o predicado que irá compor a cabeça das regras. O valor do *recall_number* é 1 para esse predicado, pois o predicado pode ter uma única resposta (sim ou não), dados os dois argumentos. 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 do *recall_number* 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 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 seguir:

$:-modeh(1, sogra\_de(+mulher, -homem)).$

$:-modeb(*, progenitor\_de(+mulher, -mulher)).$

$:-modeb(1, esposa\_de(+mulher, -homem)).$

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 $\sharp T$ só pode ser substituídos por constantes.

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

$mulher(jane).$

$mulher(sueli).$

$homem(henrique).$

$homem(junior).$

...

2.3.3 Declaração dos *Determinations*

O PROGOL utiliza $determination/2$ para declarar os predicados que podem ser usados no corpo de um predicado alvo. Essa declaração tem o formato:

$determination(PredAlvo/Aridade\_a, PredCorpo/Aridade\_c).$

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 constução 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.

2.3.4 Construção da *Bottom Clause*

A seguir temos o algoritmo de construção da *bottom clause* no PROGOL (Algoritmo 2.2). O *recall* é usado para determinar quantas vezes chamar o interpretador Prolog para cada instanciação da cláusula no passo 4. A profundidade máxima de variável determina quantas vezes o passo 4 será executado.

Para ilustrar o passo de construção de *bottom clause* considere o exemplo a seguir(M. FERRO, 2007):

$:-modeh(1, sogra\_de(+mulher, -homem)).$

$:-modeb(*, progenitor\_de(+mulher, -mulher)).$

$:-modeb(1, esposa\_de(+mulher, -homem)).$

$:-determination(sogra\_de/2, progenitor\_de/2).$

$:-determination(sogra\_de/2, esposa\_de/2).$

/* declaração dos tipos */

$homem(pai1).$

$homem(marido1).$

$homem(marido2).$

$mulher(mae1).$

$mulher(filha11).$

$mulher(filha12).$

$mulher(neta11).$

/* Conhecimento preliminar */

$progenitor\_de(mae1, filha11).$

$progenitor\_de(pai1, filha11).$

$progenitor\_de(mae1, filha12).$

$progenitor\_de(pai1, filha12).$

$progenitor\_de(filha11, neta11).$

$esposa\_de(mae1, pai1).$

$esposa\_de(filha11, marido1).$

$esposa\_de(filha12, marido2).$

/* Exemplos positivos */

$sogra\_de(mae1, marido1).$

$sogra\_de(mae1, marido2).$

/* Exemplos negativos */

$sogra\_de(mae1, pai1).$

$sogra\_de(filha11, marido2).$

$sogra\_de(filha11, marido1).$

O algoritmo 2.2 corresponde ao passo de saturação do PROGOL responsável pela geração da *bottom clause* a partir de um exemplo positivo não provado. A lista *InTerms* irá guardar os termos que instanciam argumentos de entrada no predicado da cabeça, e os termos que instanciam argumentos de saída nos predicados do corpo. A função *hash* associa a cada termo uma variável diferente.

Podemos considerar a construção da *bottom clause* como uma árvore, onde cada vez que o passo 4 é executado, um nível da árvore é contruído.

Algoritmo 2.2 Algoritmo PROGOL para construção da *bottom clause*

Seja e um exemplo positivo não provado.
$hash : Terms \rightarrow N$ é uma função que unicamente mapeia termos variáveis diferentes.
1. Adicione e ao conhecimento preliminar.
2. $InTerms = \emptyset, \bot = \emptyset$.
3. Ache a primeira declaração mode de cabeça h tal que h subsume a com substituição θ
   Para cada v/t em θ,
   Se v corresponde a um tipo $\sharp$, substitua v em h por t
   Se v corresponde a um tipo $+$ ou $-$, substitua v em h por v_k onde v_k é uma variável tal que $k = hash(t)$
   Se v corresponde a um tipo $+$, adicione t ao conjunto $InTerms$.
   Adicione h a $\bot$.
4. Para cada declaração mode de corpo b
   Para toda substituição possível θ de argumentos correspondendo a tipo $+$ por termos no conjunto $InTerms$
   Repita quanta vezes for o número *Recall*
   Se Prolog tem sucesso no objetivo b com substituição θ'
   Para cada v/t em θ e θ'
   Se v corresponde a tipo $\sharp$, substitua v em b por t
   senão substitua v em b por v_k onde $k = hash(t)$
   Se v corresponde a tipo $-$, adicione t ao conjunto $InTerms$
   Adicione $\overline{b}$ a $\bot$
5. Incremente a profundidade de variável
6. Vá para o passo 4 se a profundidade máxima de variável não foi alcançada
7. Retorne $\bot$.

A construção da *bottom clause* considerando o exemplo positivo $sogra\_de(mae1, marido1)$ segue os passos abaixo.

No início do algoritmo, a lista *InTerms* e a *bottom clause* estão vazias. O primeiro passo do algoritmo é achar uma declaração *modeh* tal que h subsume o exemplo e com uma determinada substituição θ.

A primeira e única declaração *modeh* para este exemplo é:

$$modeh(1, sogra\_de(+mulher, -homem)).$$

A substituição θ é dada por:

$$\theta = \{+mulher/mae1, -homem/marido1\}.$$

Como os argumentos de *sogra_ de* correspondem a tipos + e −, os argumentos serão substituídos por duas variáveis novas A e B, retornando o predicado:

$$sogra\_de(A, B))$$

onde o A corresponderá a *mae1* e o B a *marido1*. O termo *mae1* é colocado na lista *InTerms* pois o argumento substituído por *mae1* é do tipo +. O predicado $sogra\_de(A, B))$ é colocado na *bottom clause*. Portanto temos:

$InTerms = \{mae1\}$, $\perp = \{sogra\_de(A, B)\}$

Neste momento temos o primeiro nível da árvore mostrado na figura 2.2.

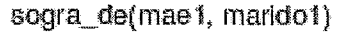

Figura 2.2: Árvore de construção da *bottom clause* - nível 1

Os próximos passos são feitos para cada declaração *modeb*. A primeira declaração é:

$modeb(*, progenitor\_de(+mulher, -mulher))$

As possíveis substituições θ de argumentos correspondentes a tipo + por termos no conjunto *InTerms* é dado por:

$\theta = \{+mulher/mae1\}$

Como o *recall* de *progenitor_ de* é $*$, será repetido o passo a seguir para cada fato do predicado *progenitor_ de* que tenha o termo *mae1* como primeiro argumento. O primeiro fato encontrado é:

$progenitor\_de(mae1, filha11)$

com substituição $\theta' = \{-mulher/filha11\}$.

Os argumentos de *progenitor_ de* são substituídos por variáveis, sendo que *mae1* já corresponde à variável A, e a variável C é introduzida para o termo *filha11.* O predicado *progenitor_ de(A,C)* é colocado na *bottom clause.* Como o termo *filha11* instancia um argumento do tipo −, ele é incluído na lista *InTerms.* Portanto temos:

$InTerms = \{mae1, filha11\}$, $\perp = \{sogra\_de(A, B), progenitor\_de(A, C)\}$

O segundo fato encontrado é:

$progenitor\_de(mae1, filha12)$

com substituição $\theta' = \{-mulher/filha12\}$.

Os argumentos de *progenitor_ de* são substituídos por variáveis, sendo que *mae1* já corresponde à variável A, e a variável D é introduzida para o termo *filha12.* O predicado *progenitor_ de(A,D)* é colocado na *bottom clause.* Como o termo *filha12*

instancia um argumento do tipo $-$, ele é incluído na lista *InTerms*. Portanto temos:

$$InTerms = \{mae1, filha11, filha12\},$$
$$\bot = \{sogra\_de(A, B), progenitor\_de(A, C), progenitor\_de(A, D)\}$$

Não há mais fatos para *progenitor_de* tendo *mae1* como primeiro argumento. Portanto passamos para a próxima declaração *modeb*, dada por:

$$modeb(1, esposa\_de(+mulher, -homem))$$

Consideramos a substituição $+mulher/mae1$. Como o *recall* de *esposa_de* é 1 será repetido o passo a seguir apenas uma vez. É procurado um fato com o predicado *esposa_de* que tenha *mae1* como primeiro argumento. O fato encontrado é:

$$esposa\_de(mae1, pai1)$$

com substituição $\theta' = \{-homem/pai1\}$.

Os argumentos de *esposa_de* são substituídos por variáveis, sendo que *mae1* já corresponde à variável A, e a variável E é introduzida para o termo *pai1*. O predicado *esposa_de(A,E)* é colocado na *bottom clause*. Como o termo *pai1* instancia um argumento do tipo $-$, ele é incluído na lista *InTerms*. Portanto temos:

$$InTerms = \{mae1, filha11, filha12, pai1\},$$
$$\bot = \{sogra\_de(A, B), progenitor\_de(A, C), progenitor\_de(A, D), esposa\_de(A, E)\}$$

Neste momento formamos o segundo nível da árvore de construção da *bottom clause*, como mostrado na figura 2.3.

Pela figura podemos ver que o nível 2 da árvore é formado por todos os fatos envolvendo os predicados do corpo *progenitor_de* e *esposa_de* que contenham como

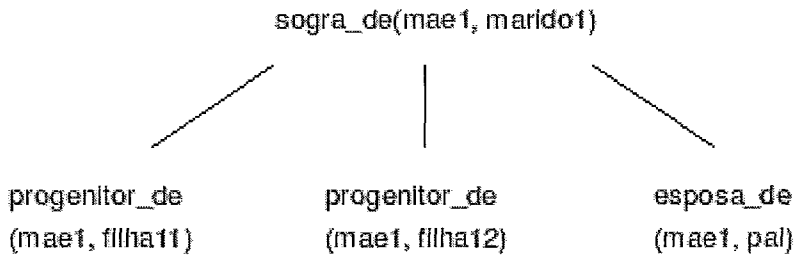

Figura 2.3: Árvore de construção da *bottom clause* - nível 2

argumento de entrada o termo *mae1*, que é justamente o termo de entrada vindo da cabeça da regra dada por $sogra\_de(mae1, marido1)$.

De acordo com a própria definição de variável de entrada, temos que uma variável de entrada em um literal do corpo deve ter aparecido antes como variável de saída em um outro literal do corpo, ou como variável de entrada no literal da cabeça. Como neste momento do algoritmo estamos formando os primeiros literais do corpo, são gerados literais cujas variáveis de entrada destes sejam variáveis de entrada do literal da cabeça, e até agora o único termo que representa uma variável de entrada no literal da cabeça é *mae1*.

Formado o segundo nível da árvore, passamos para o passo 5 do algoritmo que corresponde a incrementar a profundidade de variável. Podemos entender claramente agora o que significa esta profundidade. A profundidade de variável é um parâmetro que irá indicar quantos níveis poderemos ter na árvore de construção da *bottom clause*, e este parâmetro deverá ser definido no início do algoritmo. Em cada nível, novas variáveis serão introduzidas a partir dos termos do nível anterior. Se continuármos o algoritmo estaremos permitindo que novos literais do corpo sejam gerados a partir dos novos termos introduzidos no nível 1, dados por *filha11*, *filha12* e *marido1*. Estes termos foram introduzidos como argumentos de saída e portanto agora poderão ser argumentos de entrada nos novos literais do corpo que serão gerados.

Continuando o algoritmo e voltando para o passo 4 teremos:

A primeira declaração modeb encontrada é

$modeb(*, progenitor\_de(+mulher, -mulher))$

que terá o seu argumento de entrada substituído por *filha11* e *filha12*, ambos do tipo *mulher*. Como o *recall* é $*$, podemos buscar quantos fatos tiverem para as substituições dadas.

Apenas o fato $progenitor\_de(filha11, neta11)$ será encontrado. O termo *filha11* já está vinculado à variável C, e o termo *neta11* será vinculada à nova variável F. Portanto temos:

$$\perp = \{sogra\_de(A, B), progenitor\_de(A, C), progenitor\_de(A, D), esposa\_de(A, E), progenitor\_de(C, F)\}$$

Para a próxima declaração modeb dada por

$modeb(1, esposa\_de(+mulher, -homem))$

teremos o argumento de entrada substituído por *filha11* e *filha12*, ambos do tipo *mulher*. Não podemos utilizar o termo *pai1* pois este é do tipo *homem*. Como o *recall* é 1, só poderemos buscar um fato para *filha11* como primeiro argumento e um fato para *filha12* como primeiro argumento.

Os fatos encontrados serão $esposa\_de(filha11, marido1)$ e $esposa\_de(filha12, marido2)$. Apenas o termo marido2 não havia sido introduzido antes, e portanto é vinculado à nova variável G. Temos então:

$$\perp = \{sogra\_de(A, B), progenitor\_de(A, C), progenitor\_de(A, D), esposa\_de(A, E), progenitor\_de(C, F), esposa_de(C, G)\}$$

Neste momento formamos o terceiro nível da árvore de construção da bottom clause, mostrada na figura 2.4.

Se continuássemos o algritmo e fôssemos para o próximo nível da árvore, procuraríamos fatos que tivessem os termos *neta11*, *marido1* e *marido2* como argumentos de entrada nos literais do corpo, pois estes foram os novos termos introduzidos no nível anterior como argumentos de saída. Porém, olhando para o conhecimento pre-

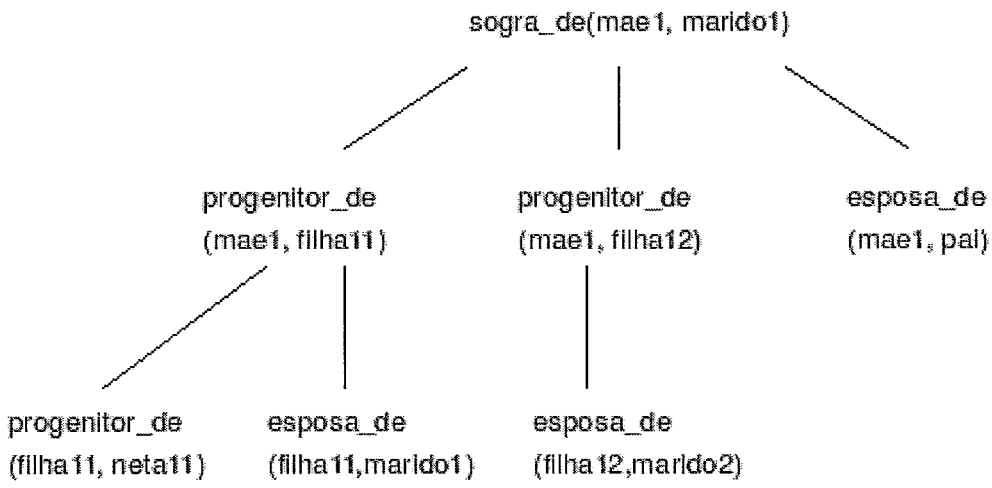

Figura 2.4: Árvore de construção da *bottom clause* - nível 3

liminar deste exemplo, veremos que tais fatos não existem e portanto o algoritmo irá terminar. Portanto a *bottom clause* final encontrada é:

$$\perp = \{sogra\_de(A,B), progenitor\_de(A,C), progenitor\_de(A,D), esposa\_de(A,E), progenitor\_de(C,F), esposa\_de(C,G)\}$$

2.4 Revisão de Teorias de Primeira-ordem a partir de Exemplos

Enquanto que a área de ILP tradicionalmente lida com o aprendizado de teorias clausais de primeira-ordem a partir do zero, dado um conjunto de exemplos e um conhecimento preliminar, esta mesma área também está focando em revisão de teorias, isto é, usar uma teoria de primeira-ordem como ponto de partida e revisá-la para que ela seja consistente com um conjunto de exemplos dados.

É possível que uma teoria previamente obtida esteja incorreta e/ou incompleta, seja porque ela foi extraída de um domínio especialista e se apoie em suposições incorretas, ou porque existem novos exemplos que não podem ser explicados pela teoria atual. Neste casos, já que a teoria original contém informação relevante, seria vantajoso utilizar a teoria original como ponto de partida no processo de aprendizado,

reparando-a ou melhorando-a.

A tarefa de revisar envolve mudar o conjunto de respostas da teoria dada, ou seja, melhorar a sua capacidade de inferência adicionando-se respostas que haviam sido perdidas ou removendo respostas incorretas. No primeiro caso estamos falando de generalização, e no segundo de especialização.

As teorias de primeira-ordem sendo revisadas normalmente possuem cláusulas que contém mais de um predicado. Portanto, a revisão de teorias deve ser capaz não apenas de modificar cláusulas individuais que correspondem a apenas um predicado, mas também deve saber lidar com o aprendizado de múltiplos predicados simultaneamente. A modificação da definição de um predicado geralmente afeta a corretude de outros predicados que usam o primeiro em suas definições. Portanto, a revisão de um predicado não pode ser feita sem levar em consideração os outros predicados.

Em revisão de teorias, nós começamos com uma teoria inicial e então a modificamos. Estamos assumindo que o conhecimento preliminar já está incluído na teoria dada, e que pode ser modificado. Em alguns sistemas, a revisão em certas partes da teoria pode ser impedida pelo usuário, de forma a não deixar alterar o conhecimento preliminar. Considerando a linguagem de teorias como sendo de cláusulas definidas e a linguagem de exemplos como sendo de átomos básicos, o problema de revisão de teorias pode ser definido como se segue(WROBEL,1996):

Definição 2.3. *Dados:*

- *Uma Teoria inicial H que pode ser modificada*
- *Um conhecimento preliminar invariante BK*
- *Um conjunto de exemplos positivos E_+ e negativos E_-*

  Achar:

- *Uma teoria revisada H' que, junto com o conhecimento preliminar BK, implique logicamente todos os exemplos positivos (completo), $BK \cup H' \models E^+$ e nenhum dos exemplos negativos (consistência), $\forall e^- \in E^- : BK \cup H' \not\models e^-$, e que esteja de acordo com o critério de qualidade, tal como o de minimalidade.*

O esquema da tarefa de revisão de teorias é exibido na Figura 2.5. Observe que ILP pode ser visto como um subconjunto da revisão de teoria. Nesse caso a teoria inicial que pode ser modificada seria vazia.

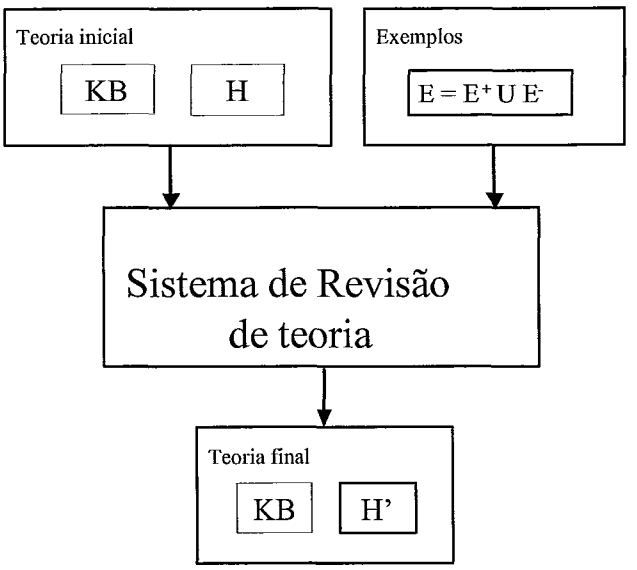

Figura 2.5: Esquema de um sistema de Revisão de teoria, onde KB é o conhecimento preliminar, H' é a teoria inicial que pode ser modificada, E é o conjunto de exemplos positivos (E^+) e negativos (E^-) e H' é a teoria revisada pelo sistema

A maioria dos sistemas de revisão usa teorias de cláusulas de Horn para definir teorias e átomos básicos ou fatos para definir os exemplos.

Sistemas de revisão de teorias assumem que a teoria inicial é aproximadamente correta. Sendo assim, apenas alguns pontos na teoria impedem que ela esteja totalmente correta. A idéia é, portanto, buscar tais pontos na teoria e revisá-los, ao invés de aprender uma nova teoria do zero. Geralmente, sistemas de revisão de teorias são compostos por três etapas de busca. Primeiro, eles buscam por pontos na teoria responsáveis pela classificação incorreta de alguns exemplos. Segundo, eles buscam por possíveis modificações nestes pontos, através do uso de operadores de revisão. E finalmente, eles buscam pela melhor modificação entre todas as revisões propostas.

2.4.1 FORTE

Um exemplo de sistema de revisão de teorias é o FORTE (*First-Order Revision of Theories from Examples*) (RICHARDS, MOONEY, 1995). FORTE é um sistema que revisa automaticamente bases de conhecimento(teorias) que contém

cláusulas Horn de primeira-ordem livres de função. O FORTE executa uma busca *hill climbing* através do espaço de operadores de especialização e generalização, na tentativa de achar uma revisão mínima para uma teoria que a torne consistente com o conjunto de exemplos de treinamento. Dada uma teoria inicial incorreta e um conjunto consistente de exemplos positivos e negativos, o FORTE deve achar uma teoria revisada que é consistente com os exemplos dados. FORTE revisa teorias iterativamente, usando uma busca *hill climbing* (ver Algoritmo 2.3). Cada iteração identifica pontos na teoria, chamados *pontos de revisão*, onde a revisão tem o potencial de melhorar a acurácia da teoria. É gerado então um conjunto de revisões, baseado nos pontos de revisão, selecionada a melhor revisão e implementada caso melhore a acurácia lógica da teoria. Para a nova teoria revisada, novos pontos de revisão são gerados. O processo de iteração continua até que nenhuma revisão melhore a teoria.

No FORTE, cada exemplo pode ser uma instância ou um conjunto de instâncias, sendo que se for um conjunto, as instâncias podem ser de predicados diferentes. Definimos instância como um átomo básico cujo predicado é um dos conceitos a serem aprendidos. Cada exemplo possui o seu próprio conhecimento preliminar (BK), e pode existir um BK comum para todos os exemplos. Ao contrário do que é feito em sistemas de aprendizado de teorias, o FORTE é capaz de aprender múltiplos predicados simultaneamente.

Para gerar pontos de revisão, a teoria corrente é testada no conjunto de treinamento. FORTE anota provas que falharam em instâncias positivas e provas que tiveram sucesso em instâncias negativas. A partir destas anotações ele identifica pontos na teoria para possível revisão. Cada ponto de revisão tem um potencial, que é o crescimento máximo em desempenho que pode ser proporcionado à teoria a partir de modificações nesse ponto (quantidade de instâncias que verificaram a necessidade de revisão no ponto). A pontuação de um operador é o crescimento real proporcionado ao desempenho da teoria após a revisão no ponto em que o operador foi aplicado (diferença entre instâncias que passaram a ser classificadas corretamente e instâncias que deixaram de ser classificadas corretamente). FORTE começa gerando revisões para o ponto de revisão com maior potencial. Para cada

ponto de revisão escolhe-se a melhor revisão proposta, proporcionada pelo operador de maior pontuação. A revisão que realmente será efetuada é aquela definida como sendo a melhor revisão proposta entre todas. Este processo está descrito no algoritmo 2.3 a seguir.

Algoritmo 2.3 Algoritmo FORTE, proposto em (RICHARDS, MOONEY, 1995)

repita
  Gera pontos de revisão
  Ordena pontos de revisão por potencial (do maior para menor)
  para cada ponto de revisão
    Gera possíveis revisões
    Atualiza melhor revisão encontrada
  até que o potencial do próximo ponto de revisão seja menor que a pontuação da melhor revisão atualizada
  se a melhor revisão melhorar a teoria
    Implementa melhor revisão
  fim se
até que nenhuma revisão melhore a teoria

Pontos de Revisão

Quando a teoria a ser revista é constituída de apenas um predicado, a instância selecionada, positiva ou negativa, indica que operação deve ser feita, se será uma generalização ou especialização.

No caso de uma teoria com múltiplas definições de predicados muitas cláusulas podem estar envolvidas na prova de uma instância negativa ou na não prova de uma instância positiva. Dessa forma a indicação do tipo da operação não é imediata tornando-se necessária a determinação dos pontos nessa teoria que precisam ser corrigidos (pontos de revisão). Dependendo do tipo de instância que esteja sendo considerada podemos ter a definição de dois tipos de pontos de revisão: pontos de especialização e pontos de generalização. Pontos na teoria onde a prova de instâncias positivas falha são lugares onde a teoria precisa ser generalizada, e cláusulas usadas em provas de instâncias negativas são pontos onde a teoria precisa ser especializada.

A generalização ou especialização é efetivada pelos operadores de revisão. A especificação do ponto de revisão determina o tipo de operador de revisão que precisará ser aplicado para tornar a teoria consistente com a base de dados. A seguir descrevemos alguns dos operadores utilizados pelo FORTE (RICHARDS, MOONEY,

1995).

Operadores de Especialização

Alguns operadores de especialização são:

- **Exclusão de regra:** esse operador remove inteiramente uma regra incorreta, sendo que se essa regra for a única para o conceito na cabeça, ela é substituída pela regra *conceito* : $-fail$.

- **Adição de antecedentes:** esse operador adiciona antecedentes em uma cláusula incorreta para excluir instâncias negativas que estão sendo provadas, não permitindo ao máximo possível que instâncias positivas deixem de ser provadas.

Operadores de Generalização

Diferentemente dos operadores de especialização que só modificam cláusulas existentes, os operadores de generalização podem adicionar cláusulas completamente novas. Como o objetivo é cobrir instâncias positivas não cobertas, qualquer operador de ILP que aceite instâncias positivas como entrada poderia ser utilizado. Abaixo definimos os operadores utilizados pelo FORTE:

- **Exclusão de antecedentes:** Esse operador exclui antecedentes de cláusulas que poderiam ser utilizadas para provar instâncias positivas. Pode ser executado usando dois métodos: *Hill-climbing* e *exclusão de múltiplos antecedentes.* O *Hill-climbing* exclui um antecedente de cada vez na cláusula, selecionando em cada vez o antecedente cuja exclusão permita que o máximo de instâncias positivas passem a ser provadas enquanto que não prova nenhuma instância negativa. Esse processo é repetido até que o desempenho da teoria não possa ser melhorado. O método de *exclusão de múltiplos antecedentes* exclui mais de um antecedente de uma vez, já que algumas vezes a prova de instâncias apenas é afetada com a exclusão de mais de um antecedente de uma só vez. Nesse método, primeiro todos os antecedentes cuja exclusão não permite que instâncias negativas sejam provadas, são reunidos; então combinações destes

antecedentes são feitas na procura daquela que permita o maior número de instâncias positivas serem provadas sem provar negativas. O algoritmo não pára quando todos as instâncias positivas foram provadas, continua excluindo tantos antecedentes quanto puder. Esse algoritmo é computacionalmente caro; entretanto, ele só é chamado quando o *hill climbing* não proporciona nenhuma revisão.

- **Adição de regra:** Esse operador deixa a cláusula original na teoria e gera novas cláusulas baseadas nesta, objetivando criar novas versões da regra que permitam a prova das instâncias que identificaram a regra original como um ponto de falha. O processo é feito em duas etapas. Primeiro copia a cláusula original e usando o algoritmo de exclusão de antecedente com *hill-climbing*, exclui antecedentes sem permitir que nenhuma instância negativa seja provada e permitindo também que algumas instâncias positivas, que antes não eram provadas passem a ser (mesmo que isso permita a prova de instâncias negativas). Então cria uma ou mais especializações desta regra, utilizando o operador de adição de antecedentes, para permitir a prova das instâncias positivas desejadas enquanto elimina a prova das instâncias negativas.

- **Identificação:** Constrói uma cláusula nova para generalizar a definição de um antecedente que falhou na prova de uma instância positiva. Melhor que desenvolver a cláusula a partir do zero, executa um passo de resolução inversa (MUGGLETON, 1992a) usando duas regras existentes no domínio da teoria.

- **Absorção:** Substitui uma cláusula já existente por uma nova a partir da substituição dos seus antecedentes falhos pela cabeça de uma cláusula, cujo o conjunto dos seus antecedentes contém os antecedentes definidos como sendo falhos.

Para mais detalhes destes operadores nos referimos a (RICHARDS, MOONEY, 1995).

Tanto operadores de especialização quanto de generalização utilizam a operação de adição de antecedentes. O primeiro o faz no próprio operador de adição

de antecedentes, enquanto que o segundo o faz no operador de adição de regras, no passo em que se cria especializações da cláusula. A busca pelos antecedentes a serem adicionados pode ser feita utilizando-se dois algoritmos: *hill climbing* e *relational pathfinding*, que serão explicados mais adiante.

O FORTE permite ao usuário definir parâmetros sobre a base de dados e sobre o processo de aprendizado. Os parâmetros descritos a seguir serão relevantes para entender que tipo de antecedente é adicionado à cláusula sendo revisada(RICHARDS B. L.):

- **Predicados alto nível:** Especificam os conceitos a serem aprendidos, e os tipos de seus argumentos. São os predicados que estão nas cabeças das regras, e para os quais existem exemplos na base de dados.
- **Predicados intermediários:** São predicados que podem estar na cabeça de alguma regra, mas para os quais não existem exemplos na base de dados.
- ***Object_Relations*:** São os predicados de mais baixo nível, ou seja, que não estão na cabeça de nenhuma regra. Tais predicados definem o conhecimento preliminar.
- ***Object_Attributes*:** Impõe restrições nos argumentos, como por exemplo definir os valores possíveis de um determinado argumento.

O Usuário pode definir que tipo de antecedente vai ser considerado na adição de antecedentes. Através dos parâmetros *use attr*, *use relations* e *use theory*, o usuário escolhe se vai querer adicionar antecedentes que representem atributos definidos no *Object_Attributes*, que sejam predicados especificados no *Object_Relations*, ou que sejam quaisquer outros predicados na teoria além dos que estão definidos no *Object_Relations*, respectivamente.

Adição de antecedentes utilizando *hill climbing*

O Algortimo de adição de antecedentes utilizando a busca *hill climbing* é mostrado em Algoritmo 2.4.

Algoritmo 2.4 Algoritmo de adição de antecedentes *hill climbing*

T = Teoria inicial
C = Cláusula incorreta
CP = Conjunto de instâncias positivas provadas pela cláusula
IP = Conjunto de instâncias negativas provadas pela cláusula
$C\_original \leftarrow C$
$CP\_original \leftarrow CP$
$IP\_original \leftarrow IP$
repita
  repita
    Calcule o score de C a partir de CP e IP
    Retorne os possíveis antecedentes e seus respectivos scores
    Escolha o melhor antecedente
    Adicione o antecedente à cláusula C, formando a cláusula C'
    $NovoCP$ = Instâncias positivas provadas por C'
    $NovoIP$ = Instâncias negativas provadas por C'
    $C \leftarrow C'$
    $CP \leftarrow NovoCP$
    $IP \leftarrow NovoIP$
  até que não se tenha mais antecedentes ou o score da cláusula não seja mais melhorado.
  Se a nova cláusula C' é diferente da cláusula original $C\_original$ **então**
    Acrescente a nova cláusula C' à teoria T, formando a nova teoria T'
    Calcule o novo conjunto de instâncias positivas provadas($NovoCP\_Teoria$) e instâncias negativas provadas($NovoIP\_Teoria$) pela nova teoria
    $C \leftarrow C\_original$
    $CP \leftarrow CP\_original - NovoCP\_Teoria$
    $IP \leftarrow IP\_original - NovoIP\_Teoria$
    $T \leftarrow T'$
  Fim Se
Até que não consiga mais especializar a cláusula, ou até que todas as instâncias positivas provadas pela cláusula original tenham sido provadas pelas novas cláusulas criadas

A operação de adição de antecedentes utilizando *hill climbing* é baseado no algoritmo do FOIL, e adiciona um antecedente de cada vez, procurando pelo que proporciona melhor desempenho à teoria. Como qualquer algoritmo *hill climbing*, o algoritmo de adição de antecedentes pode cair num máximo local. O algoritmo 2.4 descrito acima parte de uma teoria inicial a ser revisada, uma cláusula nesta teoria que está incorreta e que precisa ser modificada, e os conjuntos de instâncias positivas provadas e negativas provadas por esta cláusula. O algoritmo possui duas iterações, uma interna e uma externa. A iteração externa gera um conjunto de novas cláusulas

que são especializações da cláusula original incorreta. A cada nova cláusula que é gerada, esta é adicionada a teoria corrente, que será avaliada considerando-se o conjunto de instâncias de exemplos de treinamento, formando um novo conjunto de instâncias positivas provadas e instâncias negativas provadas pela teoria, sendo que esta nova teoria só conterá a cláusula original se estivermos utilizando o operador de generalização de adição de regras. Se alguma instância positiva que estava sendo provada antes deixou de ser provada, recomeça-se o algoritmo com a cláusula original, procurando especializações alternativas que retenham a prova das outras instâncias positivas enquanto ainda eliminam as negativas. A iteração interna é responsável pela geração de uma única cláusula modificada. A cada iteração um novo antecedente é adicionado à cláusula sendo formada, e a iteração só termina quando não houver mais antecedentes a serem adicionados ou quando o *score* da cláusula não estiver mais sendo melhorado. A iteração interna começa calculando-se o *score* da cláusula sendo revisada baseado nos conjuntos de instâncias positivas e negativas provadas por esta claúsula, dados por CP e IP, respectivamente. O *score* utilizado é o *score* do FOIL, dado por

$$-log(\frac{tamanho(CP)}{tamanho(CP) + tamanho(IP)}).$$

Depois disso são retornados todos os possíveis antecedentes a serem adicionados à cláusula, cada um com seu *score*. A busca por antecedentes feita pelo *hill climbing* está especificada no algoritmo 2.5 e será explicada a seguir. Dentre os antecedentes retornados, aquele que tiver o melhor *score* será adicionado à cláusula, formando uma nova cláusula. Neste ponto a iteração interna recomeça, só que considerando a nova cláusula até o momento e os conjuntos de instâncias positivas e negativas provadas por esta cláusula em formação.

O *hill climbing* utiliza como espaço de busca de antecedentes todos os possíveis literais da base de conhecimento, ou seja, todos os predicados intermediários e os predicados definidos no *object_relations*. O algoritmo de busca de antecedentes *hill climbing* está descrito em Algoritmo 2.5 a seguir.

O algoritmo 2.5 parte da cláusula sendo revisada. Para cada literal existente na

Algoritmo 2.5 Algoritmo de busca de antecedentes utilizando *hill climbing*

C = Cláusula sendo revisada
Para cada literal da base de cohecimento
    Retorne as variáveis e seus tipos em C
    Retorne os argumentos e seus tipos no literal em questão
    Para cada combinação de variáveis em C
        Verifique se a combinação é válida como possíveis argumentos do literal
        Se é válida **então**
            Substitua os argumentos do literal pela combinação das variáveis
            Retorne o literal como antecedente
        Senão
            Passe para a próxima combinação de variáveis
    Fim para
    N = número de argumentos do literal em questão
    $i = 1$
    Enquanto $i \leq N$ - 1
        Crie uma variável nova L
        V = Variáveis de $C + L$
        Para cada combinação em V que inclua L e alguma variável de C
            Verifique se a combinação é válida como possíveis argumentos do literal
            Se é válida **então**
                Substitua os argumentos do literal pela combinação das variáveis
                Retorne o literal como antecedente
            Senão
                Passe para a próxima combinação de variáveis
        Fim para
        $i = i + 1$
    Fim Enquanto
    Passe para o próximo literal da base de conhecimento
Fim para

base de conhecimento, o algoritmo retornará todas as instanciações feitas neste literal ao substituir seus argumentos por combinações das variáveis existente na cláusula sendo revisada. Primeiramente são retornadas quais são as variáveis da cláusula e seus respectivos tipos. Depois são retornados os tipos dos argumentos do literal em questão. A partir dessas informações são feitas todas as combinações possíveis das variáveis da cláusula. O objetivo é usar estas combinações como argumentos do literal, ou seja, deseja-se instanciar os argumentos com as variáveis disponíveis. As combinações aceitas serão aquelas que não entram em contradição com a tipagem definida para os argumentos do literal. Depois que todas as combinações possíveis foram feitas, o algoritmo irá criar $N-1$ variáveis novas que poderão ser introduzidas

como argumentos do literal, sendo N o número de argumentos da cláusula. Novas combinações serão feitas considerando essas novas variáveis. O número de novas váriáveis será sempre uma a menos do que o número de argumentos do literal, pois pelo menos um dos argumentos deve ser uma variável que já existe na cláusula, pois o novo antecedente adicionado deve estar relacionado aos outros literais da cláusula, tendo pelo menos uma variável de ligação. Só depois que se esgotaram todas as possíveis combinações de variáveis para um determinado literal, é que será considerado o próximo literal da base de conhecimento. O algoritmo continua até que todos os literais da base de conhecimento tenham sido vistos.

Para exemplicar a busca de antecedentes feita pelo *hill climbing* definimos a seguir uma parte da base de dados Amine(KING, et al., 1995), mostrando apenas a definição do *object_relations* e dos predicados de alto nível, incluindo os tipos de cada literal definido. Iremos apenas considerar o *object_relations* como espaço de busca de antecedentes de forma a simplificar o exemplo.

predicados_alto_nivel([great_ne(a,a)]).

object_relations([*x_subst(a,n,b), alk_groups(a,n), r_subst_1(a,l), r_subst_2(a,m), r_subst_3(a,n), ring_substitutions(a,n), ring_subst_1(a,b), ring_subst_2(a,b), ring_subst_3(a,b), ring_subst_4(a,b), ring_subst_5(a,b), ring_subst_6(a,b), polar(b,c), size(b,d), flex(b,e), h_doner(b,f), h_acceptor(b,g), pi_doner(b,h), pi_acceptor(b,i), polarisable(b,j), sigma(b,k), n_val(a,n), gt(n,n), great_polar(c,c), great_size(d,d), great_flex(e,e), great_h_don(f,f), great_h_acc(g,g), great_pi_don(h,h), great_pi_acc(i,i), great_polari(j,j), great_sigma(k,k)*]).

Suponha que a cláusula a ser revisada seja *great_ne(A,B)*, que não contém antecedentes em seu corpo. O primeiro literal da base de conhecimento considerado é o *x_subst(a,n,b)*, que é o primeiro literal do *object_relations*. O algoritmo irá considerar as variáveis da cláusula sendo revisada, com seus respectivos tipos, ou seja, [[A,a],[B,a]], onde ambas as variáveis A e B possuem tipo a, de acordo com o

que está definido em *predicados_alto_nivel*. Para o literal em questão deverão ser gerados três argumentos, ou seja, as combinações que serão feitas devem possuir três variáveis, cada uma instanciando um argumento do literal, sendo que os argumentos podem ter variáveis repetidas. Para as variáveis A e B da cláusula, as seguintes combinações são retornadas:

BBB,BBA,BAB,BAA,ABB,ABA,AAB,AAA.

Para a primeira combinação acima, BBB, a variável B se repete em todos os três argumentos. A variável B possui tipo a, e portanto para que a combinação seja válida os argumentos do literal devem possuir todos o tipo a, o que não acontece, já que os dois últimos argumentos do literal *x_subst(a,n,b)* possuem tipos n e b respectivamente. Seguindo a mesma linha de raciocínio, nenhuma das combinações apresentadas será válida, já que o segundo argumento do literal possui tipo n, e nem a variável A nem a B possuem tipo n.

Esgotadas as combinações com as variáveis existentes, o próximo passo do algoritmo é criar variáveis novas. Como o literal *x_subst(a,n,b)* possui três argumentos, apenas duas variáveis novas poderão ser criadas. Primeiramente é criada a variável C, que a príncipio não tem tipo definido. Agora serão geradas novas combinações, só que considerando a variável C, que passará a ter o tipo do argumento que ela estiver representando. As combinações que serão feitas devem obrigatoriamente possuir ou a variável A ou a B, que são as variáveis da cláusula original. As seguintes combinações são retornadas:

AAC,BBC,ABC,BAC,ACA,BCB,ACB,BCA,CAA,CBB,CAB,CBA,ACC,BCC, CCA,CCB,CAC,CBC.

Nenhuma destas combinações será valida, porque mesmo que a variável C substitua o argumento de tipo n, as variáveis A e B não podem substituir o terceiro argumento que possui tipo b, pois elas são do tipo a, e tampouco o pode a variável C, que já estará representando o argumento de tipo n. O mesmo acontece se a variável

C tivesse substituindo o argumento de tipo *b*. Diante disso, uma nova variável D será criada. As novas combinações considerando a variável D são:

ACD,BCD,CDA,CDB.

Percebe-se que obrigatoriamente é colocada a variável C nas combinações, pois agora estão sendo consideradas duas variáveis novas como argumentos. Entre as combinações acima, duas serão válidas: ACD e BCD, pois elas possuem ou a variável A ou a variável B como primeiro argumento, e como este tem tipo *a*, tanto A e B podem substituí-los, pois também tem tipo *a*. Os outros dois argumentos são substituídos por C e D, que não possuem tipos definidos, e que unificarão com os tipos *n* e *b*.

Para o literal *x_subst(a,n,b)* os seguintes antecedentes serão retornados:

[x_subst(A,C,D),x_subst(B,C,D)].

O algoritmo agora passará para o próximo literal do *object_relations* e fará o mesmo procedimento acima. Os antecedentes retornados após considerar todos os literais da base de conhecimento serão:

[x_subst(B,C,D), x_subst(A,C,D), alk_groups(B,C), alk_groups(A,C), r_subst_1(B,C), r_subst_1(A,C), r_subst_2(B,C), r_subst_2(A,C), r_subst_3(B,C), r_subst_3(A,C), ring_substitutions(B,C),ring_substitutions(A,C), ring_subst_1(B,C), ring_subst_1(A,C), ring_subst_2(B,C), ring_subst_2(A,C), ring_subst_3(B,C), ring_subst_3(A,C), ring_subst_4(B,C), ring_subst_4(A,C), ring_subst_5(B,C), ring_subst_5(A,C), ring_subst_6(B,C), ring_subst_6(A,C), n_val(B,C), n_val(A,C)].

Adição de antecedentes utilizando *Relational Pathfinding*:

Relational pathfinding(RICHARDS, MOONEY, 1992) é um método de adição de antecedentes desenvolvido para escapar do máximo local. Neste caso, vários antecedentes são adicionados de uma só vez, já que algumas vezes nenhum dos antecedentes diminui o desempenho, mas também não aumenta. O *relational pathfinding* é baseado na suposição de que, em domínios relacionais, geralmente existe um caminho de tamanho fixo de relações que ligam um conjunto de termos satisfazendo o conceito alvo. Considere, por exemplo, uma porção do domínio de uma família mostrada abaixo:

```
/*exemplos positivos*/
avos(pedro,ana).
avos(maria,ana).

/* exemplos negativos */
avos(pedro,arthur).

/* fatos */
casados(pedro,maria).
pais(pedro,arthur).
pais(maria,arthur).
pais(pedro,vitoria).
pais(maria,vitoria).
casados(vitoria,joao).
pais(vitoria,ana).
pais(joao,ana).
```

Figura 2.6: Parte do domínio de uma família

Para provar o exemplo positivo *avos(pedro,ana)*, ou seja, para relacionar o termo *pedro* ao termo *ana* através da relação *avos*, é preciso considerar os fatos *pais(pedro,vitoria)* e *pais(vitoria,ana)*, pois *pedro* se relaciona com *vitoria* através da relação *pais*, e *vitória* se relaciona com *ana* também atráves da relação *pais*. O mesmo acontece ao provar o exemplo positivo *avos(maria,ana)*, que considera os fatos *pais(maria,vitoria)* e *pais(vitoria,ana)*. Percebe-se que ambos os exemplos positivos são provados considerando duas relações *pais*, e que o exemplo negativo *avos(pedro,arthur)* não consegue ser provado dessa forma, pois basta uma relação *pais* para relacionar os termos *pedro* e *arthur*, definida pelo fato *pais(pedro,arthur)*.

Portanto, um algoritmo de aprendizado de teorias que estivesse aprendendo o conceito *avos* retornaria a regra dada por:

avos(A,B) :- pais(A,C),pais(C,B).

Esta regra mostra que existe um caminho de tamanho fixo de relações, neste caso duas relações *pais*, que ligam um conjunto de termos, representados pelas variáveis A e B, satisfazendo o conceito alvo, dado por *avos*.

Para visualizar claramente as relações existentes entre termos de um domínio relacional, podemos ver este domínio como um grafo, onde os nós são os termos e as arestas são as relações. Para o exemplo acima teríamos o seguinte grafo:

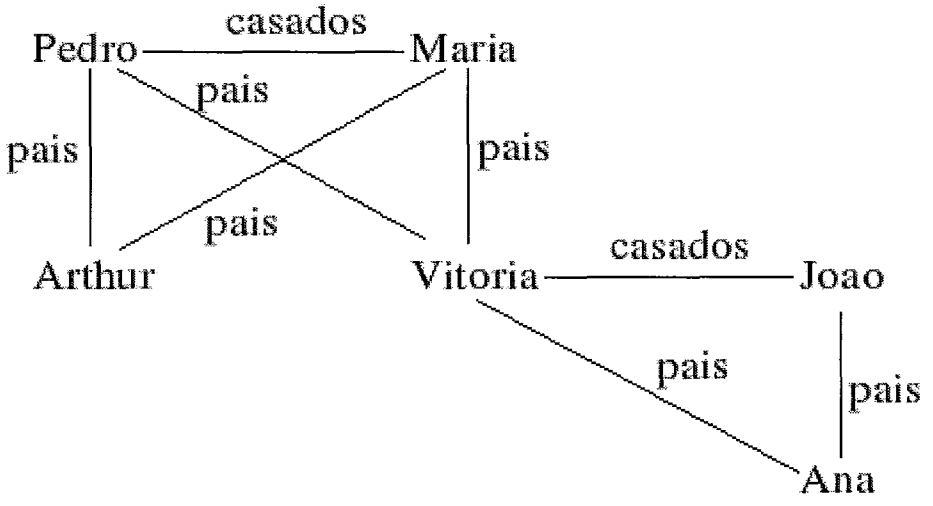

Figura 2.7: Grafo do domínio relacional

Na visão de um grafo, podemos definir *caminho relacional* como sendo o conjunto de arestas, ou relações, que ligam nós, ou termos, do grafo. No exemplo acima, o caminho relacional entre os termos *pedro* e *ana* é formado pela relação *pais* que liga *pedro* a *vitória*, e pela relação *pais* que liga *vitoria* a *ana*. Tal caminho é escrito como *pais(pedro,vitoria),pais(vitoria,ana)*, e define o conceito *avos* dado pela instância positiva *avos(pedro,ana)*, como visto anteriormente.

O *relational pathfinding* portanto, é um algoritmo para encontrar tais caminhos relacionais, considerando um domínio dado, já que na maioria dos domínios

relacionais importantes conceitos são representados por um pequeno número de caminhos fixos entre os termos que definem uma instância positiva. Do ponto de vista de revisão de teorias, o *Relational pathfinding* pode ser usado toda vez que uma cláusula precisa ser revisada e não possui caminhos relacionais ligando todas as suas variáveis. Neste caso é escolhido um exemplo positivo provado pela cláusula, para instanciar a cláusula original, e a partir daí buscar caminhos relacionais para os termos na cláusula que não se relacionam. *Relational pathfinding*, conforme descrito no algoritmo 2.6, acha caminhos através da expansão sucessiva dos nós associados aos termos em uma instância positiva, onde expansão pode ser vista como encontrar relações (arestas) partindo de um determinado nó e alcançando um nó no próximo nível do grafo.

O algoritmo 2.6 parte de uma cláusula instanciada por um exemplo positivo provado por esta cláusula. Os termos na regra instanciada serão nós no grafo do domínio, possivelmente conectados por relações definidas no corpo da regra. O algoritmo de adição de antecedentes constrói o grafo iterativamente, partindo dos nós iniciais e expandindo estes nós até que caminhos relacionando estes nós sejam encontrados. Quando explicamos o que seria um nó num grafo relacional, associamos cada termo do domínio a um nó no grafo, porém, também podemos considerar como nó um conjunto de termos que se relacionam entre si e formam um subgrafo isolado, e que em determinado momento do algoritmo não possui um caminho relacional que o ligue a outro nó do grafo. Chamamos de *end values* aos termos dos nós criados a partir da expansão de um determinado nó.

A explicação do algoritmo será feita utilizando-se exemplos que consideram uma parte da base de dados Amine(KING, et al., 1995) mostrada abaixo:

predicados_alto_nivel([great_ne(a,a)]).

object_relations([x_subst(a,n,b), alk_groups(a,n), r_subst_1(a,l), r_subst_2(a,m), r_subst_3(a,n), ring_substitutions(a,n), ring_subst_1(a,b), ring_subst_2(a,b), ring_subst_3(a,b), ring_subst_4(a,b), ring_subst_5(a,b), ring_subst_6(a,b), polar(b,c), size(b,d), flex(b,e), h_doner(b,f), h_acceptor(b,g),

Algoritmo 2.6 Algoritmo de adição de antecedentes *relational pathfinding*

Instancie a cláusula original com uma instância positiva provada escolhida aleatoriamente
Ache os subgrafos isolados
Para cada subgrafo
    Transforme os termos em *end values* iniciais
Fim para
repita
  Para cada subgrafo
    Encontre todas as relações partindo deste subgrafo
    Remova arestas que alcançam *end values* previamente vistos
  Fim cada
Até que uma interseção seja achada ou o limite de memória seja excedido
Se uma ou mais interseções foram achadas
  Para cada interseção
    Se o caminho formado contém variáveis singletons
      Adicione relações que usem as variáveis singletons
      Se não conseguiu usar todos os singletons
        Descarte o caminho
      Fim se
    Fim se
    Adicione os caminhos relacionais achado à cláusula original formando um conjunto de novas cláusulas
    Substitua os termos nas cláusulas por variáveis
  Fim para
  Selecione a cláusula com melhor acurácia
Fim se
Se cláusula selecionada continua provando instâncias negativas
    Adicione mais antecedentes usando o algoritmo *hill climbing*
Fim se

pi_doner(b,h), pi_acceptor(b,i), polarisable(b,j), sigma(b,k), n_val(a,n), gt(n,n), great_polar(c,c), great_size(d,d), great_flex(e,e), great_h_don(f,f), great_h_acc(g,g), great_pi_don(h,h), great_pi_acc(i,i), great_polari(j,j), great_sigma(k,k)]).

/* instâncias positivas */

great_ne(b1,a1).

great_ne(p1,w1).

great_ne(w1,x1).

/* instâncias negativas */

great_ne(a1,b1).

great_ne(c1,u1).

/* fatos */

alk_groups(a1,0).

alk_groups(e1,0).

r_subst_1(b1,h).

r_subst_1(d1,h).

r_subst_1(e1,h).

r_subst_2(a1,cl).

ring_subst_2(w1,och3).

ring_subst_3(x1,och3).

polar(och3,polar2).

x_subst(b1,7,cl).

gt(1,0).

O primeiro passo do algoritmo é encontrar os subgrafos isolados que serão os nós do grafo que está sendo construído. Cada nó, ou subgrafo, é formado por cada termo da cabeça da cláusula juntamente com os termos do corpo da cláusula, aos quais se relacionam através de algum caminho relacional. Seguem quatro exemplos para entender como que funciona. Os subgrafos gerados em cada exemplo são mostrados na figura 2.8:

1) Cláusula original: *great_ne(A,B) :- alk_groups(B,C), r_subst_1(A,D).* Se esta cláusula for instanciada por *great_ne(b1,a1)*, e considerando os fatos definidos, teremos:

great_ne(b1,a1) :- alk_groups(a1,0), r_subst_1(b1,h).

O termo *b1* se relaciona com o termo *h* através da relação *r_subst_1(b1,h)*, e o termo *a1* se relaciona com o termo *0* através da relação *alk_groups(a1,0)*. Portanto

os nós gerados são [*a1,0*] e [*b1,h*], que formam subgrafos isolados.

2) Cláusula original: *great_ne(A,B) :- ring_subst_2(B,C), ring_subst_3(D,C).* Se esta cláusula for instanciada por *great_ne(p1,w1)*, teremos:

great_ne(p1,w1) :- ring_subst_2(w1,och3), ring_subst_3(x1,och3).

O termo *w1* se relaciona com o termo *och3* através da relação *ring_subst_2(w1,och3)*, e o termo *och3*, por sua vez, se relaciona com o termo *x1* através da relação *ring_subst_3(x1,och3)*, portanto existe um caminho relacional vinculado os termos *w1*, *x1* e *och3*, e portanto estes termos formarão um subgrafo isolado. Já o termo *p1* não se relaciona com nenhum outro termo na cláusula, e será um outro nó isolado. Portanto os nós gerados são [*p1*] e [*w1,x1,och3*].

3) Cláusula original: *great_ne(A,B) :- ring_subst_2(A,C), polar(C,E), ring_subst_3(B,C).* Se esta cláusula for instanciada por *great_ne(w1,x1)*, teremos:

great_ne(w1,x1) :- ring_subst_2(w1,och3), polar(och3,polar2), ring_subst_3(x1,och3).

O termo *w1* se relaciona com o termo *och3* através da relação *ring_subst_2(w1,och3).* O termo *och3*, por sua vez, se relaciona com o termo *polar2* através da relação *polar(och3,polar2)*, e com o termo *x1*, através da relação *ring_subst_3(x1,och3).* Existe um caminho relacional que vincula todos os termos, e portanto apenas o nó [*w1,polar2,och3,x1*] é gerado. Como só temos um nó, e todos os termos já estão relacionados entre si, então o *relational pathfinding* não se aplica a este caso.

4) Cláusula original: *great_ne(A,B).* Neste caso não temos corpo na cláusula original, dessa forma, se a cláusula for instanciada por *great_ne(a1,b1)*, os nós criados serão [*a1*] e [*b1*].

Após a geração dos subgrafos, os termos contidos neste subgrafos serão os *end values* iniciais que serão usados na expansão dos nós. O próximo passo do algoritmo é a expansão de cada subgrafo encontrado. A expansão é feita procurando por todas as relações no conhecimento preliminar que possuem algum *end-value* do nó sendo

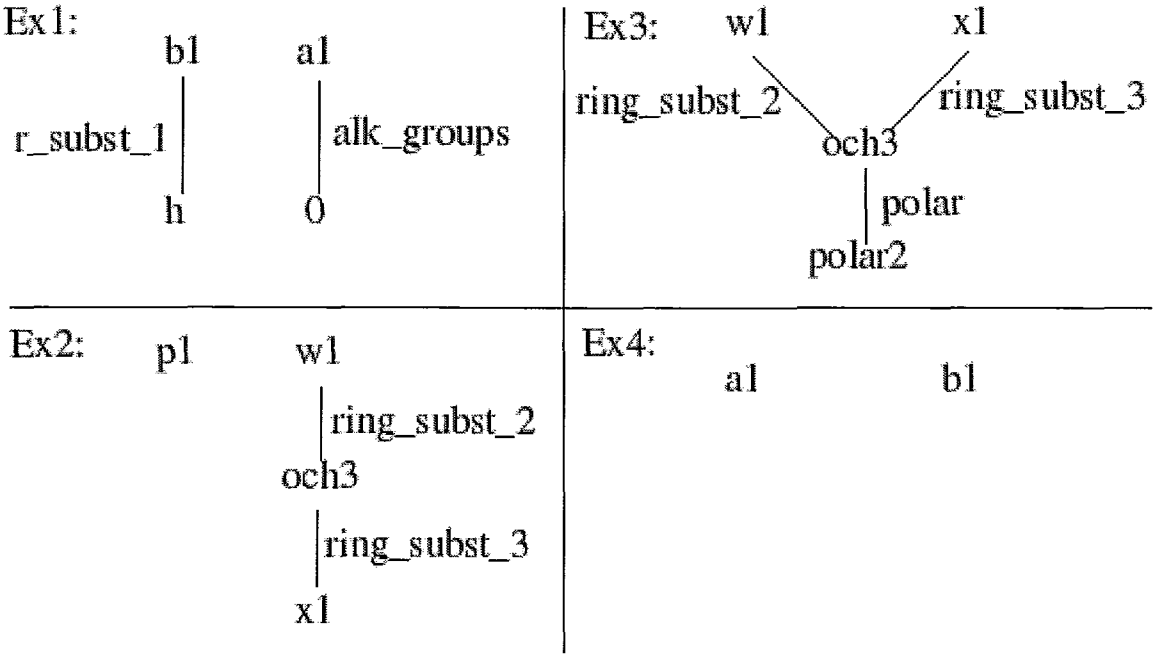

Figura 2.8: Subgrafos isolados

expandido. Novos *end values* encontrados por estas relações formarão um novo nó. As relações encontradas serão os possíveis antecedentes a serem adicionados na cláusula original, e a busca por estes antecedentes será explicada mais adiante no algoritmo 2.7. Se por acaso alguma relação retornou um *end-value* previamente visto no nó sendo expandido, esta relação é descartada, pois a intenção é formar um caminho até um outro subgrafo, e não dentro do próprio subgrafo. Os subgrafos serão expandidos intercaladamente, e a cada expansão um novo conjunto de *end values* será encontrado. Caminhos relacionais serão formados quando uma interseção entre estes conjuntos for encontrada.

Suponha a cláusula dada no primeiro exemplo da geração de nós:

great_ne(A,B) :- alk_groups(B,C), r_subst_1(A,D).

E que esta cláusula foi instanciada por *great_ne(b1,a1)*, obtendo:

great_ne(b1,a1) :- alk_groups(a1,0), r_subst_1(b1,h).

Como vimos, os nós iniciais do grafo relacional são $[a1, 0]$ e $[b1, h]$. A expansão destes nós é feita intercaladamente. Começando-se pelo nó $[b1, h]$, e de acordo com os fatos definidos no domínio dado, as relações encontradas para os *end values* $b1$ e h serão:

r_subst_1(d1,h). *r_subst_1(e1,h).* *x_subst(b1,7,cl).*

Estas relações introduziram novos *end values*, que juntos formam um novo nó no grafo dado por $[d1, e1, 7, cl]$. É verificado se este conjunto tem algum *end-value* em comum com o conjunto de *end values* do nó ainda não expandido, ou seja, com $[a1, 0]$. Como nenhuma interseção é achada, passamos para a expansão do nó $[a1, 0]$. As relações encontradas para os *end values* $a1$ e 0 serão:

r_subst_2(a1,cl). *gt(1,0).* *alk_groups(e1,0).*

Os novos *end values* encontrados para esta expansão foram $[1, e1, cl]$. Neste momento encontramos uma interseção em $e1$ e cl com o o próximo nó que ia ser expandido, ou seja, $[d1, e1, 7, cl]$. Como uma interseção foi achada, o algoritmo pára a expansão dos nós e retorna os caminhos relacionais formados, que são:

r_subst_1(e1,h),alk_groups(e1,0).
x_subst(b1,7,cl),r_subst_2(a1,cl).

O grafo final encontrado, mostrando os caminhos relacionais formados, e considerando cada nó como um conjunto de *end values*, é mostrado na figura 2.9.

Após a geração dos caminhos possíveis, estes deverão ser validados. A validação verifica se os caminhos retornados não introduzem nenhum termo que só aparece uma vez, tais termos são chamados de *singletons*. Se isso acontecer, o algoritmo tentará completar o caminho adicionando antecedentes que liguem estes termos com outros termos já existentes no caminho. Estas novas relações não podem eliminar nenhuma das instâncias positivas já provadas. Se não pudermos ligar todas estes termos, então o caminho é rejeitado. Os termos considerados na validação são

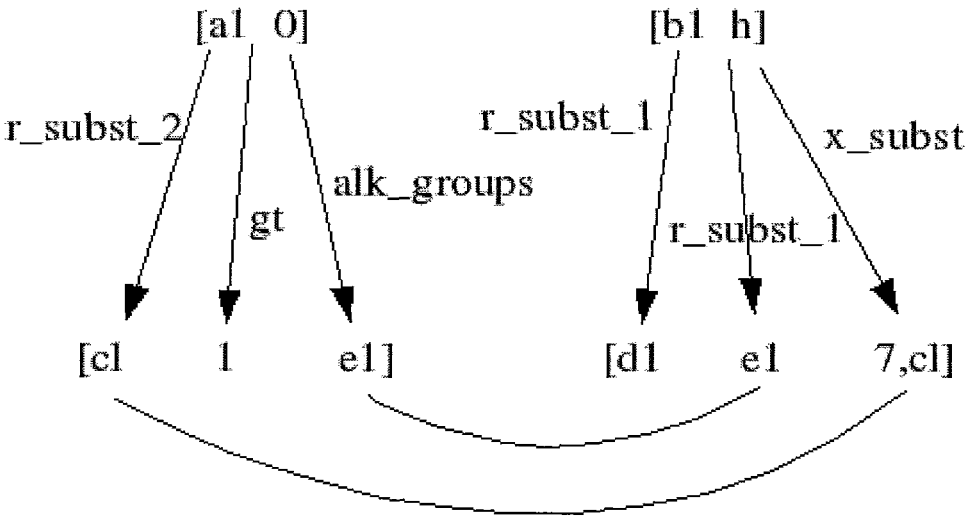

Figura 2.9: Caminhos relacionais formados

apenas aqueles que foram introduzidos durante a expansão dos nós.

Singletons são verificados porque se existem um *singleton* é porque em algum lugar do grafo ele esta desconexo, pois este termo *singleton* não se liga com nenhum outro termo no grafo.

Para o exemplo acima, o único termo que só aparece uma vez é o 7 introduzido pela relação *x_ subst(b1,7,c1)*. Porém, como este predicado é ternário, o 7 não impede que o predicado faça parte de um caminho relacional, pois o *b*1 e o *c1* já se relacionam com outros predicados no caminho. Portanto o termo 7 não é considerado um *singleton.*

Cada caminho considerado válido será adicionado à claúsula original instanciada, formando um conjunto de novas cláusulas. Os termos em cada cláusula serão substituídos por variáveis, e a melhor cláusula será selecionada. A escolha da melhor cláusula leva em consideração três critérios: O primeiro critério é o número de instâncias positivas provadas, a que tiver mais instâncias positivas ganha; O segundo critério é o número de nós criados quando se considera uma instância positiva provada e os caminhos em questão, quem tiver menos nós ganha; e o terceiro critério é o número de antecedentes no caminho relacional formado, quem tiver mais antecedentes ganha.

As cláusulas instanciadas formadas para o exemplo acima são:

great_ne(b1,a1) :- alk_groups(a1,0),r_subst_1(b1,h),r_subst_1(e1,h),alk_groups(e1,0).
great_ne(b1,a1) :- alk_groups(a1,0),r_subst_1(b1,h),x_subst(b1,7,cl),r_subst_2(a1,cl).

que após a substituição de termos por variáveis ficarão:

great_ne(A,B) :- alk_groups(B,C),r_subst_1(A,D),r_subst_1(E,D),alk_groups(E,C).
great_ne(A,B) :- alk_groups(B,C),r_subst_1(A,D),x_subst(A,E,F),r_subst_2(B,F).

Se ao final do algoritmo *relational pathfinding* a cláusula gerada continuar provando negativos, será feita a adição de antecedentes nesta cláusula utilizando-se o *hill climbing.*

O *relational pathfinding*, assim como o *hill climbing*, utiliza como espaço de busca de antecedentes todos os possíveis literais da base de conhecimento, ou seja, todos os predicados intermediários e os predicados definidos no *object_relations.* O algoritmo de busca de antecedentes *relational pathfinding*, que corresponde a buscar todas as relações na expansão de um nó, está descrito em Algoritmo 2.7 a seguir.

O algoritmo 2.7 é muito semelhante ao algoritmo 2.5 de busca de antecedentes do *hill climbing.* A diferença básica entre os dois é que, no *hill climbing*, as primeiras combinações feitas são considerando variáveis da cláusula sendo revisada, enquanto que no *relational pathfinding*, são considerados os *end values* do nó sendo expandido. Quando todas as combinações envolvendo apenas *end values* forem feitas, aí novas variáveis serão criadas, e combinações que envolvem *end values* e variáveis serão geradas.

Considerando o mesmo exemplo utilizado para explicar a adição de antecedentes, ou seja, cláusula original instanciada dada por *great_ne(b1,a1) :- alk_groups(a1,0), r_subst_1(b1,h)*, a busca por antecedentes ao expandir o nó inicial $[a1, 0]$ é mostrada a seguir. Não irei mostrar a busca completa, apenas o suficiente para demonstrar como ela é feita.

O primeiro literal da base de conhecimento considerado é o *x_subst(a,n,b)*, que é o primeiro literal do *object_relations.* O algoritmo irá considerar os *end values* do nó sendo expandido, com seus respectivos tipos, ou seja, [[a1,a],[0,n]], de acordo com o que está definido no *object_relations.* Para estes *end values* as combinações

Algoritmo 2.7 Algoritmo de busca de antecedentes do *relational pathfinding*

Para cada literal da base de conhecimento
  Retorne os *end values* e seus tipos no nó sendo expandido
  Retorne os argumentos e seus tipos no literal em questão
  Para cada combinação de *end values* do nó
    Verifique se a combinação é válida como possíveis argumentos do literal
    Se é válida **então**
      Substitua os argumentos do literal pela combinação dos *end values*
      Retorne o literal como antecedente
    Senão
      Passe para a próxima combinação de *end values*
  Fim para
  N = número de argumentos do literal em questão
  $i = 1$
  Enquanto $i \leq N$ - 1
    Crie uma variável nova L
    V = *end values* do nó + L
    Para cada combinação em V que inclua L e algum *end value* do nó
      Verifique se a combinação é válida como possíveis argumentos do literal
      Se é válida **então**
        Substitua os argumentos do literal pela combinação válida
        Busque no conjunto de fatos, um fato que consiga unificar com o literal
        Se encontrou fato **então**
          Retorne o literal como antecedente
        Senão
          Passe para a próxima combinação
      Senão
        Passe para a próxima combinação
    Fim para
    $i = i + 1$
  Fim Enquanto
  Passe para o próximo literal da base de conhecimento
Fim para

retornadas são:

(a1,a1,a1),(a1,a1,0),(a1,0,a1),(a1,0,0),(0,a1,a1),(0,a1,0),(0,0,a1),(0,0,0)

Neste conjunto nenhuma combinação é válida, então uma nova variável A é criada, e as novas combinações serão:

(a1,a1,A),(0,0,A),(a1,0,A),(0,a1,A),(a1,A,a1),(0,A,0),(a1,A,0),(0,A,a1),(A,a1,a1),

(A,0,0),(A,a1,0),(A,0,a1),(a1,A,A),(0,A,A),(A,A,a1),(A,A,0),(A,a1,A),(A,0,A).

Neste conjunto apenas a combinação (a1,0,A) é válida, e portanto o literal retornado será *x_subst(a1,0,A)*. O algoritmo irá buscar no conjunto de fatos do conhecimento preliminar algum fato que unifique com este literal, mas não encontrará, e portanto este literal é descartado como antecedente. O algoritmo continua criando uma nova variável B e fazendo novas combinações. Sabemos de antemão que para o literal *x_subst(a,n,b)* não será retornado nenhum antecedente, pois como foi mostrado anteriormente neste exemplo, a expansão do nó $[a1, 0]$ só retorna literais com os conceitos *r_subst_2*, *gt* e *alk_groups*. Portanto suponha que estamos em um ponto avançado do algoritmo e que agora estamos considerando o literal *gt(n,n)*. As primeiras combinações retornadas são:

(a1,a1),(a1,0),(0,a1),(0,0)

A combinação (0,0) é válida, porém não existe um fato *gt(0,0)* no conhecimento preliminar, e portanto esta combinação é desconsiderada e uma nova variável A é criada. As próximas combinações serão:

(a1,A),(0,A),(A,a1),(A,0)

As combinações válidas são (0,A) e (A,0), que dão origem aos literais *gt(0,A)* e *gt(A,0)*. Porém o único fato encontrado no conhecimento preliminar é *gt(1,0)*, que unifica com *gt(A,0)* e é retornado como antecedente.

Além de definir que tipo de antecedente será adicionado, o usuário também pode especificar como se dará a interação entre os algoritmos *hill climbing* e *relational pathfinding* para a operação de adição de antecedentes às cláusulas. Este parâmetro é chamado de *relation tuning* e pode ser instanciados de três formas descritas a seguir:

- ***non_relational*:** Apenas o algoritmo *hill climbing* será utilizado na adição de antecedentes.

- ***relational*:** É o padrão utilizado pelo FORTE. Neste caso, ambos os algoritmos, *hill climbing* e *relational pathfinding*, são utilizados. O algoritmo que obtiver a melhor performance é escolhida,ou seja, aquela que conseguir cobrir mais instâncias positivas.

- ***highly_relational*:** O FORTE irá executar primeiro o *relational pathfinding*, que irá retornar uma nova cláusula. Esta nova cláusula será adicionada à teoria e o FORTE tentará gerar uma nova cláusula para a cláusula original, só que a teoria considerada é a antiga mais a nova cláusula, e o conjunto de instâncias positivas será o antigo menos as instâncias positivas provadas pela nova cláusula. Este processo continua até que não consiga mais modificar a cláusula. Sendo que se o *relational pathfinding* retornar uma cláusula que é igual à cláusula original, o *hill climbing* vai ser executado para tentar gerar uma nova cláusula para a cláusula original. Resumindo, para o caso *highly_relational*, o algoritmo tenta primeiro modificar a cláusula usando o *relational pathfinding*, e se não conseguir, usa o *hill climbing* para gerar uma nova cláusula.

  A cláusula gerada pelo *relational pathfinding* não pode deixar de provar todas as instâncias positivas já provadas. Se a nova cláusula conseguir deixar de provar todas as instâncias negativas provadas, o *relational pathfinding* termina e retorna esta cláusula, pois alcançou o seu objetivo. Caso contrário, dentro do *relational pathfinding*, será chamado o *hill climbing*, para tentar adicionar mais antecedentes a esta nova cláusula que foi gerada pelo *relational pathfinding*. Quando o *hill climbing* é chamado, ele considera a nova cláusula, a teoria original, o conjunto de instâncias positivas que continuam sendo provadas pela nova cláusula, e o conjunto de instâncias negativas que continuam sendo provadas após a geração da nova cláusula. Se a cláusula alterada pelo *hill climbing* conseguir deixar de provar todos as instâncias negativas, ele termina, caso contrário vai chamar novamente o *relational pathfinding*, considerando a nova cláusula, a teoria original, o conjunto de instâncias positivas que continuam sendo provadas pela nova cláusula, e o conjunto de instâncias negativas que

continuam sendo provadas após a geração da nova cláusula. Essa intercalação entre os dois algoritmos continua até que todas as instâncias negativas tenham deixado de ser provadas ou até que não se consiga mais modificar a cláusula.

Observação 1: Percebe-se que quando o FORTE vai gerar uma nova cláusula, o conjunto de instâncias positivas usado é o anterior menos o que já foi provado pela cláusula que ele acabou de gerar, ou seja, se tinhamos 100 instâncias positivas provadas para a cláusula original e a adição de antecedentes gerou uma cláusula que prova 30 destas instâncias, na criação de uma outra cláusula para a cláusula original consideraremos apenas as 70 restantes, porque o objetivo é encontrar cláusulas novas que provem o máximo de instâncias positivas possíveis, e se uma instância positiva foi provada pela cláusula que a operação de adição de antecedentes acabou de gerar, não preciso obrigar que a próxima cláusula também prove esta instância, por isso considero só as instâncias restantes, ou seja, as 70 que sobraram. Isto é completamente diferente do que é feito dentro da geração de uma única cláusula, enquanto ainda estamos adicionando antecedentes à mesma cláusula. Neste momento, o conjunto de instâncias positivas que estamos passando como parâmetro a cada chamada de adição de antecedentes é o conjunto de instâncias positivas que foram provadas até o momento pela cláusula sendo gerada, e não a diferença do total e o que foi provado. Isso acontece porque queremos adicionar antecedentes que continuem provando as instâncias que já foram provadas pela cláusula em questão.

Observação 2: Percebe-se que dentro do algoritmo de adição de antecedentes utilizando-se a opção *highly_relational*, os algoritmos *relational pathfinding* e o *hill climbing* são intercalados, porém o *hill climbing* só é chamado se o *relational pathfinding* não conseguiu gerar uma cláusula diferente da cláusula original, ou seja, eles são intercalados para gerarem cláusulas diferentes, e não a mesma cláusula. Agora, dentro do *relational pathfinding*, estes dois algoritmos também são intercalados, só que a diferença é que agora a intercalação é feita para acrescentar antecedentes dentro da mesma cláusula.

A revisão escolhida, para ser implementada, tem que proporcionar melhora em

desempenho para a teoria. Já que a cada iteração do algoritmo é exigido um aumento em desempenho onde este é limitado a cem porcento, o algoritmo garantidamente terminará.

2.5 Trabalhos Relacionados

Nesta seção iremos falar de outros trabalhos relacionados à revisão e aprendizado de primeira-ordem(RICHARDS, MOONEY, 1995), (WROBEL, 1996), mas procurando dar ênfase às vantagens do FORTE em relação a eles.

Muitos trabalhos em ILP se preocupam em generalizar uma teoria de primeira-ordem existente através da adição de cláusulas, mas não tratam o problema de generalizar cláusulas existentes ou remover ou especializar cláusulas incorretas. O sistema MIS(SHAPIRO, 1983) é capaz de especializar teorias, porém ele requer um oráculo para responder queries para qualquer predicado da teoria. Seu sistema Prolog, PDS6, requer ainda mais interação com o usuário. Ele requer que o usuário julgue a corretude das chamadas dos predicados, determine qual cláusula de um predicado deve ser mudada, e escreva as cláusulas que estão faltando. Outro sistema interativo, CLINT(RAEDT, BRUYNOOGHE, 1992), generaliza e especializa teorias e também cria novos predicados por analogia. As revisões do sistema CLINT apenas incluem adicionar e remover cláusulas, ou seja, ele não se preocupa em modificar cláusulas existentes. Ao contrário do MIS e do CLINT, o FORTE automaticamente revisa teorias sem qualquer interação com o usuário.

O sistema CIGOL(MUGGLETON, BUNTINE, 1988) usa a técnica da resolução inversa para aprender teorias de primeira-ordem a partir de exemplos, porém ele requer que o usuário verifique interativamente certos passos. GOLEM(MUGGLETON, FENG, 1992) é um sistema mais eficiente de indução baseada no framework RLGG (*relative least-general generalization*), que Muggleton(MUGGLETON, 1992a) mostrou ser relacionado a resolução inversa. GOLEM aprende teorias de primeira-ordem "*bottom-up*", generalizando as instâncias positivas de treinamento enquanto exclui as instâncias negativas. Já que GOLEM trabalha com generalização, ele pode sobre-generalizar uma teoria. Bain e Muggleton(MUGGLETON, 1992a; MUGGLETON, 1992b) apresentaram um método de especialização para o GOLEM usando "*closed-*

world specialization"(isto é, negação por falha). Usando esta técnica no domínio KRK, GOLEM aprendeu uma teoria correta depois de 10000 exemplos. Já o FORTE integra operadores de especialização desde o início, aprendendo uma teoria correta depois de 5000 exemplos(RICHARDS, MOONEY, 1995).

O sistema AUDREY(WOGULIS, 1991) primeiro especializa teorias deletando cláusulas e então a generaliza usando um método abdutivo. Consequentemente, seu intervalo de revisões é limitado comparado ao FORTE.

O Sistema RUTH(ADé, et al., 1994) utiliza o critério de minimalidade sintática como critério adicional ao problema de revisão de teorias. Uma revisão mínima, segundo este critério, seria aquela que minimizasse o número de aplicações de operadores. Tal medida depende fortemente dos operadores de revisão disponíveis. Se algum dos operadores executar mudanças em larga escala, a teoria resultante pode não ser parecida com a teoria original no nível sintático.

O Sistema KRT(WROBEL, 1994), assim como o FORTE, usa o operador de adição de antecedentes para especializar uma cláusula, considerando tanto *hill climbing* como o *relational pathfinding* como espaço de busca de antecedentes.

Por fim vale ressaltar o trabalho realizado em (ONG, et al., 2005) que trata da inclusão do *relational pathfinding* no Aleph. Este trabalho tem como objetivo estender o algoritmo *pahtfinding* para fazer uso de declaração de modos, aplicando esta extensão à *bottom clauses*, e passando a buscar caminhos relacionais na cláusula saturada como forma de restringir o espaço de busca. Esta abordagem difere do *relational pathfinding* aplicado no FORTE primeiramente porque o Aleph é um sistema de aprendizando de teorias, enquanto que o FORTE é um sistema de revisão de teorias já existentes. Em segundo lugar, o *relational pathfinding* apresentado no FORTE busca por uma interseção entre os *end values* gerados em cada expansão de um nó, sendo que assim que uma interseção é achada, o algoritmo retorna os caminhos gerados por esta interseção. Além disso, os *end values* são termos que instanciam a cláusula sendo especializada a partir de uma instância escolhida randomicamente. Já no *relational pathfinding* direcionado a modos do Aleph, a busca dos caminhos se dá na cláusula saturada variabilizada e não instanciada, levando em consideração as definições de variáveis de entrada e saída dos literais da *bottom*

clause, e o algoritmo não termina quando uma interseção é achada, ele continua buscando por outras interseções até que todos os literais da *bottom clause* tenham sido considerados.

Capítulo 3

Utilizando a cláusula mais específica e declaração de modos na revisão de teorias de primeira-ordem a partir de exemplos

Neste capítulo apresentaremos a abordagem por nós proposta para revisar teorias de primeira-ordem utilizando modos e a cláusula mais específica. O custo da revisão de teorias de primeira-ordem no sistema FORTE é dominado pela busca por antecedentes a serem adicionados em uma cláusula, visto que o FORTE usa como espaço de busca todos os possíveis literais da base de conhecimento para serem adicionados na cláusula. Apesar da forma como os literais são gerados ser ineficiente, ou seja, fazendo combinação de variáveis, o que realmente torna o processo de revisão custoso é a avaliação de cada um destes literais. O algoritmo de geração de novos antecedentes impõe apenas restrições de tipo nos argumentos dos predicados. Não é utilizada nenhuma definição de modos, ou seja, as variáveis não são restritas a serem de entrada e/ou de saída. O algoritmo também não impõe nenhuma restrição de *recall*, e também não limita a profundidade de variável da cláusula. Ao criar literais é realizada uma combinação de todas as possíveis variáveis que podem aparecer nesses literais, fazendo com que o número de antecedentes gerados em uma iteração seja muito grande.

Freqüentemente a combinação de variáveis gera muito mais literais do que necessário. Lembre que todos os literais gerados deverão ser avaliados para escolher aquele que será adicionado à cláusula. Entretanto, alguns literais podem não

cobrir nem mesmo uma instância positiva, visto que eles são gerados sem levar em consideração os exemplos (abordagem *top-down*). Assim, um primeiro passo a ser considerado para reduzir o espaço de busca e ainda gerar predicados mais relevantes é utilizar a cláusula mais específica (*bottom clause*) variabilizada como espaço de busca de literais a serem adicionados no corpo da cláusula. A geração da *bottom clause* já leva em consideração o *recall*, e também já limita a profundidade de variável da cláusula. Propomos também a declaração de modos para validar os antecedentes retornados a partir da *bottom clause*, de forma que as variáveis contidas nestes antecedentes obedeçam às restrições de serem de entrada ou saída.

Ao utilizar declaração de modos e a cláusula mais específica na revisão de teorias de primeira-ordem, pretendemos obter uma diminuição do espaço de busca de literais na adição de antecedentes, e consequentemente reduzir o tempo de execução do processo de geração de literais, além de conseguir teorias menores e mais compreensíveis, sem que haja prejuízo da acurácia já obtida pelo processo de revisão no FORTE original.

Este capítulo está organizado da seguinte forma: Na seção 3.1 será discutido como é feita a geração da *bottom clause* no FORTE e o melhor momento para gerá-la dentro do algoritmo de adição de antecedentes. Na seção 3.2 mostraremos como é feita a unificação de variáveis da *bottom clause* com a cláusula sendo revisada, já que em revisão de teorias partimos de uma cláusula inicial que pode conter variáveis diferentes das da *bottom clause*. Na seção 3.3 mostraremos como que o uso da *bottom clause* pode restringir o espaço de busca de antecedentes. Esta seção será dividida em duas subseções que descreverão, respectivamente, o algoritmo de busca de antecedentes *hill climbing* modificado que faz uso da *bottom clause* como espaço de busca e o algoritmo de busca de antecedentes *relational pathfinding* modificado que faz uso da *bottom clause* como espaço de busca de antecedentes. Por fim, na seção 3.4 mostraremos como a declaração de modos pode tornar a operação de adição de antecedente mais correta, ao restringir as variáveis a serem de entrada ou de saída, levando em consideração os algoritmos modificados do *hill climbing* e do *relational pathfinding*.

3.1 Geração da *bottom clause* no FORTE

A *bottom clause* a ser usada no FORTE é gerada através do método de saturação do Aleph.

Para definir em que momento a *bottom clause* deve ser gerada e como ela será utilizada no FORTE, é preciso fazer um paralelo com o Aleph. A idéia é criar uma *bottom clause* a partir de uma instância e usá-la como espaço de busca para os operadores que adicionam antecedentes. Lembrando que no Aleph cada exemplo da base de dados é uma instância, e que no FORTE um exemplo pode ser uma instância ou um conjunto de instâncias.

Podemos resumir a utilização da *bottom clause* no Aleph da seguinte maneira: A partir de uma instância positiva não provada é criada uma *bottom clause*. No algoritmo de cobertura, quando vai gerar uma nova cláusula para um predicado, considera-se apenas os antecedentes que estão na *bottom clause*. Quando não puder mais cobrir instâncias positivas sem que cubra também instâncias negativas, aí desconsidera as instâncias positivas que já foram provadas, gera uma nova *bottom clause* a partir de alguma instância positiva que ainda não foi provada, e recomeça o processo de geração de uma nova cláusula, até que todas as instâncias positivas tenham sido provadas, ou até que alcance a acurácia desejada, ou até chegar a um determinado ponto pré-definido.

Assim como no Aleph, o FORTE também utiliza o algoritmo de cobertura em operadores de adição de antecedentes, para gerar novas revisões para um determinado ponto de revisão. Uma das maneiras de fazer a revisão é utilizar a operação de adição de antecedentes, que adicionam antecedentes em uma cláusula incorreta para excluir instâncias negativas que estão sendo provadas, não permitindo ao máximo possível que instâncias positivas deixem de ser provadas. Se a adição desses antecedentes também fizer com que instâncias positivas deixem de ser provadas, adiciona-se essa cláusula modificada à teoria e recomeça-se o algoritmo com a cláusula original, procurando cláusulas alternativas que retenham a prova das outras instâncias positivas enquanto ainda eliminam as negativas. O melhor momento para a geração da *bottom clause* é justamente quando a operação de adição de antecedentes

vai ser utilizada para modificar uma cláusula. Antes de começar o algoritmo de adição de antecedentes, deve-se gerar uma *bottom clause* para uma instância positiva provada por esta cláusula, e os antecedentes que serão adicionados serão os literais contidos na *bottom clause*. Quando a cláusula não puder mais ser modificada e uma nova adição de antecedentes tiver que ser feita na cláusula original, uma nova *bottom clause* será gerada para uma nova instância positiva provada.

Para criar a *bottom clause* chamamos o algoritmo de saturação do Aleph passando como parâmetro uma instância positiva, escolhida aleatóriamente dentre as instâncias positivas provadas pela cláusula sendo revisada. Lembramos aqui que só podemos criar *bottom clauses* para predicados de alto nível, pois eles são os únicos que poderão estar na cabeça de uma regra.

Percebe-se que estamos utilizando instâncias positivas provadas para gerar a *bottom clause*, pois na adição de antecedentes o objetivo é deixar de provar instâncias negativas provadas, desde que continue provando as instâncias positivas anteriomente provadas pela cláusula original. Consideramos que as instâncias positivas provadas por uma determinada cláusula seguem todas um mesmo padrão, e que usando uma destas instâncias para gerar uma *bottom clause* com todos os antecedentes relevantes para esta instância, tais antecedentes também serão relevantes para as outras instâncias positivas, e consequentemente não serão relevantes para as instâncias negativas, alcançando o objetivo que é continuar provando as instâncias positivas provadas e deixar de provar as negativas provadas. Temos que levar em consideração também que quando geramos uma *bottom clause* para uma instância, os literais do corpo da cláusula original têm que estar no corpo da *bottom clause*, porque os literais da *bottom clause* são relevantes para a instância e formam um caminho relacional, e quando for acrescentar antecedentes, o caminho tem que continuar fazendo sentido na cláusula original. Por isso que temos que gerar a *bottom clause* para uma instância positiva provada pela cláusula inicial.

3.2 Unificação de variáveis

Sabemos que na geração da *bottom clause* são adicionados todos os literais

encontrados no conhecimento preliminar cujos valores combinam com os valores da instância positiva sendo saturada, e que todos os valores encontrados para os argumentos dos predicados são substituídos por variáveis. Os argumentos substituídos por valores idênticos, são substituídos por variáveis do mesmo nome. Portanto, no final, o Aleph retorna uma *bottom clause* variabilizada onde as variáveis estão relacionadas entre si, formando um caminho relacional. Porém, vamos utilizar a *bottom clause* para adicionar antecedentes a uma cláusula inicial que já possui literais no corpo. Não estamos criando uma cláusula do zero como o Aleph faz. Aqui nos deparamos com um problema, pois as variáveis retornadas na *bottom clause* não necessariamente são as mesmas que as variáveis existentes originalmente na cláusula sendo revisada. Portanto, precisamos unificar as variáveis da *bottom clause* com as variáveis da cláusula original.

Por exemplo, suponha que a minha cláusula original seja

great_ne(A,B) :- r_subst_1(B,C)

e suponha que eu gerei a *bottom clause*

great_ne(B,C) :- r_subst_1(C,D),ring(D,E),r_subst_2(E,B)

Vê-se que o literal *r_subst_1(B,C)* contido no corpo da cláusula original está presente no corpo da *bottom clause*, porém com variáveis diferentes, e que as cabeças das duas cláusulas correspondem ao mesmo predicado mas não com as mesmas variáveis. Se eu não unificar as variáveis igualando *great_ne(A,B)* e *great_ne(B,C)* e dizendo que o A e o B da cláusula original correspondem, respectivamente, ao B e o C da *bottom clause*, quando eu for adicionar o antecedente *r_subst_2(E,B)* à cláusula original, a variável B que liga com o B de *great_(B,C)* vai passar a ligar com o B de *great_ne(A,B)*, o que está errado, pois o B de *great_ne(A,B)* na verdade é equivalente ao C de *great_ne(B,C)*. Portanto a unificação tem que ser feita, e as variáveis da *bottom clause* tem que ser as mesmas que as variáveis da cláusula original. Para as variáveis da *bottom clause* que não possuem uma correspondente

na cláusula original, como é o caso da variável E no exemplo, serão criadas novas variáveis seguindo a ordem das variáveis já existentes na cláusula original. No exemplo anterior, as variáveis já existentes na cláusula original são A,B e C, que se relacionam respectivamente às variáveis B, C e D da *bottom clause*. Como a variável E não tem correspondente, ela é desconsiderada e é gerada uma nova variável usando a próxima letra no alfabeto que ainda não foi usada. Portanto é gerada a variável D, que ficará no lugar da variável E na *bottom clause*. A *bottom clause* final retornada será:

great_ne(A,B) :- r_subst_1(B,C),ring(C,D),r_subst_2(D,A).

Nos algoritmos 3.1 e 3.2 descritos a seguir, são mostrados as modificações feitas no algoritmo de adição de antecedentes usando tanto o algoritmo *hill climbing*, descrito em Algoritmo 2.4, quanto o *relational pathfinding*, descrito em Algoritmo 2.6. Estas modificações incluem a geração da *bottom clause*, descrita na seção anterior, e a unificação das variáveis, descrita nesta seção, e estão destacadas em negrito.

No algoritmo 3.1, a geração da *bottom clause* e a unificação das variáveis são colocados como os primeiros passos da iteração externa.

No Algoritmo 3.2, a geração da *bottom clause* e a unificação das variáveis foi colocada imediatamente antes da cláusula ser instanciada, e a busca de antecedentes passou a utilizar a *bottom clause* como espaço de busca.

3.3 Utilização da *bottom clause* como espaço de busca de antecedentes

O FORTE utiliza como espaço de busca todos os possíveis literais da base de conhecimento. Após a geração da *bottom clause*, o próximo passo é utilizá-la como espaço de busca na adição de antecedentes, ao invés de considerar todos os possíveis literais.

Para adicionar antecedentes o FORTE usa dois algoritmos de busca: *hill climbing* e *relational pathfinding*. Na primeira subseção iremos descrever o novo processo de busca de antecedentes *hill climbing* considerando a *bottom clause* como espaço

Algoritmo 3.1 Algoritmo de adição de antecedentes *hill climbing* modificado

T = Teoria inicial
C = Cláusula incorreta
CP = Conjunto de instâncias positivas provadas pela cláusula
IP = Conjunto de instâncias negativas provadas pela cláusula
$C\_original \leftarrow C$
$CP\_original \leftarrow CP$
$IP\_original \leftarrow IP$
repita
  Escolha aleatoriamente uma instância I **de** CP
  Gere a *bottom clause* BC **utilizando** I
  Unifique as variáveis da *bottom clause* com as variáveis da cláusula C
  repita
    Calcule o score de C a partir de CP e IP
    Retorne os possíveis antecedentes utilizando BC
    Calcule os scores dos antecedentes retornados
    Escolha o melhor antecedente
    Adicione o antecedente à cláusula C, formando a cláusula C'
    $NovoCP$ = Instâncias positivas provadas por C'
    $NovoIP$ = Instâncias negativas provadas por C'
    $C \leftarrow C'$
    $CP \leftarrow NovoCP$
    $IP \leftarrow NovoIP$
  até que não se tenha mais antecedentes ou o score da cláusula não seja mais melhorado.
  Se a nova cláusula C' é diferente da cláusula original $C\_original$ **então**
    Acrescente a nova cláusula C' à teoria T, formando a nova teoria T'
    Calcule o novo conjunto de instâncias positivas provadas($NovoCP\_Teoria$) e instâncias negativas provadas($NovoIP\_Teoria$) pela nova teoria
    $C \leftarrow C\_original$
    $CP \leftarrow CP\_original - NovoCP\_Teoria$
    $IP \leftarrow IP\_original - NovoIP\_Teoria$
    $T \leftarrow T'$
  Fim Se
Até que não consiga mais especializar a cláusula, ou até que todas as instâncias positivas provadas pela cláusula original tenham sido provadas pelas cláusulas criadas

de busca, e na seção seguinte iremos mostrar o algoritmo *relational pathfinding* modificado que utiliza a *bottom clause* como espaço de busca.

Os exemplos que serão dados nas subseções vão se basear em uma parte da base de dados Amine(KING, et al., 1995) mostrada abaixo. Esta base também será usada na seção 3.4, portanto serão incluídas as definições de modos:

Algoritmo 3.2 Algoritmo de adição de antecedentes *relational pathfinding* modificado

C = **Cláusula incorreta**
Escolha aleatoriamente uma instância positiva provada pela cláusula C
Gere a *bottom clause* BC **utilizando esta instância**
Unifique as variáveis da *bottom clause* com as variáveis da cláusula C
Instancie a cláusula original com uma instância positiva provada escolhida aleatoriamente
Ache os subgrafos isolados
Para cada subgrafo
    Transforme os termos em *end values* iniciais
Fim para
repita
  Para cada subgrafo
    Encontre todas as relações partindo deste subgrafo, utilizando a *bottom clause* como espaço de busca
      Remova arestas que alcançam *end values* previamente vistos
  Fim cada
Até que uma interseção seja achada ou o limite de memória seja excedido
Se uma ou mais interseções foram achadas
  Para cada interseção
    Se o caminho formado contém variáveis *singletons*
      Adicione relações que usem as variáveis *singletons*
      Se não conseguiu usar todos os *singletons*
        Descarte o caminho
      Fim se
    Fim se
    Adicione os caminhos relacionais achado à cláusula original formando um conjunto de novas cláusulas
    Substitua os termos nas cláusulas por variáveis
  Fim para
  Selecione a cláusula com melhor acurácia
Fim se
Se cláusula selecionada continua provando instâncias negativas
  Adicione antecedentes usando o algoritmo *hill climbing* modificado
Fim se

modeh(1,great_ne(+a,+a)).

modeb(,x_subst(+a,+n,-b)).*

modeb(,alk_groups(+a,-n)).*

modeb(,r_subst_1(+a,-l)).*

modeb(,r_subst_2(+a,-m)).*

modeb(,r_subst_3(+a,-n)).*

modeb(,ring_substitutions(+a,-n)).*

modeb(,ring_subst_1(+a,-b)).*

modeb(,ring_subst_2(+a,-b)).*

modeb(,ring_subst_3(+a,-b)).*

modeb(,ring_subst_4(+a,-b)).*

modeb(,ring_subst_5(+a,-b)).*

modeb(,ring_subst_6(+a,-b)).*

modeb(,polar(+b,-c)).*

modeb(,size(+b,-d)).*

modeb(,flex(+b,-e)).*

modeb(,h_doner(+b,-f)).*

modeb(,h_acceptor(+b,-g)).*

modeb(,pi_doner(+b,-h)).*

modeb(,pi_acceptor(+b,-i)).*

modeb(,polarisable(+b,-j)).*

modeb(,sigma(+b,-k)).*

modeb(,n_val(+a,-n)).*

modeb(,gt(+n,-n)).*

modeb(,great_polar(+c,-c)).*

modeb(,great_size(+d,-d)).*

modeb(,great_flex(+e,-e)).*

modeb(,great_h_don(+f,-f)).*

modeb(,great_h_acc(+g,-g)).*

modeb(,great_pi_don(+h,-h)).*

modeb(,great_pi_acc(+i,-i)).*

modeb(,great_polari(+j,-j)).*

modeb(,great_sigma(+k,-k)).*

predicados_alto_nivel([great_ne(a,a)]).

object_relations([x_subst(a,n,b), alk_groups(a,n), r_subst_1(a,l),

r_subst_2(a,m), r_subst_3(a,n), ring_substitutions(a,n), ring_subst_1(a,b), ring_subst_2(a,b), ring_subst_3(a,b), ring_subst_4(a,b), ring_subst_5(a,b), ring_subst_6(a,b), polar(b,c), size(b,d), flex(b,e), h_doner(b,f), h_acceptor(b,g), pi_doner(b,h), pi_acceptor(b,i), polarisable(b,j), sigma(b,k), n_val(a,n), gt(n,n), great_polar(c,c), great_size(d,d), great_flex(e,e), great_h_don(f,f), great_h_acc(g,g), great_pi_don(h,h), great_pi_acc(i,i), great_polari(j,j), great_sigma(k,k)]).

/* instâncias positivas */

great_ne(dd1,k1).

great_ne(m1,a1).

great_ne(b1,a1).

/* instâncias negativas */

great_ne(a1,b1).

great_ne(c1,u1).

/* fatos */

alk_groups(dd1,1).

alk_groups(k1,2).

alk_groups(m1,4).

alk_groups(a1,7)

r_subst_1(dd1,single_alk(1)).

r_subst_1(k1,single_alk(2)).

r_subst_1(m1,single_alk(3)).

r_subst_2(dd1,aro(1)).

r_subst_2(k1,aro(1)).

r_subst_2(m1,double_alk(1)).

ring_subst_3(dd1,cf3).

ring_substitutions(k1,0).

polar(cf3,polar3).

gt(1,0).

gt(2,0).

gt(2,1).

gt(4,3).

x_subst(b1,7,cl).

3.3.1 Algoritmo *hill climbing* de busca de antecedentes utilizando a *bottom clause*

A única restrição imposta pelo algoritmo de busca de antecedentes *hill climbing*, descrito em Algoritmo 2.5, é o tipo dos argumentos dos predicados, e ao realizar uma combinação de todas as possíveis variáveis, muitas combinações inválidas são geradas. Claramente, conforme aumenta-se o número de variáveis novas na cláusula o número de possíveis antecedentes a serem avaliados cresce exponencialmente visto que a complexidade de enumerar todas as possíveis combinações de variáveis é exponencial de acordo com a aridade dos predicados. Além disso, os antecedentes gerados a partir das combinações válidas não levam em consideração as instâncias dos exemplos, e portanto não irão melhorar a acurácia da cláusula original se não cobrirem pelo menos uma instância positiva.

Ao usar a *bottom clause* como espaço de busca, estaremos evitando as combinações de variáveis e estaremos garantindo que o antecedente adicionado à cláusula original irá cobrir pelo menos uma instância positiva, que é a instância utilizada para a geração da *bottom clause.*

O algoritmo de busca de antecedentes *hill climbing* modificado, considerando a *bottom clause* como espaço de busca é mostrado a seguir em Algoritmo 3.3.

Ao invés de buscar um literal do *object_relations*, fazer permutação de todas as variáveis da cláusula original e retornar vários antecedentes do mesmo literal com combinações válidas das variáveis, agora o antecedente retornado será um antecedente do corpo da *bottom clause* que possua alguma variável de ligação com a cláusula sendo revisada. Isto pode ser feito porque a *bottom clause* já está variabilizada e suas variáveis já foram unificadas com as variáveis da cláusula original.

Algoritmo 3.3 Algoritmo de busca de antecedentes *hill climbing* usando a *bottom clause*

C = Cláusula incorreta
BC = Bottom clause
Para cada literal em BC
    Retorne as variáveis de C
    Retorne as variáveis do literal em questão
    Se o literal possui variável em comum com C **então**
        Retorne o literal como antecedente
    Senão
        Passe para o próximo literal de BC
Fim para

Para exemplificar a busca de antecedentes considere a base de dados definida no início da seção. Suponha que queremos adicionar antecedentes à claúsula *great_ne(A,B)*, que não tem literais no corpo, e suponha que a instância positiva *great_ne(dd1,k1)* foi escolhida aleatoriamente para gerar a *bottom clause*. Considerando os fatos existentes na base de dados, a *bottom clause* gerada é:

Bottom Clause: great_ne(A,B) :- alk_groups(B,C), alk_groups(A,D), r_subst_1(B,E), r_subst_1(A,F), r_subst_2(B,G), r_subst_2(A,G), ring_substitutions(B,H), ring_subst_3(A,I), polar(I,J), gt(D,H),gt(C,H),gt(C,D)

As variáveis da cláusula original são A e B, portanto só posso retornar antecedentes que possuam uma destas variáveis. Os possíveis antecedentes retornados serão:

[alk_groups(B,C), alk_groups(A,D), r_subst_1(B,E), r_subst_1(A,F), r_subst_2(B,G), r_subst_2(A,G), ring_substitutions(B,H), ring_subst_3(A,I)].

Depois que o algoritmo retorna todos os possíveis antecedentes da *bottom clause* que tenhas as variáveis da cláusula original, um destes antecedentes vai ser escolhido como melhor e adicionado à cláusula para gerar uma nova cláusula. A adição de antecedentes vai ser feita novamente para tentar adicionar novos antecedentes a esta nova cláusula, o que corresponde à iteração interna do algoritmo de adição

de antecedentes 3.1. Neste momento a *bottom clause* ainda é a mesma, e além de buscarmos antecedentes com as variáveis que já tinhamos antes, vamos buscar antecedentes para as novas variáveis que foram introduzidas na cláusula.

Digamos que no exemplo acima o melhor antecedente retornado foi *ring_substitutions(B,H)*. A claúsula formada será:

great_ne(A,B) :- ring_substitutions(B,H)

Quando o algoritmo for buscar antecedentes para esta nova cláusula, além de buscar literais da *bottom clause* que possuam as variáveis A e B, também vai buscar literais com a variável H.

Quando tivermos a cláusula final gerada, esta cláusula é adicionada à teoria e uma nova cláusula será gerada a partir de cláusula original, pois ainda não deixou de provar todos as instâncias negativas. Neste momento uma nova *bottom clause* vai ser criada. Se fosse a mesma *bottom clause*, ia acabar gerado a mesma cláusula, o que nao acarreta ganho nenhum.

3.3.2 Algoritmo *relational pathfinding* de busca de antecedentes utilizando a *bottom clause*

Descrevemos a seguir o novo algoritmo do *relational pathfinding*, mostrado em Algoritmo 3.4, que agora considera a *bottom clause* como espaço de busca.

O *relational pathfinding*, ao contrário do *hill climbing*, recebe *end values* ao invés de variáveis como parâmetro. No *hill climbing*, quando buscamos um antecedente na *bottom clause*, apenas verificamos se o literal da *bottom clause* tem alguma variável em comum com a cláusula. Isto pode ser feito porque estamos comparando variáveis entre si. Já no *relational pathfinding* partimos de uma cláusula instanciada, e para buscar um literal na *bottom clause* teríamos que comparar *end values* com variáveis, o que não pode ser feito, portanto precisamos saber de que forma estes *end values* se relacionam com as variáveis da *bottom clause.*

Antes da expansão do primeiro nó, temos apenas os termos que instanciaram a cláusula sendo revisada, e estes termos correspondem aos *end values* dos nós

Algoritmo 3.4 Algoritmo de busca de antecedentes do *relational pathfinding* usando a *bottom clause*

C = Cláusula incorreta
BC = Bottom clause
Para cada literal em BC
    Retorne os *end values* com as respectivas variáveis que eles representam no nó sendo expandido
    Retorne as variáveis do literal em questão
    Se o literal possui variável em comum com as variáveis vinculadas aos *end values* **então**
        Instancie apenas as variáveis no literal que possui *end values* correspondentes
        Busque no conjunto de fatos, um fato que consiga unificar com o literal
        Se encontrou fato **então**
            Retorne o literal como antecedente
        Senão
            Passe para o próximo literal de BC
    Senão
        Passe para o próximo literal de BC
Fim para

iniciais. Ao instanciar a cláusula, sabemos que termo instanciou cada variável, e podemos então estabelecer uma relação única entre termo e variável. Através desta relação poderemos saber quais literais da *bottom clause* poderão ser retornados, basta verificar se alguma das variáveis do literal tem relação com algum dos *end values* do nó sendo expandido.

Considerando a base de dados descrita no início da seção, suponha que queremos adicionar antecedentes à cláusula:

great_ne(A,B) :- alk_groups(A,C).

e que esta cláusula foi instanciada por *great_ne(m1,a1)*, obtendo:

great_ne(m1,a1) :- alk_groups(m1,4).

Portanto teremos a seguinte relação entre termos e variáveis:

[[m1,A],[a1,B],[4,C]].

Os nós iniciais desta cláusula são [m1,4] e [a1]. Se começarmos a expansão pelo nó [m1,4], poderemos retornar da *bottom clause* apenas literais que possuam pelo menos uma das variáveis A ou C.

Suponha que a instância positiva *great_ne(dd1,k1)* foi escolhida aleatoriamente para gerar a *bottom clause* dada por:

Bottom Clause: great_ne(A,B) :- alk_groups(B,D), alk_groups(A,C), r_subst_1(B,E), r_subst_1(A,F), r_subst_2(B,G), r_subst_2(A,G), ring_substitutions(B,H), ring_subst_3(A,I), polar(I,J), gt(C,H), gt(D,H),gt(D,C)

e que as variáveis da *bottom clause* já foram unificadas com as variáveis da cláusula. Dentre os literais disponíveis, aqueles que possuem a variável A ou C, e que são retornados, um por um, como antecedentes na expansão do nó [m1,4] são:

alk_groups(A,C), r_subst_1(A,F), r_subst_2(A,G), ring_subst_3(A,I), gt(C,H), gt(D,C)

A cada antecedente retornado, as variáveis dos literais são instanciadas pelos *end values* correspondentes, e as que não tem relação com nenhum *end-value* continuam sendo variáveis.

Como só temos os termos $m1$ e 4 que correspondem às variáveis A e C, respectivamente, os literais acima seriam instanciados por:

alk_groups(m1,4), r_subst_1(m1,F), r_subst_2(m1,G), ring_subst_3(m1,I), gt(4,H), gt(D,4)

Para cada um destes antecedentes, será verificado no conjunto de fatos se existe algum fato que unifique com cada um destes literais. Os seguintes fatos foram encontrados na base de dados:

alk_groups(m1,4), r_subst_1(m1,single_alk(3)), r_subst_2(m1,double_alk(1)),

gt(4,3)

Os literais *ring_subst_3(m1,I)* e *gt(D,4)* não conseguiram ser unificados, e portanto são descartados.

Para os literais que conseguiram ser instanciados, é feita uma vinculação dos novos *end values* com as variáveis que eles representam na *bottom clause*. Para o exemplo acima temos as vinculações [[single_alk(3),F],[double_alk(1),G],[3,H]].

O próximo nó a ser expandido neste exemplo é o [a1], onde o termo a1 está vinculado à variável B. Apesar de já termos feito a vinculação entre termos e as variáveis F, G e H, neste nó só temos a variável B, e portanto só poderemos buscar literais que possuam a variável B. Só quando formos expandir o nó [single_alk(3),double_alk(1),3] resultante da expansão do nó [m1,4], é que serão consideradas as variáveis F, G e H na busca por literais da *bottom clause*.

Depois que a busca de antecedentes termina e que todos os caminhos relacionais foram retornados e validados, o algoritmo irá substituir a instanciação das novas cláusulas por variáveis. Neste momento é usada as relações entre termos e variáveis. Basta substituir cada termo por sua variável correspondente.

Quando usamos a *bottom clause*, diminuímos consideravelmente o espaço de busca de antecedentes, porque já estamos buscando num conjunto de antecedentes relevantes com argumentos pré-determinados, não precisando fazer permutações de variáveis para descobrir os argumentos.

3.4 Declaração de modos nos algoritmos modificados

No FORTE não é utilizada nenhuma definição de modos ao adicionar o antecedente à cláusula sendo revisada, ou seja, as variáveis não são restritas a serem de entrada e/ou de saída. Os antecedentes podem ser adicionados à cláusula desde que respeitem os tipos dos argumentos e que possuam uma variável em comum com a cláusula. No caso da busca utilizando *relational pathfinding*, os caminhos retornados são apenas validados quanto aos *singletons*, ou seja, apenas é verificado se existe uma variável em um literal que não esteja em outro literal, de forma a fazer

um caminho relacional.

Até o momento, a definição de modos só foi utilizada para gerar a *bottom clause*, de forma que podemos garantir que na *bottom clause* os literais foram inseridos obedecendo às definições de entrada e saída das variáveis. Porém, quando escolhemos um literal da *bottom clause* para ser adicionado à cláusula sendo revisada, não estamos verificando se a cláusula formada respeita os modos. Se a cláusula formada fosse exatamente igual à *bottom clause*, com certeza estaria respeitando aos modos, mas se algum literal da *bottom clause* deixou de ser inserido não cláusula, não podemos mais garantir que os modos estão sendo respeitados, pois este literal poderia ter introduzido uma variável que seria importante para validar outros literais na cláusula.

Para verificar a consistência da cláusula gerada em relação aos modos, adaptamos o trabalho realizado em (COUTO, et al., 2008). Este trabalho foi desenvolvido em paralelo com esta dissertação, e seu objetivo foi tornar a revisão de teorias no FORTE direcionada a modos, ou seja, incluir a definição de modos no FORTE para que a busca por antecedentes se tornasse mais eficiente, além de respeitar as restrições de entrada e saída das variáveis. Como a *bottom clause* em si já é gerada considerando a definição de modos proveniente do Aleph, adaptamos a utilização de modos no FORTE apenas para validar as cláusulas geradas na adição de antecedentes.

No caso do algoritmo *hill climbing*, é adicionado um antecedente por vez, de forma que ao adicionar o antecedente passamos a verificar se as variáveis do novo literal estão de acordo com os modos definidos na cláusula existente, ou seja, se no novo literal tivesse uma variável definida como de entrada, tal variável já deveria ter aparecido a cláusula.

Considerando a base de dados descrita na seção anterior, suponha que a cláusula sendo revisada seja *great_ne(A,B)*, onde a definição de modos é dado por

modeh(1,great_ne(+a,+a)).

Agora suponha que eu tenha dois antecedentes possíveis, provenientes da *bot-*

tom clause, para adicionar na cláusula, dados por *alk_groups(B,C)* e *r_subst_1(D,A)*, cujos modos são definidos por

modeb(,alk_groups(+a,-n)).*
modeb(,r_subst_1(+a,-l)).*

O antecedente *alk_groups(B,C)* pode ser adicionado à cláusula pois a variável B é de entrada e já apareceu antes em *great_ne(A,B)*, enquanto que a variável C é de saída e por isso não precisa ter aparecido antes na cláusula. O antecedente *r_subst_1(D,A)* não pode ser adicionado à cláusula pois a variável D é de entrada e portanto deveria ter aparecido antes na cláusula, o que não acontece, pois a cláusula em questão só possui as variáveis A e B. Se não levássemos em consideração os modos, ambos os antecedentes poderiam ter sido adicionados à cláusula, já que os dois possuem uma variável em comum com a cláusula.

No caso do algoritmo *relational pathfinding* são adicionados vários antecedentes de uma só vez. Neste caso não podíamos verificar se o literal respeitava a declaração de modos no momento em que ele estava sendo adicionado à cláusula, pois podia acontecer de não respeitar naquele momento mas passar a respeitar assim que o próximo literal fosse adicionado, já que na cláusula não consideramos a ordem dos literais para verificar se os modos estão sendo obedecidos.

Por exemplo, suponha que a cláusula inicial instanciada seja *great_ne(b1,a1)*, e portanto temos os nós [b1] e [a1]. Suponha que vamos começar a expansão pelo nó [b1]. Digamos que usando a *bottom clause* conseguimos retornar o antecedente *x_subst(b1,7,cl)*. Seja a declaração de modos dada por

modeh(1,great_ne(+a,+a)).
modeb(,x_subst(+a,+n,-b)).*

Percebe-se que o termo 7 em *x_subst(b1,7,cl)* é de entrada, porém 7 não aparece na cláusula inicial, e se tivéssemos obedecendo aos modos neste momento, o antecedente *x_subst(b1,7,cl)* não poderia ser adicionado. Agora suponha que

não estamos considerando os modos, que o antecedente *x_subst(b1,7,cl)* foi adicionado e que iremos expandir o nó [a1]. Para este nó, retornamos o antecedente *alk_groups(a1,7)* que tem modos definido por

modeb(,alk_groups(+a,-n)).*

Como o termo a1 é de entrada e aparece na cláusula inicial, o antecedente pode ser adicionado. Neste momento encontraríamos uma interseção no termo 7, a partir dos antecedentes *alk_groups(a1,7)* e *x_subst(b1,7,cl)*, formando o seguinte caminho:

great_ne(a1,b1) :- *x_subst(b1,7,cl),alk_groups(a1,7).*

Podemos ver na claúsula acima que se trocássemos a ordem dos literais *alk_groups(a1,7)* e *x_subst(b1,7,cl)*, este último passaria a obedecer aos modos, pois o termo 7 que é de entrada já teria aparecido antes em *alk_groups(a1,7)*. Como a ordem dos literais em uma claúsula não importa quando estamos verificando os modos, a cláusula acima estaria respeitando os modos de qualquer jeito. Porém, se no momento que adicionamos o antecedente *x_subst(b1,7,cl)* tivéssemos levado os modos em consideração, não poderíamos ter adicionado este literal, pois o *alk_groups(a1,7)* ainda não tinha sido gerado.

Podemos concluir então que, para o algoritmo *relational pathfinding*, os modos não podem ser verificados no momento em que o antecedente é retornado. Os modos devem ser verificados depois que os caminhos relacionais são retornados, porque aí sim poderemos afirmar que um determinado literal não obedece aos modos, pois estamos olhando para cláusula como um todo, independente da ordem dos literais. Neste caso, a cláusula retornada no exemplo acima seria aceita sem problemas. Os caminhos que não obedecessem aos modos seriam desconsiderados e não seriam testados quanto à acurácia.

Com a introdução da declaração de modos para restringir as variáveis a serem de entrada ou saída, passamos a retornar cláusulas mais corretas, evitando que cláusulas que não obedecem aos modos fossem geradas e consequentemente dimi-

nuíndo o tempo gasto pelo FORTE para verificar a acurácia das cláusulas junto à teoria.

Capítulo 4

Resultados Experimentais

Neste capítulo mostraremos os experimentos realizados considerando 5 conjuntos de dados, com o objetivo de mostrar os benefícios obtidos quando revisamos uma teoria de primeira-ordem utilizando declaração de modos e a cláusula mais específica.

Os domínios considerados neste trabalho incluem:

- **Alzheimer:** Compara 37 análogos de Tacrine, que é uma droga contra o mal de Alzheimer, de acordo com 4 propriedades (KING, et al., 1995): inibir a reaceitação de **amine**, baixa **toxic**idade, alta inibição de acetil **choline**sterase e boa inversão de memória induzida por deficiência de e**scopo**lamine. Os conjuntos de instâncias são compostos de 686, 886, 1326 e 642 instâncias, respectivamente, igualmente divididos em instâncias positivas e negativas. Todas as instâncias pertencem ao mesmo exemplo.

- **Mutagênesis:** É um domínio bem conhecido para predizer o relacionamento entre a estrutura e atividade (SAR) de moléculas (SRINIVASAN, et al., 1996). O conjunto de instâncias é composto por 125 instâncias positivas e 63 instâncias negativas, onde cada instância corresponde a um exemplo. A meta é classificar compostos como "mutagênicos"ou não, dada sua estrutura química.

Nosso principal objetivo é investigar se o uso de declaração de modos e *bottom clause* na busca por antecedentes torna a revisão de teorias mais eficiente. Outra importante investigação conduzida neste trabalho é se o uso de declaração de modos e *bottom clause* melhora a acurácia de teorias obtidas a partir de sistemas ILP.

Adicionalmente, queremos verificar se a utilização da *bottom clause* e declarações de modos pode tornar o processo de aprendizado de teorias inteiras mais eficiente. Portanto, nesta dissertação analisamos os seguintes algoritmos:

1. Aleph
2. FORTE original
3. FORTE com *bottom clause* e modos

Para evitar *overfitting* durante o treinamento, utilizamos validação cruzada estratificada k-fold para dividir o conjunto de entrada em conjuntos distintos de treinamento e teste, e foi utilizada a validação cruzada estratificada t-fold para dividir o conjunto de treinamento em treinamento e validação, mantendo a taxa de exemplos positivos e negativos em cada fold.(BAIÃO, et al., 2003)(KOHAVI, 1995) Nós consideramos k=10 e t=5. Nós medimos acurácia e tempo de execução dos algoritmos e testamos diferenças significativas utilizando *corrected two-tailed paired t-test* com $p < 0.05$(NADEAU, BENGIO, 2003b).

Os algoritmos (2) e (3) foram rodados tanto para revisão quanto para geração de teorias a partir do zero, visando a comparação com o Aleph, que gera teorias a partir do zero.

O Aleph foi usado para gerar as teorias iniciais fornecidas para os algoritmos de revisão. Uma teoria diferente foi gerada para cada fold, e cada uma destas teorias foi revisada considerando seus respectivos folds, ou seja, os mesmos folds foram usados para gerar e revisar teorias.

Para execução do FORTE utilizamos computadores dual core, com 1 Gb de memória RAM. A implementação foi feita utilizando YAP Prolog, versão 5.1.2(COSTA V., DAMAS L., REIS R., AZEVEDO R.).

Os testes realizados não estão considerando os operadores de identificação e absorção para generalização de antecedentes, pois estes operadores são fortemente relacionados à existência de predicados intermediários, e nos domínios apresentados não existem predicados intermediários. Por isso, geralmente eles nao propõem revisão nenhuma.

As tabelas mostradas a seguir correspondem aos resultados obtidos pela revisão e aprendizado de teorias nos algoritmos (2) e (3), utilizando para adição de antecedentes as opções *non_relational* e *highly_relational* do *relational_tunning*, que correspondem respectivamente aos algoritmos do *hill climbing* e à intercalação dos algoritmos *hill climbing* e *relational pathfinding*. Os resultados mostrados são médias dos resultados de cada fold.

Nas tabelas a seguir o símbolo $\star$ indica que o FORTE com modos e *bottom clause* teve uma melhora significativa em relação ao FORTE original, e o símbolo $\bullet$ indica uma melhora significativa em relação ao Aleph.

Ao final do capítulo serão mostradas teorias geradas a partir da revisão usando-se o FORTE original e o FORTE com modos e *bottom clause*.

A tabela 4.1 mostra o tempo de execução e a acurácia no conjunto de teste dos algoritmos (1), (2) e (3) em todas as bases de dados quando revisando teorias a partir de toda a base de dados, e quando utilizando o algoritmo *hill climbing* para adição de antecedentes. Lembrando que o Aleph não faz revisões, ele gera a teoria do zero, mas é colocado na tabela para fins de comparação.

Tabela 4.1: Tempo de execução em segundos e acurácia de teste para os algoritmos em cada base de dados utilizando *hill climbing*

Bases	Aleph (1)		FORTE (2)		FORTE MODOS BC (3)	
	Tempo	Acurácia	Tempo	Acurácia	Tempo	Acuracia
Amine	15.83	62.68	616.90	72.47	42.32 $\star$	70.68 $\bullet$
Toxic	12.45	69.04	754.54	70.86	29.01 $\star$	72.31
Choline	37.80	56.02	376.04	65.89	282.7	64.48 $\bullet$
Scopo	15.81	51.71	3499.4	63.40	102.9 $\star$	63.53 $\bullet$
Muta	22.46	79.61	10.44	81.55	218.2	91.57 $\star\bullet$

O FORTE com modos e *bottom clause* consegue reduzir significativamente o tempo de execução em relação ao FORTE original, mantendo uma acurácia semelhante e sem diferenças estatisticamente significantes.

Na base do Mutagênesis, o tempo de execução do FORTE com Modos e *bottom clause* piorou em relação ao FORTE original e ao Aleph, mas em compensação a acurácia melhorou significativamente em relação a ambos os algoritmos, tendo uma diferença de aproximadamente 10 pontos percentuais. Isso acontece por o Mutagê-

nesis, ao contrário das outras bases, possui vários exemplos, onde cada exemplo possui uma única instância, e o conhecimento preliminar particular desse exemplo reflete apenas as informações dessa instância. Então, ao gerar uma *bottom clause*, está usará uma instância positiva que provavelmente identifica apenas um exemplo, e portanto a bottom clause refletirá o grupo de literais relevantes para aquele determinado exemplo. Quando a cláusula for revisada usando a *bottom clause*, a chance de ela provar apenas um exemplo é muito grande, o que obriga a gerar mais especializações da cláusula original, aumentando portanto o tempo de execução.

A tabela 4.2 mostra o tamanho final da teoria obtida para os algoritmos (1), (2) e (3) em todas as bases de dados quando revisando teorias a partir de toda a base de dados, e quando utilizando o algoritmo *hill climbing* para adição de antecedentes. Lembrando que o Aleph não faz revisões, ele gera a teoria do zero, mas é colocado na tabela para fins de comparação. O tamanho da teoria está sendo medido pelo número de literais que a formam.

Tabela 4.2: Tamanho da teoria para os algoritmos em cada base de dados considerando o algoritmo *hill climbing*

Bases	Aleph (1)	FORTE (2)	FORTE MODOS BC (3)
Amine	20.1	96.3	33.5 $\star$
Toxic	24.70	69.60	31.60 $\star$
Choline	30.70	66.90	54.80
Scopo	28.90	110	44.43 $\star$
Muta	20.20	24.8	46.56

O FORTE com Modos e *bottom clause* consegue reduzir o tamanho da teoria em relação ao FORTE original, com exceção na base do Mutagânesis, mas isso se dá pelo fato de que a teoria está sendo muito especializada, conforme explicado anteriormente.

A tabela 4.3 mostra o tempo de execução e acurácia no conjunto de teste dos algoritmos (1), (2) e (3) em todas as bases de dados quando aprendendo teorias do zero a partir de toda a base de dados, e quando utilizando o algoritmo *hill climbing* para adição de antecedentes.

O FORTE com modos e *bottom clause* consegue reduzir significativamente o tempo de execução em relação ao FORTE original quando aprendendo teorias do

Tabela 4.3: Tempo de execução em segundos e acurácia de teste para os algoritmos em cada base de dados utilizando *hill climbing* e gerando a teoria do zero

Bases	Aleph (1)		FORTE (2)		FORTE MODOS BC (3)	
	Tempo	Acurácia	Tempo	Acurácia	Tempo	Acuracia
Amine	15.83	62.68	140.04	67.82	4.74 ⋆ •	67.82
Toxic	12.45	69.04	611.56	68.36	8.37 ⋆	70.42
Choline	37.80	56.02	6501.3	62.96	21.2 ⋆ •	62.75 •
Scopo	15.81	51.71	1981.7	65.1	16.51 ⋆	64.16 •
Muta	22.46	79.61	4.27	76.06	121.05	88.64 ⋆

zero, equiparando-se ou até melhorando o tempo em relação ao Aleph. Novamente a única exceção é a base do mutagênesis, que piora o tempo mas consegue uma acurácia significativamente melhor.

A tabela 4.4 mostra o tamanho final da teoria obtida para os algoritmos (1), (2) e (3) em todas as bases de dados quando aprendendo teorias a partir de toda a base de dados, e quando utilizando o algoritmo *hill climbing* para adição de antecedentes.

Tabela 4.4: Tamanho da teoria para os algoritmos em cada base de dados considerando o algoritmo *hill climbing* e gerando a teoria do zero

Bases	Aleph (1)	FORTE (2)	FORTE MODOS BC (3)
Amine	20.1	42.6	6.5 ⋆ •
Toxic	24.70	50.10	12.00 ⋆ •
Choline	30.7	85.50	17.80 ⋆ •
Scopo	28.90	90.50	22.6 ⋆
Muta	20.20	19	49.78

Quando aprendendo teorias do zero, o FORTE com modos e *bottom clause* consegue aprender teorias menores que as aprendidas pelo Aleph.

Apesar do aprendizado de teorias ser reconhecidamente um problema mais custoso do que o algoritmo de cobertura (BRATKO, 1999), verificamos a partir das tabelas 4.3 e 4.4 que ao combinar as declarações de modos, a utilização da bottom clause e técnicas de revisão, o tempo de execução se torna similar ou até menor ao tempo gasto por um algoritmo de cobertura. Além disso, como já era de se esperar, acurácias melhores são obtidas e teorias menores são geradas.

A tabela 4.5 mostra o tempo de execução e a acurácia no conjunto de teste dos algoritmos (1), (2) e (3) em todas as bases de dados quando revisando teorias

a partir de toda a base de dados, e quando utilizando a intercalação dos algoritmos *hill climbing* e *relational pathfinding* para adição de antecedentes. Lembrando que o algoritmo Aleph não faz revisões, ele gera a teoria do zero, e que o Aleph também não utiliza *relational pathfinding*, mas é colocado na tabela para fins de comparação.

Tabela 4.5: Tempo de execução em segundos e acurácia de teste para os algoritmos em cada base de dados utilizando *hill climbing* e *relational pathfinding*

Bases	Aleph (1)		FORTE (2)		FORTE MODOS BC (3)	
	Tempo	Acurácia	Tempo	Acurácia	Tempo	Acuracia
Amine	15.82	62.68	7347.8	73.24	271.5 $\star$	74.24 $\bullet$
Toxic	12.45	69.04	11766	78.56	241.0 $\star$	77.75 $\bullet$
Choline	37.80	56.02	> 100	?	751.2 $\star$	64.48 $\bullet$
Scopo	15.81	51.71	20061	65.47	444.4 $\star$	64.29 $\bullet$

Novamente, o FORTE com modos e *bottom clause* consegue reduzir significativamente o tempo de execução em relação ao FORTE original. Para a base Choline, os testes estão rodando há mais de 100 horas e nenhum resultado foi obtido. Por isso não temos dados para a acurácia do FORTE original.

Para o mutagênesis não existe o caso relacional, ou seja, não é rodado o *relational pathfinding*, pois o predicado da cabeça da regra só possui um argumento, então não tem como formar caminhos relacionais.

A tabela 4.6 mostra o tamanho final da teoria obtida para os algoritmos (1), (2) e (3) em todas as bases de dados quando revisando teorias a partir de toda a base de dados, e quando utilizando a intercalação dos algoritmos *hill climbing* e *relational pathfinding* para adição de antecedentes. Lembrando que o algoritmo Aleph não faz revisões, ele gera a teoria do zero, mas é colocado na tabela para fins de comparação.

Tabela 4.6: Tamanho da teoria para os algoritmos em cada base de dados considerando o algoritmo *hill climbing* e *relational pathfinding*

Bases	Aleph (1)	FORTE (2)	FORTE MODOS BC (3)
Amine	20.1	181.1	55.8 $\star$
Toxic	24.70	129.84	65.70 $\star$
Choline	30.7	?	54.8
Scopo	28.90	253.8	88.6

A tabela 4.7 mostra o tempo de execução e acurácia no conjunto de teste dos

algoritmos (1), (2) e (3) em todas as bases de dados quando aprendendo teorias do zero a partir de toda a base de dados, e quando utilizando a intercalação dos algoritmos *hill climbing* e *relational pathfinding* para adição de antecedentes.

Tabela 4.7: Tempo de execução em segundos e acurácia de teste para os algoritmos em cada base de dados utilizando *hill climbing* e *relational pathfinding*, e gerando a teoria do zero

Bases	Aleph (1)		FORTE (2)		FORTE MODOS BC (3)	
	Tempo	Acurácia	Tempo	Acurácia	Tempo	Acuracia
Amine	15.83	62.68	3526.3	73.24	68.07 $\star$	75.35 $\bullet$
Toxic	12.45	69.04	15356	77.99	127.8 $\star$	76.92 $\bullet$
Choline	37.80	56.02	> 100	?	213.8 $\star$	64.91 $\bullet$
Scopo	15.82	51.71	14804	58.9	200.5 $\star$	63.6 $\bullet$

Quando aprendendo uma teoria do zero, o FORTE com modos e *bottom clause* continua melhorando o tempo de execução em relação ao FORTE original, porém, para o caso do *relational pathfinding*, o tempo em relação ao Aleph não é melhorado.

A tabela 4.8 mostra o tamanho final da teoria obtida para os algoritmos (1), (2) e (3) em todas as bases de dados quando aprendendo teorias a partir de toda a base de dados, e quando utilizando a intercalação dos algoritmos *hill climbing* e *relational pathfinding* para adição de antecedentes.

Tabela 4.8: Tamanho da teoria para os algoritmos em cada base de dados considerando o algoritmo *hill climbing* e *relational pathfinding*, e gerando a teoria do zero

Bases	Aleph (1)	FORTE (2)	FORTE MODOS BC (3)
Amine	20.10	96.3	48.9 $\star$
Toxic	24.70	135.68	41.40 $\star$
Choline	30.70	?	49.50
Scopo	28.90	243.5	53

Por adicionar vários antecedentes de uma só vez em uma cláusula, o *relational pathfinding* é mais custoso, e portanto demora mais tempo, gerando também teorias maiores.

O melhor resultado em relação a tempo de execução ao se fazer revisão de teorias foi na base de dados Toxic, ao usar a intercalação dos algoritmos *hill climbing*

e *relational pathfinding*. Foi alcançado um tempo 49 vezes mais rápido do que o FORTE original. Na média foi alcançado um tempo de três ordens de magnitude mais rápido.

Já quando gerando teorias do zero, foi alcançado na base Choline um tempo 306 vezes mais rápido do que o FORTE original, utilizando-se o algoritmo *hill climbing*.

Segue abaixo três tabelas que indicam a quantidade de vitórias e derrotas significativas, considerando tempo de execução, acurácia e tamanho, respectivamente, que o FORTE com modos e *bottom clause* teve em relação ao FORTE original e ao Aleph, em cada algoritmo apresentado, e considerando todas as bases de dados apresentadas.

Tabela 4.9: Tabela de vitórias e derrotas significativas quanto ao tempo de execução

Tempo	*Hill Climbing*		*Hill Climbing* do zero		*Rel Path*		*Rel Path* do zero	
	FORTE	Aleph	FORTE	Aleph	FORTE	Aleph	FORTE	Aleph
Vitórias	3	0	4	2	4	0	4	0
Derrotas	1	5	1	1	0	4	0	4

Tabela 4.10: Tabela de vitórias e derrotas significativas quanto a acurácia

Acurácia	*Hill Climbing*		*Hill Climbing* do zero		*Rel Path*		*Rel Path* do zero	
	FORTE	Aleph	FORTE	Aleph	FORTE	Aleph	FORTE	Aleph
Vitórias	1	4	1	2	0	4	0	4
Derrotas	0	0	0	0	0	0	0	0

Tabela 4.11: Tabela de vitórias e derrotas significativas quanto ao tamanho da teoria

Tamanho	*Hill Climbing*		*Hill Climbing* do zero		*Rel Path*		*Rel Path* do zero	
	FORTE	Aleph	FORTE	Aleph	FORTE	Aleph	FORTE	Aleph
Vitórias	3	0	4	3	2	0	2	0
Derrotas	1	5	1	1	0	4	0	4

Os resultados acima mostraram que quando o FORTE com modos e *bottom clause* aprende uma teoria do zero, utilizando o algoritmo *hill-climbing* para adicionar antecedentes, o tempo de execução se torna tão rápido quanto o tempo do Aleph, o que não acontecece em revisão.

O Aleph, assim como a maioria dos sistemas ILP, utiliza o algoritmo de cobertura, onde hipóteses são construídas iterativamente cláusula por cláusula. O algoritmo de cobertura é guloso no sentido de que cada iteração adiciona a melhor cláusula de acordo com algum critério de avaliação local, o que leva à geração de cláusulas localmente ótimas porém muito grandes(BRATKO, 1999). Por se adequar localmente, o Aleph acaba gerando muitas cláusulas com o mesmo predicado na cabeça. Além disso, o Aleph pode não provar todos os exemplos positivos, e quando a teoria gerada pelo Aleph é passada para o FORTE, estes exemplos não provados acabarão fazendo com que todas as cláusulas sejam pontos de revisão de generalização na teoria, o que torna a geração de revisões muito custosa.

Já quando o FORTE aprende uma teoria do zero, ele não tem que avaliar uma teoria inicial criada localmente. O FORTE já cria cláusulas pensando na teoria como um todo, e além disso a busca por pontos de revisão vai ser muito mais rápida pois a teoria vai ser pequena. Ao usar a *bottom clause* como espaço de busca de antecedentes, a revisão fica ainda mais eficiente, fazendo com que o tempo de execução se assimile ou fique melhor do que o Aleph. Por isso que quando o FORTE é usado para aprender teorias do zero ele acaba tendo um tempo muito melhor do que se tivesse que revisar uma teoria inicial. Devido a isso, acreditamos que ao usar um sistema de aprendizado de teorias como o Hyper(BRATKO, 1999) para gerar teorias iniciais, a revisão será de fato mais eficiente do que o aprendizado do zero, visto que as cláusulas já serão geradas globalmente. Este é um trabalho que já está em andamento.

Os resultados também mostraram que ao revisar uma teoria inicial aprendida com 100 porcento dos exemplos, ou seja, sem ter acesso a nenhum exemplo novo, o FORTE com declaração de modos e *bottom clause* ainda assim conseguiu melhorar a teoria. Numa situação real de revisão a teoria incial é gerada considerando apenas uma parte dos exemplos, o que permite ao sistema de revisão ter acesso a exemplos novos. Este é um trabalho que também está em andamento. O Aleph gera uma teoria inicial utilizando apenas uma parte dos exemplos e essa teoria é revisada pelo FORTE utilizando-se todo o conjunto de treinamento. O Aleph então pode aprender a teoria do zero ou pode usar a teoria gerada com menos exemplos como

conhecimento preliminar.

Mostraremos a seguir algumas das teorias geradas pelo Aleph, FORTE original e FORTE como modos e *bottom clause*, ao utilizar o algoritmo *hill climbing* em revisão e com aprendizado do zero no mesmo conjunto de exemplos. Queremos apenas exemplificar as diferenças nas teorias geradas pelos três sistemas.

Suponha o o Aleph gerou a seguinte teoria utilizando a base de dados Amine:

great_ne(A,B):-alk_groups(B,C),r_subst_3(A,D),ring_substitutions(A,C).
great_ne(A,B):-x_subst(A,C,D),r_subst_2(A,E),ring_subst_4(B,D).
great_ne(A,B):-r_subst_3(B,C),ring_substitutions(B,D),ring_substitutions(A,D).
great_ne(A,B):-r_subst_2(A,C),ring_subst_5(B,D).
great_ne(A,B):-alk_groups(B,C),ring_substitutions(A,D),gt(D,C).

Utilizando esta teoria como inicial, uma revisão feita pelo FORTE original foi:

great_ne(A,B):-r_subst_2(A,C),x_subst(A,D,E),ring_subst_5(F,E),ring_subst_3(B,G).
great_ne(A,B):-r_subst_2(A,C),ring_subst_3(A,D),ring_subst_5(E,D),ring_subst_3(B,F).
great_ne(A,B):-r_subst_2(A,C),x_subst(A,D,E),ring_subst_5(F,E),x_subst(B,G,H).
great_ne(A,B):-r_subst_2(A,C),r_subst_3(B,D),ring_subst_3(A,E).
great_ne(A,B):-r_subst_2(A,C),ring_subst_3(A,D),ring_subst_5(E,D),x_subst(B,F,G).
great_ne(A,B):-r_subst_2(A,C),ring_subst_4(B,D),r_subst_2(B,C).
great_ne(A,B):-r_subst_2(A,C),ring_subst_4(B,D),ring_subst_5(E,D).
great_ne(A,B):-r_subst_2(A,C),ring_subst_4(B,D),x_subst(A,E,F),ring_subst_5(G,F).
great_ne(A,B):-r_subst_2(A,C),ring_subst_4(B,D),ring_subst_5(E,D),r_subst_3(A,F).
great_ne(A,B):-r_subst_2(A,C),ring_subst_4(B,D),ring_subst_5(E,D),ring_subst_3(A,F).
great_ne(A,B):-r_subst_2(A,C),ring_subst_2(A,D).
great_ne(A,B):-r_subst_2(A,C),ring_subst_4(B,D),ring_subst_5(E,D),x_subst(B,F,G).
great_ne(A,B):-r_subst_2(A,C),ring_subst_4(B,D),ring_subst_5(E,D),x_subst(B,F,G),
ring_subst_2(A,G).
great_ne(A,B):-r_subst_2(A,C),ring_subst_4(B,D),ring_subst_5(E,D),x_subst(B,F,G),
r_subst_3(A,H).

great_ne(A,B):-r_subst_2(A,C),ring_subst_4(B,D),ring_subst_5(E,D),x_subst(B,F,G),
x_subst(A,F,G).
great_ne(A,B):-alk_groups(B,C),r_subst_3(A,D),ring_substitutions(A,C).
great_ne(A,B):-x_subst(A,C,D),r_subst_2(A,E),ring_subst_4(B,D).
great_ne(A,B):-r_subst_3(B,C),ring_substitutions(B,D),ring_substitutions(A,D).
great_ne(A,B):-r_subst_2(A,C),ring_subst_5(B,D).
great_ne(A,B):-alk_groups(B,C),ring_substitutions(A,D),gt(D,C).

Utilizando a mesma teoria inicial, a revisão feita pelo FORTE com *bottom clause* e declaração de modos foi:

great_ne(A,B):-r_subst_2(A,C),ring_subst_2(A,D).
great_ne(A,B):-r_subst_2(A,C),ring_subst_4(B,D).
great_ne(A,B):-alk_groups(B,C),r_subst_3(A,D),ring_substitutions(A,C).
great_ne(A,B):-x_subst(A,C,D),r_subst_2(A,E),ring_subst_4(B,D).
great_ne(A,B):-r_subst_3(B,C),ring_substitutions(B,D),ring_substitutions(A,D).
great_ne(A,B):-r_subst_2(A,C),ring_subst_5(B,D).
great_ne(A,B):-alk_groups(B,C),ring_substitutions(A,D),gt(D,C).

Agora considerando o mesmo conjunto de exemplos que gerou a teoria inicial do Aleph mostrada acima, a teoria aprendida do zero pelo FORTE original foi:

great_ne(A,B):-r_subst_2(A,C).
great_ne(A,B):-r_subst_2(A,C),ring_subst_4(B,D).
great_ne(A,B):-r_subst_2(A,C),x_subst(B,D,E),ring_subst_2(A,F).
great_ne(A,B):-r_subst_2(A,C),x_subst(B,D,E),ring_subst_2(A,F),ring_subst_2(B,G).
great_ne(A,B):-r_subst_2(A,C),x_subst(B,D,E),ring_subst_2(A,F),ring_subst_2(B,F).
great_ne(A,B):-r_subst_2(A,C),x_subst(B,D,E),ring_subst_4(B,F),x_subst(A,D,G).
great_ne(A,B):-r_subst_2(A,C),ring_subst_4(B,D),ring_subst_5(E,D),ring_subst_2(B,D).

e a teoria aprendida do zero pelo FORTE com declaração de modos e *bottom*

clause foi:

great_ne(A,B):-r_subst_2(A,C).

great_ne(A,B):-r_subst_2(A,C),ring_subst_2(A,D).

Capítulo 5

Conclusão

No sistema de revisão de teorias FORTE, a busca por antecedentes a serem adicionados numa cláusula sendo revisada considera todos os literais da base de conhecimento, o que acarreta em um espaço de busca muito grande tornando a operação de adição de antecedentes ineficiente, já que avaliar cada um destes literais é muito custoso. O presente trabalho melhorou a eficiência desta operação utilizando a *bottom clause* como espaço de busca de antecedentes, tanto na abordagem *hill climbing* quanto no algoritmo *relational pathfinding*. Foi utilizada também a declaração modos para definir quais literais da *bottom clause* podem efetivamente ser adicionados à cláusula sendo revisada.

Resultados experimentais mostraram que houve uma melhora significativa em eficiência em relação ao FORTE original, sem que a acurácia fosse prejudicada. Além disso, conseguimos reduzir o tamanho das teorias geradas, tornando-as mais compreensíveis.

Ao utilizar o sistema de revisão para aprender do zero e considerando a *bottom clause*, em alguns casos o tempo de execução foi melhor até mesmo que o Aleph, o que não acontece quando teorias iniciais são revisadas. Isso nos leva a concluir que o processo de revisão pode ser ainda mais eficiente se as teorias iniciais forem melhores, ou seja, devemos gerar teorias iniciais a partir de outros sistemas ILP, e também a partir do Aleph utilizando menos exemplos do conjunto de treinamento.

5.1 Trabalhos Futuros

As duas abordagens mais conhecidas em ILP são a *bottom-up* e a *top-down*. Métodos *bottom-up*, tal como o utilizado pelo sistema Aleph, começam com a cláusula mais específica gerada a partir de um exemplo positivo e a generalizam ao máximo sem permitir que exemplos negativos sejam cobertos. Já os métodos *top-down* começam com a cláusula mais geral, que no caso de aprendizado de teorias é a cláusula vazia, e a especializam até que não cubra mais exemplos negativos. Porém ambas as abordagens possuem problemas ao lidar com bases de dados muito grandes.

Em problemas com muitos exemplos, como acontece em EELD (*Evidence Extraction and Link Discovery*), o conhecimento preliminar contém muitos fatos contendo numerosos predicados que descrevem cada objeto complexo ou evento. Geralmente, muitos destes fatos são irrelevantes para a tarefa em questão. Entretanto, a *bottom clause* gerada pelo sistema PROGOL/Aleph inclui cada pedaço do conhecimento preliminar em seu corpo. Isto leva a *bottom clauses* muito grandes, o que gera um espaço de hipóteses de tamanho exponencial quando se está aprendendo uma cláusula. Visando resolver este problema foi proposto o sistema BETH(TANG, et al., 2003), que apresenta um novo algoritmo de aprendizado em ILP que integra as buscas *top-down* e *bottom-up* para reduzir o espaço de busca ao processar bases de dados muito grandes.

O algoritmo não mais constrói a *bottom clause* usando um exemplo positivo antes de começar a busca por uma boa cláusula. Ao invés disso, depois que o exemplo é escolhido são gerados literais de acordo com a abordagem top-down, sendo que tais literais estão restritos àqueles que cobrem o exemplo positivo. A *bottom clause* passa a ser virtual no sentido de que o algoritmo não mais precisa construí-la de antemão, como é feito no Aleph, pois ela será descoberta durante a busca por uma boa cláusula.

Pretendemos portanto, como trabalho futuro, incorporar a idéia apresentada pelo BETH ao sistema de revisão de teorias FORTE. Nesta dissertação restringimos a busca de antecedentes a serem adicionados à cláusula sendo especializada ao conjunto de literais da *bottom clause*, sendo esta gerada pelo Aleph. Queremos

então que a *bottom clause* não mais seja gerada de antemão, e sim que possamos revisar a cláusula integrando as abordagens *top-down* e *bottom-up* da forma como foi apresentada pelo sistema BETH.

Outra questão que pretendemos explorar futuramente é comparar o *relational pathfinding* direcionado a modos do Aleph(ONG, et al., 2005) com o *relational pathfinding* usado no FORTE para revisão de teorias e, segundo esta dissertação, utilizando a cláusula mais específica como espaço de busca de antecedentes. Como foi dito anteriormente na seção de trabalhos relacionados, a abordagem do *relational pathfinding* no Aleph difere do *relational pathfinding* aplicado no FORTE primeiramente porque o Aleph é um sitema de aprendizando de teorias, enquanto que o FORTE é um sistema de revisão de teorias já existentes. E em segundo lugar, o *relational pathfinding* apresentado no FORTE busca por uma interseção entre os *end-values* gerados em cada expansão de um nó, sendo que assim que uma interseção é achada, o algoritmo retorna os caminhos gerados por esta interseção. Além disso, os *end-values* são termos que instanciam a cláusula sendo especializada a partir de um exemplo escolhido randomicamente. Já no *relational pathfinding* direcionado a modos do Aleph, a busca dos caminhos se dá na cláusula saturada variabilizada e não instanciada, levando em consideração as definições de variáveis de entrada e saída dos literais da *bottom clause*, e o algoritmo não termina quando uma interseção é achada, ele continua buscando por outras interseções até que todos os literais da *bottom clause* tenham sido considerados.

Apesar de o Aleph ser um sistema de aprendizado de teorias e o FORTE um sistema de revisão, seria interessante comparar o algoritmo *relational pathfinding* de ambos quando gerando uma teoria do zero. Vimos nesta dissertação que o FORTE é capaz de aprender teorias do zero, e que muitas vezes o resultado obtido é tão bom ou melhor do que o Aleph. Portanto acreditamos que a comparação dos dois sistemas seria um trabalho relevante.

Em (PAES, et al., 2008) técnicas de busca local estocástica foram aplicadas ao sistema FORTE e foi observada uma melhora no tempo de execução bem como na acurácia. Assim como no sistema original, a geração de antecedentes era feita utilizando toda a base de conhecimento. Logo, um trabalho futuro seria introduzir

a cláusula mais específica ao gerar os antecedentes no sistema de revisão estocástica, esperando com isso obter uma eficiência ainda melhor.

Bibliografia

ADé, H., MALFAIT, B., RAEDT, L. D., 1994, "RUTH: an ILP theory revision system", In:*Proc. 8th Int. Symposium on Methodologies for Intelligent Systems (ISMIS-94) Berlin. Springer Verlag.*

AGUIAR, S., 2005, "XFORTEX: Validação Cruzada e Conjunto de Validação no Sistema FORTE", In:*Projeto Final de Curso, Departamento de Ciência da Computação do Instituto de Matemática da Universidade Federal do Rio de Janeiro.*

BAIÃO, F., MATTOSO, M., SHAVLIK, J., ZAVERUCHA, G., 2003, "Applying Theory Revision to the Design of Distributed Databases", In:*Proceedings of the the 13th Int. Conference on Inductive Logic Programming, LNAI 2835, Springer Verlag*, pp. 57–74.

BOYTCHEVA S., *Overview of Inductive Logic Programming (ILP) Systems*, `www.iit.bas.bg/Cit_en/CIT_02_en/v2-1/27-36.pdf - Acessado em 28/03/2008.`

BRATKO, I., 1999, "Refining complete hypotheses in ILP", In:*In Proceeding of the 9th International Conference on ILP, LNAI 1634,Springer*, pp. 44–55.

CASANOVA, M. A., GIORNO, F., FURTADO, A., 1987, *Programação em Lógica e a Linguagem Prolog*, Edgard Blücher.

COSTA V., DAMAS L., REIS R., AZEVEDO R., *Yap Prolog*, `http://www.ncc.up.pt/yap/ - Acessado em 28/03/2008.`

COUTO, E., PEREIRA, R., PAES, A., ZAVERUCHA, G., 2008, "Busca direcionada a modos na revisão de teorias de primeira ordem", In:*submitted.*

CUSSENS, J., *Inductive Logic Programming*, www-users.cs.york.ac.uk/~jc/teaching/GSLT/ilp.pdf - Acessado em 28/03/2008.

HINTO, G. E., 1986, "Learning Distributed Representations of Concepts", In:*Proceedings of the Eighth Annual Conference of the Cognitive Science Society.*

KING, R. D., STERNBERG, M. J. E., SRINIVASAN, A., 1995, "Relating Chemical Activity to Structure: An Examination of ILP Successes", *New Generation Computing*, v. 13, n. 3-4, pp. 411–433.

KOHAVI, R., 1995, "A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection", In:*Proceedings of the International Joint Conference on Artificial Intelligence(IJCAI)*, pp. 1137–1145.

LAVRAC, N., DZEROSKI, S., 1994, *Inductive Logic Programming: Techniques and Applications*, Ellis Horwood, New York.

LLOYD, J., 1987, *Foundations of Logic Programming*, 2 ed., Springer Verlag.

M. FERRO, F. E. P. E. M. C. M., 2007, "O Sistema de Programação Lógica Indutiva Aleph - Características e Funcionamento", Technical Report 300, Universidade de São Paulo, http://www.icmc.usp.br/~biblio/index.php?destino=relatorios_tecnicos.php.

MITCHELL, T., 1997, *Machine Learning*, McGraw-Hill, New York.

MOONEY, R., MELVILLE, P., TANG, L., SHAVLIK, J., DUTRA, I., PAGE, D., , COSTA, . V. S., 2002, "Relational Data Mining with Inductive Logic Programming for Link Discovery", In:*Proceedings of the National Science Foundation Workshop on Next Generation Data Mining, Baltimore* AAAI/MIT Press.

MUGGLETON, S., 1991, "Inductive logic programming", *New Generation Computing*, v. 13, n. 4, pp. 245–286.

MUGGLETON, S., 1992a, *Inductive logic programming*, Academic Press, New York.

MUGGLETON, S., 1992b, "Inverting implication", In:*Proceedings of the Second International Workshop on Inductive Logic Programming, Tokyo.*

MUGGLETON, S., 1995, "Inverse Entailment and Progol", *New Generation Computing Journal*, v. 13, pp. 245–286.

MUGGLETON, S., 1999, "Inductive Logic Programming", In:*The MIT Encyclopedia of the Cognitive Sciences (MITECS), MIT Press.*

MUGGLETON, S., BUNTINE, W., 1988, "Machine invention of first-order predicates by inverting resolution", In:*Proceedings of the 5th International Conference on Machine Learning*, pp. 339–352.

MUGGLETON, S., De RAEDT, L., 1994, "Inductive Logic Programming: Theory and Methods", *Journal of Logic Programming*, v. 19, n. 20, pp. 629–679.

MUGGLETON, S., FENG, C., 1992, "Efficient induction of logic programs", In:*S. Muggleton, editor, Inductive Logic Programming. Academic Press, London*, pp. 281–298.

NADEAU, C., BENGIO, Y., 2003a, "Inference for the Generalization Error", *Machine Learning*, v. 52, n. 3, pp. 239–281.

NADEAU, C., BENGIO, Y., 2003b, "Inference for the Generalization Error", *Machine Learning*, v. 52, n. 3, pp. 239–281.

ONG, I. M., DUTRA, I. C., PAGE, D., COSTA, V. C., 2005, "Mode Directed Path Finding", *ECML*, v. 3720, pp. 673–681.

PAES, A., ZAVERUCHA, G., COSTA, V. S., 2008, "Revising First-order Logic Theories from Examples through Stochastic Local Search", In:*17th Annual International Conference on Inductive Logic Programming (ILP-2007), LNAI 4894, Springer*, pp. 200–210.

PAES, A. M., 2005, *PFORTE: Revisão de Teorias Probabilísticas de Primeira Ordem através de Exemplos*, Master's thesis, COPPE - UFRJ, Rio de Janeiro, RJ.

PINA, A. C., ZAVERUCHA, G., 2004, "An Algorithmic Presentation of Accuracy Comparison of Classification Learning Algorithms", Technical report, Universidade Federal do Rio de Janeiro, COPPE/PESC, Rio de Janeiro, Brasil.

PLOTKIN, G. D., 1970, "A note on inductive generalisation", *Machine Intelligence*, v. 5, pp. 153–163.

QUINLAN, J., 1990, "Learning logical definitions from relations", *Machine Learning*, v. 5, pp. 239–266.

RAEDT, L. D., BRUYNOOGHE, M., 1992, "Interactive concept learning and constructive induction by analogy", *Machine Learning*, v. 8, pp. 107–150.

RICHARDS, B. L., MOONEY, R. J., 1992, "Learning Relations by Pathfinding", In:*Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92), San Jose, CA*, pp. 50–55.

RICHARDS, B. L., MOONEY, R. J., 1995, "Automated Refinement of First-Order Horn-Clause Domain Theories", *Machine Learning*, v. 19, pp. 95–131.

RICHARDS B. L., *Instructions for Using FORTE*, `http://www.cs.utexas.edu/ftp/pub/mooney/forte/ - Acessado em 28/03/2008.`

RUSSELL, S., NORVIG, P., 2002, *Artificial intelligence: A modern approach, 2nd edition*, Englewook Cliffs, NJ: Prentice-Hall.

SHAPIRO, E. Y., 1983, *Algorithmic Program Debugging*, ACM Distinguished Doctoral Dissertations. The MIT Press, Cambridge, Mass.

SRINIVASAN, A., 2001, "The Aleph manual", Technical report, Oxford University.

SRINIVASAN, A., KING, R. D., MUGGLETON, S., STERNBERG, M. J. E., 1997, "Carcinogenesis Predictions Using ILP", In:*Proceedings of the 7th Int. Workshop on ILP*, volume 1297, pp. 273–287 Springer-Verlag.

SRINIVASAN, A., MUGGLETON, S., STERNBERG, M. J. E., KING, R. D., 1996, "Theories for Mutagenicity: A Study in First-Order and Feature-Based Induction", *Artificial Intelligence*, v. 85, n. 1-2, pp. 277–299.

TANG, L. P. R., 2003, *Integrating Top-down and Bottom-up Approaches in Inductive Logic Programming: Applications in Natural Language Processing and Relational Data Mining*, Ph.D. thesis, University of Texas, Austin, TX.

TANG, L. R., MOONEY, R. L., MELVILLE, P., 2003, "Scaling Up ILP to Large Examples: Results on Link Discovery for Counter-Terrorism", In:*Proceedings of the KDD-2003 Workshop om Multi-Relational Data Mining. Washington, DC*, pp. 107–121.

WOGULIS, J., 1991, "Revising relational domain theories", In:*Proceedings of the Eighth International Workshop on Machine Learning. San Mateo, CA: Morgan Kaufman*, pp. 462–466.

WROBEL, S., 1994, "Concept formation during interactive theory revsion", *Machine Learning*, v. 14, pp. 169–191.

WROBEL, S., 1996, "First-order Theory Refinement", In:Raedt, L. D., , editor, *Advances in Inductive Logic Programming*, pp. 14–33 IOS Press.

ZAVERUCHA, G., 2008, "Apostila de Lógica - Notas de aula - material distribuido nos cursos Cos230 e cos705".