Answer Set Programming probabilístico

Eduardo Menezes de Morais

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Programa: Ciência da Computação
Orientador: Prof. Dr. Marcelo Finger

Durante o desenvolvimento deste trabalho o autor recebeu auxílio financeiro da CAPES

São Paulo, dezembro de 2012

Answer Set Programming probabilístico

Esta versão definitiva da tese/dissertação
contém as correções e alterações sugeridas pela
Comissão Julgadora durante a defesa realizada
por Eduardo Menezes de Morais em 10/12/2012.

Comissão Julgadora:

- Profa. Dra. Renata Wassermann (presidente) - IME-USP
- Prof. Dr. Flávio Soares Corrêa da Silva - IME-USP
- Prof. Dr. Paulo E. Santos - FEI

Resumo

Answer Set Programming probabilístico

Este trabalho introduz uma técnica chamada Answer Set Programming Probabilístico (PASP), que permite a modelagem de teorias complexas e a verificação de sua consistência em relação a um conjunto de dados estatísticos. Propomos métodos de resolução baseados em uma redução para o problema da satisfazibilidade probabilística (PSAT) e um método de redução de Turing ao ASP.

Palavras-chave: programação lógica, lógica probabilística, satisfazibilidade probabilística (PSAT), answer set programming (ASP), redução muitos-para-um.

Abstract

Probabilistic Answer Set Programming

This dissertation introduces a technique called Probabilistic Answer Set Programming (PASP), that allows modeling complex theories and check its consistence with respect to a set of statistical data. We propose a method of resolution based in the reduction to the probabilistic satisfiability problem (PSAT) and a Turing reduction method to ASP.

Keywords: logic programming, probabilistic logic, probabilistic satisfiability (PSAT), answer set programming (ASP), many-to-one reduction.

Sumário

Lista de Abreviaturas

AnsProlog	*Answer Set Prolog.*
ASP	*Answer Set Programming.*
NP	Classe dos problemas polinomiais não determinísticos.
PSAT	Problema da satisfazibilidade probabilística (*Probabilistic Satisfiability*).
PASP	*Answer Set Programming* Probabilístico.
SAT	Problema da satisfazibilidade da lógica proposicional clássica.

Capítulo 1

Introdução

O problema da satisfazibilidade probabilística (Boole (1854); Nilsson (1986)) provê um arcabouço muito poderoso e expressivo para a análise de sentenças lógicas e relações entre dados estatísticos. Porém, representar teorias mais complexas e grandes bases de conhecimento utilizando a lógica proposicional pura é uma tarefa bastante difícil.

Answer Set Programming (Gelfond e Lifschitz (1988); Marek e Truszczyński (1998)) é uma forma de linguagem de programação declarativa que torna muito mais fácil a representação de bases de conhecimento complexas devido ao uso de *defaults* e outras construções de alto nível. Graças ao uso da negação como falha, torna-se mais fácil capturar o bom senso e tratar o *frame problem* (Gelfond (1989); Lifschitz (2002)).

1.1 Objetivos

Considere a Figura 1.1 e imagine que queremos chegar do vértice 1 ao vértice 6. Porém, por restrições de tráfego ou algum outro motivo qualquer, queremos limitar o número de vezes que passamos pela aresta $(1, 3)$ a 50% das vezes e o número de vezes que passamos pela aresta $(3, 4)$ para 40% da vezes.

Existe alguma distribuição de probabilidade sobre os possíveis caminhos de modo que as arestas demarcadas só sejam utilizadas na frequência pedida? E se a probabilidade para as arestas $(1, 3)$ e $(3, 4)$ for modificada para 30% e 40% respectivamente?

Este problema pode ser formulado como um problema de satisfazibilidade probabilística. Porém, formulá-lo desta maneira é trabalhoso e resulta em uma fórmula grande.

Por outro lado, formular um programa ASP para encontrar caminhos em um grafo é bem mais simples e resulta em um programa relativamente pequeno (veja o Exemplo 2.25).

O objetivo deste trabalho é desenvolver o Answer Set Programming Probabilístico, uma técnica que junta a facilidade de representação do Answer Set Programming (ASP) com o poder de tratamento de dados estatísticos do problema da satisfazibilidade probabilística. Esperamos que esta técnica permita o raciocínio sobre grandes bases de dados de fatos e regras que podem ser expressos mais facilmente graças ao poder do ASP.

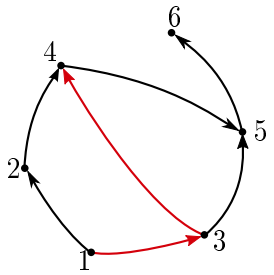

Figura 1.1: *Um grafo dirigido com possíveis caminhos entre 1 e 6*

Neste trabalho também descreveremos dois métodos de se encontrar respostas para o Answer Set Programming Probabilístico que foram implementados e podem ser usados para avaliar a sua utilidade na resolução de problemas com dados estatísticos.

1.2 Organização do trabalho

Nos Capítulos 2 e 3 serão revistos alguns conceitos do Answer Set Programming e do problema da satisfazibilidade probabilística necessários para os capítulos posteriores. No capítulo 4 será introduzido formalmente o Answer Set Programming Probabilístico e este será comparado com outra abordagem envolvendo ASP e probabilidades, o *P-log* (Baral *et al.* (2004, 2009)).Algoritmos para a resolução do Answer Set Programming Probabilístico serão mostrados no capítulo 5, assim como resultados experimentais. Finalmente no capítulo 6 avaliamos alguns possíveis usos do Answer Set Programming Probabilístico e indicamos possíveis trabalhos futuros. O apêndice A contém os conceitos de programação linear utilizados no trabalho.

Capítulo 2

Answer Set Programming

Answer Set Programming (ASP; usaremos o termo em inglês por ser o termo estabelecido na literatura) é um paradigma de programação declarativo e não monotônico orientado para problemas combinatórios difíceis e que facilita a modelagem de problemas reais.

Ao contrário de Prolog e outras linguagens de programação lógica, onde se resolve um problema definindo recursivamente o domínio e realizando *queries* (Clocksin (2003)), para resolver um problema em ASP nós criamos um programa de modo que os modelos-resposta (*Answer Sets*) deste programa sejam a resposta do problema modelado.

Neste capítulo, vamos apresentar a origem, sintaxe e semântica do ASP, além de apresentar propriedades relevantes ao nosso trabalho.

2.1 Negação em programação lógica

O conceito de modelos-resposta, que é a base do ASP, surgiu a partir da investigação da semântica da negação em linguagens de programação lógica.

Implementações de linguagens de programação lógica são dotadas de operadores para negação há muito tempo, por exemplo, Prolog possui o operador de negação desde sua criação. Porém, a semântica desta negação nem sempre foi clara.

A negação utilizada na maioria das linguagens de programação não é a negação clássica, mas a negação como falha.

A negação como falha de uma fórmula, `not p`, é verdadeira se não for possível concluir `p` de nenhuma das regras presentes no programa.

Para ilustrar a diferença entra a negação clássica e a negação com falha, considere o seguinte exemplo, inspirado em McCarthy (1980): se estamos tentando decidir se devemos ou não cruzar uma linha de trem e utilizarmos como critério a seguinte regra:

$$siga \leftarrow \textbf{not}\ trem$$

(onde **not** simboliza a negação como falha), então se não tivermos nenhuma indicação da presença do trem, cruzaremos a linha.

Isto, porém, significa que é aceitável cruzar uma linha de trem na ausência de informação sobre o trem, o que não é muito aconselhável. Nesta situação, o recomendável seria utilizar a negação clássica:

$$siga \leftarrow \neg trem$$

que indica que só é aceitável cruzar a linha do trem se pudermos saber com certeza que não existe trem.

Uma das primeiras semânticas propostas para negação clássica é a completação de um programa. No caso proposicional, a negação clássica e a negação como falha coincidem quando consideramos a completação de um programa, como visto em Clark (1978).

A completação de um programa é um conjunto de fórmulas proposicionais que expressam explicitamente que um predicado é verdadeiro se e somente se uma das regras do programa que inferem este predicado é verdadeira.

Exemplo 2.1. Dado o programa:

$$\begin{array}{lcl} a & \leftarrow & b \wedge \textbf{not } c \\ b & \leftarrow & d \\ b & \leftarrow & e \\ d & & \end{array}$$

Sua completação é

$$\begin{array}{lcl} a & \equiv & b \wedge \neg c \\ b & \equiv & d \vee e \\ c & \equiv & \bot \\ d & \equiv & \top \\ e & \equiv & \bot \end{array}$$

A completação de um programa deixa claro que a negação como falha está intimamente ligada com a hipótese de um mundo fechado, ou seja, é assumido implicitamente que tudo que não sabemos ser verdadeiro é considerado falso.

Alternativamente, podemos interpretar a `not p` como um operador que significa "não se acredita que p".

2.2 Modelos-resposta

A negação, porém, introduz um problema. Se não temos negação, a semântica de um programa é clara: um predicado é verdadeiro se e somente se ele pertence ao modelo de Herbrand mínimo do programa.

Um **modelo de Herbrand** é um modelo que utiliza apenas elementos do universo de Herbrand (Definição 2.11). Em outras palavras, é o modelo cujos termos são formados utilizando as constantes e símbolos de função que aparecem no programa. Um modelo de Herbrand M é **mínimo** se não existir nenhum outro modelo M' tal que $M' \subset M$.

Porém, quando a negação é introduzida pode ser que não tenhamos mais um único modelo de Herbrand mínimo para um programa.

Exemplo 2.2. O programa

$$\begin{array}{lcl} a & \leftarrow & \textbf{not } b. \\ b & \leftarrow & \textbf{not } a. \end{array}$$

Possui dois modelos de Herbrand mínimos: $\{a\}$ e $\{b\}$.

Quando um programa Prolog sem negação recebe uma *query*, deverá responder *true* se e somente se a *query* pertencer a este modelo de Herbrand. Porém quando existe mais de um modelo de Herbrand, não se pode afirmar isso.

Tanto o GNUProlog[1] quando o SWIProlog[2], implementações livres de Prolog, entram em um loop infinito quando se introduz o Exemplo 2.2, eventualmente parando com um *stack overflow*.

[1] http://www.gprolog.org/
[2] http://www.swi-prolog.org/

Michael Gelfond e Vladimir Lifschitz introduziram em Gelfond e Lifschitz (1988) o conceito de **modelos estáveis** (*stable models*) na tentativa de esclarecer a semântica da negação para programas lógicos. Posteriormente, com a adição da negação clássica neste arcabouço, o conceito de modelos estáveis foi estendido e chamado **modelo-resposta** (*answer set*).

O modelo-resposta de um programa P é definido com base no conceito da **redução** deste programa:

Definição 2.3 (Gelfond e Lifschitz (1988))**.** Seja M um conjunto de átomos de P, o programa P_M obtido a partir de P removendo:

- todas as regras que possuem um literal negativo **not** B no seu corpo, com $B \in M$;
- todos os literais negativos no corpo das regras restantes.

é chamado de **redução de P em relação à M** .

Claramente, o programa P_M não possui negação e portanto possui um único modelo de Herbrand mínimo. Se o modelo de Herbrand mínimo de P_M coincidir com M, então dizemos que M é um modelo-resposta do programa P.

Exemplo 2.4. Considere o programa P:

$$\begin{array}{lcl} a & \leftarrow & b, \textbf{not } c. \\ b. & & \end{array}$$

$M_1 = \{a, b\}$ é um modelo-resposta, pois P_{M_1} é:

$$\begin{array}{lcl} a & \leftarrow & b. \\ b. & & \end{array}$$

e M_1 é o modelo de Herbrand mínimo de P_{M_1}.

Porém, $M_2 = \{b, c\}$, apesar de ser um modelo de P, não é um modelo-resposta, pois P_{M_2} consiste de uma única regra, b, e portanto o Modelo de Herbrand mínimo de P_{M_2} não é M_2.

Percebemos também que no exemplo anterior, M_1 além de ser modelo-resposta de P, também é modelo de P. Isso é algo que acontece sempre, conforme mostrado em Gelfond e Lifschitz (1988).

Teorema 2.5 (Teorema 1 de Gelfond e Lifschitz (1988))**.** *Todo modelo-resposta de P é um modelo de Herbrand mínimo de P.*

Nesta semântica, se um programa possui mais de um modelo-resposta, o programa é considerado inválido e, portanto, pode ter um comportamento inesperado ou travar, como ocorre com o Exemplo 2.2.

2.3 ASP como um paradigma de programação

Marek e Truszczyński (1998) propuseram que se tratasse modelos-resposta não só como uma ferramenta auxiliar para Prolog, mas como um paradigma de programação lógica próprio, com sintaxe e semântica diferenciada do paradigma de Prolog. Nascia então o *Answer Set Programming* (ASP).

No ASP não podemos utilizar funções ou variáveis, pois elas tornariam o universo de Herbrand infinito. Por isto, programas ASP com variáveis deve estar sempre totalmente instanciado (veja adiante). A resposta de um problema é obtida não através de *querys*, mas na forma de modelos-resposta.

Antes de definir a semântica de ASP na seção 2.3.3, vamos olhar a sintaxe da linguagem normalmente usada para programar no paradigma ASP, O *AnsProlog*.

2.3.1 Answer Set Prolog

Answer Set Prolog, abreviada *AnsProlog*, é a linguagem normalmente utilizada para fazer programas em ASP.

Existem muitas variações sintáticas de *AnsProlog*. Neste trabalho usaremos $AnsProlog^{\neg,\bot}$, que inclui a negação clássica e permite o uso de restrições.

Vamos assumir uma assinatura de primeira ordem (C, F, P) contendo um conjunto de constantes C, um conjunto de símbolos funcionais F e um conjunto de predicados P. Assuma também um conjunto contável e não vazio de variáveis V.

Definição 2.6. Um **termo** é definido como o menor conjunto tal que:

- Uma variável é um termo;
- Uma constante é um termo;
- Uma função $f(t_1, \ldots, t_n)$ onde $t_1, \ldots, t_n$ são termos é um termo.

Definição 2.7. Um **átomo** possui a forma $p(a_1, \ldots, a_n)$ onde p é um predicado n-ário em P e $a_1, \ldots, a_n$ são termos.

Um átomo ou termo são ditos ***ground*** ou **instanciados** se eles não possuem variáveis.

Definição 2.8. Um **literal** é um átomo ou a sua negação clássica (precedido por $\neg$). Eles são chamados literais positivos e negativos respectivamente.

Um **literal estendido** é um literal ou a sua negação como falha (precedido por **not**). Um literal estendido também pode ser dividido em positivo e negativo devido à presença ou ausência da negação como falha. Perceba que **not** $\neg a$ é um literal estendido válido, que pode ser interpretado como "não se pode provar que a é falso".

Definição 2.9. Uma **regra** é da forma

$$h \leftarrow L_1, \ldots, L_m.$$

onde h é um literal e $L_1, \ldots, L_m$ são literais estendidos.

Em uma regra r, h é chamado a **cabeça** da regra e representado por $Head(r)$. Analogamente, $L_1, \ldots, L_m$ é chamado **corpo** de r e representado por $Body(r)$.

Se em uma regra $m = 0$, ou seja, se a regra não possui corpo, ela é chamada **fato**.

Além disso, em $AnsProlog^{\neg,\bot}$ uma regra da forma

$$\bot \leftarrow L_1, \ldots, L_m.$$

é chamada **restrição** e simboliza uma inconsistência entre os elementos do corpo. É possível omitir o símbolo $\bot$ das restrições e assumir a sua presença em qualquer regra com uma cabeça vazia.

Percebemos que restrições podem ser facilmente substituídas. Por exemplo, a restrição acima pode ser escrita como

$$p \leftarrow L_1, \ldots, L_m, \textbf{not } p. \tag{2.1}$$

com p sendo um átomo novo. Como p é um átomo novo, que não aparece na cabeça de nenhuma outra regra, então **not** p será verdadeiro. Portanto, se o corpo da restrição for satisfeito, isso implicará na derivação de p, fazendo com que esse modelo-resposta seja recusado.

Do mesmo modo, é possível eliminar a negação clássica de um programa (definido abaixo) simplesmente substituindo toda aparição de uma negação clássica $\neg a$ por um novo átomo a' e adicionando ao programa a regra:

$$\leftarrow a, a' \tag{2.2}$$

Definição 2.10. Um **programa** em $AnsProlog^{\neg,\bot}$ é um conjunto finito de regras

Conforme citado anteriormente, um programa ASP não pode possuir funções ou variáveis. As funções e variáveis apresentadas nesta seção são, na verdade, apenas uma abreviação sintática e antes de serem interpretadas devem ser eliminadas por um processo chamado *instanciação*, ou *grounding*. Para realizar a instanciação é necessário primeiro determinar o Universo de Herbrand de um programa.

Definição 2.11. O **Universo de Herbrand** de um programa P, HU_P, é o conjunto de todos os termos *ground* que podem ser formados utilizando as constantes e símbolos de função deste programa.

Para evitar um Universo de Herbrand infinito, muitos resolvedores de ASP limitam o número de vezes que uma função pode ser aninhada.

Definição 2.12. A **Base de Herbrand** de um programa P, HB_P, é o conjunto de todos os átomos *ground* que podem ser formados utilizando os predicados de P e os elementos de HU_P.

Exemplo 2.13 (Retirado de Syrjänen (2000))**.** Considere o programa P:

$$\begin{aligned} d(a). \quad d(b). \\ foo(X) \leftarrow \textbf{not } bar(X). \\ bar(X) \leftarrow \textbf{not } foo(X). \end{aligned}$$

O Universo de Herbrand de P é $\{a, b\}$ e a Base de Herbrand de P é $\{foo(a), foo(b), bar(a), bar(b)\}$.

Definição 2.14. A **instanciação** de um programa P é o conjunto de regras obtidas substituindo todas as variáveis em P pelos elementos de HU_P.

Exemplo 2.15. A instanciação do programa 2.13 é:

$$\begin{aligned} d(a). \quad d(b). \\ foo(a) \leftarrow \textbf{not } bar(a). \\ foo(b) \leftarrow \textbf{not } bar(b). \\ bar(a) \leftarrow \textbf{not } foo(a). \\ bar(b) \leftarrow \textbf{not } foo(b). \end{aligned}$$

2.3.2 Extensões

Existem várias extensões ao *AnsProlog*. A seguir veremos algumas que serão relevantes para o nosso trabalho.

Como descrever em detalhes cada uma dessas extensões estaria além do escopo deste trabalho, nos restringiremos aos pontos mais importantes.

Regras de escolha

Uma das extensões mais comuns e úteis é a **regra de escolha** (*choice rules*), introduzida em Gelfond e Lifschitz (1991).

Uma regra de escolha permite escolher não deterministicamente um átomo de um conjunto de possíveis átomos. Elas são da forma:

$$h_1, \ldots, h_n \leftarrow L_1, \ldots, L_m. \tag{2.3}$$

Esta regra afirma que se o corpo for verdadeiro, um entre $h_1, \ldots, h_n$ deve ser verdadeiro.

Exemplo 2.16. O programa:

$$a.$$
$$b, c \leftarrow a.$$

possui os modelos-resposta $\{a, b\}$ e $\{a, c\}$.

Em muitos casos, é possível eliminar as regras de escolha trocando as regras do tipo 2.3 pela seguinte construção:

$$\begin{array}{rcl} h_1 & \leftarrow & L_1, \ldots, L_m, \textbf{not } h_2, \ldots, \textbf{not } h_n \\ & \vdots & \\ h_n & \leftarrow & L_1, \ldots, L_m, \textbf{not } h_1, \ldots, \textbf{not } h_{n-1} \end{array}$$

Podemos verificar que essa técnica funciona com o exemplo 2.16. Porém, Ben-Eliyahu e Dechter (1992) mostrou que só podemos garantir que essa substituição funciona se não houver ciclos no *grafo de dependências positivas* (Definição 2.31) entre os literais que aparecem nas cabeças das regras. Por exemplo, considere o programa com ciclos abaixo:

Exemplo 2.17. O programa

$$a, b.$$
$$a \leftarrow b.$$
$$b \leftarrow a.$$

possui modelos-resposta $\{a, b\}$, enquanto

$$a \leftarrow \textbf{not } b.$$
$$b \leftarrow \textbf{not } a.$$
$$a \leftarrow b.$$
$$b \leftarrow a.$$

não possui modelos-resposta.

De fato, enquanto encontrar modelos-resposta de um programa $AnsProlog^{\neg,\bot}$ é um problema NP-completo, conforme será visto na seção 2.3.4, encontrar modelos-resposta de programas com regras de escolha com ciclos entre os literais da cabeça é mais difícil e pertence à classe de complexidade $\sum_2^P$ *−completo* (Ben-Eliyahu e Dechter (1992)).

Regras de peso

As regras de peso (*weight constraints rules*), introduzidas em Niemelä *et al.* (1999), podem ser vistas como generalizações das regras de escolha e serão muito úteis para o nosso trabalho.

Vamos começar definindo uma **restrição de peso**. Uma restrição de peso é da forma:

$$L \leq \{h_1 = p_1, \ldots, h_n = p_n, \textbf{not } n_1 = p_{n+1}, \ldots, \textbf{not } n_m = p_{n+m}\} \leq U. \tag{2.4}$$

onde h_i e n_i são literais, L e U são inteiros chamados respectivamente **limite inferior** e **limite superior** e $p_1, \ldots, p_{n+m}$ são os **pesos** associados a um literal l (também representado por $peso(l)$).

Intuitivamente, uma restrição de peso é satisfeita quando a soma dos pesos dos literais satisfeitos está entre L e U. Se L ou U forem omitidos, se assume que eles tenham o valor $-\infty$ e $+\infty$ respectivamente.

Exemplo 2.18. Considere as restrições de peso

$$C_1 = 2 \leq \{a = 1, b = 2, \textbf{not } c = 1\} \leq 3$$

$$C_2 = 1 \leq \{\textbf{not } a = 2, b = 1, c = 1\} \leq 2$$

O conjunto $\{a, b\}$ satisfaz C_2 e não satisfaz C_1.

Sem perda de generalidade, podemos assumir que todos os pesos são positivos, pois pesos negativos podem ser eliminados, como provado em Niemelä *et al.* (1999), substituindo:

$$L \leq \{h_1 = -p_1, \textbf{not } h_2 = -p_2\} \leq U$$

por

$$L + p_1 + p_2 \leq \{\textbf{not } h_1 = p_1, h_2 = p_2\} \leq U + p_1 + p_2$$

Uma **regra de peso** é da forma

$$C_0 \leftarrow C_1, \ldots, C_n \tag{2.5}$$

onde $C_0, \ldots C_n$ são restrições de peso e em C_0 não existem literais negativos. Uma regra de peso é satisfeita se sempre que $C_1, \ldots, C_n$ são satisfeitos, C_0 é satisfeito.

Exemplo 2.19. São exemplos de regras de peso:

$$1 \leq \{a = 1, b = 1, c = 1\} \leq 2 \quad \leftarrow \quad \begin{array}{l} 2 \leq \{d = 1, \textbf{not } b = 1, \textbf{not } e = 3\} \leq 4. \\ 1 \leq \{d = 3, e = 2\} \leq 5. \end{array}$$

A redução (Definição 2.3) de uma restrição de peso é um pouco diferente da redução de regras comuns. Removemos o limite superior e os literais negativos, descontando os pesos dos literais satisfeitos removidos.

Definição 2.20. Seja P um programa com restrições de peso e M um conjunto de literais desse programa, a **redução de uma restrição de peso** C da forma do esquema (2.4) é

$$C_M = L' \leq \{h_i = p_i | h_i \in M\}$$

onde

$$L' = L - \sum_{n_i \notin M} peso(n_i)$$

Para as regras de peso temos:

Definição 2.21. Seja M um conjunto de literais, a **redução de uma regra de peso** da forma do esquema (2.5) é o conjunto de regras

$$R_M = \begin{cases} \emptyset & \text{se } \exists C_{i \geq 1} : M \nvDash C_i \\ \{h \leftarrow C_{1M}, \ldots, C_{nM} | h \in M \text{ e } h \in C_0\} & \text{caso contrário} \end{cases}$$

onde $M \nvDash C$ significa que M não satisfaz a restrição C.

Finalmente, uma regra de escolha pode ser expressa utilizando uma regra de peso. Uma regra da forma (2.3) pode ser escrita como:

$$1 \leq \{h_1 = 1, \ldots, h_n = 1\} \leq 1 \leftarrow L_1, \ldots, L_m.$$

No *lparse*[3], um famoso instanciador ASP, as regras de escolha são, na verdade, regras de peso onde todos os literais possuem peso 1 (Syrjänen (2000)).

2.3.3 A semântica de ASP

Tendo definido a sintaxe que utilizaremos para descrever programas ASP, iremos agora discutir a semântica destes programas. Nesta seção, consideramos que os programas estejam instanciados e portanto que a sua base de Herbrand seja finita. Além disso, vamos considerar que as restrições e negações clássicas tenham sido eliminadas utilizando os esquemas (2.1) e (2.2).

Definição 2.22. Uma **interpretação** de um programa P é qualquer subconjunto da base de Herbrand H_P.

Definição 2.23. Uma interpretação I de um programa P que não contém negação como falha é chamada um **modelo** se e somente se para cada regra r do programa P, se $Body(r) \subseteq I$, então $Head(r) \in I$.

Um modelo M de P é chamado **modelo mínimo** de P se não existir um modelo M' de P tal que $M' \subset M$.

Ainda falta levar em consideração a negação como falha. Para isso, utilizaremos o conceito de redução de um programa mostrado na Definição 2.3, considerando agora que se trata de um programa ASP e não de um programa Prolog.

Agora podemos definir modelos-resposta de maneira análoga.

Definição 2.24. Seja P_I a redução do programa P em relação à interpretação I, I é um **modelo-resposta** (*answer set*) de P se o modelo mínimo de P_I for I.

O fato de que possivelmente possam existir diversos modelos-resposta para um programa não é mais visto como sinal de que o programa está "errado". Ao contrário, isto é parte da força do ASP e torna possível resolver problemas com mais de uma solução.

Exemplo 2.25. Considere o programa P abaixo, onde as letras em maiúsculo representam variáveis e as letras em minúsculo representam constantes:

$$passar(X,Y) \leftarrow aresta(X,Y), caminhoPara(X), \textbf{not } naoPassar(X,Y). \tag{2.6}$$

$$naoPassar(X,Y) \leftarrow aresta(X,Y), \textbf{not } passar(X,Y). \tag{2.7}$$

$$caminhoPara(1). \tag{2.8}$$

$$caminhoPara(Y) \leftarrow caminhoPara(X), passar(X,Y), aresta(X,Y). \tag{2.9}$$

$$\leftarrow passar(X,Y1), passar(X,Y2), Y1! = Y2, vertice(X), aresta(X,Y1), aresta(X,Y2). \tag{2.10}$$

$$\leftarrow \textbf{not } caminhoPara(n). \tag{2.11}$$

Se concatenado com a descrição de um grafo, escrita utilizando os predicados *aresta* e *vertice*, os modelos-resposta do programa P descrevem um caminho entre o vértice 1 e o vértice n. Uma

[3] http://www.tcs.hut.fi/Software/smodels/

descrição do grafo do Exemplo 1.1, P_G seria:

$$
\begin{aligned}
vertice(1..6).\\
aresta(1,2).\\
aresta(1,3).\\
aresta(2,4).\\
aresta(3,4).\\
aresta(4,5).\\
aresta(3,5).\\
aresta(5,6).
\end{aligned}
$$

As regras 2.6 e 2.7 escolhem arestas que farão parte do caminho. As regras 2.8 e 2.9 definem quais vértices são acessíveis pelo caminho. A regra 2.10 proíbe que num mesmo modelo-resposta sejam escolhidos dois caminhos, e finalmente a restrição 2.11 retira todos os modelos-resposta que não chegam ao vértice n.

O programa P utiliza variáveis. Portanto, para encontrar os modelos-resposta deste programa, é necessário instanciá-lo. O resultado da instanciação do programa P concatenado com a descrição do grafo do Exemplo 1.1 é:

$$
\begin{aligned}
&\leftarrow passar(1,2), passar(1,3).\\
&\leftarrow passar(1,3), passar(1,2).\\
&\leftarrow passar(3,4), passar(3,5).\\
&\leftarrow passar(3,5), passar(3,4).\\
&\leftarrow \textbf{not}\ caminhoPara(6).\\
caminhoPara(1).&\\
caminhoPara(6) &\leftarrow caminhoPara(5), passar(5,6).\\
&\vdots\\
caminhoPara(2) &\leftarrow caminhoPara(1), passar(1,2).\\
passar(5,6) &\leftarrow caminhoPara(5), \textbf{not}\ naoPassar(5,6).\\
&\vdots\\
passar(1,2) &\leftarrow caminhoPara(1), \textbf{not}\ naoPassar(1,2).\\
naoPassar(5,6) &\leftarrow \textbf{not}\ passar(5,6).\\
&\vdots\\
naoPassar(1,2) &\leftarrow \textbf{not}\ passar(1,2).\\
vertice(1).&\\
&\vdots\\
vertice(6).&\\
aresta(1,2).&\\
&\vdots\\
aresta(5,6).&
\end{aligned}
$$

Tomando $I_1 = \{passar(1,3), passar(3,5), passar(5,6), naoPassar(1,2), naoPassar(2,4), naoPassar(3,4),$

$naoPassar(4,5)$, $caminho(1)$, $caminho(3)$, $caminho(5)$, $caminho(6)\} \cup P_G$, a redução do programa P, P^{I_1} é:

$$
\begin{aligned}
& \leftarrow passar(1,2), passar(1,3).\\
& \leftarrow passar(1,3), passar(1,2).\\
& \leftarrow passar(3,4), passar(3,5).\\
& \leftarrow passar(3,5), passar(3,4).\\
caminhoPara(1).& \\
caminhoPara(6) & \leftarrow caminhoPara(5), passar(5,6).\\
& \vdots\\
caminhoPara(2) & \leftarrow caminhoPara(1), passar(1,2).\\
passar(1,3) & \leftarrow caminhoPara(1)\\
passar(3,5) & \leftarrow caminhoPara(3)\\
passar(5,6) & \leftarrow caminhoPara(5)\\
naoPassar(1,2).& \\
naoPassar(2,4).& \\
naoPassar(3,4).& \\
naoPassar(4,5).& \\
vertice(1).& \\
& \vdots\\
vertice(6).& \\
aresta(1,2).& \\
& \vdots\\
aresta(5,6).&
\end{aligned}
$$

cujo modelo mínimo é exatamente I_1. Portanto, I_1 é modelo-resposta de P.

A semântica das regras de peso é um pouco diferente. Ela foi definida em Niemelä *et al.* (1999), juntamente com métodos para encontrar os modelos-resposta de um programa com estas regras.

Um **modelo de uma regra de peso**, equivalentemente a um modelo de um programa sem regras de peso, é uma interpretação que, para cada regra do programa, contém um literal I que está na cabeça da regra somente se todas as restrições do corpo da regra são satisfeitas.

Um modelo mínimo é definido de maneira equivalente, porém um modelo-resposta não pode ser definido da mesma maneira, pois os limites superiores foram descartados na redução.

Por isso, um modelo M é chamado modelo-resposta de um programa com regras de peso se e somente se, além de M ser o modelo mínimo de P_M, M satisfaz todas as regras de peso de P.

Exemplo 2.26. Considere o programa P formado pela regras do Exemplo 2.19:

$$
\begin{aligned}
1 \leq \{a=1, b=1, c=1\} \leq 2 \quad \leftarrow \quad & 2 \leq \{d=1, \textbf{not } b=1, \textbf{not } e=3\} \leq 4.\\
& 1 \leq \{d=3, e=2\} \leq 5.
\end{aligned}
$$

$M_1 = \{a, d, e\}$ é um modelo-resposta deste programa, pois a redução P^{M_1} é:

$$
\begin{aligned}
a \quad \leftarrow \quad & 1 \leq \{d=1\}.\\
& d.\\
& e.
\end{aligned}
$$

E o modelo mínimo de P^{M_1} é M_1 que, além disso, satisfaz todas as regras de P.

Por outro lado $M_2 = \{d, a\}$ não é modelo-resposta pois o corpo da primeira regra não é satisfeito o que faz com que a primeira regra seja eliminada e não se possa deduzir a.

Utilizando regras de peso, pode-se simplificar muito a escrita de programas. O programa do Exemplo 2.25 pode ser escrito apenas como:

$$\begin{aligned} 1 \leq \{passar(X,Y) = 1 \text{ para cada } aresta(X,Y)\} \leq 1 &\leftarrow vertice(X), caminhoPara(X). \\ caminhoPara(1). & \\ caminhoPara(Y) &\leftarrow caminhoPara(X), passar(X,Y), aresta(X,Y). \\ &\leftarrow \textbf{not } caminhoPara(n). \end{aligned}$$

Existe mais um resultado a apresentar que será útil mais tarde:

Teorema 2.27. *Dado um programa ASP S com os modelos-resposta Ψ, ao adicionarmos uma regra de restrição, o novo programa S' terá modelos-resposta $\Psi' \subseteq \Psi$.*

Demonstração. Seja M um modelo-resposta de S'. Mostraremos que M é também um modelo-resposta de S.

Digamos que a nova regra, r, seja da forma $\leftarrow L_1, \ldots, L_x, \textbf{not } L_{x+1}, \ldots, \textbf{not } L_y$. Considerando a transformação mostrada na Equação (2.1), podemos expressá-la da forma

$$h \leftarrow L_1, \ldots, L_x, \textbf{not } L_{x+1}, \ldots, \textbf{not } L_y, \textbf{not } h.$$

onde h é um literal novo, não presente antes em HB_S.

Sabemos que $h \notin M$ pois, caso contrário, M não seria modelo de S'. Portanto, na redução S'^M, "**not** h" é eliminado. Se algum elemento de $\{L_{x+1}, \ldots, L_y\}$ pertencer a M, então a regra r é eliminada de S'^M, tornando $S'^M = S^M$, o que faz com que M seja modelo-resposta de S.

Caso contrário, teremos $S'^M = S^M \cup \{h \leftarrow L_1, \ldots, L_x\}$. Porém, sabemos que algum literal de $\{L_1, \ldots, L_x\}$ não pertence a M, caso contrário teríamos que $h \in M$. Isso faz com que a regra r seja ignorada ao se tentar decidir se M é ou não modelo mínimo de S . Portanto M também é modelo-resposta de S.

Caso tenhamos uma restrição de peso, a prova é análoga. Digamos que a regra adicionada seja da forma $\leftarrow C_1, \ldots, C_x$ onde $C_1, \ldots, C_x$ sejam restrições de peso. Utilizando novamente a transformação da Equação (2.1), teremos $S'^M = S^M \cup \{h \leftarrow C_1, \ldots, C_x\}$ e, utilizando o mesmo argumento, sabemos que alguma restrição de $\{C_1, \ldots, C_x\}$ não é satisfeita pelos literais de M. Então, novamente, a regra r é ignorada e M também é modelo-resposta de S. □

Este teorema afirma que ao adicionar restrições a um programa ASP não estaremos adicionando novos modelos-resposta a ele, apenas retirando os que violam a restrição.

2.3.4 Complexidade

Em Marek e Truszczyński (1991) foi provado que o problema de decidir se um programa lógico na sintaxe *AnsProlog* possui modelos-resposta é um problema NP-Completo.

Isto significa que o ASP possui poder suficiente para representar qualquer problema de decisão presente na classe NP.

Além disso, em Ben-Eliyahu e Dechter (1992) é mostrado que se incluirmos regras de escolha ou regras de peso ao ASP, encontrar modelos-resposta se torna $\sum_2^P -completo$, permitindo a modelagem de problemas que não podem ser traduzidos para SAT em tempo polinomial, considerando que $P \neq NP$.

Apesar de, no pior caso, o tempo para resolver problemas NP-completos ser exponencial, na prática, muitas instâncias são tratáveis pelos resolvedores atuais (Zhao e Lin (2003)).

2.4 ASP e SAT

É fácil fazer um programa ASP que encontre a solução para um problema SAT. Dado um conjunto de clausas C, considere o programa formado pelas seguintes regras (retiradas de Marek e Truszczyński (1998)):

$$\begin{aligned} in(V) &\leftarrow \textbf{not } out(V). \\ out(V) &\leftarrow \textbf{not } in(V). \end{aligned}$$

e para cada clausa c da forma:

$$\neg a_1 \vee \cdots \vee \neg a_n \vee b_1 \vee \ldots \vee b_m$$

nós adicionamos a regra.

$$\leftarrow in(a_1), \ldots, in(a_n), out(b_1), \ldots, out(b_m).$$

Os átomos $in(a)$ no modelo-resposta deste programa coincidirão com uma valoração que torna o problema C satisfazível.

Porém, existem muitas dificuldades em se fazer o contrário, transformar um programa ASP em um problema SAT, apesar de isto ser muito interessante devido à grande quantidade de resolvedores SAT. Como vimos na seção 2.3.4, se o programa ASP contiver regras de escolha, isto é impossível.

Porém, mesmo que o programa não contenha regras de escolha e se restrinja à sintaxe $AnsProlog^{\neg,\bot}$, reduzir um programa ASP para uma fórmula SAT é bastante difícil, mesmo que o teorema de Cook-Levin (Cook (1971); Trakhtenbrot (1984)), que afirma que qualquer programa NP pode ser reduzido a uma instância SAT, garanta que é possível.

Consideremos as seguintes propriedades sobre funções de tradução $Tr : \mathcal{C} \to \mathcal{C}'$ que mapeiam um programa lógico finito em uma linguagem $\mathcal{C}$ em um outro programa lógico finito na linguagem $\mathcal{C}'$ (adaptado de Janhunen (2006)):

Definição 2.28. Dizemos que uma função de tradução $Tr : \mathcal{C} \to \mathcal{C}'$ é **fiel** se para todo $P \in \mathcal{C}$ houver uma bijeção entre as soluções de P e $Tr(P)$.

Definição 2.29. Uma função de tradução $Tr : \mathcal{C} \to \mathcal{C}'$ é dita **modular** se, dados programas P e $Q \in \mathcal{C}$, tal que $P \cap Q = \emptyset$, temos:

$$Tr(P \cup Q) = Tr(P) \cup Tr(Q)$$

Proposição 2.30 (Baseado em Niemelä (1999) e Janhunen (2006))**.** *É impossível obter uma tradução fiel e modular de um programa em $AnsProlog^{\neg,\bot}$ para um conjunto de clausas SAT.*

Demonstração. Vamos assumir que exista uma tradução Tr que seja fiel e modular.

Consideremos os programas ASP $P_1 = \{a \leftarrow \textbf{not } a.\}$ e $P_2 = \{a.\}$. O programa P_1 não possui modelos-resposta e P_2 possui o modelo-resposta $\{a\}$.

Como a tradução Tr é fiel, $Tr(P_1)$ deve ser inconsistente, ou seja, não deve possuir nenhuma interpretação que a satisfaça.

Considere agora o programa $P_1 \cup P_2$. Pela modularidade de Tr, $Tr(P_1 \cup P_2) = Tr(P_1) \cup Tr(P_2)$ e também é inconsistente. Porém, $P_1 \cup P_2$ possui um modelo-resposta, $\{a\}$, o que contradiz a condição de fidelidade de Tr. □

Isto significa que, ao traduzir um programa lógico para uma fórmula SAT, precisamos abrir mão ou da fidelidade ou da modularidade.

Além disso, em Lifschitz e Razborov (2006) foi apontado que muito provavelmente[4] a fórmula SAT resultante da tradução de certos programas ASP sempre terá tamanho exponencial a menos

[4] se $P \nsubseteq NC^1/poly$, o que se acredita ser verdade

que a função que relaciona os modelos SAT e os modelos-resposta do programa ASP não seja uma função identidade (em outras palavras, devem existir variáveis auxiliares para poder evitar um crescimento exponencial). Algo semelhante acontece com a transformação de uma fórmula booleana qualquer para uma fórmula na Forma Normal Conjuntiva.

Mas mesmo com essas limitações, existem diversos métodos para se encontrar modelos-resposta usando fórmulas SAT. Um dos primeiros métodos apareceu em Ben-Eliyahu e Dechter (1992), porém esta tradução introduz $O(n^2)$ variáveis e não é fiel: vários modelos poderiam mapear para um mesmo modelo-resposta. Posteriormente, o sistema ASSAT[5], discutido em Lin e Zhao (2002), mostrou-se bastante eficiente em encontrar modelos-resposta utilizando outro método de tradução entre programas lógicos e SAT. Este método não introduz novas variáveis, mas pode crescer a fórmula exponencialmente. Em Janhunen (2004), foi apresentado um método onde o tamanho da tradução é da ordem de $\|P\| \times \log_2 |HB_P|$ e que introduz $\log_2 |HB_P + 2|$ variáveis.

Para exemplificar, discutiremos a seguir brevemente estes dois últimos métodos de tradução.

2.4.1 Tradução via completação e fórmulas de loop

Este é o método utilizado pelo sistema ASSAT e introduz diversas propriedades interessantes do ASP. Primeiro precisamos definir dois conceitos auxiliares:

Definição 2.31. O **grafo de dependências positivas** de um programa P, que chamaremos G_P, é um grafo dirigido cujos vértices são os átomos de P e existe uma aresta entre dois vértices p e q, de p para q, se existir uma regra r em P tal que $p \in Head(r)$ e $q \in Body(r)$.

Intuitivamente, as arestas indicam de quais átomos um certo átomo depende para poder se tornar verdadeiro.

Definição 2.32. Um subconjunto não vazio L de átomos de P é chamado um **loop** se o subgrafo de G_L formado pelos vértices em L for fortemente conexo, ou seja, para cada par de nós $\{u, v\}$ existe um caminho dirigido de u para v.

Conjuntos com um único elemento a também são chamados **loops** se houver uma aresta de a para a.

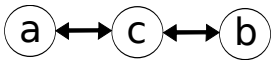

Figura 2.1: *Exemplo de grafo de dependências positivas com três loops*

Perceba que um loop não é a mesma coisa que um componente conexo. Por exemplo o grafo 2.1 possui um único componente conexo, porém possui três loops: $\{a, c\}$, $\{b, c\}$ e $\{a, b, c\}$.

Intuitivamente, um loop ocorre quando existe uma dependência mútua entre um conjunto de átomos, por exemplo "$a \leftarrow b. \quad b \leftarrow a.$".

Fages (1994) mostrou que se um programa não contém loops, então todo modelo da completação deste programa também é um modelo-resposta. Como a completação de um programa é um conjunto de clausas proposicionais, isto garante um caminho para traduzir um programa ASP em fórmulas SAT, mas apenas para uma classe de programas.

Em Lin e Zhao (2002) procura-se fortalecer este resultado adicionando-se regras que filtram os modelos indesejados da completação de um programa.

Dado um programa P e um loop L neste programa, gera-se dois conjuntos de regras:

$$\begin{aligned} R^+(L, P) &= \{r | r \in P, Head(r) \in L, (\exists q).q \in Body(r) \wedge q \in L\} \\ R^-(L, P) &= \{r | r \in P, Head(r) \in L, \neg(\exists q).q \in Body(r) \wedge q \in L\} \end{aligned}$$

[5]Disponível em http://www.cs.ust.hk/assat/

As regras em R^+ são as regras que estão "dentro do loop", tanto os literais da cabeça quanto os literais do corpo. As regras em R^- são as regras que estão "fora do loop", mas implicam elementos no loop.

Exemplo 2.33 (Lin e Zhao (2002))**.** Considere o programa:

$$\begin{array}{lll} a \leftarrow b. & b \leftarrow a. & a \leftarrow \textbf{not } c. \\ c \leftarrow d. & d \leftarrow c. & c \leftarrow \textbf{not } a. \end{array}$$

Este programa contém dois loops: $L_1 = \{a, b\}$ e $L_2 = \{c, d\}$. Para estes loops temos:

$$\begin{array}{lll} R^+(L_1) = \{a \leftarrow b. & b \leftarrow a.\}, & R^-(L_1) = \{a \leftarrow \textbf{not } c.\} \\ R^+(L_2) = \{c \leftarrow d. & d \leftarrow c.\}, & R^-(L_1) = \{c \leftarrow \textbf{not } a.\} \end{array}$$

A diferença entre os modelos da completação de um programa e os modelos-resposta deste programa no caso do programa conter loops é que a completação do programa considera como possível modelo o caso em que os elementos do loop "suportam" uns aos outros, mesmo que nenhum elemento de fora do loop o faça. Isto não ocorre na semântica do modelo-resposta.

Por isso, em Lin e Zhao (2002) se definiu uma "fórmula de loop" que, intuitivamente, força os elementos de um loop a serem verdadeiros apenas no caso em que regras de fora de um loop o fazem.

Definição 2.34. Seja P um programa lógico, L um loop neste programa e $R^-(L) = \{r_1, \ldots, r_n\}$, a **fórmula de loop** de L, denotado $LF(L)$, é a implicação:

$$\neg \left(\bigvee_{i=1}^{n} Body(r_i) \right) \rightarrow \bigwedge_{i=1}^{n} \neg Head(r_i)$$

Teorema 2.35 (Teorema 1 de Lin e Zhao (2002))**.** *Seja P um programa lógico, $Comp(P)$ a completação de P e LF o conjunto de todas as fórmulas de loop de P, temos que um conjunto é um modelo-resposta de P se e somente se este conjunto é um modelo de $Comp(P) \cup LF$.*

Exemplo 2.36. Considere o programa do exemplo 2.33. A sua completação é:

$$\begin{array}{lll} a & \equiv & \neg c \vee b \\ b & \equiv & a \\ c & \equiv & \neg a \vee d \\ d & \equiv & c \end{array}$$

Os modelos da sua completação são: $\{a, b\}$, $\{c, d\}$ e $\{a, b, c, d\}$, dos quais apenas os dois primeiros são modelos-resposta. Porém, se adicionarmos as regras $LF(L_1) = \{c \rightarrow (\neg a \wedge \neg b)\}$ e $LF(L_2) = \{a \rightarrow (\neg c \wedge \neg d)\}$, o último modelo é eliminado e os modelos restantes são exatamente os modelos-resposta procurados.

Um corolário deste teorema é que a tradução por fórmulas de loop é uma tradução fiel (Definição 2.28), pois existe uma bijeção trivial entre os modelos-resposta de P e os modelos de $Comp(P) \cup LF$.

Consequentemente, podemos concluir que esta tradução não é modular (Definição 2.29). De fato, ao adicionar novas regras podemos modificar os loops e suas fórmulas.

Um problema desta técnica é que é possível que tenhamos um número exponencial de fórmulas de loop. Para contornar este problema o sistema ASSAT adiciona apenas algumas fórmulas de loop e chama um resolvedor SAT para encontrar uma resposta. Caso a resposta encontrada não seja um modelo-resposta, adicionam-se mais fórmulas de loops iterativamente até obter alguma resposta.

2.4.2 Tradução via *level numbering*

Em Janhunen (2004) foi apresentado um algoritmo fiel (e, portanto, não modular) para traduzir programas ASP para clausas SAT em que o tamanho da fórmula SAT gerada pela tradução do programa ASP é subquadrático, da ordem de $\|P\| \times \log_2 |HB_P|$, e introduz um número de literais da orderm de $\log_2 |HB_P + 2|$.

Esta tradução utiliza uma caracterização diferente, porém equivalente, de modelos-resposta. Um dos elementos desta caracterização é o conceito de **modelos suportados**.

Definição 2.37. Um modelo M de um programa P é um modelo suportado (*supported model*) deste programa se e somente se para cada literal $a \in M$ existe uma regra $r \in P$ cuja cabeça é a e cujo corpo é satisfeito por M.

Corolário 2.38. *Todo modelo-resposta é um modelo suportado.*

A recíproca, porém, não é sempre verdadeira, como pode-se ver no exemplo a seguir.

Exemplo 2.39. Considere o programa

$$a \leftarrow b.$$
$$b \leftarrow a.$$

Este programa possui apenas o modelo-resposta $\emptyset$, porém, seus modelos suportados são $\emptyset$ e $\{a, b\}$.

Além do conceito de modelo suportado, também é necessário apresentar o conceito de ***level numbering*** para caracterizar modelos-resposta.

Definição 2.40. Seja M um modelo suportado de um programa P. A função $\# : M \cup P \to \mathbb{N}$ é um *level numbering* com respeito a M se para todo literal $a \in M$:

$$\#(a) = \min\{\#(r) | r \in P \text{ e } a = Head(r)\}$$

e para toda regra r de P:

$$\#(r) = \max\{\#(b) | b \in Body^+(r)\} + 1$$

Consideramos $\max(\emptyset) = 0$ para cobrir as regras sem corpo positivo.

Intuitivamente, um *level numbering* captura a ordem em que os literais e as regras de um programa positivo são "descobertos" se, partindo de um conjunto vazio M, adicionarmos as cabeças das regras satisfeitas por M até que não haja mais nada para adicionar.

Finalmente podemos definir a nova caracterização de modelo-resposta.

Teorema 2.41 (Teorema 4 de Janhunen (2004))**.** *Seja P um programa, M é um modelo-resposta de P se e somente se M é um modelo suportado de P e existe um* level numbering *único com respeito a M.*

Com o teorema 2.41 podemos finalmente definir uma tradução. A tradução utiliza dois passos: no primeiro passo, transforma-se um programa ASP qualquer em um programa onde todos os literais do corpo de uma regra estão negados pela negação como falha, ou seja, em um programa sem literais positivos. Feito isso, o segundo passo é transformar o programa em uma fórmula SAT usando a completação. Como nenhuma regra tem literais positivos, o grafo de dependências positivas não possuirá nenhuma aresta, o que faz com que não haja loops e, portanto, a completação será suficiente para capturar modelos-resposta.

Para tirar os literais positivos das regras, para cada literal b_i que aparece positivamente em uma regra $a \leftarrow b_1, \ldots, b_n, \textbf{not } c_1, \ldots, \textbf{not } c_m$, define-se o literal $\bar{b_i}$ e adicionam-se na tradução as regras

$$\bar{b_i} \leftarrow \textbf{not } b_i.$$
$$a \leftarrow \textbf{not } \bar{b_1}, \ldots, \textbf{not } \bar{b_n}, \textbf{not } c_1, \ldots, c_m$$

Utilizando apenas estas duas regras, conseguiríamos obter modelos suportados do programa original, mas não modelos-resposta. Para conseguir modelos-resposta precisamos também garantir a existência de um *level numbering*. Utilizamos uma codificação binária para os números de *level numbering*. Como no pior caso cada literal pode ter um *level number* diferente, precisamos de $j = \lceil \log_2(|HB_P| + 2) \rceil$ variáveis para representar um único *level number*. Este número de bits é um motivo para a não-modularidade deste método.

Para cada literal a, definimos $a_1, \ldots, a_j$ novos literais além do seu complemento $\bar{a_1}, \ldots, \bar{a_j}$. Estes literais serão usados para representar o *level number* do literal a.

Como mostrado em Janhunen (2004), é possível expressar operações aritméticas básicas com regras ASP sem literais positivos. Desta maneira pode-se escrever a restrição de que os valores dos literais a_i e $\bar{a_i}$ descrevam um *level numbering*, filtrando assim modelos suportados que não são modelos-resposta.

Capítulo 3

O problema da satisfazibilidade probabilística

Neste capítulo faremos uma breve revisão de algumas propriedades relevantes do problema da satisfazibilidade probabilística (PSAT), que inspirou o PASP.

O problema PSAT é bastante antigo, foi estudado inicialmente por Boole (1854) e redescoberto por diversos pesquisadores.

O problema da satisfazibilidade probabilística é o problema de decidir se há ou não consistência entre um conjunto de probabilidades associadas a fórmulas lógicas. Por isso vamos primeiro rever algumas definições da lógica proposicional.

3.1 Lógica proposicional clássica

A linguagem da lógica proposicional clássica é formada por um conjunto enumerável de símbolos representando variáveis booleanas combinadas pelos conectivos lógicos binários $\wedge$, $\vee$ e $\rightarrow$ e pelo conectivo unário $\neg$, possível com parênteses para esclarecer a ordem das operações.

Uma **fórmula bem formada** da lógica proposicional, ou simplesmente fórmula, neste alfabeto é definida pelas seguintes regras:

- Toda variável é uma fórmula;
- Seja α uma fórmula, $\neg\alpha$ também é uma fórmula;
- Sejam α e β fórmulas, $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$ e $(\alpha \rightarrow \beta)$ também o são;
- Nada além destas regras é uma fórmula.

Por conveniência podemos definir o conectivo $\leftrightarrow$ de forma que $\alpha \leftrightarrow \beta$ é apenas uma abreviação para $(\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$.

As variáveis assumem um valor verdade, verdadeiro ou falso, representados como 1 e 0 respectivamente. Uma **valoração** é uma associação de uma valor verdade para cada variável presente em uma fórmula. Dado uma valoração v para as variáveis de uma fórmula, podemos definir a valoração de uma fórmula considerando a semântica dos operadores lógicos.

- $v(\alpha \wedge \beta) = 1$ se, e somente se, $v(\alpha) = 1$ e $v(\beta) = 1$;
- $v(\alpha \vee \beta) = 1$ se, e somente se, $v(\alpha) = 1$ ou $v(\beta) = 1$;
- $v(\alpha \rightarrow \beta) = 1$ se, e somente se, $v(\alpha) = 0$ ou $v(\alpha) = v(\beta) = 1$;
- $v(\neg\alpha) = 1$ se, e somente se, $v(\alpha) = 0$.

Chamamos de **literal** uma variável (v) ou sua negação ($\neg v$). Uma disjunção ($\vee$) de literais é chamada de **clausa**. É dito que uma fórmula está na **Forma Normal Conjuntiva** (**CNF**) se ela é composta de uma conjunção de clausas. De maneira semelhante, uma fórmula está na **Forma Normal Disjuntiva** (**DNF**) se ela é composta de uma disjunção de conjunções de literais.

Duas fórmulas α e β são equivalentes se $v(\alpha) = v(\beta)$ para toda valoração. Para toda fórmula bem formada da lógica proposicional existe uma fórmula equivalente na Forma Normal Conjuntiva. Por isto a CNF normalmente é usada como formato padrão para expressar fórmulas da lógica proposicional.

Como a transformação de uma fórmula para o formato CNF pode, eventualmente, levar a um crescimento exponencial, normalmente ao transformar um fórmula para o formato CNF usa-se um algoritmo que introduz novas variáveis e cria novas valorações que tornam a fórmula falsa. Porém garante-se que as valorações que tornam a nova fórmula verdadeira possuem um equivalente que também tornam a fórmula original verdadeira. Este algoritmo é polinomial no tempo e no espaço e pode ser visto em da Silva *et al.* (2006).

3.2 O problema PSAT

Uma instância de um problema PSAT é dada por um conjunto de fórmulas da lógica proposicional clássica sobre um conjunto de n variáveis, $S = \{s_1, \ldots, s_k\}$, e um conjunto de probabilidades $P = \{p_1, \ldots, p_k\}$ com $0 \leq p_i \leq 1$ para todo i.

Uma **distribuição de probabilidades discreta** sobre um conjunto enumerável X é uma função que associa a cada elemento deste conjunto uma probabilidade (a chance de que uma variável assuma o valor deste elemento) de modo que a soma de todas as probabilidades seja um.

$$\sum_{x \in X} p(x) = 1$$

Dado uma distribuição de probabilidades sobre todas as possíveis valorações das fórmulas do conjunto S, podemos definir a probabilidade de uma fórmula. A probabilidade de uma fórmula é a soma da probabilidade das valorações onde esta fórmula é verdadeira. Ou seja, se π for uma distribuição de probabilidade sobre o conjunto de valorações V_S, a probabilidade da fórmula s é dado por:

$$p(s) = \sum_{v \in V_S} \{\pi(v) | v(s) = 1\} \tag{3.1}$$

Intuitivamente, a probabilidade de uma fórmula é a probabilidade de todos os "mundos possíveis" onde esta fórmula é verdadeira.

Dizemos que esta instância do problema PSAT é satisfazível se a associação $p(s_i) = p_i$ é consistente.

Definição 3.1. Seja $V_S = \{v_1, \ldots, v_{2^n}\}$ o conjunto de valorações possíveis sobre as variáveis de S, dizemos que a associação de probabilidades $p(s_i) = p_i$ é **consistente** se existir uma distribuição de probabilidades sobre V_S, chamada π, que satisfaça a equação 3.1

Exemplo 3.2. Considere o problema PSAT:

$$P(a \vee b \vee c) = 1$$
$$P(a \wedge b) = 0,61; P(a \wedge c) = 0,60; P(b \wedge c) = 0,59$$

Considere as valorações $v_1 = \{a = b = c = 1\}$, $v_2 = \{a = b = 1; c = 0\}$, $v_3 = \{a = c = 1; b = 0\}$ e $v_4 = \{a = 0; b = c = 1\}$.

A distribuição de probabilidades π que associa os seguintes valores a estas valorações: $\pi(v_1) = 0,4; \pi(v_2) = 0,21; \pi(v_3) = 0,2; \pi(v_4) = 0,19$ é uma solução para este problema PSAT, como podemos ver substituindo estes valores nas equações abaixo, equivalentes a equação 3.1 para cada

fórmula:

$$\begin{aligned}
\pi(v_1) + \pi(v_2) + \pi(v_3) + \pi(v_4) &= 1 && (3.2)\\
\pi(v_1) + \pi(v_2) &= 0,61 && (3.3)\\
\pi(v_1) + \pi(v_3) &= 0,6 && (3.4)\\
\pi(v_1) + \pi(v_4) &= 0,59 && (3.5)
\end{aligned}$$

Expressando estas equações na forma matricial, temos:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0,4 \\ 0,21 \\ 0,2 \\ 0,19 \end{bmatrix} = \begin{bmatrix} 1 \\ 0,61 \\ 0,60 \\ 0,59 \end{bmatrix} \tag{3.6}$$

O problema PSAT pode ser expresso como um problema de programação linear. Definimos a matriz $A_{k\times 2^n} = [a_{ij}]$, tal que $a_{ij} = 1$ se a valoração j satisfaz a fórmula i, caso contrário $a_{ij} = 0$, e definimos também o vetor $p_{k\times 1} = [p_i]$. Uma instância PSAT é satisfazível se há um vetor π que satisfaz:

$$\begin{aligned}
A\pi &= p \\
\pi &\geq 0 \\
\sum \pi_i &= 1
\end{aligned} \tag{3.7}$$

A última restrição pode ser omitida adicionando a A e a p uma linha inteira de uns.

Por causa do Lema de Carathéodory (Prasolov e Tikhomirov (2001)), não precisamos de uma matriz A com tantas colunas, pois este lema garante que se este problema tem solução, ele tem uma solução com apenas $k+1$ elementos de π diferentes de zero. Então, podemos eliminar os elementos com zero da matriz e reduzir a matriz A a uma matriz $k+1 \times k+1$ (a matriz básica), tornando o problema PSAT um problema NP-completo.

Na forma de problema de programação linear o Exemplo 3.2 é expresso como a equação matricial (3.6).

O trabalho de Bona (2011) apresentou a Forma Normal Atômica à qual todas as instâncias PSAT podem ser convertidas em tempo polinomial. Uma instância PSAT está na Forma Normal Atômica se pode ser particionada em dois conjuntos (Γ, Ψ) onde Γ é um conjunto de fórmulas proposicionais com probabilidade associada 1 e Ψ possui probabilidades associadas apenas a átomos.

Esta transformação de uma instância PSAT qualquer em uma instância na Forma Normal Atômica consiste basicamente em criar k novas variáveis $y_1, \ldots, y_k$ e então atribuir $\Gamma = \{p(y_i \leftrightarrow s_i) = 1 | 1 \leq i \leq k\}$ e $\Psi = \{p(y_i) = p_i | 1 \leq i \leq k\}$. Claramente, se o programa original for consistente, este novo também será.

Esta forma separa o problema PSAT em dois problemas: um problema SAT e um problema de encontrar uma distribuição de probabilidades compatível e facilita o tratamento do problema, permitindo a criação de um formato de entrada padrão para um resolvedor PSAT.

Exemplo 3.3. O problema do exemplo 3.2 pode ser escrito na Forma Normal Atômica, levando Γ para a forma CNF, como:

$$\Gamma = \left\{ \begin{array}{cccc} \neg x \vee a & \neg y \vee a & \neg z \vee b & a \vee b \vee c \\ \neg x \vee b & \neg y \vee c & \neg z \vee c & \\ \neg a \vee \neg b \vee x & \neg a \vee \neg c \vee y & \neg b \vee \neg c \vee z & \end{array} \right\}$$

$$\Psi = \{P(x) = 0.61; P(y) = 0.60; P(z) = 0.59\}$$

Como as fórmulas de Γ devem ter probabilidade 1 e a soma de uma distribuição de probabilidade é 1, a distribuição de probabilidades só pode associar valores maiores que 0 a valorações que satisfazem Γ.

Então se expressarmos em forma matricial o equivalente às equações (3.2)-(3.5) do problema na Forma Normal, veremos que obrigatoriamente a primeira linha da matriz é formada por "uns" pois, sejam quais forem as valorações escolhidas na distribuição, elas devem satisfazer as fórmulas de Γ.

Se, e somente se, na i-ésima valoração com probabilidade não nula temos $v_i(x) = 1$, então na segunda linha, i-ésima coluna, teremos o valor 1. Isto se repete para todas as variáveis com probabilidade, isto é, para todas as linhas da matriz. Por causa disso, as colunas da matriz gerada, a menos da primeira linha, representam o valor verdade que uma valoração associa às variáveis com probabilidade. Se uma coluna da matriz possui os valores $(1, 0, 1, 0)$, isto significa que em uma das valorações, $v(x) = 0$, $v(y) = 1$ e $v(z) = 0$. Esta correspondência entre colunas da matriz e valorações sempre acontece em instâncias na Forma Normal Atômica.

As valorações v_1, v_2, v_3 e v_4 do Exemplo 3.2 só podem satisfazer Γ se tivermos $v_1 = \{x = y = z = 1\}$, $v_2 = \{a = 1; y = z = 0\}$, $v_3 = \{y = 1; x = z = 0\}$ e $v_4 = \{z = 1; x = y = 1\}$. Considerando essas valorações, representando o problema na Forma Normal Atômica como uma série de equações na forma matricial, teremos exatamente a Equação matricial (3.6).

Capítulo 4

Answer Set Programming Probabilístico

Neste trabalho estenderemos o Answer Set Programming com informações sobre probabilidades. Chamaremos esta extensão de Answer Set Programming Probabilístico, ou PASP.

A motivação para esta extensão é a análise da consistência entre uma *teoria*, expressa por meio de um programa ASP, e um conjunto de *dados*, expressos por um conjunto de probabilidades através de uma interpretação frequentista.

4.1 Definição do PASP

Analogamente ao problema PSAT, o problema PASP é um problema de decisão onde pergunta-se se um conjunto de probabilidades é *consistente* com as regras de um programa.

Portanto, a entrada de um problema PASP é dada por um programa ASP, S, instanciado e cuja negação clássica tenha sido eliminida de acordo com a Regra 2.2, por uma base de Herbrand $HB_s = \{a_1, \ldots, a_n\}$, e um conjunto de probabilidades[1] $P = \{p_1, \ldots, p_k\}$ onde cada elemento p_i deste conjunto está associado a um elemento de HB_S.

Uma instância PASP é consistente se o programa S e a associação $p(a_i) = p_i$ forem consistentes.

Definição 4.1. Seja $2^{HB_S} = \{v_1, \ldots, v_k\}$ o conjunto de todos os subconjuntos da base de Herbrand de S (ou seja, todos os candidatos à modelos-resposta de S).

Dizemos que o conjunto de probabilidades P é **consistente** com o programa S se existir uma distribuição de probabilidades, π, sobre 2^{HB_S} onde $p_i = \sum \{\pi(v_l) | a_i \in v_l \text{ e } v_l \text{ é modelo-resposta de } S\}$.

Como podemos ver, a definição de consistência do PASP é muito semelhante à consistência no PSAT.

Exemplo 4.2. Considere o programa abaixo, junto com uma descrição do grafo 1.1:

$$
\begin{aligned}
1 \leq \{passar(X,Y) = 1 \text{ para cada } aresta(X,Y)\} \leq 1 &\leftarrow vertice(X), caminhoPara(X).\\
caminhoPara(1).&\\
caminhoPara(Y) &\leftarrow caminhoPara(X), passar(X,Y), aresta(X,Y).\\
&\leftarrow \textbf{not } caminhoPara(n).
\end{aligned}
$$

e o conjunto de probabilidades associadas $p(passar(1,3)) = 0,5$ e $p(passar(3,4)) = 0.4$.

Excluindo-se literais auxiliares, como *caminhoPara* e *vertice*, os modelos-resposta do programa são $I_1 = \{passar(1,2), passar(2,4), passar(4,5), passar(5,6)\}$, $I_2 = \{passar(1,3), passar(3,5), passar(5,6)\}$ e $I_3 = \{passar(1,3), passar(3,4), passar(4,5), passar(5,6)\}$.

Este problema PASP é consistente, pois tomando um π tal que $\pi(I_1) = 0.5$, $\pi(I_2) = 0.1$ e $\pi(I_3) = 0.4$ e $\pi(v_i) = 0$ para todos os outros v_i, pode-se verificar que π satisfaz o critério de consistência.

[1] Nos restringiremos ao caso em que as probabilidades são números racionais e, portanto, computáveis.

Da mesma maneira que o PSAT, este problema pode ser visto do ponto de vista da programação linear, definindo uma matriz $A_{k\times 2^{\|HB_S\|}} = [a_{ij}]$, tal que $a_{ij} = 1$ se o subconjunto de átomos j contém o átomo i e é modelo-resposta de P, caso contrário $a_{ij} = 0$. Então temos os mesmos critérios para saber se uma instância PASP é consistente:

$$\begin{array}{rcl} A\pi & = & p \\ \pi & \geq & 0 \\ \sum \pi_i & = & 1 \end{array} \tag{4.1}$$

Novamente podemos eliminar o último critério adicionando uma nova linha de 1's.

Exemplo 4.3. O exemplo 4.2 na forma de um problema de programação linear, eliminando as muitas linhas e colunas onde π tem valor 0 (ou seja, usando a matriz básica), seria:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0.2 \\ 0.6 \\ 0.2 \end{bmatrix} = \begin{bmatrix} 1 \\ 0.2 \\ 0.6 \\ 0.8 \end{bmatrix}$$

Proposição 4.4. *Se existe uma solução para um PASP, existe uma solução com no máximo $k+1$ elementos de π diferentes de 0.*

O fato do problema PASP ser apresentado na forma de programação linear faz com que o Lema de Carathéodory seja válido para ele também, uma vez que o Lema de Carathéodory se aplica a qualquer problema de programação linear.

4.2 Complexidade Computacional

Podemos reduzir qualquer problema ASP em um problema PASP, basta adicionar um átomo novo ao programa, como fato, e associar a ele uma probabilidade 1. O problema PASP só terá solução se for possível encontrar um modelo-resposta ao qual se pode associar um valor de π igual a 1.

Isto significa que o problema PASP é pelo menos tão difícil quanto encontrar um modelo-resposta de um programa ASP, ou seja, $\sum_2^P$-difícil se estamos utilizando regras de escolha no programa ASP, NP-difícil, caso contrário.

Encontrar uma solução para o PASP em sua forma de programação linear é um problema basicamente de escolher os $k+1$ elementos que serão diferentes de zero. Em uma máquina de Turing não determinística, podemos fazer esta escolha não deterministicamente e encontrar a resposta esperada para o caso sem regras de escolha e regras de peso. Portanto, neste caso, o PASP com $AnsProlog^{\neg,\perp}$ é NP-completo.

O caso com regras de escolha e regras de peso é um pouco mais complicado, pois mesmo depois de escolher os elementos diferentes de zero, ainda temos o problema de testar os vários subconjuntos de átomos possíveis das regras de escolha.

Com uma Máquina de Turing Alternante (Chandra *et al.* (1981)), porém, é possível resolver o problema em tempo polinomial, alternando a máquina entre escolher um elementos presentes nas regras de escolha e de peso e alternando novamente para encontrar um modelo-resposta para o problema resultante. Isto faz com que o PASP com regras de escolha seja $\sum_2^P$-completo.

4.3 Comparação com a literatura

Na literatura pode-se encontrar outros trabalhos que misturam ASP com probabilidades. Nesta seção iremos comparar o PASP com o *P-log*, de Baral *et al.*.

Introduzido em Baral *et al.* (2004) e aperfeiçoado em Baral *et al.* (2009), **P-log** é uma extensão do ASP com primitivas para fazer inferências sobre informações de probabilidade.

P-log foi criado como uma linguagem para representar conhecimento envolvendo lógica e probabilidade. P-log estende o ASP adicionando regras da forma

$$random \quad a(t) : B \tag{4.2}$$

$$pr(a(t) = y|_c B) = v \tag{4.3}$$

onde $a(t)$ é um atributo aleatório e B é um conjunto de literais. A expressão (4.2) indica que, dado B, $a(t)$ é escolhido aleatoriamente e a expressão (4.3) é uma **probabilidade causal** que pode ser interpretada como "fatores determinados por B fazem com que $a(t)$ seja igual a y com probabilidade v". Implicitamente é assumido que dado a causa B, o efeito $a(t) = y$ é probabilisticamente independente de todos os fatores, exceto efeitos de $a(t) = y$.

O P-log também representa **observações** e **ações** que informam, respectivamente, o resultado assumido por um atributo aleatório em um certo instante e ações não aleatórias que modificam atributos aleatórios.

Também existe o conceito de consistência para um programa P-log, porém ele é definido de maneira a garantir que não existam literais contraditórios e que as regras de probabilidade e as hipóteses de independência sejam cumpridas.

Como podemos ver, P-log e PASP têm enfoques bastante diferentes. P-log foi feito para representar conhecimento e como mecanismo de inferência de agentes inteligentes e sobre redes Bayesianas, enquanto o PASP procura consistência entre um conjunto de regras e um conjunto de dados probabilísticos.

Outro ponto de grande divergência é o modo como estes sistemas tratam a independência probabilística. Todas as probabilidades do P-log são probabilidades causais e, portanto, têm como hipótese a independência de todos os outros fatores, dado a sua causa. Por outro lado, o PASP não possui nenhuma hipótese de independência e a dependência ou independência de variáveis não é levada em conta.

P-log é, pelo menos, $\sum_2^P$-difícil, pois um programa ASP com extensões pode ser trivialmente reduzido a P-log. Isto significa que é possível reduzir uma instância PASP a uma instância P-log. Porém, esta redução não é trivial. Os dois sistemas têm estruturas bastante diferentes.

Apesar de ser possível reduzir um programa $AnsProlog^{\neg,\perp}$ para SAT e vice-versa, isto não significa que seja prático em todas as aplicações substituir um pelo outro. O mesmo ocorre entre o P-log e o PASP. São ferramentas diferentes para tarefas diferentes.

Capítulo 5

Métodos de resolução do PASP

Neste capítulo estudaremos alguns métodos que podem ser usados para se resolver uma instância PASP, ou seja, decidir se um conjunto de probabilidades P é consistente com um programa S. Quando a instância PASP for consistente, gostaríamos ainda de obter a distribuição de probabilidades π que satisfaz os critérios de consistência.

A seguir estão dois possíveis métodos de resolução.

5.1 Solução via programação linear

Este método é semelhante ao método apresentado em de Bona (2011). Porém, ele foi modificado para levar em conta peculiaridades do PASP, como a não monotonicidade.

Como mostrado na Seção 4.1, uma instância PASP pode ser descrita como um problema de programação linear sem função de custo. O problema de encontrar uma distribuição de probabilidades que torne a instância PASP consistente se reduz então para o problema de encontrar uma solução básica viável, que é um problema de programação linear.

Podemos então considerar o uso do problema auxiliar utilizado no algoritmo simplex de duas fases visto na Seção A.2.1. Para isso precisamos introduzir $n+1$ variáveis artificiais ψ, onde n é o tamanho do vetor de probabilidades P. Perceba que essas variáveis não são variáveis no sentido do ASP. São variáveis do ponto de vista do problema de programação linear. No nosso caso, estamos falando da distribuição de probabilidades π que terá sua dimensão aumentada.

O uso dessas variáveis artificiais pode ser visto como o relaxamento do problema PASP. No problema original procurávamos uma distribuição de probabilidades sobre modelos-resposta. Estas variáveis extras podem ser vistas como a associação de probabilidade a conjuntos de literais que não necessariamente são modelos-resposta. Por isso consideraremos como função de custo $\sum \psi_i$, pois saberemos que só temos probabilidades não nulas associadas a modelos-resposta quando a função de custo tiver valor 0.

No algoritmo simplex de duas fases é utilizada a matriz identidade como matriz básica. Apesar de ser possível utilizar a matriz identidade, ela não é a melhor escolha. Lembre-se que, devido a restrição que $\sum \pi_i = 1$, a primeira linha da matriz básica deve conter apenas uns. Por isso a matriz (5.1) pode ser uma matriz básica melhor.

$$I^* = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & \dots & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & \dots & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & \dots & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & \dots & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & \dots & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & \dots & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & \dots & 1 \end{bmatrix} \tag{5.1}$$

Teorema 5.1. *Se o vetor de probabilidades P estiver ordenado em ordem decrescente, o sistema $I^*\pi = P$ sempre tem solução não-negativa.*

Demonstração. Seja $\|P\| = k$, considerando a última linha da matriz I^*, concluímos que $\pi_k = P_k$ e portanto um valor para π_k foi encontrado.

Vamos assumir que as variáveis $\pi_{n+1} \dots \pi_k$ possuem valores. Portanto devemos fazer $\pi_n = P_n - \sum_{i=n+1}^{k} \pi_i$. Como P está em ordem decrescente, $\pi_n > 0$ e assim teremos um valor para π_n.

Pelo princípio da indução, teremos uma solução para π. □

Uma prova alternativa consiste em simplesmente apontar que a matriz $[I^*|\pi]$ está na forma escalonada reduzida (Weisstein (2012)), o que faz com que o sistema sempre tenha solução.

A restrição de que o vetor de probabilidades P esteja ordenado não é nenhum problema, pois pode-se mudar a ordem que as variáveis aparecem no problema sem nenhuma perda, desde que se ordene as variáveis nos modelos-resposta corretamente.

Agora podemos inicializar o algoritmo simplex de duas fases. Porém, precisamos de um jeito de escolher colunas para entrar na base e assim minimizar a função $\sum \psi_i$.

A coluna escolhida deve representar um modelo-resposta do programa S e deve possuir um custo reduzido negativo. Uma possibilidade é simplesmente utilizar um resolvedor ASP, obter todos os modelos-resposta e em cada iteração calcular o custo reduzido de cada. É possível que exista um número exponencial de modelos-resposta, porém em muitos problemas práticos este número é bem menor.

Na subseção 5.1 a seguir, vemos um método alternativo para obter colunas com custo reduzido negativo.

O algoritmo 1 demonstra este método de resolução assumindo a existência da função *selecionaColuna* que devolve uma coluna com custo reduzido negativo ou "$\emptyset$" caso não exista nenhuma.

Entrada: Um programa ASP S, um conjunto de literais de S, l, e um vetor de k probabilidades p associadas aos elementos de l

Saída: Uma matriz B e um vetor π_B tal que $B\pi_B = [1|p]'$ se a instância PASP for consistênte, "INCONSISTENTE" caso contrário

1 $B \leftarrow I^*_{(k+1)\times(k+1)}$ /* Base Inicial */
2 $c_B \leftarrow [1 \dots 1]_{k+1}$
3 **para** $j = 1$ *até* k **faça**
4 **se** *Consistente*(B_j, S) **então**
5 $c_{B_j} \leftarrow 0$
6 **fim**
7 **fim**
8 $\pi_B \leftarrow B^{-1}p$ /* Solução básica viável inicial */
9 **enquanto** $c'_B\pi_b > 0$ **faça**
10 $A \leftarrow selecionaColuna(S, l, B, c_B)$
11 **se** $A = \emptyset$ **então**
12 **retorna** *INCONSISTENTE*
13 **fim**
14 $TrocaBase(B, A, \pi, c_B)$
15 $\pi \leftarrow B^{-1}p$ /* Nova solução básica viável inicial */
16 **fim**
17 **retorna** B, π_B

Algoritmo 1: Método de resolução do PASP por programação linear

A função $TrocaBase$ na linha 1 substitui alguma coluna de B por A e atualiza c_B. *Consistente* na linha 1 verifica se a coluna B_J representa um modelo-resposta de S.

Para implementar a função *Consistente* temos uma dificuldade: nem todos os literais necessariamente estão associados a uma probabilidade. Então se quisermos, por exemplo, verificar se um programa S com n variáveis é consistente com uma coluna j de k variáveis, não sabemos a

valoração de $n-k$ variáveis e portanto não podemos simplesmente verificar se esta valoração é um modelo-resposta de S.

O que precisamos é encontrar um modelo-resposta de S onde as variáveis com probabilidade tenham o mesmo valor que o apresentado na coluna da matriz. Para isso, é preciso utilizar um resolvedor ASP. Devido ao Teorema 2.27, a melhor maneira é adicionar a série de restrições

$$\begin{array}{ll} \leftarrow & L_1 \\ \vdots & \\ \leftarrow & L_m \\ \leftarrow & \textbf{not } L_{m+1} \\ \vdots & \\ \leftarrow & \textbf{not } L_k \end{array}$$

Onde $L_1, \ldots, L_m$ são os literais que não devem aparecer no modelo-resposta, ou seja, os literais que são 0 em B_j e $L_{m+1}, \ldots, L_k$ são os literais que devem aparecer no modelo-resposta, ou seja, os literais que são 1 em B_j.

Adicionadas essas restrições, verificamos se existem modelos-resposta que satisfazem o programa resultante. Isso vai significar que as variáveis associadas a probabilidades têm a valoração necessária: se uma variável L_i tem valor 0 na coluna, restringimos modelo-resposta onde este literal aparece. Se uma variável tem valor 1, restringimos os modelo-resposta onde este literal não aparece.

Exemplo 5.2. Considere o seguinte programa ASP:

$$\begin{array}{rcl} 0 \leq \{a = 1\} \leq 1. & & \\ b, c & \leftarrow & a. \\ b & \leftarrow & \textbf{not } a. \\ c & \leftarrow & \textbf{not } a. \end{array}$$

Associamos as probabilidades $p(b) = 0.7$ e $p(c) = 0.4$.

Vamos resolvê-lo utilizando este método. Primeiro, vamos inicializar a matriz básica com a matriz

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

Como b e c possuem probabilidades, a segunda linha representa b e a terceira representa c. Percebemos que a segunda coluna e a terceira coluna são consistentes com o programa, pois o programa possui modelos-resposta onde b ocorre e c não ocorre ($\{a, b\}$) e onde ambos ocorrem ($\{b, c\}$).

O vetor de custo c_B fica com o valor $[1, 0, 0]$. Calculamos π e temos o sistema:

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 0.3 \\ 0.3 \\ 0.4 \end{bmatrix} = \begin{bmatrix} 1 \\ 0.7 \\ 0.4 \end{bmatrix}$$

que possui custo total 0.3.

Vamos então utilizar a função $SelecionaColuna$, cuja implementação veremos adiante. Digamos que ela retorne o modelo-resposta $\{c, a\}$. A função $TrocaBase$ trocaria a primeira coluna por uma coluna representando este modelo-resposta, e teríamos:

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 0.3 \\ 0.6 \\ 0.1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0.7 \\ 0.4 \end{bmatrix}$$

que possui custo total 0. Portanto o nosso problema está resolvido, neste caso com apenas uma iteração.

SelecionaColuna pode ser escrito simplesmente consultando o custo reduzido de todos os modelos-resposta de S, mas veremos a seguir uma maneira mais eficiente.

5.1.1 Geração de colunas

Para obter colunas com custo reduzido negativo, podemos utilizar uma **redução de Turing** (Rogers (1967)) a ASP. Uma redução de Turing é um algoritmo que resolve um problema utilizando chamadas a um algoritmo que resolve outro problema (oráculo). Este método é baseado no método encontrado em Finger e De Bona (2010) para resolver problemas PSAT.

Queremos obter um modelo-resposta com custo reduzido (Definição A.6) negativo para tirar uma das variáveis auxiliares da base, isto é, queremos um modelo-resposta j tal que $\bar{c}_j = c_j - c'_B B^{-1} A_j < 0$. Como $c_j = 0$ para qualquer modelo-resposta, precisamos apenas que

$$c'_B B^{-1} A_j > 0 \tag{5.2}$$

para este modelo-resposta.

Para obter este modelo-resposta vamos gerar um novo programa ASP S' cujos modelos-resposta também sejam modelo-resposta de S e que além disso, tenha custo reduzido negativo. Uma dificuldade extra na geração desses modelos-resposta é o fato do ASP ser não monotônico. No caso geral, não podemos garantir que ao incluir novas regras em um programa S não criaremos novos modelos-resposta que não são modelos-resposta de S.

Para podermos garantir que todos os modelos-resposta de S' sejam modelos-resposta de S, usaremos o Teorema 2.27. Devido ao Teorema 2.27 devemos usar apenas restrições para garantir que o custo reduzido seja negativo.

Existem duas maneiras de gerar essas restrições: utilizando regras de peso ou expressando a desigualdade (5.2) como fórmulas SAT.

Regras de peso

Se chamarmos de u o vetor $c'_B B^{-1}$ e de $l_1, \ldots, l_k$ o valor dos literais de S associados a probabilidades, então a desigualdade $c'_B B^{-1} A_j > 0$ pode ser expressa como:

$$u_0 + \sum_{i=1}^{k} u_i l_i > 0 \tag{5.3}$$

O u_0 aparece sempre com coeficiente 1 devido ao 1 presente na primeira posição do vetor A_j.

Temos então um peso associado a cada literal. Lembrando da semântica de uma restrição de peso, a regra que deve ser adicionada ao programa S para que apenas modelos-resposta com custo reduzido negativo sejam produzidas é:

$$\leftarrow \{l_1 = u_1, \ldots, l_k = u_k\} \leq -u_0$$

Esta restrição filtra modelos-resposta com custo reduzido não negativo. Adicionando esta restrição ao programa S e utilizando um resolvedor ASP podemos obter uma coluna que reduza o custo do problema.

Muitos resolvedores ASP só aceitam pesos inteiros. Neste caso, precisamos multiplicar os pesos u_i por algum valor que os torne inteiros. Lembrando que $u = c'_B B^{-1}$ e que os elementos de c_B

são apenas 0 ou 1, como $B^{-1} = \frac{Adj(B)}{det(B)}$ (onde $Adj(B)$ é a matriz adjunta de B e $det(B)$ é sua determinante) e os elementos de $Adj(B)$ são inteiros, tudo que precisamos fazer é multiplicar o vetor u por $det(B)$ para obter pesos inteiros.

A desvantagem dessa técnica é que ela restringe o uso de resolvedores ASP a aqueles que utilizam regras de peso.

O Algoritmo 2 demonstra a técnica para seleção de colunas utilizando regras de peso. Assumimos a existência de uma função $ResolvedorASP$ que encontra um modelo-resposta de um programa se existir algum modelo-resposta.

Entrada: Um programa ASP S, um conjunto de k literais com probabilidades $l_{1\ldots k}$, uma matriz básica B e um vetor de custos das colunas básicas c_B
Saída: Uma coluna A com custo reduzido negativo. $\emptyset$ se não existir

```
1  u ← c'_B B^{-1} det(B)
2  S' ← S ∪ {l_1 = u_1, ..., l_k = u_k} ≤ −u_0
3  AS ← ResolvedorASP(S')
4  se ∃AS então
5      A ← [1 ... 1]_{k+1}
6      para j = 1 até k faça
7          se l_j ∉ AS então A_j ← 0
8      fim
9      retorna A
10 senão
11     retorna ∅
12 fim
```

Algoritmo 2: Seleção de colunas utilizando regras de peso

Desigualdades como instâncias SAT

Em Warners (1998), mostrou-se um algoritmo capaz de transformar qualquer desigualdade linear binária com coeficientes inteiros não negativos em uma fórmula da lógica proposicional em um tempo linear sobre o número de variáveis e o número de bits usados para representar os coeficientes. Em de Bona (2011) na seção 5.2.1, demonstrou-se com detalhes como fazer esta transformação para o caso de uma instância PSAT.

O primeiro passo é garantir que os coeficientes da inequação (5.3) sejam todos inteiros. Utilizaremos exatamente a mesma técnica utilizada para as regras de peso e multiplicamos u por $det(B)$ para obter um vetor onde todos os elementos são inteiros. Chamaremos este novo vetor de u'. Reordenando a inequação e adicionando 1 à direita para transformar $>$ em $\geq$ temos:

$$u'_1 l_1 + \cdots + u'_k l_k \geq -u'_0 + 1$$

O próximo passo é garantir que os coeficientes sejam não negativos. Para isso, para cada coeficiente u'_i negativo basta adicionar $-u'_i$ em ambos os lados da inequação e substituir $-u'_i + u'_i l_i$ por $-u'_i \bar{l}_i$ onde $\bar{l}_i$ é o inverso do valor l_i (ou seja, $1 - l_i$). Caso u'_0 seja negativo, isso significa que a inequação sempre é verdadeira e portanto não é necessária.

Finalmente precisamos representar todos os coeficientes como bits. A maior dificuldade é saber o número de bits necessários para representar a soma dos coeficientes. Utilizando a desigualdade de Hadamard (Faddeev e Somins'kyi, 1971, problema 523), que limita a determinante de uma matriz, podemos limitar o valor absoluto da soma de coeficiente em $k^2(k+1)^{\frac{k+1}{2}}/2^k$ e o número de bits necessários ao logaritmo na base 2 deste valor.

Terminada esta preparação, utilizaremos os esquemas de fórmulas SAT para multiplicação, soma e o operador $\geq$ apresentados em Warners (1998).

Com a fórmula proposicional gerada, precisamos prepará-la para que possa ser expressa por

meio restrições ASP. Considerando que a fórmula está na forma CNF, ela é da forma

$$(a_{11} \vee \cdots \vee a_{n1}) \wedge \cdots \wedge (a_{1m} \vee \cdots \vee a_{jm})$$

Primeiro iremos negar esta fórmula, transformando ela no formato DNF:

$$(\neg a_{11} \wedge \cdots \wedge \neg a_{n1}) \vee \cdots \vee (\neg a_{1m} \wedge \cdots \wedge \neg a_{jm})$$

Se para algum modelo-resposta alguma das conjunções $(\neg a_{1i} \wedge \cdots \wedge \neg a_{ni})$ for satisfeita, a negação da fórmula proposicional gerada se torna verdadeira, o que significa que a inequação $u_0 + \sum_{i=1}^{k} u_i l_i > 0$ não é satisfeita e o custo reduzido deste modelo-resposta é não negativo. Por outro lado, se nenhuma dessas conjunções for satisfeita para algum modelo-resposta, este modelo-resposta tem custo reduzido negativo.

Para gerar modelos-resposta com custo negativo devemos então exigir que nenhuma dessas conjunções seja satisfeita por meio das restrições:

$$\begin{gathered} \leftarrow \textbf{not } a_{11}, \ldots, \textbf{not } a_{n1}. \\ \vdots \\ \leftarrow \textbf{not } a_{1m}, \ldots, \textbf{not } a_{jm}. \end{gathered}$$

Se algum a_{ij} for um literal negativo, então não usamos **not** , como uma dupla negação comum.

O Algoritmo 3 representa esta técnica para seleção de colunas. Novamente assumimos a existência de uma função $ResolvedorASP$ que encontra um modelo-resposta de um programa se existir algum modelo-resposta e $InequacaoParaSAT$ que cria uma fórmula SAT para a inequação passada com o número de bits passado de acordo com os esquemas apresentados em Warners (1998).

5.2 Solução via PSAT

Como citado na Seção 2.4, existem diversos modos de se transformar o problema de encontrar modelos-resposta em um problema SAT. Por isso, um possível método para encontrar soluções para problemas PASP é através de uma transformação em um problema PSAT.

Isto, porém, torna impossível resolver problemas PASP com extensões como regras de escolha e regras de peso devido à expressividade maior destas extensões que são capazes de expressar problemas que não estão em NP. Só é possível transformar programas $AnsProlog^{\neg,\perp}$sem regras de escolha em fórmulas SAT.

O método envolve simplesmente transformar o programa S do PASP em um conjunto de clausas proposicionais. Associar-se-ia então às clausas uma probabilidade 1 e às probabilidades dos átomos no problema PASP seriam associadas a fórmulas proposicionais que representassem estes átomos nas clausas geradas.

Isto significa que seria necessário que houvesse um mapeamento simples de uma variável ASP para um átomo na fórmula SAT gerada. Por isso é necessário que a transformação do programa ASP no programa SAT utilizada seja fiel e, portanto, que haja uma bijeção entre os modelos SAT e modelos-resposta, caso contrário não seria possível afirmar que o vetor π que serve de testemunha para a solução do problema PSAT seja útil para resolver o problema PASP.

Devido ao requerimento de fidelidade, a transformação de ASP para SAT apresentada em Ben-Eliyahu e Dechter (1992) não pode ser utilizada. Como a transformação utilizada em Lin e Zhao (2002) pode fazer o tamanho da fórmula crescer exponencialmente, melhor candidata seria a transformação utilizada em Janhunen (2004). Ainda assim o método utilizado pelo ASSAT de iterativamente adicionar fórmulas de loop é uma melhora.

De qualquer maneira, o crescimento da instância ao se traduzir do PASP para o PSAT é inevitável, tanto em número de variáveis quanto em número de regras/clausas, devido à falta de modularidade em consequência ao teorema 2.30.

Entrada: Um programa ASP S, um conjunto de k literais com probabilidades $l_{1\ldots k}$, uma matriz básica B e um vetor de custos das colunas básicas c_B
Saída: Uma coluna A com custo reduzido negativo. $\emptyset$ se não existir

1 $u \leftarrow c'_B B^{-1} det(B)$
2 $a_0 \leftarrow u_0 + 1$
3 **para** $i = 1$ *até* k **faça**
4 **se** $u_i \geq 0$ **então**
5 $a_i \leftarrow u_i$
6 $z_i \leftarrow l_i$
7 **senão**
8 $a_i \leftarrow -u_i$
9 $z_i \leftarrow \bar{l}_i$
10 $a_0 \leftarrow a_0 - u_i$
11 **fim**
12 **fim**
13 $\Lambda \leftarrow \emptyset$
14 **se** $a_0 > 0$ **então**
15 $b \leftarrow k^2(k+1)^{\frac{k+1}{2}}/2^k$
16 $L \leftarrow InequacaoParaSAT(\sum_{i=1}^{k} a_i z_i \geq a_0, b)$
17 **para** *cada clause* c *em* L **faça**
18 $R \leftarrow \emptyset$
19 **para** *cada literal* l *em* c **faça**
20 **se** l *está negado* **então**
21 $R \leftarrow R \cup l$
22 **senão**
23 $R \leftarrow R \cup$ **not** l
24 **fim**
25 **fim**
26 $\Lambda \leftarrow \Lambda \cup \{\leftarrow R\}$
27 **fim**
28 **fim**
29 $AS \leftarrow ResolvedorASP(S \cup \Lambda)$
30 **se** $\exists AS$ **então**
31 $A \leftarrow [1 \ldots 1]_{k+1}$
32 **para** $j = 1$ *até* k **faça**
33 **se** $l_j \notin AS$ **então** $A_j \leftarrow 0$
34 **fim**
35 **retorna** A
36 **senão**
37 **retorna** $\emptyset$
38 **fim**

Algoritmo 3: Seleção de colunas utilizando desigualdades transformadas em fórmulas SAT

Exemplo 5.3. Considere o problema do exemplo 5.2:

$$0 \leq \{a = 1\} \leq 1.$$
$$\begin{aligned} b, c &\leftarrow a. \\ b &\leftarrow \textbf{not } a. \\ c &\leftarrow \textbf{not } a. \end{aligned}$$

com as probabilidades $P(b) = 0.7$ e $P(c) = 0.4$.

Ele possui regras de escolha, porém estas regras não formam um loop no grafo de dependências positivas e, portanto, podem ser eliminadas. Neste caso, podemos também eliminar a regra de peso adicionando uma nova variável. Obteremos o programa:

$$\begin{array}{rcl} a & \leftarrow & \textbf{not } a'. \\ a' & \leftarrow & \textbf{not } a. \\ b & \leftarrow & a, \textbf{not } c. \\ c & \leftarrow & a, \textbf{not } b. \\ b & \leftarrow & a'. \\ c & \leftarrow & a'. \\ & \leftarrow & a, a'. \end{array}$$

Novamente o grafo de dependências positivas não possui loops. A transformação deste programa, utilizando a completação é:

$$\begin{array}{rcl} a & \equiv & \neg a' \\ a' & \equiv & \neg a \\ b & \equiv & (a \wedge \neg c) \vee (a') \\ c & \equiv & (a \wedge \neg b) \vee (a') \\ \bot & \leftarrow & a \wedge a'. \end{array}$$

Temos então uma instância PSAT na forma normal atômica onde Γ é a completação mostrada acima e $\Psi = \{P(b) = 0.7, P(c) = 0.4\}$.

Este programa possui os modelos $\{a = 0, a' = 1, b = 1, c = 1\}$, $\{a = 1, a' = 0, b = 1, c = 0\}$ e $\{a = 1, a' = 0, b = 0, c = 1\}$. Tomando-se $\pi = [0.1, 0.6, 0.3]$, temos uma solução para o problema PSAT e este mesmo π serve de solução para o problema PASP.

5.3 Implementação

Como parte do projeto, foram implementados dois programas utilizando os algoritmos aqui citados para resolver instâncias PASP. O código-fonte de ambos os programas está disponível em um repositório Git no endereço https://gitorious.org/pasp sob a licença GPLv3[1].

Existe um programa, *pasp-psat*, que utiliza o método discutido na Seção 5.2, ou seja, ele transforma uma instância PASP e uma instância PSAT equivalente utilizando a tradução por fórmulas de loop. Utilizamos a tradução por fórmulas de loop devido à sua facilidade de implementação. Ele se encontra no repositório git://gitorious.org/pasp/pasp-psat.git. O outro, *pasp-asp*, utiliza o método da Seção 5.1 fazendo uma redução de Turing para ASP utilizando regras de peso para selecionar colunas. Este último está no repositório git://gitorious.org/pasp/pasp-asp.git.

Estes programas foram desenvolvidos em C++ utilizando diversas ferramentas do novo padrão da linguagem, C++11. Por isso, para compilar os projetos é necessário um compilador que atenda estes novos requisitos. GCC[2] 4.6 e clang[3] 3.1 compilam o projeto corretamente. Clang 3.0 falha ao compilar o projeto por falta de suporte a *initializer lists* e GCC 4.5 não possui suporte a *range-based fors*. Do mesmo modo, a versão mais recente do compilador da Microsoft, MSVC[4] 2010, atualmente não consegue compilar os projetos por falta de suporte a diversas características do C++11 (Microsoft Corporation (2010)). Utilizando versões adequadas do GCC e do clang é possível compilar o programa nos sistemas operacionais Windows, GNU/Linux e Mac OS X, além de outras plataformas compatíveis. O *pasp-asp* utiliza *pipes* e outras funções POSIX para a comunicação entre processos e por isso para ser executado no Windows necessita de uma camada de compatibilidade, como o Cygwin[5]. Para os testes de unidade foi usada a biblioteca cppunit[6] e para as operações de

[1] Texto integral da licença disponível em http://www.gnu.org/licenses/gpl.html
[2] http://gcc.gnu.org
[3] http://clang.llvm.org
[4] http://msdn2.microsoft.com/en-us/visualc/default.aspx
[5] http://www.cygwin.com
[6] http://www.freedesktop.org/wiki/Software/cppunit

álgebra linear do *pasp-asp* foi usada a biblioteca Eigen[7].

Ambos os programas recebem a instância PASP em duas partes separadas: um programa ASP, já instanciado por um *instanciador*, pela entrada padrão e uma atribuição de probabilidades em um arquivo. Isto faz com que o programa aja como um *filtro* de sistemas Unix. Um exemplo de invocação do programa *pasp-psat* seria:

```
lparse exemplo.lp | pasp-psat exemplo.prob | resolvedor-psat
```

Exemplo 5.4. Considere o seguinte o arquivo `exemplo.lp` que contém o programa ASP do exemplo 2.25 no formato *lparse*:

```
passar(X,Y) :- aresta(X,Y), caminhoPara(X), not naoPassar(X,Y).
naoPassar(X,Y) :- aresta(X,Y), not passar(X,Y).
caminhoPara(1).
caminhoPara(Y) :- caminhoPara(X), passar(X,Y), aresta(X,Y).
:- passar(X,Y1) : aresta(X,Y1), passar(X,Y2) : aresta(X,Y2), Y1!=Y2,
   vertice(X), aresta(X,Y1), aresta(X,Y2).
:- not caminhoPara(n).

#hide caminhoPara(X).
#hide aresta(X,Y).
#hide vertice(X).
#hide naoPassar(X,Y).
```

Após ser processado pelo *lparse*, junto com a descrição do grafo 1.1, `grafo.lp`, este programa é instanciado e escrito em um formato interno do *lparse*.

```
$ lparse grafo.lp exemplo.lp
1 1 1 1 2
1 1 2 0 3  4
2 4 2 0 2 5 6
(...)
3 1 31 1 0 15
0
5 passar(1,2)
6 passar(1,3)
11 passar(3,4)
12 passar(3,5)
31 passar(5,6)
32 passar(4,5)
33 passar(2,4)
0
B+
0
B-
1
0
1
```

O programa *pasp-psat* lê este arquivo da entrada padrão e recebe como argumento o arquivo `exemplo.prob` com associações de probabilidade.

```
passar(1,3) 0.5
passar(3,4) 0.4
```

[7] http://eigen.tuxfamily.org/

O resultado da execução do *pasp-psat* com estas entradas é uma instância PSAT no formato de entrada definido em de Bona (2011):

```
$ lparse grafo.lp exemplo.lp | pasp-psat exemplo.prob
p pcnf 80 179 2 1
a 1 0.5
a 2 0.4
-36 30 0
-36 -15 0
-30 15 36 0
-36 1 0
-1 36 0
(...)
```

Finalmente enviando esta instância PSAT para um resolvedor PSAT, como o PSATtoMaxSAT[8], um resolvedor PSAT eficiente desenvolvido com base no artigo Finger e De Bona (2010), obtemos uma resposta:

```
$ PSATtoMaxSat exemplo.psat
(...)
Probabilities:
0.500000 0.100000 0.400000
Columns inconsistent with Gamma:
       0        0        0 ******
Total number of iterations:         3
SAT
```

Estas probabilidades apontadas, 0,5, 0,1 e 0,4, de fato são uma resposta para o problema ilustrado pela figura 1.1

A escolha desta forma de entrada foi feita para aproveitar o máximo possível o instanciador. Os instanciadores modernos, como o lparse[9] e o gringo[10] são bastantes sofisticados e facilitam a criação de programas ASP eficientes e concisos. Esta forma de entrada é um dos jeitos mais simples de evitar que o programa tenha que interpretar os símbolos de uma instância ASP ou ter que fazer a instanciação por si próprio. Assim o programa se concentra em fazer a sua função deixando a tarefa de interpretação e, possivelmente, definição de um formato mais amigável, nas mãos de outro programa.

No caso do *pasp-asp*, como o resolvedor ASP deve ser chamado diversas vezes, o caminho do executável é passado como argumento. Por exemplo, a execução do *pasp-asp* para resolver o problema do exemplo utilizando o smodels[9] seria:

```
$ lparse grafo.lp exemplo.lp | pasp-asp exemplo.prob smodels
ANSWER: CONSISTENT
Base:
1 1 1
0 1 1
0 0 1

Probability assignment:
0.5
0.1
0.4
```

[8] http://sourceforge.net/projects/psat/
[9] http://www.tcs.hut.fi/Software/smodels/
[10] http://potassco.sourceforge.net/

O formato de entrada do programa ASP instanciado é o mesmo que o utilizado pelo smodels e gerado pelo lparse. Qualquer instanciador compatível pode ser usado.

5.4 Resultados experimentais

Executamos testes para analisar o comportamento de ambos os resolvedores PASP. Todos os testes foram executados em um computador com 12 processadores Intel Core i7 e 48Gb de memória RAM rodando Ubuntu Linux 12.04 64bits.

Os testes executavam os programas *pasp-psat* e *pasp-asp* em uma série de programas PASP aleatórios. As regras ASP dos programas foram geradas utilizando o procedimento descrito em Zhao e Lin (2003) para criar programas 3-LP, ou seja, programas onde cada regra possui exatamente 3 literais distintos, 1 literal na cabeça e 2 no corpo, e cada literal do corpo tem 50% de chance de aparecer com uma negação como falha.

Foram geradas diferentes associações de probabilidade para cada programa ASP. As associações de probabilidade variavam quanto à percentagem de literais que recebiam probabilidades (20%, 50% e 80%) e quanto ao intervalo do qual probabilidades eram escolhidas ($[0; 0,3]$, $[0,7; 1]$ e $[0; 1]$). Portanto, para cada programa ASP, haviam 9 associações de probabilidade diferentes.

Da mesma maneira as regras ASP variavam no número de literais (50, 100 e 150) e no número de regras (entre $0,5$ e $6,5$ vezes o número de literais, com incrementos de $0,5$). Isto significa um total de 39 programas ASP.

Combinando cada programa ASP com as possíveis associações de probabilidade, temos um total de 351 configurações de instâncias PASP, considerando o número de literais, número de regras, intervalo de probabilidades e percentagem de literais com probabilidade. Para cada configuração possível, 200 instâncias eram geradas e resolvidas por ambos os programas.

Podemos ver nos gráficos de dispersão 5.1, 5.2 e 5.3 os tempos médios de 200 execuções com diferentes configurações utilizando o *pasp-asp* e nos gráficos de dispersão 5.4, 5.5 e 5.6 os tempos médios utilizando o *pasp-psat*. Destes gráficos podemos obter várias informações sobre o comportamento destes métodos. Apesar do *pasp-asp* ser consistentemente mais rápido que o *pasp-psat*, ambos apresentam um comportamento parecido, com um padrão "fácil-difícil-fácil" característico de uma transição de fase.

A transição de fase é um fenômeno encontrado em muitos problemas NP-completos onde uma região em torno de certos valores críticos de algum parâmetros das instâncias é muito mais difícil que as regiões adjacentes, em geral quando a probabilidade da resposta do problema ser "verdadeiro" é 50%. Em Zhao e Lin (2003) vemos que para programas ASP 3-LP com 150 literais, as instâncias mais difíceis se encontram quando a razão entre o número de regras e o número de literais é aproximadamente 5, mas este valor não corresponde a uma probabilidade de 50% da resposta da instância ser "verdadeiro". Isto parece se repetir no caso do PASP com programas 3-LP.

Os gráficos 5.8 e 5.8 ilustram os tempos médios dos testes ordenados pela percentagem de literais que estavam associados a probabilidades. Apesar desta quantidade definir o tamanho da matriz que deve ser gerada para resolver o problema, os dados mostram que esta quantidade parece ter pouca influência sobre o tempos de execução dos programas.

Dos 70.200 testes rodados, apenas 175 instâncias PASP, $0,249\%$ do total, são consistentes e praticamente todas possuem a razão entre regras e literais igual a $0,5$, não sendo significativamente influenciadas pelo intervalo dos valores das probabilidades. Isto pode ser reflexo do número médio de modelos-resposta em instâncias 3-LP. Segundo Zhao e Lin (2003), a probabilidade da existência de pelo menos um modelo-resposta cai bruscamente de 100% para 30% quando a razão entre regras e literais se move de 0 para 1 e continua a cair mais suavemente.

A menos que todas as probabilidades sejam iguais a 1, é necessário mais de um modelo-resposta para que uma instância PASP seja consistente. Como a probabilidade de existência de modelos-resposta é tão pequena para instâncias com razão alta entre regras e literais, é de se esperar que a maioria seja inconsistente.

Sabendo que não existem modelos-resposta para uma instância, trivialmente conclui-se que ela

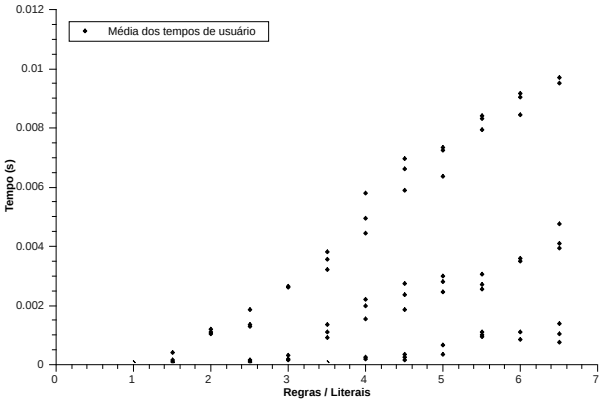

Figura 5.1: *Gráfico de dispersão dos testes com o* pasp-asp *utilizando 50 literais*

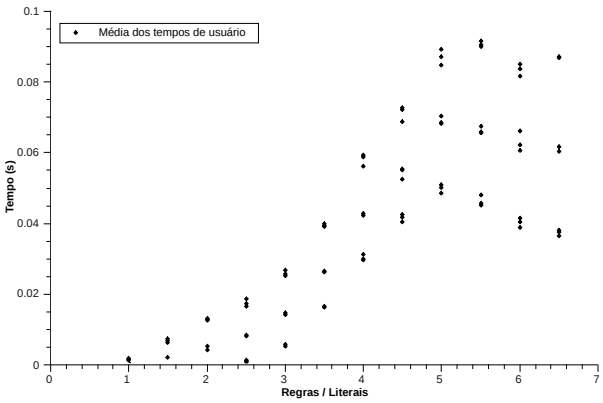

Figura 5.2: *Gráfico de dispersão dos testes com o* pasp-asp *utilizando 100 literais*

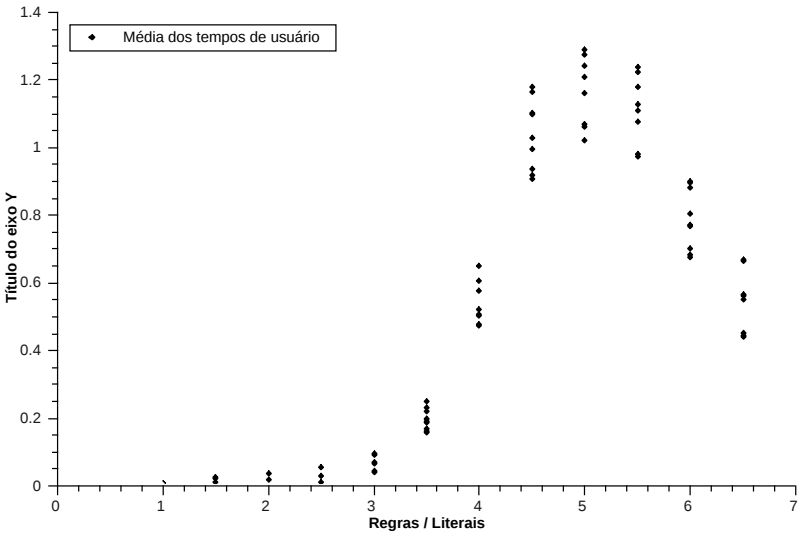

Figura 5.3: *Gráfico de dispersão dos testes com o* pasp-asp *utilizando 150 literais*

seja inconsistente. É possível que este seja o motivo da semelhança entre as curvas de tempo do ASP e do PASP: mostrar a existência de um único modelo-resposta é a maior parte do trabalho.

Conforme explicado na seção A.2.2 é possível, em casos degenerados, que o algoritmo simplex entre em *loop* infinito se não forem tomadas precauções. Tratando-se de uma prova de conceito, no *pasp-asp* não foram implementadas estratégias para evitar o *loop* infinito. Mesmo assim, apenas 7 instâncias, 0.009% das instâncias testadas, entraram em *loop* infinito. Foram considerados como estando em *loop* infinito todas as execuções que superaram 3 horas de execução.

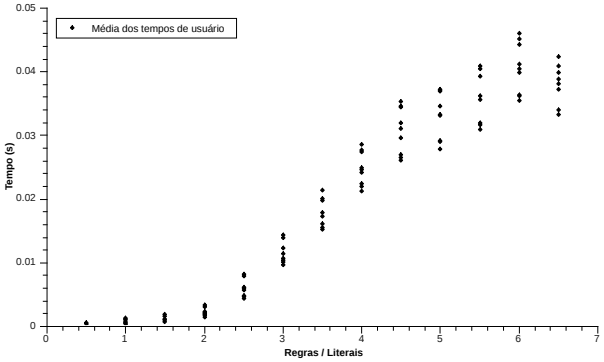

Figura 5.4: *Gráfico de dispersão dos testes com o* pasp-psat *utilizando 50 literais*

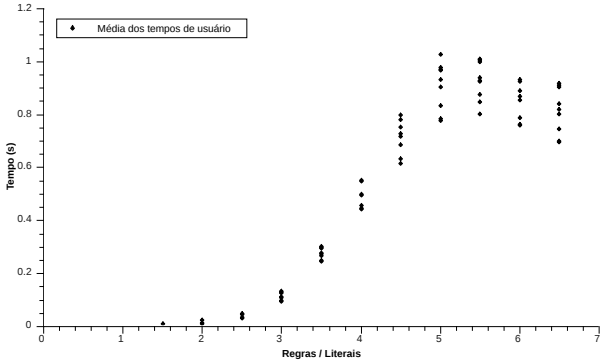

Figura 5.5: *Gráfico de dispersão dos testes com o* pasp-psat *utilizando 100 literais*

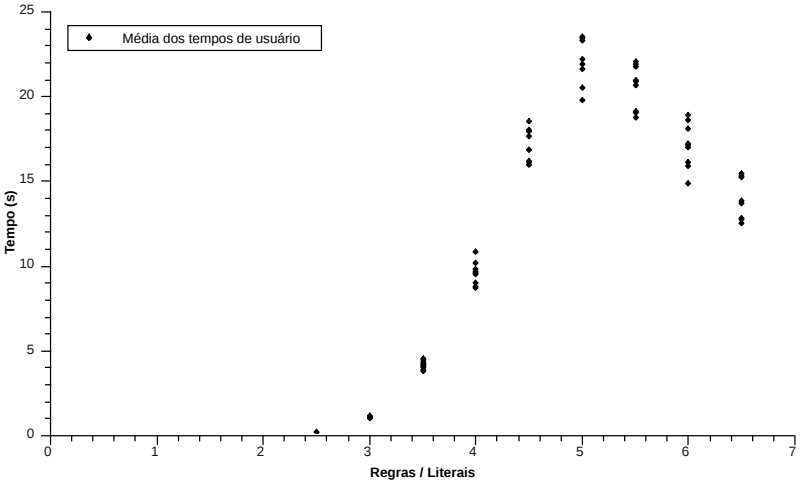

Figura 5.6: *Gráfico de dispersão dos testes com o* pasp-psat *utilizando 150 literais*

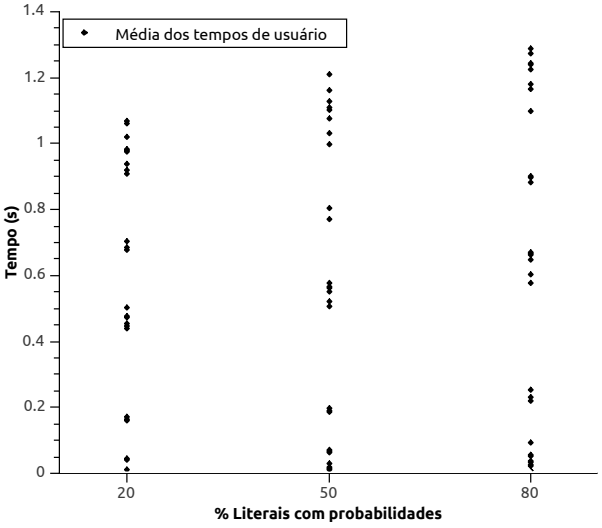

Figura 5.7: *Gráfico de dispersão relacionando a percentagem de literais com probabilidade e o tempo médio dos testes com o* pasp-asp *com 150 literais*

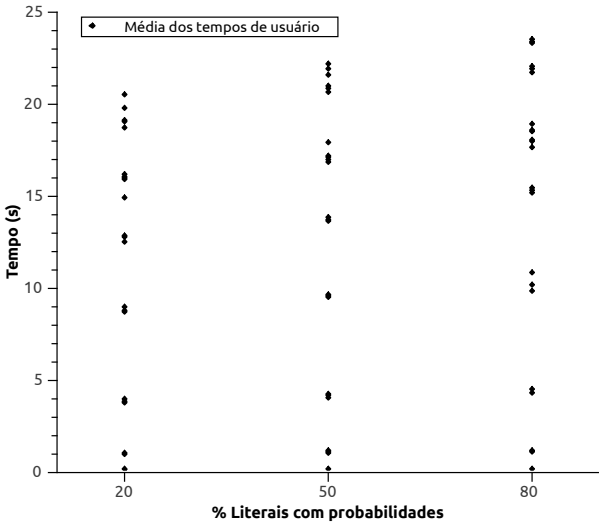

Figura 5.8: *Gráfico de dispersão relacionando a percentagem de literais com probabilidade e o tempo médio dos testes com o* pasp-psat *com 150 literais*

Capítulo 6

Conclusão

Neste trabalho nosso objetivo foi tentar juntar a expressividade do Answer Set Programming com o poder da satisfazibilidade probabilística.

Com o Answer Set Programming probabilístico isto se torna possível. A semântica do PASP é equivalente à do PSAT, porém com a parte lógica sendo substituída pelo ASP com todas as suas facilidades.

Uma possível aplicação bastante simples do PASP seria a verificação de hipóteses científicas. Se for possível expressar uma hipótese na forma de regras de um programa ASP podemos obter um conjunto de dados experimentais de fenômenos descritos por esta teoria, expressá-los como probabilidades (considerando a interpretação frequentista de probabilidade) e utilizar o PASP para avaliar se a hipótese e os dados são consistentes.

É claro que isto não pode ser usado para provar uma hipótese, apenas para desprová-la caso os dados e a hipótese não se mostrem consistentes.

Caso a hipótese e os dados se mostrem consistentes, a distribuição de probabilidade π encontrada que satisfaz o problema pode conter informações interessantes. A distribuição de probabilidade π expressa a probabilidade de ocorrência dos vários "mundos possíveis" que estão de acordo com a hipótese. Se um desses "mundos possíveis" tiver probabilidade muito baixa em todos os possíveis π seria uma indicação que talvez este caso seja impossível e um refinamento da hipótese seja possível.

De maneira mais geral, a variança ou entropia da distribuição de probabilidade pode dar valiosas dicas sobre a hipótese. Mais pesquisas são necessárias nesta área para avaliar a função destes indicadores.

Perceba que podemos ter vários π diferentes que tornam um problema PASP consistente. Alguns inclusive com mais valores maiores que zero do que o necessário. O método de resolução baseado na redução de Turing para ASP (seção 5.1) pode ser ligeiramente modificado para encontrar distribuições interessantes.

O método se baseia na primeira fase do algoritmo simplex. Podemos continuar executando a segunda fase do algoritmo simplex e assim minimizar ou maximizar alguma função convexa sobre os elementos do vetor π. Com isso, podemos encontrar distribuições de probabilidade interessantes.

Encontrar distribuições de probabilidade que maximizam ou minimizam funções pode ser útil para outras aplicações. Considere o problema de fazer uma análise sintática em uma frase numa linguagem natural. É possível expressar as regras gramaticais de formação de frases como regras ASP e retirar de um extensivo corpus as probabilidades relativas de cada construção gramatical. Podemos então utilizar o PASP para encontrar o sintagma mais provável. Esta técnica de realizar o *parsing* de linguagem natural é bastante sofisticada e merece mais pesquisa para avaliar sua capacidade.

Concluindo, o Answer Set Programming probabilístico oferece muitas ferramentas para resolver problemas probabilístico e pode ajudar a resolver muitos problemas envolvendo lógica e probabilidade.

Apêndice A

Programação linear

Neste capítulo teremos uma pequena revisão sobre programação linear. Leitores familiarizados com o assunto podem pular este capítulo sem maiores problemas. A maior parte deste material foi retirado de Bertsimas e Tsitsiklis (1997). Para manter a brevidade, as provas dos teoremas não são inclusas e podem ser vistas no livro citado.

A.1 Conceitos iniciais

Programação linear, ou *otimização linear*, é o problema de minimizar uma função linear sujeita a equações lineares como restrições.

Exemplo A.1.

$$\begin{array}{lrcl} \text{minimize} & -x_1 - 2x_2 & & \\ \text{restrito à} & -x_1 + 6x_2 & \geq & -5 \\ & x_1 + x_2 & \leq & 10 \\ & 2x_1 + 20x_2 & \leq & 50 \\ & x_1 & \geq & 0 \\ & x_2 & \geq & 0 \end{array}$$

As variáveis $x_1, \ldots, x_n$ são chamadas *variáveis de decisão*. Um vetor x satisfazendo todas as restrições é chamado *solução viável*. O conjunto de toda as soluções viáveis é chamado *região viável*.

A função $c'x$ é chamada de *função de custo*. Um problema de programação linear é o problema de encontrar uma solução viável que minimiza a função de custo. Essa função é chamada *solução ótima viável*, e o seu custo é chamado de *custo ótimo*.

Todos os problemas de programação linear podem ser expressos na *forma padrão*. Um problema está na forma padrão se ele é da forma:

$$\begin{array}{lrcl} \text{minimize} & c'x & & \\ \text{restrito à} & Ax & = & b \\ & x & \geq & 0 \end{array}$$

A região viável de um problema de programação linear pode ser descrita como um poliedro, como mostrado na figura A.1.

Considere a reta $c'x = C$, que é a reta de todos os valores que fazem a função de custo ter valor igual a C. Para minimizar esta função devemos mover esta reta, diminuindo o valor de C o máximo possível sem que a reta saia da região viável.

Para o caso de duas dimensões, intuitivamente, parece que se existir um valor ótimo ele conterá um vértice do poliedro, pois ou a reta $c'x = C$ é paralela a outra reta que forma os limites do poliedro, e neste caso dois vértices possuem o custo ótimo, ou então a reta $c'x = C$ é concorrente a todas as retas que formam os limites do poliedro. Neste caso também, o máximo que podemos diminuir C nos leva a um vértice do poliedro.

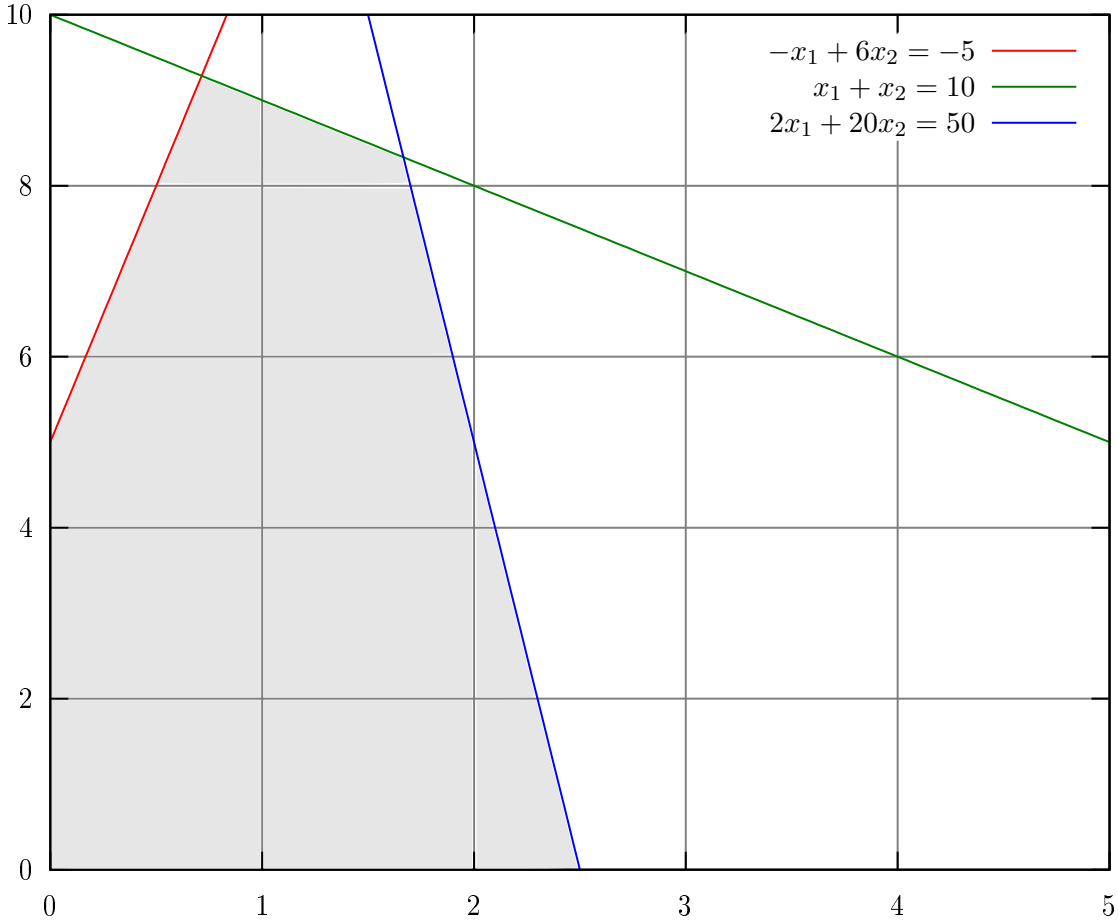

Figura A.1: *Representação da região viável do exemplo A.1*

De fato, a intuição está correta e para qualquer problema de programação linear, se existir um valor ótimo, um vértice possui este valor ótimo.

Porém, para permitir um tratamento algorítmico usaremos uma definição de vértice pouco usual.

Definição A.2. Se um vetor x^* satisfaz $a_i' x^* = b_i$ para algum i (um índice de uma restrição), dizemos que a restrição i está *ativa* em x^*.

Definição A.3. Considere um poliedro P e seja x^* um elemento de $\mathbb{R}^n$.

- O vetor x^* é uma *solução básica* se:
  - Todas as restrições de igualdade estão ativas.
  - Das restrições que estão ativas em x^*, existem n que são linearmente independentes.
- Se x^* é uma solução básica que satisfaz todas as restrições, dizemos que é uma *solução básica viável*

O conceito de solução básica viável é equivalente ao conceito usual de vértice de um poliedro.

Perceba que se o número de restrições usadas para definir o poliedro $P \subset \mathbb{R}^n$ é menor que n, o número de restrições ativas a qualquer momento deve ser menor que n e não existem soluções básicas.

Para um problema na forma padrão, como toda solução básica deve satisfazer a restrição de igualdade $Ax = b$, toda solução viável possui m restrições ativas, onde m é o número de restrições. Podemos assumir que estas soluções são linearmente independentes, pois, caso não sejam, isto significa que alguma restrição do problema é redundante e pode ser eliminada. Para obter um total

de n restrições ativas, precisamos escolher $n-m$ das variáveis de x_i e setá-las como 0, o que faz a restrição de não negatividade correspondente, $x_i \geq 0$ ativa. Porém, para o conjunto resultante de n restrições ativas ser linearmente independente, a escolha das $n-m$ restrições não é totalmente arbitrária, como mostrado a seguir.

Teorema A.4. *Considere a restrição $Ax = b$ e $x \geq 0$ e assuma que a matriz $m \times n$ A possua linhas linearmente independentes. Um vetor $x \in \mathbb{R}^n$ é uma solução básica se e somente se temos $Ax = b$ e existem os índices $B(1), \ldots, B(m)$ tal que:*

- *As colunas $A_{B(1)}, \ldots, A_{B(m)}$ são linearmente independentes;*
- *Se $i \neq B(1), \ldots, B(m)$, então $x_i = 0$*

Deste teorema podemos chegar ao seguinte método para construir soluções básicas:

- Escolha m colunas linearmente independentes $A_{B(1)}, \ldots, A_{B(m)}$.
- Ajuste $x_i = 0$ para todo $i \neq B(1), \ldots, B(m)$.
- Resolva o sistema de m equações $Ax = b$ para as incógnitas $x_{B(1)}, \ldots, x_{B(m)}$.

Se x é uma solução básica, as variáveis $x_{B(1)}, \ldots, x_{B(m)}$ são chamadas **variáveis básicas**, $A_{B(1)}, \ldots, A_{b(m)}$ são **colunas básicas** e a matriz formada por essas colunas é uma **matriz básica**, normalmente representada pela letra B.

A.2 Algoritmo simplex

O método simplex é um método para resolução de problemas de programação linear. Apesar de no pior caso ele ter uma complexidade exponencial, na prática o método simplex é bastante rápido, geralmente mais rápido que algoritmos polinomiais.

Ele procura por uma solução ótima movendo de uma solução básica viável para outra nos cantos do conjunto viável, sempre em uma direção que reduz a função de custo.

Esta estratégia gulosa funciona pois, para problemas de programação linear, um ponto ser ótimo localmente implica ele ser um ótimo global, uma vez que estamos minimizando uma função convexa em um conjunto convexo.

Se estivermos em um ponto x qualquer dentro de um poliedro P, precisamos saber em quais direções podemos nos mover para continuar dentro do poliedro. Por isso a próxima definição.

Definição A.5. Seja x um elemento do poliedro P. Um vetor $d \in \mathbb{R}^n$ é chamado uma *direção viável* de x se existe um escalar positivo Θ para o qual $x + \Theta d \in P$

O algoritmo simplex não se move em qualquer direção viável. Ele se move de uma solução básica viável para uma próxima que difere por apenas uma coluna básica. Isto significa que todos os índices do vetor d correspondentes às variáveis não básicas devem ser 0, com exceção de uma variável não básica que deve se tornar básica. Se esta variável tiver índice j, teremos $d_j = 1$.

Como pode ser verificado, para os índices básicos temos

$$d_B = -B^{-1}A_j$$

Resumindo, seja j o índice de uma variável não básica que se tornará básica, temos um vetor d tal que:

$$d = \begin{cases} d_j &= 1 \\ d_i &= 0, \text{ para variáveis não básicas diferentes de } j \\ d_B &= B^{-1}A_j \end{cases}$$

Essa direção é chamada j-ésima direção básica.

Porém, precisamos saber qual direção básica queremos seguir. Para isso verificamos qual direção é vantajosa.

Definição A.6. Seja x uma solução básica e B a matriz básica associada. Para cada j, definimos o *custo reduzido* $\bar{c}_j$ da variável x_j de acordo com a fórmula

$$\bar{c}_j = c_j - c'_B B^{-1} A_j$$

Intuitivamente, o custo reduzido $\bar{c}_j$ mede o quanto x_j contribui individualmente para o custo total. Se $\bar{c}_j < 0$ para algum j, então é interessante, se possível, aumentar x_j, pois isso reduzirá o custo total. Por outro lado, se o vetor de todos os $\bar{c}_j$, $\bar{c}$, for maior ou igual a zero e o ponto atual é viável, chegamos à solução ótima (o contrário só é válido para casos não degenerados. Veja seção A.2.2).

Digamos que estejamos em uma solução básica viável x e computamos o custo reduzido $\bar{c}_j$ das variáveis não básicas. Se todas são não negativas, temos uma solução ótima e paramos. Caso contrário, o custo reduzido $\bar{c}_j$ de uma variável não básica x_j é negativo, o que significa que a j-ésima direção básica d é uma direção viável que diminui o custo. Movendo-se pela direção d, a variável não básica x_j se torna positiva e as outras variáveis não básicas continuam 0. Dizemos que x_j "entra na base".

Como o custo diminui na direção d, queremos andar nessa direção o máximo possível sem sair da região viável. Isso nos leva ao ponto $x + \Theta^* d$ onde $\Theta^* = \max\{\Theta \geq 0 | x + \Theta d \in P\}$. A mudança no custo é $\Theta^* c' d$, que é o mesmo que $\Theta^* \bar{c}_j$.

Agora derivamos a fórmula para Θ^*. $x+\Theta d$ se torna inviável somente se um dos seus componentes se torna negativo.

Se todos os componentes de $d \geq 0$, a solução nunca se torna inviável e portanto $\Theta^* = \infty$.

Se $d_i < 0$ para algum i, temos:

$$\Theta^* = \min_{\{i | d_i < 0\}} \left(-\frac{x_i}{d_i} \right)$$

Como só precisamos considerar as variáveis básicas, temos

$$\Theta^* = \min_{\{i=1,\ldots,m | d_{B(i)} < 0\}} \left(-\frac{x_{B(i)}}{d_{B(i)}} \right)$$

Depois de escolher o Θ^*, movemos para a nova solução viável $y = x + \Theta^* d$. Após mover, teremos $y_j = \Theta^*$. Além disso, a variável básica cujo índice minimiza a expressão $\left(-\frac{x_{B(i)}}{d_{B(i)}} \right)$ se torna zero. Então nós substituímos $A_{B(l)}$ por A_j na base, obtendo uma nova base.

Teorema A.7. *(a) As colunas $A_{B(i)}$, $i \neq l$ e A_j são linearmente independentes e portanto a nova matriz é básica. (b) o vetor y é uma solução básica viável associada com a nova matriz básica*

Juntando tudo o que vimos nessa seção, podemos escrever uma iteração do algoritmo simplex com visto na tabela A.1.

1.	Começamos com uma base com as colunas $A_{B(1)}, \ldots, A_{B(m)}$ e uma solução básica viável x.
2.	Computamos o custo reduzido $\bar{c}_j = c_j - c'_B B^{-1} A_j$ para todos os índices não básicos j. Se todos são não negativos, a solução atual é ótima. Se não, escolhemos algum j para o qual $\bar{c}_j < 0$
3.	Compute o vetor $u = -d_B = B^{-1} A_j$. Se nenhum componente de u é positivo, temos $\Theta^* = \infty$ e o custo ótimo é $-\infty$.
4.	Se algum componente de u é positivo, calculamos $\Theta^* = \min_{\{i=1,\ldots,m \| u_i > 0\}} \frac{x_{B(i)}}{u_i}$
5.	Seja l o índice em que $\Theta^* = \frac{x_{B(l)}}{u_l}$ formamos uma nova base substituindo $A_{B(l)}$ com A_j. Os valores das novas variáveis básicas são $y_j = \Theta^*$ e $y_{B(i)} = x_{B(i)} - \Theta^* u_i$, $i \neq l$.

Tabela A.1: *Uma iteração do algoritmo simplex*

Uma otimização comum feita na implementação do algoritmo simplex é manter a matriz B^{-1} pré-calculada e atualizá-la utilizando *operações elementares de linha*.

A.2.1 Solução básica inicial

A única coisa que falta para poder executar o algoritmo simplex é uma solução básica viável inicial para poder executar a primeira iteração. Podemos fazer isso utilizando um problema auxilar. Esta técnica é chamada de algoritmo simplex de duas fases.

Considere o problema:

$$\begin{array}{lrcl} \text{minimize} & c'x & & \\ \text{restrito à} & Ax & = & b \\ & x & \geq & 0 \end{array}$$

Introduziremos o vetor de variáveis artificiais $y \in \mathbb{R}^m$ e usamos o simplex para resolver o problema auxiliar.

$$\begin{array}{lrcl} \text{minimize} & y_1 + \cdots + y_m & & \\ \text{restrito à} & Ax + y & = & b \\ & x & \geq & 0 \\ & y & \geq & 0 \end{array}$$

Inicializar o problema auxiliar é fácil. Ao setar $x = 0$ e $y = b$, temos uma solução básica viável e a matriz básica correspondente é a identidade.

Perceba que no problema auxiliar a função de custo é a soma das variáveis auxiliares. Se houver uma solução para o problema original, teremos um custo 0 do problema auxiliar, pois poderemos zerar todas as variáveis auxiliares.

Porém, mesmo se chegarmos ao custo 0, ainda podem existir variáveis artificiais na base caso tenhamos uma solução degenerada do problema auxiliar. Neste caso precisamos tirar essas variáveis da base introduzindo variáveis não básicas de custo 0 na base.

A.2.2 Degeneração

Definição A.8. Uma solução básica $x \in \mathbb{R}^n$ é dita *degenerada* se mais de n de suas restrições estão ativas em x.

Intuitivamente, em $\mathbb{R}^2$ temos uma solução degenerada se tivermos três retas se interceptando em um mesmo ponto.

Para problemas degenerados, em alguns casos $\Theta*$ pode ser igual a zero, e não mudamos a solução x, pois estamos trocando uma variável básica de valor 0 por outra não básica de valor 0. Isso pode levar a uma sequência de mudanças de base que leva de volta à base inicial, fazendo com que o algoritmo entre em *loop* infinito. Isso pode ser evitado com uma boa escolha de quais variáveis vão entrar ou sair da base.

Existem várias regras para evitar o loop. Uma das regras possíveis é a regra de Bland. Por essa regra, a coluna a entrar na base deve ser a coluna de menor índice que possua um custo reduzido negativo e se for possível tirar mais de uma coluna da base, sempre escolher a de menor índice.

Referências Bibliográficas

Baral *et al.*(**2004**) Chitta Baral, Michael Gelfond, e Nelson Rushton. Probabilistic reasoning with answer sets. Em Vladimir Lifschitz e Ilkka Niemelä, editores, *LPNMR*, volume 2923 de *Lecture Notes in Computer Science*, páginas 21–33. Springer. ISBN 3-540-20721-X. Citado na pág. 2, 24

Baral *et al.*(**2009**) Chitta Baral, Michael Gelfond, e Nelson Rushton. Probabilistic reasoning with answer sets. *Theory Pract. Log. Program.*, 9(1):57–144. ISSN 1471-0684. doi: 10.1017/S1471068408003645. Citado na pág. 2, 24

Ben-Eliyahu e Dechter(1992) Rachel Ben-Eliyahu e Rina Dechter. Propositional sematics for disjunctive logic programs. Em *JICSLP'92*, páginas 813–827. Citado na pág. 8, 13, 15, 32

Bertsimas e Tsitsiklis(1997) Dimitris Bertsimas e John N. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, Belmont, Massachusetts, EUA. ISBN 1-886529-19-1. Citado na pág. 43

Boole(1854) G. Boole. *An investigation of the laws of thought: on which are founded the mathematical theories of logic and probabilities*. Walton and Maberly. URL http://books.google.com/books?id=SWgLVT0otY8C. Citado na pág. 1, 19

Chandra *et al.*(**1981**) Ashok K. Chandra, Dexter C. Kozen, e Larry J. Stockmeyer. Alternation. *J. ACM*, 28:114–133. ISSN 0004-5411. Citado na pág. 24

Clark(1978) Keith L. Clark. Negation as failure. Em *Logic and Data Bases*, páginas 293–322. Plenum Press. Citado na pág. 3

Clocksin(2003) W Clocksin. *Programming in Prolog*. Springer-Verlag, Berlin New York. ISBN 9783540006787. Citado na pág. 3

Cook(1971) Stephen A. Cook. The complexity of theorem-proving procedures. Em *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, páginas 151–158, New York, NY, USA. ACM. Citado na pág. 14

da Silva *et al.*(**2006**) Flávio Soares Corrêa da Silva, Ana Cristina Vieira de Melo, e Marcelo Finger. *Lógica para computação*. Thomson Pioneira. ISBN 9788522105175. Citado na pág. 20

de Bona(2011) Glauber de Bona. Satisfazibilidade probabilística. Dissertação de Mestrado, Instituto de Matemática e Estatística da Universidade de São Paulo, 2011. Citado na pág. 21, 27, 31, 36

Faddeev e Somins'kyi(1971) D.K. Faddeev e I.S. Somins'kyi. *Book of Problems in Higher Algebra*. Vyshcha Shkola. Citado na pág. 31

Fages(1994) François Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1(1):51–60. Citado na pág. 15

Finger e De Bona(2010) Marcelo Finger e Glauber De Bona. A logic based algorithm for solving probabilistic satisfiability. Em *Proceedings of the 12th Ibero-American conference on Advances*

in artificial intelligence, IBERAMIA'10, páginas 453–462, Berlin, Heidelberg. Springer-Verlag. ISBN 3-642-16951-1, 978-3-642-16951-9. Citado na pág. 30, 36

Gelfond e Lifschitz(1991) M. Gelfond e V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New generation computing*, 9(3):365–385. Citado na pág. 7

Gelfond(1989) Michael Gelfond. Autoepistemic logic and formalization of commonsense reasoning preliminary report. Em *Non-Monotonic Reasoning*, páginas 176–186. Springer. Citado na pág. 1

Gelfond e Lifschitz(1988) Michael Gelfond e Vladimir Lifschitz. The stable model semantics for logic programming. Em *International Conference on Logic Programming/Joint International Conference and Symposium on Logic Programming*, páginas 1070–1080. Citado na pág. 1, 5

Janhunen(2006) Tomi Janhunen. Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics*, 16:35–86. Citado na pág. 14

Janhunen(2004) Tomi Janhunen. Representing normal programs with clauses. Em *In Proc. of the 16th European Conference on Artificial Intelligence*, páginas 358–362. IOS Press. Citado na pág. 15, 17, 18, 32

Lifschitz(2002) Vladimir Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1):39–54. Citado na pág. 1

Lifschitz e Razborov(2006) Vladimir Lifschitz e Alexander Razborov. Why are there so many loop formulas? *ACM Transactions on Computational Logic*, 7(2):261–268. ISSN 1529-3785. Citado na pág. 14

Lin e Zhao(2002) Fangzhen Lin e Yuting Zhao. Assat: computing answer sets of a logic program by sat solvers. Em *Eighteenth national conference on Artificial intelligence*, páginas 112–117, Menlo Park, CA, USA. American Association for Artificial Intelligence. ISBN 0-262-51129-0. Citado na pág. 15, 16, 32

Marek e Truszczyński(1991) Victor W. Marek e Miroslaw Truszczyński. Autoepistemic logic. *Journal of the ACM*, 38:587–618. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/116825.116836. Citado na pág. 13

Marek e Truszczyński(1998) Victor W. Marek e Miroslaw Truszczyński. Stable models and an alternative logic programming paradigm. *CoRR*, cs.LO/9809032. Citado na pág. 1, 5, 14

McCarthy(1980) J. McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial intelligence*, 13(1):27–39. Citado na pág. 3

Microsoft Corporation(2010) Microsoft Corporation. C++0x core language features in vc10: The table, 2010. Disponível em: <http://blogs.msdn.com/b/vcblog/archive/2010/04/06/c-0x-core-language-features-in-vc10-the-table.aspx>. Acesso em: 10 Set. 2012. Citado na pág. 34

Niemelä(1999) Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25:241–273. ISSN 1012-2443. doi: 10.1023/A:1018930122475. Citado na pág. 14

Niemelä *et al.***(1999)** Ilkka Niemelä, Patrik Simons, e Timo Soininen. Stable model semantics of weight constraint rules. Em *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99), Volume 1730 of lecture*, páginas 317–331. Springer-Verlag. LNAI. Citado na pág. 8, 9, 12

Nilsson(1986) Nils J. Nilsson. Probabilistic logic. *Artif. Intell.*, 28(1):71–87. Citado na pág. 1

Prasolov e Tikhomirov(2001) V.V. Prasolov e V.M. Tikhomirov. *Geometry*. American Mathematical Society. Citado na pág. 21

Rogers(1967) H. Rogers. *Theory of recursive functions and effective computability*. McGraw-Hill series in higher mathematics. McGraw-Hill. Citado na pág. 30

Syrjänen(2000) Tommi Syrjänen. *Lparse 1.0 User's Manual*, 2000. Disponível em: <http://www.tcs.hut.fi/Software/smodels/lparse.ps>. Acesso em: 6 Nov. 2010. Citado na pág. 7, 10

Trakhtenbrot(1984) B. A. Trakhtenbrot. A survey of russian approaches to perebor (brute-force searches) algorithms. *IEEE Ann. Hist. Comput.*, 6(4):384–400. ISSN 1058-6180. Citado na pág. 14

Warners(1998) Joost P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, 68(2):63–69. Citado na pág. 31, 32

Weisstein(2012) Eric W. Weisstein. Echelon form. De MathWorld—A Wolfram Web Resource, 2012. Disponível em: <http://mathworld.wolfram.com/EchelonForm.html>. Acesso em: 4 Set. 2012. Citado na pág. 28

Zhao e Lin(2003) Yuting Zhao e Fangzhen Lin. Answer set programming phase transition: A study on randomly generated programs. Em Catuscia Palamidessi, editor, *Logic Programming*, volume 2916 de *Lecture Notes in Computer Science*, páginas 239–253. Springer Berlin / Heidelberg. ISBN 978-3-540-20642-2. Citado na pág. 13, 37