Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Título da monografia

Matheus C.S.C. Pimenta

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof.ª Dr.ª Cláudia Nalon

Brasília
2015

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Título da monografia

Matheus C.S.C. Pimenta

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof.ª Dr.ª Cláudia Nalon (Orientador)
CIC/UnB

Prof.ª Dr.ª Maria Emilia Machado Telles Walter
CIC/UnB

Dr. Membro2
CIC/UnB

Prof. Dr. Rodrigo Bonifácio de Almeida
Coordenador do Bacharelado em Ciência da Computação

Brasília, 18 de dezembro de 2015

Dedicatória

Dedico este trabalho ao leitor.

Agradecimentos

Agradeço à minha família, aos meus amigos e ao rock 'n' roll, cujas existências, sem dúvida, foram fundamentais para a realização deste trabalho.

Agradeço à professora Cláudia Nalon, por ser uma orientadora extremamente solícita, atenciosa e preocupada.

Agradeço finalmente aos colegas e professores de programação competitiva, cujos ensinamentos constituem a essência deste trabalho.

Resumo

O *resumo* é um texto inaugural para quem quer conhecer o trabalho, deve conter uma breve descrição de todo o trabalho (apenas um parágrafo). Portanto, só deve ser escrito após o texto estar pronto. Não é uma coletânea de frases recortadas do trabalho, mas uma apresentação concisa dos pontos relevantes, de modo que o leitor tenha uma ideia completa do que lhe espera. Uma sugestão é que seja composto por quatro pontos: 1) o que está sendo proposto, 2) qual o mérito da proposta, 3) como a proposta foi avaliada/validada, 4) quais as possibilidades para trabalhos futuros. É seguido de (geralmente) três palavras-chave que devem indicar claramente a que se refere o seu trabalho. Por exemplo: *Este trabalho apresenta informações úteis a produção de trabalhos científicos para descrever e exemplificar como utililzar a classe LaTeX do Departamento de Ciência da Computação da Universidade de Brasília para gerar documentos. A classe `UnB-CIC` define um padrão de formato para textos do CIC, facilitando a geração de textos e permitindo que os autores foquem apenas no conteúdo. O formato foi aprovado pelos professores do Departamento e utilizado para gerar este documento. Melhorias futuras incluem manutenção contínua da classe e aprimoramento do texto explicativo.*

Palavras-chave: LaTeX, metodologia científica, trabalho de conclusão de curso

Abstract

O *abstract* é o resumo feito na língua Inglesa. Embora o conteúdo apresentado deva ser o mesmo, este texto não deve ser a tradução literal de cada palavra ou frase do resumo, muito menos feito em um tradutor automático. É uma língua diferente e o texto deveria ser escrito de acordo com suas nuances (aproveite para ler `http://dx.doi.org/10.6061%2Fclinics%2F2014(03)01`). Por exemplo: *This work presents useful information on how to create a scientific text to describe and provide examples of how to use the Computer Science Department's LaTeX class. The `UnB-CIC` class defines a standard format for texts, simplifying the process of generating CIC documents and enabling authors to focus only on content. The standard was approved by the Department's professors and used to create this document. Future work includes continued support for the class and improvements on the explanatory text.*

Keywords: LaTeX, scientific method, thesis

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Lógicas têm sido utilizadas em Computação para representar e raciocinar sobre problemas. A representação se dá através de uma linguagem formal, que define a *sintaxe* de uma lógica em particular, ou seja, a *forma* dos enunciados que estão presentes na lógica. Cada palavra na linguagem formal é dita uma *fórmula bem formada*, ou, simplesmente, uma *fórmula*.

Para cada lógica é definida também uma *semântica*, um instrumento para atribuir *significado* às fórmulas. Isto é feito através da definição de diferentes *interpretações*. Sob uma mesma interpretação, cada fórmula deve possuir um único significado. Em lógicas clássicas, como *lógica proposicional* e *lógica de primeira ordem*, o significado de uma fórmula sob uma interpretação deve ser somente um entre dois valores possíveis: *verdadeiro* ou *falso*.

Satisfatibilidade, o problema de determinar se existe uma interpretação sob a qual uma fórmula é verdadeira, é de grande interesse prático. Tal problema aparece, por exemplo, em vários problemas da microeletrônica, como síntese [3], otimização [8] e verificação [12] de *hardware*. Aparece também em problemas de raciocínio automático [9] e em muitos outros problemas relevantes [10].

Satisfatibilidade é também de grande interesse teórico. Em 1971, Cook definiu a classe dos problemas *NP-completos*, sendo satisfatibilidade proposicional o primeiro problema a ser descoberto como representativo desta classe. A partir destes resultados, Cook formalizou o enunciado do maior problema ainda não resolvido da Ciência da Computação: P versus NP [4].

Fortemente ligado ao problema da satisfatibilidade é o problema da *validade*: determinar se uma dada fórmula é verdadeira sob *qualquer* interpretação. Por esta forte relação com satisfatibilidade, que detalhamos no Capítulo 2, o problema da validade é também extensamente investigado.

Grande avanço já foi feito em direção a algoritmos de busca rápidos para satisfatibilidade e validade [6, 5, 2], apesar de ser conjecturado que qualquer um terá custo de tempo exponencial determinístico no pior caso [4].

Uma característica comum a diversos dos algoritmos para satisfatibilidade e validade é a redução do problema a fórmulas em uma determinada *forma normal*. Uma forma normal é uma imposição de restrições sobre a forma de uma fórmula, ou seja, é um subconjunto das fórmulas de uma lógica. Formas normais criam vantagens ao lidar com problemas da lógica, pois fórmulas em formas mais restritas possuem mais propriedades em comum que podem ser exploradas, além do fato de que menos situações precisam ser consideradas pelos algoritmos.

Em algoritmos que utilizam formas normais, etapas de pré-processamento são necessárias, pois é necessário transformar uma fórmula qualquer para outra que esteja na respectiva forma normal. É importante ainda que o pré-processamento tenha custo de tempo polinomial determinístico. Somente assim a técnica implementada será de fato vantajosa, em vista da conjectura sobre o custo de tempo de qualquer algoritmo de busca para satisfatibilidade e validade.

Considerando a possibilidade de melhorar a eficiência total de pré-processamento e busca, conjectura-se que fórmulas menores produzem respostas mais rápido [13]. O objetivo deste trabalho é testar esta hipótese através de experimentos. Em particular, investigamos algoritmos baseados na *forma normal clausal*, definida formalmente no Capítulo 2. Para obter fórmulas menores, implementamos algoritmos que reduzem o *número de cláusulas* através de *renomeamento*, conceitos também definidos no Capítulo 2. Propomos um algoritmo para este fim e o comparamos com um algoritmo proposto por Boy de la Tour [7]. O algoritmo que propomos permite ainda encontrar boas transformações restringindo o tamanho máximo do renomeamento.

Capítulo 2

Referencial teórico

2.1 Lógica proposicional

Esta seção apresenta os conceitos básicos que definem a lógica proposicional.

Definição 1 (Sintaxe)

Seja o conjunto infinito e enumerável $\mathcal{P} = \{a, b, ..., a_1, a_2, ..., b_1, b_2, ...\}$. Dizemos que $\mathcal{P}$ é o conjunto dos *símbolos proposicionais*.

Se $\phi \in \mathcal{P}$, dizemos que ϕ é uma *fórmula bem formada*, ou simplesmente que ϕ é uma *fórmula*. Além disso, se $\phi_1, ..., \phi_n$ são fórmulas quaisquer, onde $n \in \mathbb{N} \cup \{0\}$ e ϕ é de uma das formas a seguir, então ϕ também é uma fórmula:

1. *Negação*: $\neg\phi_1$
2. *Conjunção*: $\phi_1 \wedge ... \wedge \phi_n$
3. *Disjunção*: $\phi_1 \vee ... \vee \phi_n$
4. *Implicação*: $\phi_1 \rightarrow \phi_2$
5. *Equivalência*: $\phi_1 \leftrightarrow \phi_2$

Em todos os casos acima, dizemos que ϕ_i é *subfórmula imediata* de ϕ, para $i = 1, ..., n$.

Denotamos a conjunção vazia ($\phi_1 \wedge ... \wedge \phi_n$ com $n = 0$) por $\top$, a disjunção vazia por $\bot$ e o conjunto das fórmulas por $\mathcal{L}$.

Denotamos ainda por $SFI(\phi)$ o conjunto das subfórmulas imediatas de ϕ.

Exemplo 1 São fórmulas:

- $\phi = (p \to q) \to \neg s$
- $\psi = (p \vee q) \leftrightarrow (r \wedge s)$
- $\xi = \neg(p \to q)$

Note ainda que $p \to q$ é subfórmula imediata de ϕ e de ξ.

Definição 2 (Semântica)

Dizemos que $\Bbbv_0$ é uma *valoração booleana* se $\Bbbv_0$ é uma função tal que $\Bbbv_0 : \mathcal{P} \longmapsto \{V, F\}$.

Seja $\Bbbv_0$ uma valoração booleana. Dizemos que $\Bbbv$ é uma *interpretação definida por* $\Bbbv_0$ se $\Bbbv$ é uma função tal que $\Bbbv : \mathcal{L} \longmapsto \{V, F\}$ e:

1. Se $\phi_1 \in \mathcal{P}$, então $\Bbbv(\phi_1) = \Bbbv_0(\phi_1)$.
2. $\Bbbv(\neg\phi_1) = V$ se, e somente se, $\Bbbv(\phi_1) = F$.
3. $\Bbbv(\phi_1 \wedge ... \wedge \phi_n) = V$ se, e somente se, $\Bbbv(\phi_i) = V$, para todo i.
4. $\Bbbv(\phi_1 \vee ... \vee \phi_n) = V$ se, e somente se, $\Bbbv(\phi_i) = V$, para algum i.
5. $\Bbbv(\phi_1 \to \phi_2) = V$ se, e somente se, $\Bbbv(\phi_1) = F$ ou $\Bbbv(\phi_2) = V$.
6. $\Bbbv(\phi_1 \leftrightarrow \phi_2) = V$ se, e somente se, $\Bbbv(\phi_1) = \Bbbv(\phi_2)$.
7. $\Bbbv(\top) = V$.
8. $\Bbbv(\bot) = F$.

No que segue, usaremos tão somente *interpretação*, ao invés de *interpretação definida por* $\Bbbv_0$.

Exemplo 2 Seja a interpretação $\Bbbv$ definida por $\Bbbv_0 = \{(p, V), (q, F), (r, V), (s, V)\}$ e considere a fórmula $\phi = \neg((p \vee (q \wedge r \wedge s)) \leftrightarrow (q \to \neg s))$. Temos que:

1. $\Bbbv(q \wedge r \wedge s) = F$
2. $\Bbbv(p \vee (q \wedge r \wedge s)) = V$
3. $\Bbbv(\neg s) = F$
4. $\Bbbv(q \to \neg s) = V$

5. $\mathbb{v}((p \vee (q \wedge r \wedge s)) \leftrightarrow (q \rightarrow \neg s)) = V$
6. $\mathbb{v}(\phi) = F$

Definição 3 Se existe uma interpretação $\mathbb{v}$ tal que $\mathbb{v}(\phi) = V$, então dizemos que $\mathbb{v}$ *satisfaz* ϕ, ou ainda, que ϕ é *satisfatível*. De maneira análoga, se $\mathbb{v}$ é tal que $\mathbb{v}(\phi) = F$, então dizemos que $\mathbb{v}$ *falsifica* ϕ, ou ainda, que ϕ é *falsificável*.

Se toda interpretação satisfaz ϕ, então dizemos que ϕ é uma *tautologia*. Por outro lado, se toda interpretação falsifica ϕ, ou seja, se nenhuma interpretação satisfaz ϕ (logo ϕ não é satisfatível), então dizemos que ϕ é uma *contradição*, ou que ϕ é *insatisfatível*.

Se ϕ é simultaneamente satisfatível e falsificável, então dizemos que ϕ é uma *contingência*.

Exemplo 3 São tautologias:

- $\phi \vee \neg\phi$
- $\phi \rightarrow \phi$
- $\phi \leftrightarrow \phi$
- $\top$

São contradições:

- $\phi \wedge \neg\phi$
- $\phi \leftrightarrow \neg\phi$
- $\bot$

São contingências:

- p
- $\neg p$
- $p \wedge q$
- $p \vee q$

- $p \rightarrow q$
- $p \leftrightarrow q$

2.2 Problemas da lógica proposicional

Apresentamos nesta seção os principais problemas envolvendo a lógica proposicional, que são o alvo deste trabalho.

Definição 4 Seja L um conjunto de cadeias sobre um alfabeto. Se nos referimos a L como um *problema*, referimo-nos ao problema de decidir se uma dada cadeia w pertence a L. Ou seja, referimo-nos a L como um *problema de decisão*.

Dizemos que um problema L é *decidível* quando existe um algoritmo A tal que:

1. A realiza um número finito de passos sobre a entrada w e responde "sim", para toda cadeia $w \in L$.
2. A realiza um número finito de passos sobre a entrada w e responde "não", para toda cadeia $w \notin L$.

Neste caso, dizemos que A *decide* L, ou ainda que A é um *decisor* para L.

Denotamos por $\overline{L}$ o complemento de L, ou seja, o conjunto $\{w \mid w \notin L\}$.

Definição 5 Definimos SAT $= \{\phi \in \mathcal{L} \mid \phi \text{ é satisfatível}\}$ como o problema da *satisfatibilidade* e UNSAT $= \{\phi \in \mathcal{L} \mid \phi \text{ é insatisfatível}\} = \{\phi \in \mathcal{L} \mid \phi \notin \text{SAT}\} = \overline{SAT}$ como o problema da *insatisfatibilidade*.

Definimos ainda VAL $= \{\phi \in \mathcal{L} \mid \phi \text{ é tautologia}\}$ como o problema da *validade*.

Observe que UNSAT $= \{\phi \in \mathcal{L} \mid \phi \text{ é contradição}\}$.

O conceito de *fórmula válida* é usualmente definido através do conceito de *consequência lógica*. Como estes dois conceitos não são necessários para este trabalho, não incluímos suas definições neste texto. Entretanto, é possível mostrar que o conjunto das tautologias é igual ao conjunto das fórmulas válidas. Por esta razão e por ser o uso mais famoso, referimo-nos ao problema de decidir se uma fórmula é uma tautologia por *problema da validade*.

Davis et al. apresentam um algoritmo que decide SAT [6]. Além disso, é claro que se um problema é decidível, então o seu complemento também é. Isto é, com um algoritmo que decide SAT, é claro que temos um algoritmo para decidir UNSAT. De forma geral,

podemos dizer que um problema decidível e seu complemento são *redutíveis* um ao outro, ou seja, podemos construir um decisor para $\overline{L}$ usando um decisor para L e vice-versa.

Mostramos agora que SAT e VAL são redutíveis um ao outro, reduzindo VAL a UNSAT e vice-versa. O teorema que apresentamos a seguir é útil nesta tarefa.

Teorema 1 Se ϕ é uma:

1. tautologia, então $\neg\phi$ é uma contradição.
2. contradição, então $\neg\phi$ é uma tautologia.
3. contingência, então $\neg\phi$ é uma contingência.

Se ϕ é uma tautologia, então $\mathbb{v}(\phi) = V, \forall \mathbb{v}$. Logo $\mathbb{v}(\neg\phi) = F, \forall \mathbb{v}$. Portanto, $\neg\phi$ é uma contradição.

Se ϕ é uma contradição, então $\mathbb{v}(\phi) = F, \forall \mathbb{v}$. Logo $\mathbb{v}(\neg\phi) = V, \forall \mathbb{v}$. Portanto, $\neg\phi$ é uma tautologia.

Se ϕ é uma contingência, então existem interpretações $\mathbb{v}_1, \mathbb{v}_2$ tais que $\mathbb{v}_1(\phi) = V$ e $\mathbb{v}_2(\phi) = F$. Logo, $\mathbb{v}_1(\neg\phi) = F$ e $\mathbb{v}_2(\neg\phi) = V$. Portanto, $\neg\phi$ é uma contingência.

Com isto, provamos o Teorema 1.

Agora, seja A_1 um decisor para UNSAT e considere o seguinte algoritmo, que chamaremos de R_1: "Sobre a entrada $\phi \in \mathcal{L}$, dê a resposta de A_1 sobre a entrada $\neg\phi$."

Vamos examinar o comportamento de R_1 para todas as possibilidades, ou seja, $\forall \phi \in \mathcal{L}$.

Quando ϕ é uma contradição ou uma contingência, do Teorema 1, temos que $\neg\phi$ é uma tautologia ou uma contingência, respectivamente. Em ambos os casos, R_1 responde "não", pois A_1 responde "não" sobre a entrada $\neg\phi$. Observe ainda que, em ambos os casos, $\phi \notin$ VAL.

Quando ϕ é uma tautologia (logo $\phi \in$ VAL), temos que $\neg\phi$ é uma contradição. Neste caso, R_1 responde "sim", pois A_1 responde "sim" sobre a entrada $\neg\phi$.

Mostramos então que R_1 decide VAL, ou seja, VAL é redutível a UNSAT.

Seja agora A_2 um decisor para VAL e considere o seguinte algoritmo, que chamaremos de R_2: "Sobre a entrada $\phi \in \mathcal{L}$, dê a resposta de A_2 sobre a entrada $\neg\phi$."

Quando ϕ é uma tautologia ou uma contingência (ou seja, $\phi \notin$ UNSAT), temos que $\neg\phi$ é uma contradição ou uma contingência, respectivamente. Em ambos os casos, R_2 responde "não", pois A_2 responde "não" sobre a entrada $\neg\phi$.

Finalmente, quando ϕ é uma contradição, ou seja, quando $\phi \in$ UNSAT, temos que $\neg\phi$ é uma tautologia. Neste caso, R_2 responde "sim", pois A_2 responde "sim" sobre a entrada $\neg\phi$.

Mostramos então que R_2 decide UNSAT, ou seja, UNSAT é redutível a VAL.

Com as *reduções* R_1 e R_2 e com o fato de que SAT e UNSAT são redutíveis um ao outro, mostramos que VAL e SAT são também redutíveis um ao outro.

2.3 Formas normais

Esta seção apresenta as definições e resultados que envolvem formas normais, conceito chave para este trabalho.

Definição 6 Seja ϕ uma fórmula. Definimos $tam(\phi)$, o *tamanho* de ϕ, como o seguinte número:

1. Se $\phi \in \mathcal{P}$, então $tam(\phi) = 1$.
2. $tam(\neg\phi) = 1 + tam(\phi)$.
3. $tam(\phi_1 \wedge ... \wedge \phi_n) = tam(\phi_1 \vee ... \vee \phi_n) = n - 1 + \sum_i tam(\phi_i)$.
4. $tam(\phi_1 \rightarrow \phi_2) = tam(\phi_1 \leftrightarrow \phi_2) = 1 + tam(\phi_1) + tam(\phi_2)$.
5. $tam(\top) = tam(\bot) = 1$.

Exemplo 4 Seja $\phi = p \rightarrow (\neg r \vee (q \leftrightarrow s))$. Então

$$\begin{aligned} tam(\phi) &= 1 + tam(p) + tam(\neg r \vee (q \leftrightarrow s)) \\ &= 1 + 1 + (1 + tam(\neg r) + tam(q \leftrightarrow s)) \\ &= 1 + 1 + (1 + (1 + tam(r)) + (1 + tam(q) + tam(s))) \\ &= 1 + 1 + (1 + (1 + 1) + (1 + 1 + 1)) \\ &= 8 \end{aligned}$$

Definição 7 Dizemos que ϕ é *subfórmula* de ψ, escrito $\phi \sqsubseteq \psi$, se, e somente se, alguma das possibilidades ocorre:

1. $\phi = \psi$.
2. ϕ é subfórmula imediata de ψ.
3. ϕ é subfórmula de ξ e ξ é subfórmula imediata de ψ.

Denotamos o conjunto $\{\psi \mid \psi \sqsubseteq \phi\}$ das subfórmulas de ϕ por $SF(\phi)$.

Se $\phi \sqsubseteq \psi$ e $\phi \neq \psi$, então dizemos que ϕ é *subfórmula própria* de ψ e escrevemos $\phi \sqsubset \psi$.

Denotamos o conjunto $\{\psi \mid \psi \sqsubset \psi\}$ das subfórmulas próprias de ϕ por $SFP(\phi)$.

Exemplo 5 Considere a fórmula $\phi = \neg((p \vee (q \wedge r \wedge s)) \leftrightarrow (q \rightarrow \neg s))$. Note que:

- A única subfórmula imediata de ϕ é $\phi_1 = (p \vee (q \wedge r \wedge s)) \leftrightarrow (q \rightarrow \neg s)$.
- $SF(\phi) = \{p, q, r, s, \neg s, q \wedge r \wedge s, q \rightarrow \neg s, p \vee (q \wedge r \wedge s), \phi_1, \phi\}$.
- $SFP(\phi) = SF(\phi) - \{\phi\}$; e, por definição, isto é verdade para toda ϕ.

Definição 8 Uma *posição* é uma sequência finita de números naturais. Usaremos as notações alternativas ε, para a posição vazia (), e $a_1.\cdots.a_n$, para a posição $(a_1, \cdots, a_n)$, onde $n \in \mathbb{N} \cup \{0\}$. Além disso, se $\pi = a_1.\cdots.a_n$ é uma posição e i é um número natural, então $i.\pi$ denota a posição $i.a_1.\cdots.a_n$ e $\pi.i$ denota a posição $a_1.\cdots.a_n.i$.

Definimos o *conjunto de posições* de uma fórmula ϕ, $pos(\phi)$, da seguinte maneira:

1. Se $\phi \in \mathcal{P}$, então $pos(\phi) = \{\varepsilon\}$.
2. Se ϕ é da forma $\neg\phi_1$, $\phi_1 \wedge ... \wedge \phi_n$, $\phi_1 \vee ... \vee \phi_n$, $\phi_1 \rightarrow \phi_2$, ou $\phi_1 \leftrightarrow \phi_2$, então

$$pos(\phi) = \{\varepsilon\} \cup \left(\bigcup_i \{i.\pi \mid \pi \in pos(\phi_i)\} \right)$$

Agora, definimos a *subfórmula de ϕ começando na posição π*, escrito $\phi|_\pi$, da seguinte forma:

1. Se $\pi = \varepsilon$, então $\phi|_\pi = \phi$.
2. Se ϕ é da forma $\neg\phi_1$, $\phi_1 \wedge ... \wedge \phi_n$, $\phi_1 \vee ... \vee \phi_n$, $\phi_1 \rightarrow \phi_2$, ou $\phi_1 \leftrightarrow \phi_2$, e π é da forma $i.\pi'$, para algum natural i e alguma posição $\pi' \in pos(\phi_i)$, então $\phi|_\pi = \phi_i|_{\pi'}$.

Exemplo 6 Seja $\phi = p \vee (q \wedge \neg r)$. Observe que:

$$\begin{aligned} pos(\neg r) &= \{\varepsilon\} \cup \{1.\pi \mid \pi \in pos(r)\} \\ &= \{\varepsilon\} \cup \{1.\pi \mid \pi \in \{\varepsilon\}\} \\ &= \{\varepsilon, 1\} \end{aligned}$$

$$\begin{aligned} pos(q \wedge \neg r) &= \{\varepsilon\} \cup \{1.\pi \mid \pi \in pos(q)\} \cup \{2.\pi \mid \pi \in pos(\neg r)\} \\ &= \{\varepsilon\} \cup \{1.\pi \mid \pi \in \{\varepsilon\}\} \cup \{2.\pi \mid \pi \in \{\varepsilon, 1\}\} \\ &= \{\varepsilon, 1, 2, 2.1\} \end{aligned}$$

$$\begin{aligned} pos(\phi) &= \{\varepsilon\} \cup \{1.\pi \mid \pi \in pos(p)\} \cup \{2.\pi \mid \pi \in pos(q \wedge \neg r)\} \\ &= \{\varepsilon\} \cup \{1.\pi \mid \pi \in \{\varepsilon\}\} \cup \{2.\pi \mid \pi \in \{\varepsilon, 1, 2, 2.1\}\} \\ &= \{\varepsilon, 1, 2, 2.1, 2.2, 2.2.1\} \end{aligned}$$

Além disso, note que:

- $\phi|_\varepsilon = \phi = p \vee (q \wedge \neg r)$
- $\phi|_1 = p|_\varepsilon = p$
- $\phi|_2 = (q \wedge \neg r)|_\varepsilon = q \wedge \neg r$
- $\phi|_{2.1} = (q \wedge \neg r)|_1 = q|_\varepsilon = q$
- $\phi|_{2.2} = (q \wedge \neg r)|_2 = (\neg r)|_\varepsilon = \neg r$
- $\phi|_{2.2.1} = (q \wedge \neg r)|_{2.1} = (\neg r)|_1 = r|_\varepsilon = r$

Observe que $\{\phi|_\pi \mid \pi \in pos(\phi)\} = SF(\phi)$.

Definição 9 Definimos a *polaridade da subfórmula de ϕ começando na posição π*, escrito $pol(\phi, \pi)$, como o seguinte número:

1. $pol(\phi, \varepsilon) = 1$.
2. Se $\phi|_\pi$ é da forma $\neg\phi_1$, então $pol(\phi, \pi.1) = -pol(\phi, \pi)$.
3. Se $\phi|_\pi$ é da forma $\phi_1 \wedge ... \wedge \phi_n$, ou $\phi_1 \vee ... \vee \phi_n$, então $pol(\phi, \pi.i) = pol(\phi, \pi)$, para $i = 1, ..., n$.
4. Se $\phi|_\pi$ é da forma $\phi_1 \to \phi_2$, então $pol(\phi, \pi.1) = -pol(\phi, \pi)$ e $pol(\phi, \pi.2) = pol(\phi, \pi)$.

5. Se $\phi|_\pi$ é da forma $\phi_1 \leftrightarrow \phi_2$, então $pol(\phi, \pi.1) = pol(\phi, \pi.2) = 0$.

Exemplo 7 Seja $\phi = (p \rightarrow q) \rightarrow \neg(p \leftrightarrow (r \vee s))$. Temos que:

- $pol(\phi, \varepsilon) = 1$
- $pol(\phi, 1) = -1$
- $pol(\phi, 1.1) = 1$
- $pol(\phi, 1.2) = -1$
- $pol(\phi, 2) = 1$
- $pol(\phi, 2.1) = -1$
- $pol(\phi, 2.1.1) = 0$
- $pol(\phi, 2.1.2) = 0$
- $pol(\phi, 2.1.2.1) = 0$
- $pol(\phi, 2.1.2.2) = 0$

Definição 10 Denotamos por

$$\phi \longmapsto \psi$$

a *regra de reescrita* que transforma uma fórmula da forma de ϕ em ψ.

Se uma regra de reescrita transforma ϕ em ψ, dizemos que esta regra:

1. *preserva equivalência* se, e somente se, $\Bbbv(\phi) = \Bbbv(\psi), \forall \Bbbv$, ou seja, $\phi \leftrightarrow \psi \in \text{VAL}$.
2. *preserva satisfatibilidade* se, e somente se, $\phi \in \text{SAT} \iff \psi \in \text{SAT}$.

Definição 11 Dizemos que uma fórmula ϕ está na *forma normal negada* (FNN) se, e somente se, ϕ não contém implicações, não contém equivalências e negações ocorrem somente em símbolos proposicionais.

Teorema 2 As transformações

1. $\neg\neg\phi_1 \longmapsto \phi_1$ (eliminação de dupla negação)
2. $\neg(\phi_1 \wedge ... \wedge \phi_n) \longmapsto \neg\phi_1 \vee ... \vee \neg\phi_n$ (De Morgan)
3. $\neg(\phi_1 \vee ... \vee \phi_n) \longmapsto \neg\phi_1 \wedge ... \wedge \neg\phi_n$ (De Morgan)
4. $\phi_1 \rightarrow \phi_2 \longmapsto \neg\phi_1 \vee \phi_2$
5. Se $\phi|_\pi$ é da forma $\phi_1 \leftrightarrow \phi_2$, então
   (a) $\phi|_\pi \longmapsto (\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1)$, se $pol(\phi, \pi) = 1$
   (b) $\phi|_\pi \longmapsto (\phi_1 \wedge \phi_2) \vee (\neg\phi_2 \wedge \neg\phi_1)$, se $pol(\phi, \pi) = -1$

preservam equivalência e produzem fórmulas na FNN.

A prova do Teorema 2 segue por indução na estrutura de uma fórmula.

A transformação de equivalências dependente de polaridade do Teorema 2 evita que tautologias difíceis de detectar apareçam nas fórmulas transformadas, como mostra o Exemplo 8, dado por Nonnengart et al. [13]. Observe que não é necessário considerar o caso em que a polaridade é zero, pois, para evitar este caso, basta transformar equivalências em posições mais curtas primeiro.

Definição 12 Dizemos que ϕ é um *literal* se, e somente se, $\phi \in \mathcal{P}$, ou ϕ é da forma $\neg p$, onde $p \in \mathcal{P}$.

Dizemos que uma disjunção de literais é uma *cláusula*.

Dizemos que uma fórmula ϕ está na *forma normal clausal* (FNC) se, e somente se, ϕ é uma conjunção de cláusulas.

Teorema 3 A transformação

$$\phi \vee \left(\bigwedge_i \phi_i\right) \longmapsto \bigwedge_i (\phi \vee \phi_i)$$

chamada de *distribuição*, preserva equivalência e, se aplicada a fórmulas na FNN, produz fórmulas na FNC.

A prova do Teorema 3 segue por indução na estrutura de uma fórmula.

Exemplo 8 Considere ϕ da forma $\neg(\phi_1 \leftrightarrow \phi_2)$. Aplicando as transformações dos Teoremas 2 e 3 à exaustão, começando pela transformação do item 5.a do Teorema

2 e então aplicando distribuição, temos:

$$\begin{aligned}
\neg(\phi_1 \leftrightarrow \phi_2) &\longmapsto \neg((\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1)) \\
&\longmapsto \neg((\neg\phi_1 \vee \phi_2) \wedge (\neg\phi_2 \vee \phi_1)) \\
&\longmapsto \neg(\neg\phi_1 \vee \phi_2) \vee \neg(\neg\phi_2 \vee \phi_1) \\
&\longmapsto (\neg\neg\phi_1 \wedge \neg\phi_2) \vee (\neg\neg\phi_2 \wedge \neg\phi_1) \\
&\longmapsto (\phi_1 \wedge \neg\phi_2) \vee (\phi_2 \wedge \neg\phi_1) \\
&\longmapsto ((\phi_1 \wedge \neg\phi_2) \vee \phi_2) \wedge ((\phi_1 \wedge \neg\phi_2) \vee \neg\phi_1) \\
&\longmapsto (\phi_1 \vee \phi_2) \wedge (\neg\phi_2 \vee \phi_2) \wedge (\phi_1 \vee \neg\phi_1) \wedge (\neg\phi_2 \vee \neg\phi_1)
\end{aligned}$$

Se $\phi_1, \phi_2 \in \mathcal{P}$, então a última fórmula já está na FNC, de modo que é fácil identificar e remover as tautologias $\neg\phi_2 \vee \phi_2$ e $\phi_1 \vee \neg\phi_1$. Caso contrário, as transformações aplicadas à exaustão transformam $\neg\phi_i$ em uma fórmula $\psi \neq \neg\phi_i$, dificultando identificar e remover as tautologias mencionadas.

Considere agora uma transformação que leva em conta polaridade, ou seja, desta vez começamos com a transformação do item 5.b do Teorema 2.

$$\begin{aligned}
\neg(\phi_1 \leftrightarrow \phi_2) &\longmapsto \neg((\phi_1 \wedge \phi_2) \vee (\neg\phi_1 \wedge \neg\phi_2)) \\
&\longmapsto \neg(\phi_1 \wedge \phi_2) \wedge \neg(\neg\phi_1 \wedge \neg\phi_2) \\
&\longmapsto (\neg\phi_1 \vee \neg\phi_2) \wedge (\neg\neg\phi_1 \vee \neg\neg\phi_2) \\
&\longmapsto (\neg\phi_1 \vee \neg\phi_2) \wedge (\phi_1 \vee \phi_2)
\end{aligned}$$

Agora, o número de passos de transformação é menor, o tamanho da fórmula resultante é menor e as tautologias indesejadas não aparecem.

Por fim, note que os fatos ilustrados por este exemplo ocorrem para qualquer subfórmula da forma $\phi_1 \leftrightarrow \phi_2$ ocorrendo com polaridade negativa em qualquer posição de ϕ, $\forall\phi, \phi_1, \phi_2 \in \mathcal{L}$.

2.4 Renomeamento

Nesta seção introduzimos a transformação feita com renomeamento, que utilizamos para reduzir o tamanho de uma fórmula neste trabalho.

Definição 13 Sejam ϕ e ψ fórmulas tais que $\psi \in SFP(\phi)$ e $p \in \mathcal{P}$ tal que p não ocorre em ϕ. A *substituição de ψ por p em ϕ*, denotada por $rep(\phi, \psi, p)$, é a fórmula ϕ com todas as ocorrências de ψ trocadas por p.

Um *renomeamento de ϕ* é um conjunto de subfórmulas próprias de ϕ.

Seja R um renomeamento de ϕ. Uma *substituição de R em ϕ* é uma função injetora $s : R \longmapsto \mathcal{P}$ tal que p não ocorre em ϕ, $\forall p \in \mathrm{Im}(s)$.

Seja $s = \{(\phi_1, p_1), ..., (\phi_n, p_n)\}$ uma substituição de $R = \{\phi_1, ..., \phi_n\}$ em ϕ. Definimos

$$rep(\phi, s) = \begin{cases} \phi & \text{se } n = 0 \\ rep(rep(\phi, \phi_1, p_1), \{(\phi_2, p_2), ..., (\phi_n, p_n)\}) & \text{se } n > 0 \end{cases}$$

Exemplo 9 Seja $\phi = (p \vee q) \rightarrow (r \wedge (p \vee q) \wedge (p \wedge q))$ e $s = \{(p \vee q, a), (p \wedge q, b)\}$. Então

$$rep(\phi, s) = a \rightarrow (r \wedge a \wedge b)$$

Definição 14 Sejam ϕ e ψ tais que $\psi \sqsubseteq \phi$ e s uma substituição em ϕ tal que $(\psi, p) \in s$. Denote ainda $rep(\psi, s - \{(\psi, p)\})$ por ξ. Definimos

$$def(\phi, \psi, s) = \begin{cases} p \rightarrow \xi & \text{se } pol(\phi, \pi) = 1, \forall \pi \in pos(\phi) \text{ tal que } \phi|_\pi = \psi \\ \xi \rightarrow p & \text{se } pol(\phi, \pi) = -1, \forall \pi \in pos(\phi) \text{ tal que } \phi|_\pi = \psi \\ p \leftrightarrow \xi & \text{caso contrário} \end{cases}$$

Teorema 4 Seja $s = \{(\phi_1, p_1), ..., (\phi_n, p_n)\}$ uma substituição de $R = \{\phi_1, ..., \phi_n\}$ em ϕ, onde R é um renomeamento de ϕ. Então, a transformação

$$\phi \longmapsto \mathcal{R}(\phi, s)$$

onde

$$\mathcal{R}(\phi, s) = rep(\phi, s) \wedge def(\phi, \phi_1, s) \wedge ... \wedge def(\phi, \phi_n, s)$$

que chamaremos de *transformação por renomeamento*, preserva satisfatibilidade.

Plaisted et al. provam o Teorema 4 [14].

Observe que, dado um renomeamento R, encontrar uma substituição de R em ϕ para uma transformação por renomeamento é uma tarefa fácil, pois $\mathcal{P}$ é infinito e enumerável,

enquanto o conjunto dos símbolos proposicionais que ocorrem em ϕ é finito.

Exemplo 10 Considere $\phi = (p \leftrightarrow q) \leftrightarrow (p \leftrightarrow q)$ e $s = \{(p \leftrightarrow q, r)\}$. Então

$$\mathcal{R}(\phi, s) = (r \leftrightarrow r) \wedge (r \leftrightarrow (p \leftrightarrow q))$$

Note que tanto ϕ quanto $\mathcal{R}(\phi, s)$ são satisfatíveis, mas que ϕ é uma tautologia da forma $\psi \leftrightarrow \psi$, enquanto $\mathcal{R}(\phi, s)$ é uma contingência, pois

1. Se $\mathbb{v}(r) = V$, $\mathbb{v}(p) = V$ e $\mathbb{v}(q) = V$, então $\mathbb{v}(\mathcal{R}(\phi, s)) = V$.
2. Se $\mathbb{v}(r) = V$, $\mathbb{v}(p) = V$ e $\mathbb{v}(q) = F$, então $\mathbb{v}(\mathcal{R}(\phi, s)) = F$.

O Exemplo 10 mostra que transformações que preservam satisfatibilidade, diferentemente das que preservam equivalência, podem não preservar o significado de uma fórmula em todas as interpretações. No entanto, isto não é um obstáculo se quisermos determinar precisamente se uma fórmula é uma tautologia, uma contradição ou uma contingência. Isto segue do fato que mostramos na Seção 2.2: SAT, UNSAT e VAL são todos problemas redutíveis uns aos outros. Se uma transformação preserva satisfatibilidade, então é claro que as respostas de decisores para SAT e UNSAT são preservadas por esta transformação. Neste caso, as respostas das reduções que mostramos anteriormente também são preservadas. Portanto, com os procedimentos apropriados, é perfeitamente possível determinar o tipo de uma fórmula através de sua transformação, ou da transformação de sua negação. Este resultado é fundamental para que possamos utilizar renomeamento para obter fórmulas pequenas.

2.5 Reduzindo o número de cláusulas

Definição 15 Denotamos por $p(\phi)$ o *número de cláusulas* da fórmula obtida por aplicar as transformações dos Teoremas 2 e 3 à exaustão na fórmula ϕ. Denotamos ainda $p(\neg\phi)$ por $\overline{p}(\phi)$.

O número $p(\phi)$ pode ser calculado pela Tabela 2.1 [7].

Tabela 2.1: Número de cláusulas de uma fórmula.

Forma de ϕ	$p(\phi)$	$\overline{p}(\phi)$
$\neg\phi_1$	$\overline{p}(\phi_1)$	$p(\phi_1)$
$\phi_1 \wedge ... \wedge \phi_n$	$\sum_{i=1}^{n} p(\phi_i)$	$\prod_{i=1}^{n} \overline{p}(\phi_i)$
$\phi_1 \vee ... \vee \phi_n$	$\prod_{i=1}^{n} p(\phi_i)$	$\sum_{i=1}^{n} \overline{p}(\phi_i)$
$\phi_1 \rightarrow \phi_2$	$\overline{p}(\phi_1)p(\phi_2)$	$p(\phi_1) + \overline{p}(\phi_2)$
$\phi_1 \leftrightarrow \phi_2$	$\overline{p}(\phi_1)p(\phi_2) + \overline{p}(\phi_2)p(\phi_1)$	$p(\phi_1)p(\phi_2) + \overline{p}(\phi_1)\overline{p}(\phi_2)$
$\phi \in \mathcal{P}$	1	1

Denotaremos ainda $p(\mathcal{R}(\phi, s))$ simplesmente por $p(\phi, R)$, onde $R = D(s)$ é o renomeamento associado à substituição s.

Exemplo 11 Considere a fórmula $\phi = (r \leftrightarrow s) \leftrightarrow (r \leftrightarrow s)$. Primeiro, seja $\phi_1 = r \leftrightarrow s$. Então

$$p(\phi_1) = \overline{p}(r)p(s) + \overline{p}(s)p(r) = 1 \cdot 1 + 1 \cdot 1 = 2$$

e

$$\overline{p}(\phi_1) = p(r)p(s) + \overline{p}(s)\overline{p}(r) = 1 \cdot 1 + 1 \cdot 1 = 2$$

Agora,

$$p(\phi) = \overline{p}(\phi_1)p(\phi_1) + \overline{p}(\phi_1)p(\phi_1) = 2 \cdot 2 + 2 \cdot 2 = 8$$

Agora, dada uma fórmula ϕ, queremos escolher um renomeamento R de ϕ tal que o número $p(\phi, R)$ seja o menor possível.

O Algoritmo 1, apresentado por Boy de la Tour [7], encontra um renomeamento que produz o número ótimo (mínimo) de cláusulas, desde que $\pi_1 \neq \pi_2 \implies \phi|_{\pi_1} \neq \phi|_{\pi_2}$, ou seja, desde que qualquer subfórmula ocorra somente uma vez. Seu custo de tempo é $O(|pos(\phi)|^2)$ determinístico no pior caso. Os números $a_{\psi_i}^{\mathcal{R}(\phi,s)}$ e $b_{\psi_i}^{\mathcal{R}(\phi,s)}$ são computados a partir de $a = a_{\psi}^{\mathcal{R}(\phi,s)}$, $b = b_{\psi}^{\mathcal{R}(\phi,s)}$, $\psi_i.p$ e $\psi_i.\overline{p}$, seguindo as fórmulas da Tabela 2.2. A função $nbcl(\psi)$ calcula os números de cláusulas $p(\psi)$ e $\overline{p}(\psi)$ a partir de $\psi_i.p$ e $\psi_i.\overline{p}$.

Exemplo 12 Considere a fórmula $\phi = (x_1 \wedge x_2 \wedge x_3) \vee (y_1 \wedge y_2 \wedge y_3)$. Vamos executar $R\_rec(\phi, 1, 0, 1)$.

Na primeira chamada, a condição da linha 3 é satisfeita, pois $\psi.p = 9$. A condição da linha 4, no entanto, não é satisfeita, pois $a \cdot \psi.p + b \cdot \psi.\overline{p} = 9$ e $a + b + if\_pos(r, \psi.p) + if\_pos(-r, \psi.\overline{p}) = 10$. Portanto, a partir da linha 9, $\psi_1 = x_1 \wedge x_2 \wedge x_3$ e $\psi_2 = y_1 \wedge y_2 \wedge y_3$ são subfórmulas imediatas de $\psi = \phi$. Na linha 11, a função recursiva é chamada primeiro com $R\_rec(\psi_1, 3, 0, 1)$.

Tabela 2.2: Coeficientes a e b do Algoritmo 1.

Forma de ψ	$a^{\phi}_{\psi_i}$	$b^{\phi}_{\psi_i}$
$\neg\psi_1$	b^{ϕ}_{ψ}	a^{ϕ}_{ψ}
$\psi_1 \wedge ... \wedge \psi_n$	a^{ϕ}_{ψ}	$b^{\phi}_{\psi} \prod_{j \neq i} \overline{p}(\psi_j)$
$\psi_1 \vee ... \vee \psi_n$	$a^{\phi}_{\psi} \prod_{j \neq i} p(\psi_j)$	b^{ϕ}_{ψ}
$\psi_1 \rightarrow \psi_2, i = 1$	b^{ϕ}_{ψ}	$a^{\phi}_{\psi} p(\psi_2)$
$\psi_1 \rightarrow \psi_2, i = 2$	$a^{\phi}_{\psi} \overline{p}(\psi_1)$	b^{ϕ}_{ψ}
$\psi_1 \leftrightarrow \psi_2, j = 3 - i$	$a^{\phi}_{\psi} \overline{p}(\psi_j) + b^{\phi}_{\psi} p(\psi_j)$	$a^{\phi}_{\psi} p(\psi_j) + b^{\phi}_{\psi} \overline{p}(\psi_j)$
$\psi_i = \phi$	1	0

Algoritmo 1 Algoritmo de Boy de la Tour para encontrar renomeamentos.

seja R um conjunto vazio global
função $R\_rec(\psi, a, b, r)$
    se $(\psi.p, \psi.\overline{p}) \neq (1, 1)$ **então**
        se $a \cdot \psi.p + b \cdot \psi.\overline{p} > a + b + if\_pos(r, \psi.p) + if\_pos(-r, \psi.\overline{p})$ **então**
            $R \leftarrow R \cup \{\psi\}$
            $R\_rec(\psi, if\_pos(r, 1), if\_pos(-r, 1), r)$
            $(\psi.p, \psi.\overline{p}) \leftarrow (1, 1)$
        senão
            seja $SFI(\psi) = \{\psi_1, ..., \psi_n\}$
            para $i \leftarrow 1$ **até** n **faça**
                $R\_rec(\psi_i, a^{\mathcal{R}(\phi,s)}_{\psi_i}, b^{\mathcal{R}(\phi,s)}_{\psi_i}, r \cdot pol(\psi, i))$
            fim para
            $(\psi.p, \psi.\overline{p}) \leftarrow nbcl(\psi)$
        fim se
    fim se
fim função
função $if\_pos(x, y)$
    se $x \geq 0$ **então**
        retorne y
    fim se
    retorne 0
fim função
para cada $\psi \in SF(\phi)$ **faça**
    $(\psi.p, \psi.\overline{p}) \leftarrow (p(\psi), \overline{p}(\psi))$
fim para
$R\_rec(\phi, 1, 0, 1)$

Na segunda chamada, temos que $\psi = x_1 \wedge x_2 \wedge x_3$. A condição da linha 3 é satisfeita, pois $\psi.p = 3$. A condição da linha 4 é satisfeita, pois $a \cdot \psi.p + b \cdot \psi.\overline{p} = 9$ e $a + b + if\_pos(r, \psi.p) + if\_pos(-r, \psi.\overline{p}) = 6$. Portanto, após a linha 5, teremos $R = \{x_1 \wedge x_2 \wedge x_3\}$. A linha 6 é executada: $R\_rec(\psi, 1, 0, 1)$.

Na terceira chamada, a condição da linha 3 é novamente satisfeita, pois $\psi.p$ ainda vale 3. A condição da linha 4, no entanto, não é satisfeita, pois agora $a \cdot \psi.p + b \cdot \psi.\overline{p} = 3$ e $a + b + if\_pos(r, \psi.p) + if\_pos(-r, \psi.\overline{p}) = 4$. Portanto, a partir da linha 9, temos $\psi_i = x_i$, para $i = 1, 2, 3$. Assim, o algoritmo recursivo é chamado três vezes, para três literais. Mas é fácil ver que o algoritmo nunca escolhe literais para serem renomeados. Portanto, a terceira chamada retorna para a segunda, que está agora na linha 7.

De volta à segunda chamada, na linha 7, os valores $\psi.p$ e $\psi.\overline{p}$ são atualizados. Isto encerra a segunda chamada recursiva.

De volta à primeira chamada, a linha 11 é agora executada com $R\_rec(\psi_2, 1, 0, 1)$. No entanto, já vimos que uma chamada com estes parâmetros e ψ da forma $y_1 \wedge y_2 \wedge y_3$ não modifica o conjunto R.

O algoritmo termina então com $R = \{x_1 \wedge x_2 \wedge x_3\}$.

É fácil ver que $p(\phi, R) = 6$. Para ver que este resultado é ótimo, observe que basta considerar a inclusão das subfórmulas $\phi_1 = x_1 \wedge x_2 \wedge x_3$ e $\phi_2 = y_1 \wedge y_2 \wedge y_3$, pois qualquer outra subfórmula é um símbolo proposicional ou a própria ϕ. Se incluímos ambas ϕ_1 e ϕ_2, a transformação gera 7 cláusulas. Se $R = \emptyset$, a fórmula permanece com 9 cláusulas. Se somente ϕ_1, ou somente ϕ_2 for incluída, a transformação gera 6 cláusulas. Logo, um renomeamento ótimo gera 6 cláusulas e o Algoritmo 1 é capaz de encontrar um destes renomeamentos.

No Capítulo 3, que vem a seguir, apresentamos um algoritmo de custo de tempo igualmente polinomial determinístico, mas que, para algumas famílias de fórmulas, encontra renomeamentos melhores que os construídos pelo Algoritmo 1.

Capítulo 3

Um algoritmo para escolher renomeamentos

Afirmação 1 Seja $R \subseteq SFP(\phi)$ um renomeamento ótimo entre os que contêm no máximo j subfórmulas e seja $\psi \in R$. Então, permitindo que só as subfórmulas em $SFP(\phi) - \{\psi\}$ sejam escolhidas, $R - \{\psi\}$ é um renomeamento ótimo entre os que contêm no máximo $j - 1$ subfórmulas.

Contraexemplo: Seja $\phi = ((p_1 \wedge p_2 \wedge p_3 \wedge p_4) \vee (q_1 \wedge q_2)) \wedge ((r_1 \wedge r_2) \vee (s_1 \wedge ... \wedge s_{100}))$. Então $p(\phi) = 208$.

Vamos calcular um renomeamento ótimo para ϕ que tenha no máximo 2 subfórmulas.

Observe primeiro que se ψ é da forma $\xi_1 \vee \xi_2$, onde ξ_1 e ξ_2 são conjunções de literais todos distintos entre si, então:

1. $p(\psi) = p(\xi_1)p(\xi_2)$;
2. $p(\psi, \{\xi_1\}) = p(\psi, \{\xi_2\}) = p(\xi_1) + p(\xi_2)$; e
3. $p(\psi, \{\xi_1, \xi_2\}) = 1 + p(\xi_1) + p(\xi_2)$

Ou seja, se $p(\xi_1), p(\xi_2) \geq 2$, então $\{\xi_1\}$ e $\{\xi_2\}$ são renomeamentos ótimos para ψ. Em outras palavras, em uma disjunção de duas conjunções de literais, basta renomear uma das conjunções para obter um renomeamento ótimo.

Sejam então $\xi_1 = p_1 \wedge p_2 \wedge p_3 \wedge p_4$, $\xi_2 = q_1 \wedge q_2$, $\xi_3 = q_1 \wedge q_2$ e $\xi_4 = s_1 \wedge ... \wedge s_{100}$. Usando o fato acima, temos que $R = \{\xi_1, \xi_4\}$ é ótimo e $|R| \leq 2$. No entanto,

$$p(\phi, R - \{\xi_4\}) = 206 > 110 = p(\phi, \{\xi_3\})$$

Ou seja, $R' = R - \{\xi_4\} = \{\xi_1\}$ é tal que $\xi_4 \notin R'$ e $|R'| \leq 1$, mas R' não é ótimo.

Com isto, provamos que a Afirmação 1 é falsa, até mesmo para fórmulas que satisfazem $\pi_1 \neq \pi_2 \implies \phi|_{\pi_1} \neq \phi|_{\pi_2}$, como a do contraexemplo.

Se a Afirmação 1 fosse verdadeira, ela caracterizaria o que chamamos de *subestrutura ótima* do problema de encontrar renomeamentos que geram o menor número possível de cláusulas. Ou seja, quando uma solução ótima para um problema é sempre feita de soluções ótimas para subproblemas, isto é, para versões menores do problema, dizemos que o problema exibe subestrutura ótima.

Propomos agora uma solução baseada na Afirmação 1 adaptada como uma heurística. A Afirmação 1 não é verdade. Ou seja, como vimos, $R - \{\psi\}$ não é necessariamente ótimo para a versão menor do problema. Mas a heurística é: $R - \{\psi\}$ pode não ser ótimo, mas é, em grande parte das vezes, "muito bom". Verificamos este fato experimentalmente, comparando o algoritmo aqui proposto com um outro, e apresentamos os resultados no Capítulo 4.

3.1 Uma fórmula recursiva

Nesta seção, tomamos a Afirmação 1 como verdade para construir uma fórmula recursiva que encontra bons renomeamentos. Como provamos anteriormente que a afirmação é falsa para algumas fórmulas, deve ficar claro que aqui a usamos como uma heurística, ou seja, quando nos referirmos nesta seção a um renomeamento ótimo, referimo-nos na verdade a um renomeamento que "provavelmente é muito bom".

Seja então $SFP(\phi) = \{\phi_1, ..., \phi_n\}$ e defina $f(i, j)$ como um renomeamento ótimo que contém no máximo j subfórmulas, considerando somente as subfórmulas em $\{\phi_1, ..., \phi_i\}$.

Por definição, é claro que $f(i, 0) = f(0, j) = \emptyset, \forall i, j$.

Agora, note que, ou $\phi_i \in f(i, j)$, ou $\phi_i \notin f(i, j)$.

Suponha que $\phi_i \in f(i, j)$. Neste caso, a Afirmação 1 nos garante que $f(i, j) - \{\phi_i\}$ é um renomeamento ótimo com no máximo $j - 1$ subfórmulas, considerando somente as subfórmulas em $\{\phi_1, ..., \phi_{i-1}\}$, ou seja, $p(\phi, f(i, j) - \{\phi_i\}) = p(\phi, f(i - 1, j - 1))$.

Suponha agora que $\phi_i \notin f(i, j)$. Neste caso, é claro que $p(\phi, f(i, j)) = p(\phi, f(i-1, j))$.

Portanto, podemos definir $f(i, j)$ da seguinte maneira. Se $i > 0$ e $j > 0$, então

$$f(i,j) = \begin{cases} f(i-1, j-1) \cup \{\phi_i\} & \text{se } p(\phi, f(i-1, j-1) \cup \{\phi_i\}) < p(\phi, f(i-1, j)) \\ f(i-1, j) & \text{caso contrário} \end{cases}$$

Observe que não há dependência cíclica na definição da fórmula, pois $f(i, j)$ depende somente de $f(i - 1, k)$, para todo $i > 0$ e $k = j - 1, j$.

Para obter então um renomeamento ótimo considerando todas as subfórmulas próprias de ϕ, basta calcular $f(n, n)$.

Observe ainda que resultados intermediários da fórmula são utilizados múltiplas vezes. Por exemplo, suponha que $n = 3$. Neste caso, $f(n, n)$ depende de $f(2, 2)$ e $f(2, 3)$. Agora, $f(2, 2)$ depende de $f(1, 1)$ e $f(1, 2)$; e $f(2, 3)$ depende de $f(1, 2)$ e $f(1, 3)$. Ou seja, tanto $f(2, 2)$ quanto $f(2, 3)$ dependem de $f(1, 2)$. Portanto, o valor de $f(1, 2)$ é usado no mínimo duas vezes para calcular $f(n, n) = f(3, 3)$. Chamamos esta propriedade de *sobreposição de subproblemas*.

Quando expressamos uma solução para um problema desta maneira, onde ficam evidentes a subestrutura ótima e a sobreposição de subproblemas, podemos aplicar *programação dinâmica* [1]. Começando pelos subproblemas menores, calculamos as soluções de cada subproblema uma única vez e, no fim da computação, calculamos a solução do problema principal.

Portanto, se a Afirmação 1 fosse verdadeira, o algoritmo que propomos a seguir seria considerado uma programação dinâmica implementada por abordagem ascendente [1].

3.2 Uma implementação por computação ascendente

O Algoritmo 2 calcula e armazena $f(n, n)$ em $dp[n]$.

Algoritmo 2 Computação ascendente de $f(n, n)$.

```
1: seja dp[0..n] um novo arranjo com dp[j] = ∅ para todo j
2: para i ← 1 até n faça
3:     para j ← n descendo até 1 faça
4:         alt ← dp[j − 1] ∪ {φ_i}
5:         se p(φ, alt) < p(φ, dp[j]) então
6:             dp[j] ← alt
7:         fim se
8:     fim para
9: fim para
```

3.2.1 Prova de correção

A correção do Algoritmo 2 segue das invariantes de laço a seguir.

Invariante 1: No início de cada iteração do laço **para** das linhas 2–9, temos que $dp[j] = f(i - 1, j), \forall j \leq n$.

Inicialização da Invariante 1: No início da primeira iteração do laço, temos que $i = 1$ e $dp[j] = \emptyset = f(0, j) = f(i - 1, j), \forall j \leq n$, logo a invariante vale.

Manutenção da Invariante 1: Suponha que, antes de uma iteração do laço, $dp[j] = f(i-1, j)$, $\forall j \leq n$. Para provar que o mesmo vale antes da iteração seguinte, enunciamos uma segunda invariante.

Invariante 2: No início de cada iteração do laço **para** das linhas 3–8:

1. Se $k \leq j$, então $dp[k] = f(i-1, k)$.
2. Se $j < k \leq n$, então $dp[k] = f(i, k)$.

Inicialização da Invariante 2: No início da primeira iteração do laço, temos que $j = n$. Pela hipótese de manutenção da Invariante 1, antes da primeira iteração do laço, temos que $dp[k] = f(i-1, k)$, $\forall k \leq n = j$. Portanto, o item 1 da invariante vale. Além disso, o item 2 é satisfeito por vacuidade, pois, $\forall k > j$, temos que $k > n$.

Manutenção da Invariante 2: Suponha que, antes de uma iteração do laço, os itens 1 e 2 da invariante sejam verdade. Neste caso, após a linha 4, temos que $alt = dp[j-1] \cup \{\phi_i\} = f(i-1, j-1) \cup \{\phi_i\}$. Além disso, $dp[j] = f(i-1, j)$. Portanto, a linha 6 só será executada se $p(\phi, f(i-1, j-1) \cup \{\phi_i\}) < p(\phi, f(i-1, j))$. Se isto ocorre, então $f(i, j) = f(i-1, j-1) \cup \{\phi_i\}$ e, após a linha 7, $dp[j] = f(i, j)$. Se isto não ocorre, então $f(i, j) = f(i-1, j)$ e, após a linha 7, $dp[j] = f(i, j)$. Portanto, sob qualquer hipótese, temos que $dp[j] = f(i, j)$ após a iteração do laço, o que prova que os itens 1 e 2 da invariante irão valer na iteração seguinte.

Terminação da Invariante 2: A condição para que o laço termine é $j < 1$. Como cada iteração subtrai 1 de j, em algum momento teremos $j = 0$, ou seja, o laço termina. Além disso, mostramos que, neste ponto, $dp[k] = f(i, k), \forall k > j = 0$, ou seja, $dp[j] = f(i, j), \forall j$.

A propriedade provada na terminação da Invariante 2 prova que, se a Invariante 1 vale antes de uma iteração, então ela também irá valer na iteração seguinte, o que conclui a manutenção da Invariante 1.

Terminação da Invariante 1: A condição para que o laço termine é $i > n$. Como cada iteração adiciona 1 a i, em algum momento teremos $i = n + 1$, ou seja, o laço termina. Além disso, mostramos que, neste ponto, $dp[j] = f(i-1, j) = f(n, j), \forall j$, ou seja, $dp[n] = f(n, n)$.

Note que o algoritmo computa não somente o renomeamento $f(n, n)$, mas também renomeamentos para versões mais restritas do problema. Por exemplo, se quisermos permitir que no máximo $j < n$ subfórmulas sejam escolhidas, basta utilizar o resultado $dp[j] = f(n, j)$.

3.2.2 Análise

O custo de tempo do Algoritmo 2 é o custo das linhas 4–7 multiplicado por n^2. A linha 4, que cria um conjunto de no máximo n elementos e acrescenta a ele um novo elemento, custa $O(n)$ no pior caso. A linha 5 custa o tempo de calcular o número de cláusulas de duas transformações por renomeamento. É possível mostrar que este custo é $O(|pos(\phi)|)$ no pior caso [13]. A linha 6 custa o tempo de copiar e destruir um conjunto com no máximo n elementos, ou seja, $O(n)$ no pior caso. Por fim, temos que $n \leq |pos(\phi)|$. Portanto, o custo das linhas 4–7 é $O(2n + 2|pos(\phi)|) = O(|pos(\phi)|)$ no pior caso; e o custo de tempo total do algoritmo é $O(n^2|pos(\phi)|) = O(|pos(\phi)|^3)$ no pior caso.

O custo de espaço do Algoritmo 2 é dado pela soma dos custos do arranjo *dp*, da variável *alt* e do custo de espaço para calcular o número de cláusulas de duas transformações por renomeamento. O arranjo *dp* contém $n + 1$ conjuntos de no máximo n elementos, logo seu custo de espaço é $O(n^2)$ no pior caso. A variável *alt* é um conjunto de no máximo n elementos, logo seu custo é $O(n)$ no pior caso. É possível mostrar que o custo de espaço para calcular o número de cláusulas de duas transformações por renomeamento é $O(|pos(\phi)|)$ no pior caso [13]. Portanto, o custo de espaço total do algoritmo é $O(n^2 + n + |pos(\phi)|) = O(|pos(\phi)|^2)$ no pior caso.

3.3 Conjectura para árvores lineares

Definição 16 Uma fórmula ϕ é dita uma *árvore linear* se:

1. $\pi_1 \neq \pi_2 \implies \phi|_{\pi_1} \neq \phi|_{\pi_2}$; e
2. ϕ está na FNN.

Conjectura 1 Se ϕ é uma árvore linear e $SFP(\phi) = \{\phi_1, ..., \phi_n\}$, então $f(n, n)$ é ótimo.

Assim como ocorre com os algoritmos de Boy de la Tour [7] e Jackson et al. [11], afirmamos que nosso algoritmo encontra renomeamentos ótimos para árvores lineares. Não provamos esta afirmação analiticamente, mas, para confirmar este fato, apresentamos resultados experimentais no Capítulo 4, que vem a seguir.

Capítulo 4

Resultados experimentais

Apresentamos neste capítulo a metodologia empregada e os resultados experimentais obtidos ao testar se fórmulas menores produzem respostas mais rápido e ao comparar algoritmos para escolha de renomeamento.

4.1 Metodologia

Apresentamos nesta seção os detalhes necessários para reproduzir os experimentos realizados.

4.1.1 Da representação de uma fórmula

A forma que utilizamos para representar fórmulas lógicas em textos teóricos, ou seja, cadeias da respectiva linguagem formal, se mostra pouco prática para implementar transformações ou algoritmos de busca.

Observe, no entanto, que definimos a linguagem formal da lógica proposicional como uma *gramática livre-do-contexto* [15] e, portanto, é garantido que temos um algoritmo de custo de tempo polinomial determinístico para obter a *árvore sintática*, ou simplesmente *árvore*, de uma fórmula [16]. Tal estrutura se mostra muito mais prática para a implementação de algoritmos de transformação e busca do que cadeias de linguagens formais.

Ainda melhor que árvores para representar fórmulas, são *grafos acíclicos dirigidos*, ou simplesmente DAGs na sigla do inglês, onde vértices que representam subfórmulas distintas compartilham arestas com o vértice de uma mesma subfórmula que ambos possuem em comum [11]. Tais estruturas equivalem a representar o conjunto de subfórmulas de fato como um conjunto, ou seja, $|\{\psi \in SF(\phi) \mid \psi = \xi\}| = 1$, para todas ϕ e $\xi \sqsubseteq \phi$. É então uma forma de representação oposta a árvores, que representam o conjunto de subfórmulas

$(p \leftrightarrow p) \leftrightarrow (p \leftrightarrow p)$

Figura 4.1: Representações de $\phi = (p \leftrightarrow p) \leftrightarrow (p \leftrightarrow p)$. (a) Cadeia de $\mathcal{L}$. (b) Árvore sintática. (c) Grafo acíclico dirigido.

como um multiconjunto, ou seja, é possível acontecer $|\{\psi \in SF(\phi) \mid \psi = \xi\}| > 1$. Em outras palavras, árvores sempre representam fórmulas como se nunca houvesse repetição de subfórmulas, criando cópias distintas para as distintas posições em que uma mesma subfórmula ocorre, enquanto DAGs permitem uma representação "honesta", ou seja, repetições de subfórmulas podem ser consideradas e cada subfórmula é representada por um único objeto. Além disso, DAGs são exponencialmente menores que árvores no melhor caso, como mostra a Figura 4.1, e, portanto, propícios a algoritmos de programação dinâmica exponencialmente mais rápidos no melhor caso.

As três formas de representação de fórmulas apresentadas nesta seção são utilizadas em diferentes etapas da implementação, como discutimos a seguir.

porque NNF (pq tem q por na CNF e pq fica tudo mais facil, renaming e tal. fora o fato de poder testar a optimalidade restrita do nosso algoritmo)

por que DAG depois de NNF (arvore mais facil pra fazer NNF)

detalhes de implementacao do Boy para usar DAG (ordenacao topologica e prog dinamica no calculo dos ai's)

detalhes de implementacao do Knapsack (aritmetica de precisao arbitraria)

para simplificar o trabalho, soh aplicamos simplificacao no final, durante a distribuicao

pipeline

pipelina de ordem fixa com estagios flexiveis

Capítulo 5

Conclusões

Conclusões

Referências

[1] Richard E Bellman e Stuart E Dreyfus. *Applied dynamic programming*. Princeton university press, 2015. 21

[2] Armin Biere, Marijn Heule, Hans van Maaren, e Toby Walsh. Conflict-driven clause learning sat solvers. *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications*, pages 131–153, 2009. 2

[3] Roderick Bloem, Uwe Egly, Patrick Klampfl, Robert Könighofer, e Florian Lonsing. Sat-based methods for circuit synthesis. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, pages 31–34. FMCAD Inc, 2014. 1

[4] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971. 1, 2

[5] Martin Davis, George Logemann, e Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962. 2

[6] Martin Davis e Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960. 2, 6

[7] Thierry Boy de la Tour. An optimality result for clause form translation. *Journal of Symbolic Computation*, 14(4):283–301, 1992. 2, 15, 16, 23

[8] Aarti Gupta, Malay K Ganai, e Chao Wang. Sat-based verification methods and applications in hardware verification. In *Formal Methods for Hardware Verification*, pages 108–143. Springer, 2006. 1

[9] John Harrison. *Handbook of practical logic and automated reasoning*. Cambridge University Press, 2009. 1

[10] Eric J Horvitz. *Automated reasoning for biology and medicine*. Knowledge Systems Laboratory, Section on Medical Informatics, Stanford University, 1992. 1

[11] Paul Jackson e Daniel Sheridan. Clause form conversions for boolean circuits. In *Theory and applications of satisfiability testing*, pages 183–198. Springer, 2004. 23, 24

[12] Robert Nieuwenhuis e Albert Oliveras. On sat modulo theories and optimization problems. In *Theory and Applications of Satisfiability Testing-SAT 2006*, pages 156–169. Springer, 2006. 1

[13] Andreas Nonnengart e Christoph Weidenbach. Computing small clause normal forms. *Handbook of automated reasoning*, 1:335–367, 2001. 2, 12, 23

[14] David A Plaisted e Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986. 14

[15] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012. 24

[16] Daniel H Younger. Recognition and parsing of context-free languages in time n 3. *Information and control*, 10(2):189–208, 1967. 24

Apêndice A

Fichamento de Artigo Científico

Anexo I

Documentação Original UnB-CIC (parcial)

```
% -*- mode: LaTeX; coding: utf-8; -*-
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% File    : unb-cic.cls (LaTeX2e class file)
%% Authors : Flávio Maico Vaz da Costa
%%               (based on previous versions by José Carlos L. Ralha)
%% Version : 0.96
%% Updates : 0.5  [??/11/2004] - Initial release. don't remember the day.
%%         : 0.75 [04/04/2005] - Fixed font problems, UnB logo
%%                               resolution, keywords and palavras-chave
%%                               hyphenation and generation problems,
%%                               and a few other problems.
%%         : 0.8  [08/01/2006] - Corrigido o problema causado por
%%                               bancas com quatro membros. O quarto
%%                               membro agora é OPCIONAL.
%%                               Foi criado um novo comando chamado
%%                               bibliografia. Esse comando tem dois
%%                               argumentos onde o primeiro especifica
%%                               o nome do arquivo de referencias
%%                               bibliograficas e o segundo argumento
%%                               especifica o formato. Como efeito
%%                               colateral, as referências aparecem no
%%                               sumário.
%%         : 0.9 [02/03/2008]  - Reformulação total, com nova estrutura
%%                               de opções, comandos e ambientes, adequação
%%                               do logo da UnB às normas da universidade,
%%                               inúmeras melhorias tipográficas,
```

```
%%                              aprimoramento da integração com hyperref,
%%                              melhor tratamento de erros nos comandos,
%%                              documentação e limpeza do código da classe.
%%          : 0.91 [10/05/2008] - Suporte ao XeLaTeX, aprimorado suporte para
%%                              glossaries.sty, novos comandos \capa, \CDU
%%                              e \subtitle, ajustes de margem para opções
%%                              hyperref/impressao.
%%          : 0.92 [26/05/2008] - Melhora do ambiente {definition}, suporte
%%                              a hypcap, novos comandos \fontelogo e
%%                              \slashedzero, suporte [10pt, 11pt, 12pt].
%%                              Corrigido bug de seções de apêndice quando
%%                              usando \hypersetup{bookmarksnumbered=true}.
%%          : 0.93 [09/06/2008] - Correção na contagem de páginas, valores
%%                              load e config para opção hyperref, comandos
%%                              \ifhyperref e \SetTableFigures, melhor
%%                              formatação do quadrado CIP.
%%          : 0.94 [17/04/2014] - Inclusão da opção mpca.
%%          : 0.95 [06/06/2014] - Remoção da opção "mpca", inclusão das opções
%%                              "doutorado", "ppginf", e "ppca" para identificar
%%                              o programa de pós-graduação. Troca do teste
%%                              @mestrado por @posgraduacao.
%%          : 0.96 [24/06/2014] - Ajuste do nome do curso/nome do programa.
%%
```