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

# Um algoritmo baseado em programação dinâmica e renomeamento para minimização de formas normais

Matheus Costa de Sousa Carvalho Pimenta

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

Orientadora
Prof.ª Dr.ª Cláudia Nalon

Brasília
2016

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

# Um algoritmo baseado em programação dinâmica e renomeamento para minimização de formas normais

Matheus Costa de Sousa Carvalho Pimenta

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

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

Prof. Dr. Bruno Lopes Vieira
UFF

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

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

Brasília, 20 de junho de 2016

# Dedicatória

Ao leitor.

# Agradecimentos

Agradeço à minha família, aos meus amigos e ao *rock and 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

Satisfatibilidade booleana é um problema de grande interesse prático, além de ser o primeiro problema caracterizado como NP-completo. Desde que ganhou atenção, diversos algoritmos para este problema já foram descobertos e alguns deles são baseados em formas normais. As transformações usuais para formas normais podem acarretar aumento exponencial do tamanho da fórmula, o que não é desejável. No entanto, é sabido que técnicas como renomeamento podem auxiliar a reduzir o tamanho da fórmula transformada. Neste trabalho, apresentamos uma heurística para o problema de encontrar renomeamentos que reduzem o tamanho de fórmulas e um algoritmo baseado em programação dinâmica para calcular o renomeamento dado pela heurística, junto com sua prova de correção, análise de complexidade e resultados experimentais. A avaliação experimental inclui a comparação do algoritmo proposto com um algoritmo guloso que é ótimo para uma classe restrita do problema. Os resultados indicam que a heurística proposta, na prática, é tão eficiente quanto a do algoritmo ótimo para esta classe, apresentando em alguns casos resultados ainda melhores quando não há restrições sobre a entrada do problema.

**Palavras-chave:** satisfatibilidade, formas normais, renomeamento, programação dinâmica

# Abstract

Boolean satisfiability is a problem of great practical interest and the first problem ever to be stated as NP-complete. Many algorithms for this problem are based on normal forms. Also, naive translations to normal forms may lead to exponential growth on the size of a formula, a highly undesirable effect. It is known, however, that techniques such as formula renaming may help reducing the size of such translation. In this work, we present a heuristic to find renamings that may reduce the size of translated formulæ and a dynamic programming algorithm to compute the renaming given by the heuristic, together with its proof of correctness, complexity analysis and experimental results. We also present comparison results with a greedy algorithm which is optimal for a restricted class of the problem. Empirical evaluation shows that, in most cases, the proposed heuristic produces as fewer clauses as the greedy algorithm for this restricted class of the problem and behaves even better for some specific unrestricted inputs.

**Keywords:** satisfiability, normal forms, renaming, dynamic programming

# 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 [15] e verificação [9] de *hardware*. Aparece também em problemas de raciocínio automático [10] e em muitos outros problemas relevantes [11].

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 [5].

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 [8, 7, 2], apesar de ser conjecturado que qualquer um terá custo de tempo exponencial determinístico no pior caso [5].

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 preciso 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 [16]. 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. Boy de la Tour [4] e Jackson et al. [13] propõem algoritmos para este fim. No Capítulo 3, propomos o nosso algoritmo para este fim, que permite ainda encontrar boas transformações restringindo o tamanho máximo do renomeamento. Comparamos o algoritmo proposto com o de Boy de la Tour no Capítulo 4 e, por fim, propomos trabalhos futuros no Capítulo 5.

# 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 $\phi$ é uma *fórmula bem-formada*, ou simplesmente que $\phi$ é uma *fórmula*. Além disso, se $\phi_1, ..., \phi_n$ são fórmulas quaisquer, onde $n \in \mathbb{N} \cup \{0\}$ e $\phi$ é de uma das formas a seguir, então $\phi$ 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 $\phi_i$ é *subfórmula imediata* de $\phi$, 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 $\phi$.

**Exemplo 1.** *São fórmulas:*

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

- $\xi = \neg(p \rightarrow q)$

*Note ainda que $p \rightarrow q$ é subfórmula imediata de $\phi$ e de $\xi$.*

**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. $\Bbbv(\phi_1 \rightarrow \phi_2) = V$ 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. $\Bbbv(\bot) = F$.

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

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

**Definição 3** Se existe uma interpretação $\Bbbv$ tal que $\Bbbv(\phi) = V$, então dizemos que $\Bbbv$ *satisfaz* $\phi$, ou ainda, que $\phi$ é *satisfatível*. De maneira análoga, se $\Bbbv$ é tal que $\Bbbv(\phi) = F$,

então dizemos que $\Vdash$ *falsifica* $\phi$, ou ainda, que $\phi$ é *falsificável*.

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

Se $\phi$ é simultaneamente satisfatível e falsificável, então dizemos que $\phi$ 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 os alvos 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 $= \{\phi \in \mathcal{L} \mid \phi \text{ é insatisfatível}\} = \{\phi \in \mathcal{L} \mid \phi \notin \text{SAT}\} = \overline{\text{SAT}}$ como o problema da *insatisfatibilidade*.

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

O conceito de *fórmula válida* é usualmente definido através do conceito de *consequência lógica* [14]. 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 [14]. 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 [8]. 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 $\phi$ 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.

**Prova.** *Se $\phi$ é uma tautologia, então $\nu(\phi) = V, \forall \nu$. Logo $\nu(\neg\phi) = F, \forall \nu$. Portanto, $\neg\phi$ é uma contradição.*

*Se $\phi$ é uma contradição, então $\nu(\phi) = F, \forall \nu$. Logo $\nu(\neg\phi) = V, \forall \nu$. Portanto, $\neg\phi$ é uma tautologia.*

*Se $\phi$ é uma contingência, então existem interpretações $\nu_1, \nu_2$ tais que $\nu_1(\phi) = V$ e $\nu_2(\phi) = F$. Logo, $\nu_1(\neg\phi) = F$ e $\nu_2(\neg\phi) = V$. Portanto, $\neg\phi$ é uma contingência.* □

**Teorema 2** SAT e VAL são redutíveis um ao outro.

**Prova.** *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 $\phi$ 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 $\phi$ 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 $\phi$ 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 $\phi$ 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 $\phi$ uma fórmula. Definimos $tam(\phi)$, o *tamanho* de $\phi$, 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. $tam(\phi_1 \to \phi_2) = tam(\phi_1 \leftrightarrow \phi_2) = 1 + tam(\phi_1) + tam(\phi_2)$.
5. $tam(\top) = tam(\bot) = 1$.

**Exemplo 4.** *Seja* $\phi = p \to (\neg r \vee (q \leftrightarrow s))$. *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 $\phi$ é *subfórmula* de $\psi$, escrito $\phi \sqsubseteq \psi$, se, e somente se, alguma das possibilidades ocorre:

1. $\phi = \psi$.
2. $\phi$ é subfórmula imediata de $\psi$.
3. $\phi$ é subfórmula de $\xi$ e $\xi$ é subfórmula imediata de $\psi$.

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

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

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

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

- *A única subfórmula imediata de* $\phi$ *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* $\phi$.

**Definição 8** Uma *posição* é uma sequência finita de números naturais. Usaremos as notações alternativas $\varepsilon$, 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 $\phi$, $pos(\phi)$, da seguinte maneira:

1. Se $\phi \in \mathcal{P}$, então $pos(\phi) = \{\varepsilon\}$.
2. Se $\phi$ 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* $\phi$ *começando na posição* $\pi$, escrito $\phi|_\pi$, da seguinte forma:

1. Se $\pi = \varepsilon$, então $\phi|_\pi = \phi$.
2. Se $\phi$ 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 $\pi$ 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 $\phi$ começando na posição $\pi$*, 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 $\phi = (p \to q) \to \neg(p \leftrightarrow (r \vee s))$. 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 $\phi$ em $\psi$.

Se uma regra de reescrita transforma $\phi$ em $\psi$, dizemos que esta regra:

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

**Teorema 3** Seja $\phi$ da forma $\phi_1 \wedge ... \wedge \phi_n$ e $\phi_1$ da forma $\psi_1 \wedge ... \wedge \psi_k$. Então a transformação

$$\phi \longmapsto \psi_1 \wedge ... \wedge \psi_k \wedge \phi_2 \wedge ... \wedge \phi_n$$

que chamaremos de *aplainamento*, preserva equivalência.

A regra é análoga para disjunções.

O Teorema 3 é apenas uma formalização do fato que podemos considerar uma conjunção de conjunções como uma única conjunção, e que o mesmo vale para disjunções. Usaremos este fato mais adiante, no Capítulo 4.

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

**Teorema 4** 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. $\phi_1 \to \phi_2 \longmapsto \neg\phi_1 \vee \phi_2$

5. Se $\phi|_\pi$ é da forma $\phi_1 \leftrightarrow \phi_2$, então

   (a) $\phi|_\pi \longmapsto (\phi_1 \to \phi_2) \wedge (\phi_2 \to \phi_1)$, 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 4 segue por indução na estrutura de uma fórmula.

A transformação de equivalências dependente de polaridade do Teorema 4 evita que tautologias difíceis de detectar apareçam nas fórmulas transformadas, como mostra o Exemplo 8, dado por Nonnengart et al. [16]. 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 $\phi$ é um *literal* se, e somente se, $\phi \in \mathcal{P}$, ou $\phi$ é da forma $\neg p$, onde $p \in \mathcal{P}$.

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

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

**Teorema 5** 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 5 segue por indução na estrutura de uma fórmula.

**Exemplo 8.** *Considere $\phi$ da forma $\neg(\phi_1 \leftrightarrow \phi_2)$. Aplicando as transformações dos Teoremas 4 e 5 à exaustão, começando pela transformação do item 5.a do Teorema 4 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 4.*

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

*Observe que 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 $\phi$, $\forall \phi, \phi_1, \phi_2 \in \mathcal{L}$.*

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

Um *renomeamento de $\phi$* é um conjunto de subfórmulas próprias de $\phi$.

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

Seja $s = \{(\phi_1, p_1), ..., (\phi_n, p_n)\}$ uma substituição de $R = \{\phi_1, ..., \phi_n\}$ em $\phi$. 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* $\phi = (p \vee q) \to (r \wedge (p \vee q) \wedge (p \wedge q))$ *e* $s = \{(p \vee q, a), (p \wedge q, b)\}$. *Então*

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

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

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

como a *definição de $\psi$ em $\phi$ segundo $s$.*

Note que, na definição acima, $rep(\psi, s - \{(\psi, p)\})$ representa a fórmula $\psi$ com todas as outras substituições definidas por $s - \{(\psi, p)\}$ já aplicadas.

**Teorema 6** Seja $s = \{(\phi_1, p_1), ..., (\phi_n, p_n)\}$ uma substituição de $R = \{\phi_1, ..., \phi_n\}$ em $\phi$, onde $R$ é um renomeamento de $\phi$. 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 6 [17].

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

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

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

Apresentamos nesta seção o algoritmo proposto por Boy de la Tour para obter um renomeamento que procura reduzir o número de cláusulas geradas por uma fórmula [4].

**Definição 15** Denotamos por $p(\phi)$ o *número de cláusulas* da fórmula obtida por aplicar as transformações dos Teoremas 4 e 5 à exaustão na fórmula $\phi$. Denotamos ainda $p(\neg\phi)$ por $\overline{p}(\phi)$.

O número $p(\phi)$ pode ser calculado pela Tabela 2.1 [4].

**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*

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

Tabela 2.1: Número de cláusulas de uma fórmula.

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

*Agora,*

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

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

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

**Definição 16** Uma fórmula $\phi$ é dita uma *árvore linear* se:

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

O Algoritmo 1, apresentado por Boy de la Tour, encontra um renomeamento que produz o número ótimo (mínimo) de cláusulas, desde que $\phi$ seja uma árvore linear [4], ou seja, desde que não ocorram repetições de subfórmulas e $\phi$ esteja na FNN. Seu custo de tempo é $O(|pos(\phi)|^2)$ determinístico no pior caso.

Os coeficientes $a$ e $b$ que aparecem no algoritmo são exatamente os números de vezes que os números $p(\psi)$ e $\overline{p}(\psi)$, respectivamente, são somados ao valor de $p(\phi)$. Em outras palavras, o algoritmo utiliza a forma irredutível de $p(\phi)$ em função de $p(\psi)$ e $\overline{p}(\psi)$:

$$p(\phi) = a^{\phi}_{\psi} p(\psi) + b^{\phi}_{\psi} \overline{p}(\psi) + c$$

onde $c$ não depende de $p(\psi)$ ou $\overline{p}(\psi)$.

As escritas $\psi.p$ e $\psi.\overline{p}$ que aparecem no algoritmo representam inicialmente $p(\psi)$ e $\overline{p}(\psi)$, respectivamente. No entanto, se $\psi$ é incluída em $R$, sua contribuição para $p(\phi, R)$ muda. Para esta mudança ser refletida nos cálculos do algoritmo, os valores de $p(\psi)$ e $\overline{p}(\psi)$ são armazenados em campos da respectiva subfórmula e atualizados conforme a execução do algoritmo. As notações $\psi.p$ e $\psi.\overline{p}$ indicam estes campos. Note então que pode ocorrer em

Tabela 2.2: Coeficientes $a$ e $b$ do Algoritmo 1.

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

algum momento que $p(\psi) \neq \psi.p$ ou $\overline{p}(\psi) \neq \psi.\overline{p}$.

Os números $a^{\mathcal{R}(\phi,s)}_{\psi_i}$ e $b^{\mathcal{R}(\phi,s)}_{\psi_i}$ são computados a partir de $a = a^{\mathcal{R}(\phi,s)}_{\psi}$, $b = b^{\mathcal{R}(\phi,s)}_{\psi}$, $\psi_i.p$ e $\psi_i.\overline{p}$, seguindo as fórmulas da Tabela 2.2. A função $nbcl(\psi)$ retorna um par ordenado com os números de cláusulas de $\psi$ e $\neg\psi$, respectivamente. Este par ordenado é calculado a partir de $\psi_i.p$ e $\psi_i.\overline{p}$, ou seja, o estado atual do renomeamento é levado em consideração.

**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* $R\_rec(\psi_1, 3, 0, 1)$.

*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* $\psi$ *da forma* $y_1 \wedge y_2 \wedge y_3$ *não modifica o conjunto* $R$.

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

**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**
                $R\_rec(\psi_i, a_{\psi_i}^{\mathcal{R}(\phi,s)}, b_{\psi_i}^{\mathcal{R}(\phi,s)}, r \cdot pol(\psi, i))$
            **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**
$R\_rec(\phi, 1, 0, 1)$

*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* $\phi$. *Se incluímos ambas* $\phi_1$ *e* $\phi_2$, *a transformação gera 7 cláusulas. Se* $R = \emptyset$, *a fórmula permanece com 9 cláusulas. Se somente* $\phi_1$, *ou somente* $\phi_2$ *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.*

Note que o algoritmo proposto por Boy de la Tour é um *algoritmo guloso*, ou seja, um algoritmo que faz uma série de escolhas que são ótimas em contextos locais [6]. Mas, como dito anteriormente, Boy de la Tour prova que se a fórmula de entrada for uma árvore linear, então as escolhas ótimas locais de seu algoritmo são também ótimas no contexto global.

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

# O algoritmo

Apresentamos neste capítulo um algoritmo de renomeamento para reduzir o número de clásulas geradas por uma fórmula. Começamos por uma afirmação essencial.

**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.

**Prova.** *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* $R \cup \{\psi\}$ *para* $\phi$ *que tenha no máximo 2 subfórmulas.*

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

1. $p(\psi) = p(\xi_1)p(\xi_2)$
2. $p(\psi, \{\xi_1\}) = p(\psi, \{\xi_2\}) = p(\xi_1) + p(\xi_2)$
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* $\psi$. *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. Neste caso,* $\phi$ *é um contraexemplo para a Afirmação 1.* □

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. Por outro lado, se a afirmação fosse verdadeira, ela caracterizaria o que chamamos de uma *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 $k = j-1$ ou $k = j$.

Para obter então um renomeamento ótimo considerando todas as subfórmulas próprias de $\phi$, 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 se $j \leq n$, então $dp[j] = f(i-1, j)$.

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

**Manutenção da Invariante 1**: Suponha que, antes de uma iteração do laço, se $j \leq n$, então $dp[j] = f(i-1, j)$. 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 se $k \leq n = j$, então $dp[k] = f(i - 1, k)$. Portanto, o item 1 da invariante vale. Além disso, o item 2 é satisfeito por vacuidade, pois se $k > j$, então $k > n$.

**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, se $k > j = 0$, então $dp[k] = f(i, k)$, 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 clásulas de duas transformações por renomeamento. É possível mostrar que este custo é $O(|pos(\phi)|)$ no pior caso [16]. 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 clásulas 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 clásulas de duas transformações por renomeamento é $O(|pos(\phi)|)$ no pior caso [16]. 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

**Conjectura 1** Se $\phi$ 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 [4] e Jackson et al. [13], afirmamos que nosso algoritmo também encontra renomeamentos ótimos para árvores lineares. Não provamos esta afirmação analiticamente, mas apresentamos resultados experimentais no Capítulo 4, que vem a seguir, que exemplificam este fato.

# Capítulo 4

# Resultados experimentais

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

Os códigos-fonte e dados citados neste capítulo estão disponíveis no endereço `https://github.com/matheuscscp/TG`.

## 4.1 Metodologia

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

### 4.1.1 Representações de fórmulas

A forma de representação de fórmulas lógicas utilizada em textos teóricos, ou seja, cadeias de uma determinada linguagem formal, complica bastante a tarefa de implementar transformações ou algoritmos de busca.

Observe, no entanto, que definimos a linguagem formal da lógica proposicional como uma *gramática livre-do-contexto* [21] e, portanto, é garantido que existe um algoritmo de custo de tempo polinomial determinístico para obter a *árvore sintática*, ou simplesmente *árvore*, de uma fórmula [23]. 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. Em um DAG, qualquer subfórmula é representada por um único objeto [13], ou seja, mesmo que uma subfórmula ocorra em várias posições da fórmula principal, ela não é replicada para cada ocorrência. Tais estruturas equivalem a

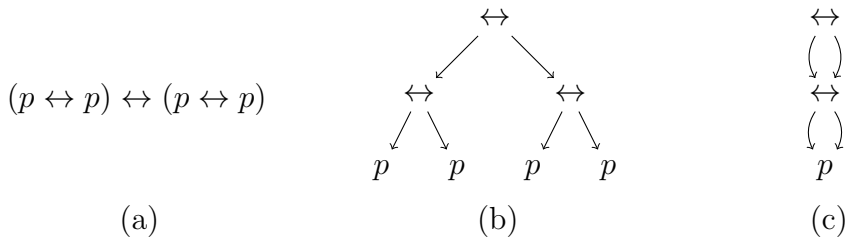


Figura 4.1: Representações de $(p \leftrightarrow p) \leftrightarrow (p \leftrightarrow p)$. (a) Cadeia. (b) Árvore. (c) DAG.

representar o conjunto de subfórmulas de fato como um conjunto, ou seja, $|\{\psi \in SFP(\phi) : \psi = \xi\}| = 1$, para todas $\phi$ e $\xi \sqsubseteq \phi$. É então uma forma de representação que se contrapõe a de uma árvore, que representa o conjunto de subfórmulas como um multiconjunto, ou seja, é possível acontecer $|\{\psi \in SFP(\phi) : \psi = \xi\}| > 1$. 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. Portanto, DAGs permitem uma representação mais fiel ao tratamento teórico dado neste trabalho, ou seja, múltiplas ocorrências de subfórmulas podem ser observadas na representação de uma fórmula. 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 [1] 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.

### 4.1.2 Implementação

Implementamos um programa, utilizando a linguagem de programação C++ versão 11 [12], que recebe uma fórmula sob a forma de cadeia, realiza uma série de transformações e dá como saída uma fórmula resultante também sob a forma de cadeia, juntamente com três medidas desta última fórmula: seu tamanho, número de cláusulas e número de símbolos proposicionais. As transformações realizadas são detalhadas a seguir. Optamos por deixar flexível a execução ou não de algumas transformações, mas fixamos a ordem em que as transformações são realizadas.

#### Análise sintática

Convertemos uma fórmula sob a forma de cadeia para uma árvore sintática. Implementamos um algoritmo de pilha [21, 6], que faz uma única varredura na cadeia e produz a árvore. Esta transformação é sempre executada.

### Conversão para FNN

A próxima transformação coloca a fórmula na forma normal negada. Esta etapa é feita através de uma busca em profundidade [6], onde o vértice pai é processado antes dos filhos, ou seja, em *pré-ordem*. Realizamos esta transformação por dois motivos em particular:

1. As versões dos algoritmos para escolha de renomeamento e do algoritmo de conversão para a FNC são extremamente mais simples para fórmulas na FNN;
2. Optando por não realizar a conversão para DAG mais adiante, a transformação que obtemos simula uma árvore linear. É claro que, em geral, as fórmulas não são necessariamente árvores lineares. No entanto, se for representada por uma árvore sintática, qualquer fórmula na FNN é considerada uma árvore linear pelos algoritmos de renomeamento, pois eles não conseguem identificar vértices distintos representando uma mesma subfórmula. Portanto, podemos testar casos em que os algoritmos não necessariamente escolhem renomeamentos ótimos, mas necessariamente escolhem renomeamentos que geram o mesmo número de cláusulas.

Esta transformação é necessária somente para a realização de renomeamento e conversão para FNC. Pode ser desligada, caso o objetivo da execução do programa seja apenas obter as medidas da fórmula original.

### Aplainamento

Aplicamos aplainamento à exaustão, também através de busca em profundidade pré-ordem. Isto não é necessário para nenhuma outra transformação e portanto pode-se optar por não realizar aplainamento. No entanto, esta transformação faz com que apareçam mais situações em que é possível aplicar simplificação [20]. E, ainda, no contexto de renomeamento, o exemplo a seguir evidencia como aplainamento pode ser proveitoso.

Considere $\phi$ da forma $(p \wedge (q \wedge r)) \vee ... \vee ((p \wedge q) \wedge r)$. Na forma aplainada de $\phi$, $(p \wedge q \wedge r)$ pode ser renomeada por um único novo símbolo proposicional em suas duas ocorrências, se $\phi$ estiver representada por um DAG.

### Conversão para DAG

A conversão é feita através de um algoritmo ascendente, ou seja, construímos o DAG de baixo para cima, indo das folhas até a raiz da árvore. Esta transformação não é necessária para nenhuma transformação seguinte e é possível optar por não executá-la.

Vale observar ainda que optamos por executar a conversão para DAG somente após colocar a fórmula na FNN, dado que isto evita o seguinte problema.

Suponha que $\psi$ ocorra em $\pi_1 \in pos(\phi)$ e em $\pi_2 \in pos(\phi)$ e suponha ainda que $pol(\phi, \pi_1) = 1 = -pol(\phi, \pi_2)$. Neste caso, é provável que, durante a conversão para FNN, $\psi$ deva ser transformada de duas maneiras distintas, uma em cada posição. Isto complicaria a conversão, caso a representação de $\phi$ fosse um DAG. Em uma árvore, no entanto, onde diferentes cópias de $\psi$ são criadas para diferentes posições em que $\psi$ ocorre, este problema é resolvido transparentemente.

**Renomeamento**

Nesta etapa, há três opções:

1. Não realizar nenhum renomeamento;
2. Aplicar o renomeamento escolhido pelo algoritmo de Boy de la Tour;
3. Aplicar o renomeamento $f(n, n)$, que propomos e descrevemos no Capítulo 3.

Observamos a seguir os detalhes cruciais das implementações dos algoritmos de renomeamento.

Como dito anteriormente, ambos os algoritmos de renomeamento são implementados para fórmulas na FNN. Como na FNN só há negações na frente de símbolos proposicionais, temos que

$$p(\psi) > 1 \implies b^{\phi}_{\psi} = 0, \forall \phi \in \mathcal{L} \text{ e } \psi \in SF(\phi)$$

Isto reduz a condição da Linha 4 do Algoritmo 1:

$$a \cdot \psi.p > a + if\_pos(r, \psi.p) + if\_pos(-r, \psi.\overline{p})$$

Observe ainda que o valor de $r$, no Algoritmo 1, só deixa de ser 1 na Linha 11, quando $pol(\psi, i) \neq 1$. Mas, na FNN, ocorre que

$$p(\phi|_{\pi}) > 1 \implies pol(\phi, \pi) = 1, \forall \phi \in \mathcal{L} \text{ e } \pi \in pos(\phi)$$

Logo, a condição da Linha 4 é reduzida mais uma vez:

$$a \cdot \psi.p > a + \psi.p$$

Por fim, como é garantido que $\psi.p > 1$ quando a Linha 4 é executada, esta última condição se reduz novamente:

$$a > 2 \text{ ou } (a = 2 \text{ e } \psi.p > 2)$$

Este último resultado nos permite implementar o algoritmo de Boy de la Tour sem aritmética de precisão arbitrária, pois, como as únicas operações aritméticas realizadas para calcular os valores de $a$ e $p$ são a adição e a multiplicação, podemos simplesmente truncar valores grandes para evitar que transbordamento aritmético ocorra. Acontece, no entanto, que este truncamento nos leva a outro problema.

Há duas alternativas para implementar o Algoritmo 1 em sua complexidade correta (quadrática). A mais simples é calcular o produto $a^{\phi}_{\psi_i} = a \prod_j \psi_j.p$, utilizado na Linha 11, uma única vez e dividí-lo por $\psi_i.p$ apropriadamente em cada iteração do laço da Linha 10. Como optamos por uma representação numérica que não permite divisão, utilizamos uma segunda alternativa. Calculamos previamente uma tabela $dp$ que satisfaz $dp[i] = \prod_{j=i+1}^{n} \psi_j.p,\ i = 1, 2, ..., n$, e atualizamos o valor de $a$ a cada iteração com $a \leftarrow a \cdot \psi_i.p$. Desta forma, o produto é corretamente calculado a cada iteração pela expressão $a \cdot dp[i]$. Nesta alternativa, nenhuma divisão é necessária.

Um detalhe adicional precisa ser considerado para a correção do método que utilizamos para calcular $a^{\phi}_{\psi_i}$. Observe que este valor precisa ser calculado a partir dos campos $\psi_j.p$. Acontece que, se a fórmula estiver representada por um DAG, o valor de $\psi_j.p$ poderia mudar durante a execução da chamada recursiva para $\psi_i$, para algum $j > i + 1$, de modo que a tabela que calculamos previamente forneceria um valor incorreto para a iteração seguinte, quando $i$ passa a valer $i + 1$. Evitamos este problema fazendo uma ordenação topológica nas arestas que saem de $\psi$, de modo que se $\psi_j \sqsubset \psi_k$, então $j < k$. Isto é de fato uma solução correta para o problema, porque qualquer DAG possui pelo menos uma ordenação topológica [6].

Na implementação do Algoritmo 2, por outro lado, não utilizar aritmética de precisão arbitrária no cálculo de $p(\phi)$ levaria a resultados possivelmente incorretos. Assim, para que os resultados fossem corretos, e portanto no mínimo fiéis à heurística, optamos por implementar e utilizar aritmética de precisão arbitrária.

Por fim, é importante mencionar que, para calcular $f(n, n)$ através do Algoritmo 2, a ordem de $SFP(\phi) = \{\phi_1, ..., \phi_n\}$ é dada por uma busca em largura, ou seja, os primeiros vértices considerados pelo Algoritmo 2 estão mais perto da raiz.

### Conversão para FNC

A última etapa do programa executa distribuição para colocar a fórmula na FNC e dá como saída a fórmula final sob a forma de cadeia. Esta etapa também é feita através de busca em profundidade pré-ordem. É possível ainda optar por não colocar a fórmula na FNC, ou ainda, durante a distribuição, aplicar as seguintes simplificações: eliminação de literais e cláusulas repetidos e eliminação de tautologias.

Tabela 4.1: Combinações de transformações executadas.

| Combinação | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Análise sintática | X | X | X | X | X | X | X | X | X | X |
| Conversão para FNN | X | X | X | X | X | X | X | X | X | X |
| Aplainamento | X | X | X | X | X | X | X | X | X | X |
| Conversão para DAG | | | | | | | X | X | X | X |
| Renomeamento | | | 1 | 1 | 2 | 2 | 1 | 1 | 2 | 2 |
| Conversão para FNC | 1 | 2 | 1 | 2 | 1 | 2 | 1 | 2 | 1 | 2 |

Escolhemos aplicar simplificação somente no final do programa, para que, justamente optando por não aplicar, fosse possível obter melhores comparações dos algoritmos de renomeamento. Não obstante, apresentamos resultados em que simplificação é aplicada.

### 4.1.3 Experimentos propostos

Para tentar responder às perguntas propostas, executamos o programa implementado para diversas combinações de transformações sobre um *benchmark* tradicional de 1200 fórmulas intuicionistas [18]. Optamos por utilizar este *benchmark*, pois a maioria dos *benchmarks* tradicionais apresentam fórmulas já em uma forma normal, ou possuem poucas fórmulas proposicionais, como o proposto por Sutcliffe [22]. Portanto, com o *benchmark* escolhido, pudemos testar nossas hipóteses de maneira mais exaustiva. Em seguida, executamos sobre as fórmulas transformadas o provador de teoremas E, um decisor para VAL baseado em forma normal clausal [19]. Foram anotadas as medidas de cada fórmula fornecidas ao final da etapa de pré-processamento, os tempos de execução nas duas etapas e o resultado do decisor de validade.

As dez combinações de transformações testadas são descritas na Tabela 4.1. Em cada combinação, executamos um subconjunto das transformações apresentadas na Seção 4.1.2. Na tabela, a presença de um X indica que a transformação da respectiva linha foi executada na combinação da respectiva coluna. Uma célula vazia indica que a respectiva transformação não foi executada na respectiva combinação. Na linha de renomeamento, o número 1 indica que o algoritmo de renomeamento executado é o proposto por Boy de la Tour (Algoritmo 1), enquanto o número 2 indica que o algoritmo executado é o que propomos neste trabalho (Algoritmo 2). Finalmente, na linha de conversão para FNC, o número 1 indica que foi executada somente distribuição, enquanto o número 2 indica que foi executada distribuição com simplificação.

Os experimentos foram executados em um computador com sistema operacional Ubuntu 14.04 (GNU/Linux 3.19.0-30-generic x86_64), processador Intel® Xeon® Processor E5-2620 v3, que possui frequência de *clock* 2.4GHz e 15M de memória *cache*, e memória RAM de 64GiB. Cada combinação foi executada com um limite máximo

de 4GB de memória virtual, tamanho ilimitado para o segmento de pilha e mil segundos de limite de tempo.

## 4.2 Resultados e análise

Dos dados coletados, foi feita uma análise exaustiva. Apresentamos nesta seção os principais destaques.

### 4.2.1 Combinações sem renomeamento

Primeiro, observamos os resultados das Combinações 1 e 2, onde não foi feito nenhum renomeamento e alternamos somente entre aplicar ou não simplificação durante a etapa de distribuição.

Na Combinação 1, onde simplificação não foi aplicada, 73% das fórmulas excederam o limite de memória na etapa de pré-processamento e o restante foi transformado com sucesso. Na Combinação 2, 67% excedeu o limite de memória e 1% excedeu o limite de tempo. Além disso, dos 27% em que todas as fórmulas foram transformadas com sucesso por ambas as combinações, 5 fórmulas (menos de 1%) ficaram com o mesmo tamanho, enquanto todas as restantes ficaram menores ao aplicar simplificação. Isto faz bastante sentido, pois se uma fórmula contém exatamente $n$ símbolos proposicionais distintos, então:

1. Ao eliminar literais repetidos e tautologias, cada cláusula gerada pela fórmula pode conter no máximo $n$ literais.

2. Ao eliminar cláusulas repetidas, a fórmula pode gerar no máximo $3^n$ cláusulas distintas. Isto ocorre porque, para cada símbolo proposicional e uma cláusula fixa sem tautologias e literais repetidos, há somente três possibilidades: ou o símbolo não ocorre na cláusula, ou ele ocorre, ou ele ocorre negado.

Em outras palavras, uma fórmula com as simplificações que aplicamos possui um tamanho máximo ao ser convertida para a FNC.

A observação principal é que apenas simplificação não é suficiente para obter bons resultados, já que mais da metade das fórmulas sequer pôde ser pré-processada com sucesso dentro das restrições de tempo e memória utilizadas.

### 4.2.2 Testando a conjectura para árvores lineares

Das Combinações 3 e 5, que aplicam a árvores respectivamente os Algoritmos 1 e 2 para escolha de renomeamento e não aplicam simplificação durante a etapa de distribui-

Tabela 4.2: Árvore × DAG. Número de cláusulas.

| | $C_3 \times C_7$ | $C_4 \times C_8$ | $C_5 \times C_9$ | $C_6 \times C_{10}$ | |
|---|---|---|---|---|---|
| $C_i$ foi melhor em | 0 (0%) | 5 (0%) | 0 (0%) | 6 (1%) | fórmulas. |
| $C_{i+4}$ foi melhor em | 179 (15%) | 177 (15%) | 367 (31%) | 358 (30%) | fórmulas. |

ção, obtemos uma amostra da correção da Conjectura 1 (optimalidade do Algoritmo 2 para árvores lineares). Como discutido na Seção 4.1.2, estas combinações *simulam* árvores lineares e, portanto, sob essa condição, não necessariamente os Algoritmos 1 e 2 devem encontrar renomeamentos que geram o menor número possível de cláusulas para uma fórmula qualquer, mas ambos devem encontrar renomeamentos que geram exatamente o mesmo número de cláusulas para esta fórmula. Para todas as fórmulas que puderam ser transformadas com sucesso em ambas as combinações (49%), constatamos esta propriedade. Além disso, nenhuma fórmula excedeu o limite de memória somente na Combinação 5. Caso ocorresse, esta situação seria um leve indício de que a conjectura talvez não fosse válida. Estes dois fatos indicam fortemente que a Conjectura 1 provavelmente é um teorema.

### 4.2.3 Comparações entre árvores e DAGs

Comparamos agora os resultados das Combinações de 3 a 6, onde a estrutura de representação utilizada foi árvore, com as Combinações de 7 a 10, onde a estrutura utilizada foi DAG, ou seja, comparamos cada combinação com sua igual, a menos do tipo de representação utilizada. Portanto, referindo-nos à Combinação $i$ abreviadamente por $C_i$, comparamos $C_i$ (árvore) com $C_{i+4}$ (DAG), para $i = 3, 4, 5, 6$.

O primeiro resultado observado, que é relativo ao número de cláusulas produzido em cada combinação, é apresentado na Tabela 4.2. As seguintes regras foram usadas para determinar quando cada representação foi melhor. Para uma fórmula fixa $\phi$, $C_i$ foi melhor que $C_{i+4}$ se, e somente se:

1. ambas $C_i$ e $C_{i+4}$ transformaram $\phi$ com sucesso, isto é, não excederam o limite de tempo ou memória, e a transformação produzida por $C_i$ gera menos cláusulas que a transformação produzida por $C_{i+4}$; ou

2. $C_i$ transformou $\phi$ com sucesso e $C_{i+4}$ não.

O critério análogo obtido trocando cada ocorrência de $C_i$ por $C_{i+4}$ e vice-versa foi utilizado para determinar quando $C_{i+4}$ foi melhor que $C_i$.

Da Tabela 4.2, note que as combinações com representação por DAG, em geral, produziram fórmulas com menos cláusulas. Observe ainda que nos pares onde simplificação não é aplicada ($C_3 \times C_7$ e $C_5 \times C_8$), nenhuma fórmula gerou menos cláusulas na transformação

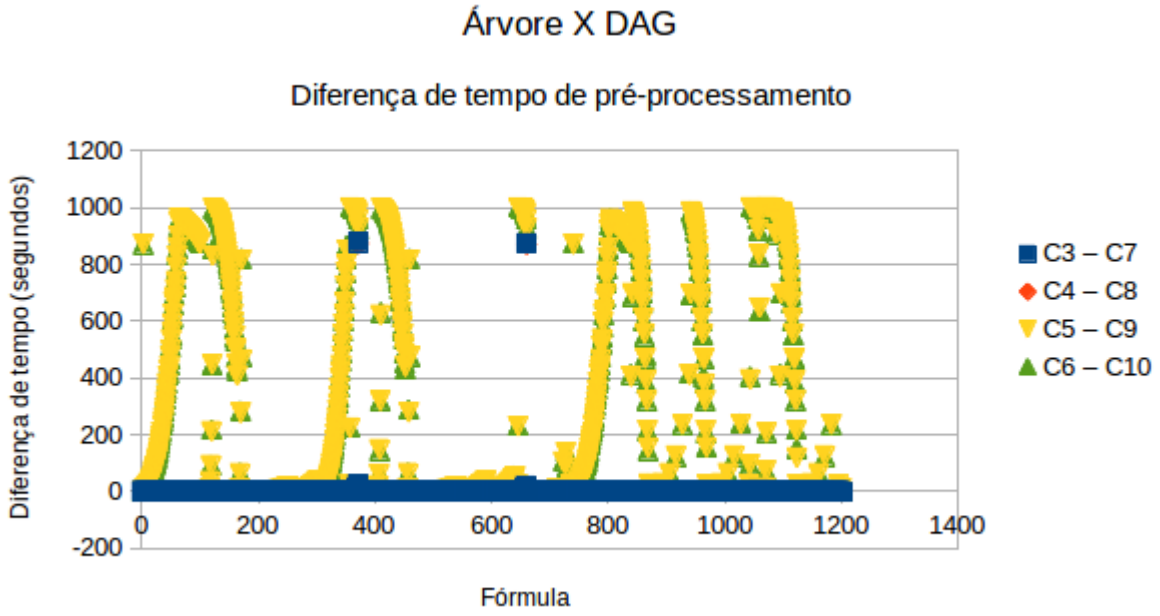


Figura 4.2: Árvore × DAG. Tempo de pré-processamento.

por $C_i$ (árvore). Estes fatos fazem bastante sentido, pois, do ponto de vista de algoritmos de renomeamento, a única diferença entre as duas formas de representação é que DAGs permitem que todas as ocorrências de uma mesma subfórmula sejam renomeadas por um único novo símbolo proposicional. Note que mesmo aplicando simplificação ao final das transformações (pares $C_4 \times C_8$ e $C_6 \times C_{10}$), DAGs ainda geram menos cláusulas em grande parte das vezes.

Em segundo lugar, observamos o tempo de execução da etapa de pré-processamento. Os resultados são apresentados no gráfico da Figura 4.2. Cada ponto sobre o eixo horizontal identifica uma das 1200 fórmulas do *benchmark*. O eixo vertical é o tempo de $C_i$ menos o tempo de $C_{i+4}$, em segundos. Do gráfico, nota-se que a maior parte das diferenças é positiva e com um grande valor absoluto. Isto quer dizer que, na maior parte das vezes, DAGs obtiveram melhor desempenho durante a etapa de pré-processamento. O que não é surpresa, pois, mesmo que $C_{i+4}$ não gere menos cláusulas, algoritmos de transformação executam mais rápido em representações mais compactas.

Por último, analisamos o tempo de execução na etapa de busca, ou seja, o tempo de execução do provador E. O gráfico da Figura 4.3, que ilustra estes resultados, foi construído de maneira similar ao gráfico da Figura 4.2, ou seja, cada ponto no eixo horizontal representa uma fórmula e cada altura indica a diferença entre o tempo medido para $C_i$ e o tempo medido para $C_{i+4}$. Mais uma vez, notamos que a maior concentração de pontos está acima ou sobre o eixo horizontal de diferença igual a zero. E, novamente,

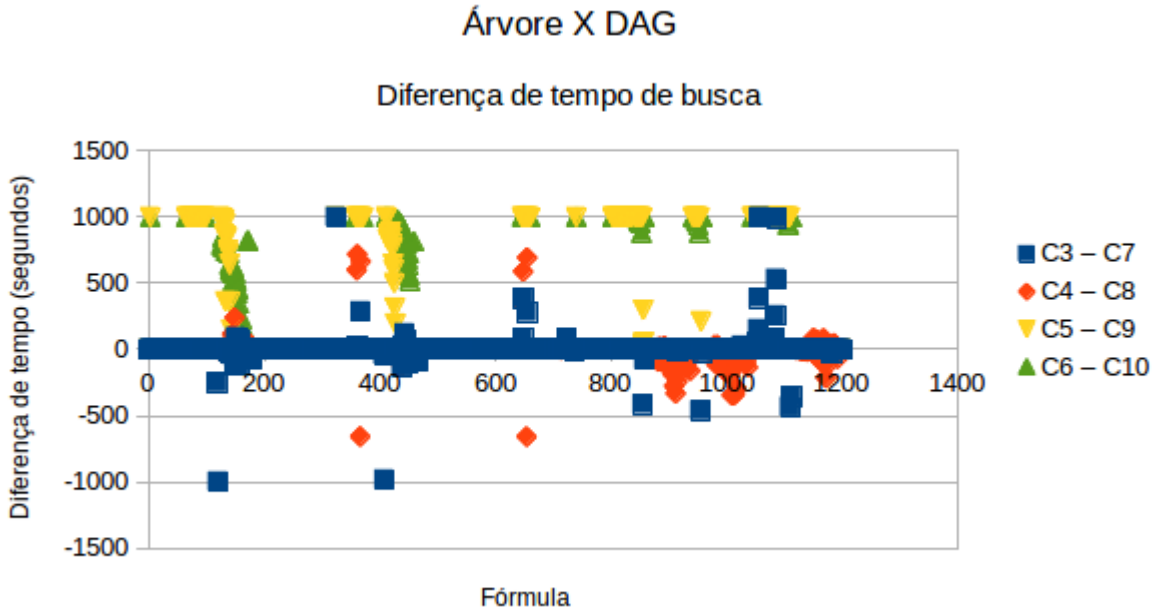


Figura 4.3: Árvore × DAG. Tempo de busca.

isto indica que, em grande parte das vezes, DAGs obtiveram melhor desempenho, mas agora na etapa de busca.

Com esta última observação, temos o primeiro indício de que, muitas vezes, fórmulas com menos cláusulas podem produzir respostas mais rápido. Esta conclusão é bastante razoável, já que, como observamos antes, a principal diferença entre as fórmulas produzidas por cada um dos dois grupos de combinações é justamente o número de cláusulas, que é menor para DAGs em grande parte das vezes.

Com os resultados apresentados nesta seção, há um forte indício de que, independentemente dos resultados de optimalidade restrita dos algoritmos de renomeamento serem para árvores lineares, é melhor realizar conversão para DAG na grande maioria das vezes. Portanto, a partir de agora, voltamos nossa atenção para comparações envolvendo somente combinações que realizam conversão para DAG.

### 4.2.4 Comparações entre os algoritmos de renomeamento

O primeiro resultado que observamos é bastante surpreendente e animador. Comparamos as Combinações 7 e 9, que aplicam a DAGs respectivamente os Algoritmos 1 e 2, sem aplicar simplificação ao final. No total, 73% das fórmulas foram transformadas com sucesso por ambas as combinações. Ocorre ainda que 3% das fórmulas foram transformadas com sucesso e a combinação com o algoritmo de Boy de la Tour produziu menos

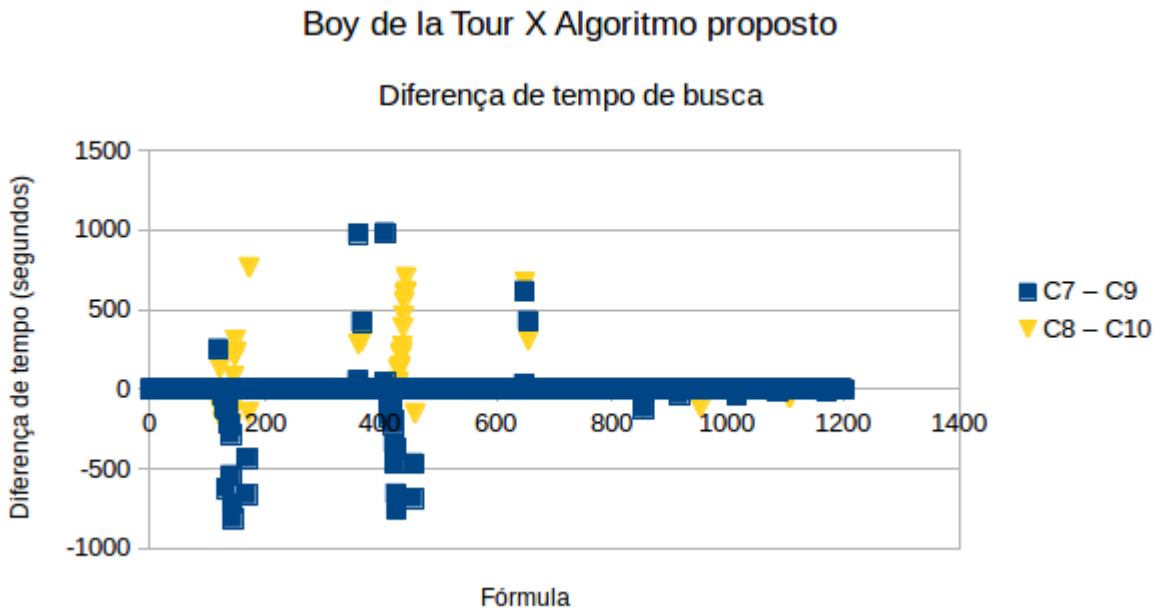


Figura 4.4: Boy de la Tour × Algoritmo proposto. Tempo de busca.

cláusulas, com diferença máxima de 3 cláusulas. Antes de mais nada, isto prova que o algoritmo que propomos não encontra renomeamentos ótimos para qualquer fórmula. Por outro lado, 8% das fórmulas foram transformadas com sucesso e, além disso, a combinação com o algoritmo que propomos produziu menos cláusulas. Desta vez, a diferença máxima chegou a 1.572.786 cláusulas, onde a fórmula produzida por nosso algoritmo gera apenas 78 cláusulas. Primeiro, isto prova que o algoritmo de Boy de la Tour também não encontra renomeamentos ótimos para qualquer fórmula. Mas este resultado também indica que o algoritmo que propomos, apesar de possuir a complexidade do algoritmo de Boy de la Tour multiplicada por um fator linear, pode produzir renomeamentos que geram muito menos cláusulas para algumas fórmulas.

Estes resultados ocorrem dentro de famílias de fórmulas específicas. O algoritmo de Boy de la Tour produziu menos cláusulas somente na família de fórmulas *nishimura*. O algoritmo que propomos produziu menos cláusulas principalmente nas famílias *SYJ205*, *SYJ206* e *SYJ212*. As diferenças mais discrepantes ocorrem nas famílias *SYJ206* e *SYJ212*, onde o algoritmo que propomos é capaz de encontrar subfórmulas em níveis mais altos para renomear, enquanto o algoritmo de Boy de la Tour não é. Observando ainda os resultados das combinações análogas que aplicam simplificação ao final, notamos que os resultados são semelhantes, apenas trocando as famílias em que as diferenças são pequenas e mantendo as famílias onde as diferenças são maiores.

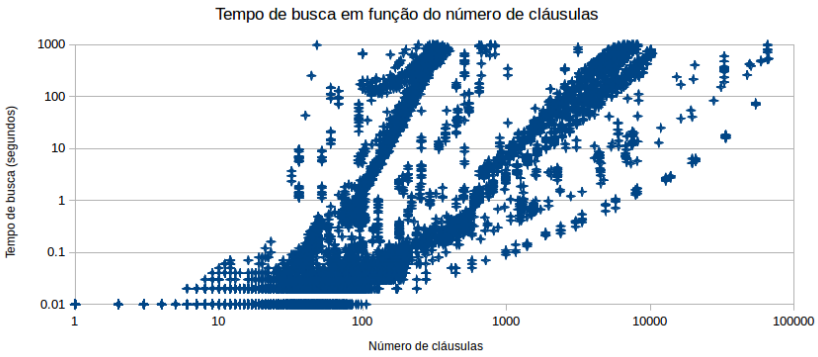


Figura 4.5: Tempo de busca em função do número de cláusulas.

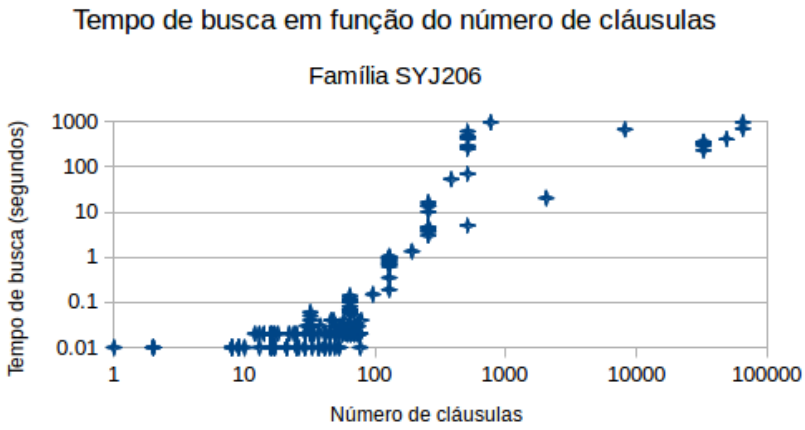


Figura 4.6: Tempo de busca em função do número de cláusulas. Família *SYJ206*.

Por último, comparamos os desempenhos das transformações produzidas pelos algoritmos de renomeamento durante a etapa de busca.

O gráfico da Figura 4.4 foi construído similarmente aos gráficos da Seção 4.2.3. O eixo horizontal identifica as fórmulas, enquanto o eixo vertical é a diferença entre o tempo de busca de $C_i$ (Boy de la Tour) e $C_{i+2}$ (algoritmo proposto), para $i = 7, 8$. Do gráfico, observamos que as concentrações de pontos acima e abaixo do eixo horizontal de diferença igual a zero são bastante parecidas, no sentido de que um algoritmo leva vantagem em algumas famílias de fórmulas, enquanto o outro leva vantagem em outras.

### 4.2.5 Tempo de busca em função do número de cláusulas

Finalmente, tentamos responder à pergunta inicial. O gráfico da Figura 4.5 é o tempo de busca em função do número de cláusulas. Neste gráfico, incluímos todas as execuções

feitas com sucesso. Como algumas transformações geraram o mesmo número de cláusulas, há múltiplos pontos em diversas posições ao longo do eixo horizontal. Do gráfico, notamos que o comportamento do tempo de busca em função do número de cláusulas não é estritamente crescente ou suave. Ora, curiosa seria a realidade em que esta relação funcional fosse de fato tão bem comportada. No entanto, é sim notável um certo grau de bom comportamento crescente em diversas partes do gráfico, que se misturam. E, muito provavelmente, isto ocorre dentro de cada família de fórmulas do *benchmark*, como mostra o gráfico da Figura 4.6, feito exclusivamente para a família *SYJ206*. Portanto, uma conclusão razoável é que, considerando fórmulas similares, a resposta para a nossa conjectura é positiva, ou seja, os algoritmos de busca são mais rápidos para fórmulas com menos cláusulas.

# Capítulo 5

# Conclusão

Neste trabalho, propomos a seguinte pergunta: fórmulas com menos cláusulas produzem respostas mais rápido? Como esforço para tentar responder esta pergunta, revisamos técnicas para reduzir o número de cláusulas geradas por uma fórmula proposicional, como as propostas por Boy de la Tour [4], Nonnengart et al. [16] e Jackson et al. [13], que são baseadas em renomeamento [17]. Após estudar estas técnicas, desenvolvemos também um algoritmo (Algoritmo 2), baseado em programação dinâmica [1], para obter renomeamentos que geram poucas cláusulas. Propomos ainda experimentos para: verificar uma propriedade de optimalidade restrita deste algoritmo; comparar este algoritmo com o de Boy de la Tour (Algoritmo 1); e tentar responder à pergunta levantada inicialmente. Dos resultados experimentais obtidos, apresentados no Capítulo 4, tiramos as seguintes conclusões principais:

1. O esforço de tentar reduzir o tamanho de uma fórmula ao convertê-la para uma forma normal compensa na grande maioria das vezes;
2. A representação de fórmulas através de DAGs é altamente recomendável para algoritmos de minimização baseados em renomeamento;
3. É bastante provável que o algoritmo que propomos produza renomeamentos ótimos para árvores lineares. Propomos, como trabalho futuro, provar esta afirmação analiticamente;
4. Diferentes algoritmos de renomeamento levam vantagem em diferentes famílias de fórmulas. Em particular, para as famílias *SYJ206* e *SYJ212* [18], o algoritmo que propomos gera muito menos cláusulas que o de Boy de la Tour;
5. Por fim, se consideramos fórmulas parecidas, ou seja, fórmulas com uma mesma estrutura principal em comum, as com menos cláusulas de fato produzem respostas mais rápido.

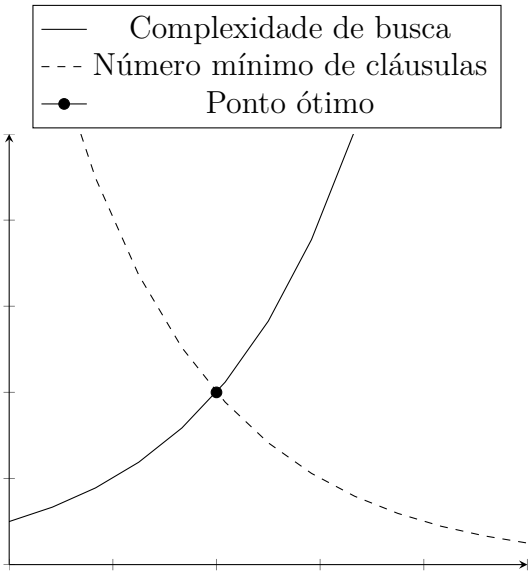


Figura 5.1: Ponto limite ótimo para o número de símbolos proposicionais.

Um ponto ainda não explorado do algoritmo proposto é sua capacidade de obter bons renomeamentos com um dado limite fixo para o número de subfórmulas que podem ser escolhidas. Observamos que esta propriedade pode ser bastante útil, dado que as complexidades dos algoritmos de busca para SAT e VAL crescem justamente com o número de símbolos proposicionais de uma fórmula [8, 7, 2]. Ao limitar o número de subfórmulas que podem ser escolhidas para renomeamento, deixamos de reduzir completamente o número de cláusulas, mas também impedimos que o número de símbolos proposicionais da transformação resultante cresça demais. Como trabalho futuro, propomos encontrar uma maneira de determinar um ponto limite ótimo para o número de subfórmulas que podem ser escolhidas para renomeamento. Esta ideia está ilustrada no gráfico da Figura 5.1. Uma vantagem adicional que pode surgir ao explorar este ponto é o fato de que um dos fatores lineares da complexidade do algoritmo vem justamente do número máximo de subfórmulas que podem ser escolhidas. No limite em que este número tende a zero, a complexidade do algoritmo tende a ficar quadrática, como a complexidade do algoritmo de Boy de la Tour. Portanto, o trabalho futuro agora proposto tem potencial para reduzir os tempos de execução de ambas as etapas: pré-processamento e busca.

Por último, observamos que o Algoritmo 2 se baseia principalmente na tomada de decisão da Linha 5. É provável que este critério possa ser apropriadamente substituído para resolver outros problemas, como minimização de outras formas normais, por exemplo. Como último trabalho futuro, deixamos a tarefa de investigar estas possíveis substituições.

# Referências


[1] Richard E. Bellman e Stuart E. Dreyfus. *Applied Dynamic Programming*. Princeton University Press, 1962. 22, 26, 38

[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, 39

[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] Thierry Boy de la Tour. An optimality result for clause form translation. *Journal of Symbolic Computation*, 14(4):283–301, 1992. 2, 15, 16, 24, 38

[5] 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

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, e Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3 edition, 2009. 19, 26, 27, 29

[7] Martin Davis, George Logemann, e Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962. 2, 39

[8] Martin Davis e Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960. 2, 6, 39

[9] 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

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

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

[12] ISO. ISO International Standard ISO/IEC 14882:2011(E) - Programming Language C++, 2011. 26

[13] Paul Jackson e Daniel Sheridan. Clause form conversions for boolean circuits. In *Theory and applications of satisfiability testing*, pages 183–198. Springer, 2004. 2, 24, 25, 38

[14] Stephen C. Kleene. *Mathematical Logic*. Wiley and Sons, 1968. 6

[15] 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

[16] Andreas Nonnengart e Christoph Weidenbach. Computing small clause normal forms. *Handbook of automated reasoning*, 1:335–367, 2001. 2, 12, 24, 38

[17] David A. Plaisted e Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986. 15, 38

[18] Thomas Raths, Jens Otten, e Christoph Kreitz. The ILTP problem library for intuitionistic logic. *Journal of Automated Reasoning*, 38(1):261–271, 2007. 30, 38

[19] Stephan Schulz. System Description: E 1.8. In Ken McMillan, Aart Middeldorp, e Andrei Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*. Springer, 2013. 30

[20] Roberto Sebastiani e Michele Vescovi. Automated reasoning in modal and description logics via SAT encoding: the case study of K(m)/ALC-satisfiability. *Journal of Artificial Intelligence Research*, 35(1):343, 2009. 27

[21] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 3rd edition, 2012. 25, 26

[22] Geoff Sutcliffe. The TPTP problem library and associated infrastructure. *Journal of Automated Reasoning*, 43(4):337–362, 2009. 30

[23] Daniel H. Younger. Recognition and parsing of context-free languages in time $n^3$. *Information and control*, 10(2):189–208, 1967. 25