UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Murillo Lagranha Flores

MELHORAMENTOS PARA O ALGORITMO DE TRANSFORMAÇÃO DUAL

Florianópolis(SC)

2012

Murillo Lagranha Flores

MELHORAMENTOS PARA O ALGORITMO DE TRANSFORMAÇÃO DUAL

TCC submetido ao Curso de Bacharelado em Sistemas de Informação para a obtenção do Grau de Bacharel.
Orientadora: Jerusa Marchi, Dra.

Florianópolis(SC)

2012

Murillo Lagranha Flores

MELHORAMENTOS PARA O ALGORITMO DE TRANSFORMAÇÃO DUAL

Este TCC foi julgado aprovado para a obtenção do Título de "Bacharel", e aprovado em sua forma final pelo Curso de Bacharelado em Sistemas de Informação.

Florianópolis(SC), 19 de dezembro 2012.

Renato Cislaghi, Dr.
Coordenador

Jerusa Marchi, Dra.
Orientadora

Banca Examinadora:

Olinto José Varela Furtado, Dr.

Eduardo Camponogara, Dr.

Elder Santos, Dr.

À minha mãe, meu maior exemplo por toda vida.

AGRADECIMENTOS

A Deus, por ter criado um enredo tão incrível para a vida, e por sua eterna e imutável sabedoria.

Gostaria de agradecer imensamente a todos aqueles que participaram junto comigo destes 16 anos de aulas. Desde a primeira aula com a tia Glacir no "pré " até a última aula no CTC, todas foram grandes lições. A todos do Lauro, do TAN, do Médici e do CAC (certamente o Rafa escreveria C|A|C), um muito obrigado. É muito bom olhar pra trás e ver como as coisas se conectam e como fizemos grandes amigos.

Também gostaria muito de agradecer aos meus colegas da 081 por terem feito as longas noites e as aulas chatas, bem mais divertidas. Ao Alex, ao Leandro, à Debora, ao Andrei, ao Grolli e a todos, um valeu galera :-). Sigamos todos o exemplo da sabia tia adriana.

A todos aqueles com quem eu trabalhei, e com quem eu aprendi muito, lá na SEaD, na CERTI, no Stella e na Chaordic: Senhores e senhoritas, obrigado.

Ao clã dos Flores e dos Lagranha, lá no Alegrete ou ali em Porto Alegre um especial obrigado. Vocês foram e continuam sendo uma grande fonte de inspiração e de aprendizagem para mim.

Aos mestres, da Tia Tânia, até o Prof. Dr. João Candido Lima Dovicchi a minha eterna gratidão. Eu realmente tenho paixão em aprender e vocês me deram as maiores alegrias da vida.

E por fim, um especial e carinhoso obrigado às mulheres sem as quais nada disso estaria acontecendo: À minha orientadora, Professora Jerusa Marchi, por ter tido a paciência e a coragem de me orientar por esse caminho pantanoso dos trabalhos acadêmicos. À minha namorada, futura Dra. Kamilla, por ter sempre acreditado em mim, por ter realmente admirado o meu trabalho e por ter feito a minha vida bem mais bela. E à minha mãe, dona Carmen ou Professora Carmen (mas pra mim só mãe mesmo) por ter estado ao meu lado estes anos todos, por ter demonstrado amor eterno, por ter me ensinado tudo o que há para se saber na vida e por ter me tornado um homem livre em alma e pensamento. Tu és o meu maior exemplo e motivo de orgulho.

A capacidade de nos surpreendermos é a única coisa de que precisamos para nos tornarmos bons filósofos.

Jostein Gaarder

RESUMO

O problema da satisfazibilidade Booleana, ou problema SAT, é um problema clássico em computação. Foi o primeiro problema provado NP-Completo e melhorar a forma como trabalhamos com ele significa melhorar a forma como trabalhamos com complexidade em geral. Este trabalho apresenta o Algoritmo de Transformação Dual, que é um algoritmo de resolução do problema SAT, pronpondo-lhe melhorias, e demonstrando que tais melhorias reduzem o custo computacional associado à resolução de SAT por tal algoritmo.

Palavras-chave: Satisfazibilidade, SAT, Algoritmo de Transformação Dual, melhorias

ABSTRACT

The problem of Boolean satisfiability, or SAT problem is a classic problem in computing. It was the first proven NP-Complete problem and improve the way we deal with it means improving the way we deal with complexity in general. This report presents the "Dual Transformation Algorithm", which is an algorithm for solving the SAT problem, proposing improvements to it, and demonstrating that these improvements reduce the computational cost associated with solving SAT by this algorithm.
Keywords: satisfiability, SAT, Algoritmo de Transformação Dual, melhorias

SUMÁRIO

1 INTRODUÇÃO

Dentro da teoria da computação, o estudo da complexidade de algoritmos é de fundamental importância pois nos permite classificar os problemas de acordo com os algoritmos associados à sua resolução. Nestas classificações os problemas NP-Completos são tidos como os de resolução associada mais custosa (ROSEN, 2010).

O problema da satisfazibilidade Booleana, ou problema SAT, é um problema NP-Completo, pertencendo portanto à classe de problemas de resolução custosa. O problema SAT é bastante importante em uma série de aplicações práticas, tais quais a automatização de desenho de componentes eletrônicos, a verificação de software e de hardware e a inteligência artificial. Contudo, desde há muito tempo, algoritmos e métodos para a resolução de tal problema vêm sendo propostos e aprimorados para diferentes contextos/usos específicos, tornando possível inclusive a classificação de tais estudos de acordo com uma série de características. Uma destas características que permite a classificação é a completude do algoritmo - se consegue retornar a resposta, ao invés de apenas dizer que ela existe. O Algoritmo de Transformação Dual é um algoritmo completo para a resolução do problema SAT.

Este trabalho tem por objetivo propor três melhorias ao Algoritmo de Transformação Dual e comprovar que as mesmas reduzem o tempo de resposta e a quantidade de memória utlizada pelo mesmo, permitindo assim que instâncias maiores (com mais cláusulas e variáveis proposicionais) sejam resolvidas com menos recursos computacionais. Como resultado é esperada a comprovação desta hipótese de trabalho através da análise dos resultados obtidos nos testes. Alcançar tal objetivo significa descrever as melhorias propostas, implementá-las, executar os testes e comparar os dados obtidos, comprovando a hipótese.

1.1 MOTIVAÇÃO

Atualmente existem basicamente dois tipos de métodos para determinar a satisfazibilidade de uma fórmula lógica, ou seja, resolver uma instância do problema SAT: variantes modernas do algoritmo proposto em 1962 por Davis-Putnam-Logemann-Loveland - DPLL - tais como Chaff, Grasp e march, e algoritmos estocásticos, como o WalkSat. Os primeiros precisam que a fórmula lógica de entrada esteja na forma normal conjuntiva, que é uma representação em uma forma padrão bem definida, para que seja possível computar sua satisfazibilidade. Os segundos, estocásticos, não tem esta restrição

e por isso tendem a ser vantajosos para problemas que são naturalmente mais fáceis de descrever com uma fórmula proposicional arbitrária (que não está na forma normal conjuntiva) (MARIć, 2009).

Guilherme Bittencourt propôs um método de resolução não-estocástico baseado em uma notação especial, chamada notação Quantum (BITTENCOURT; MARCHI; PADILHA, 2003). A hipótese deste trabalho é a de que este algoritmo proposto por Bittencourt, Marchi e Padilha (2003) tem espaço para melhorias teóricas e práticas que reduzirão o seu tempo de execução e a quantidade de memória utilizada.

1.2 OBJETIVOS DO TRABALHO

O objetivo geral deste trabalho é implementar o Algoritmo de Transformação Dual propondo-lhe melhorias, adicionar estas melhorias à implementação e comparar o desempenho das versões com e sem melhorias na resolução de instâncias do SAT, a fim de verificar se existe um melhor desempenho da versão com melhorias.

Objetivos específicos

Os seguintes são objetivos específicos do presente trabalho:

- Apresentar o problema da satisfazibilidade Booleana e as diferentes abordagens de resolução que este pode ter (ex.: resolução completa x incompleta);
- Implementar o Algoritmo de Transformação Dual;
- Propor melhorias ao Algoritmo de Transformação Dual e implementá-las;
- Executar os testes com as duas implementações desenvolvidas;
- Analisar os resultados dos testes comparando as duas versões.

1.3 APRESENTAÇÃO DO TRABALHO

Este trabalho está dividido em nove capítulos; esta introdução e outros oito.

Os capítulo dois, três e quatro apresentam a fundamentação teórica do trabalho, descrevendo lógica proposicional, complexidade computacional e, por fim, conceituando o problema SAT em função destes dois.

O capítulo cinco apresenta o Algoritmo de Transformação Dual, enquanto o capítulo seis apresenta as melhorias propostas ao mesmo. Já o capítulo sete apresenta a implementação do Algoritmo. Estes três capítulos (cinco, seis e sete) compreendem juntos o "desenvolvimento" do trabalho. É neles que o trabalho feito para se comprovar a hipótese é descrito.

O capítulo oito apresenta os resultados dos testes e a análise dos mesmo. É neste capítulo que os resultados esperados, expostos nesta introdução, serão apresentados.

O capítulo nove encerra com as conclusões alcançadas e com sugestões para trabalhos futuros.

2 LÓGICA PROPOSICIONAL

A lógica proposicional, ou lógica Booleana, estuda as proposições e a combinação destas através de conectivos (ROSEN, 2010). Como toda lógica, ela pode ser definida em função da sua sintaxe e da sua semântica (RUSSELL; NORVIG, 2010).

Neste capítulo apresenta-se a sintaxe e a semântica da lógica proposicional, pois ela é a base para o entendimento do problema SAT. Conceitos relacionados, que também estão envolvidos com este trabalho, são aqui também apresentados.

2.1 SINTAXE

A sintaxe de uma lógica define quais sentenças são permitidas (RUSSELL; NORVIG, 2010). Em lógica proposicional, a menor sentença permitida é a sentença atômica, que consiste em um único símbolo proposicional. Cada símbolo proposicional representa uma proposição que pode ser verdadeira ou falsa. Assim, o símbolo proposicional referente a uma proposição falsa, tem o valor falso, e o símbolo referente a uma proposição verdadeira, tem o valor verdadeiro. Um símbolo proposicional representa uma variável proposicional. Cada variável proposicional é geralmente representada por uma letra (RUSSELL; NORVIG, 2010).

Florianópolis é uma cidade é um exemplo de proposição, que pode ser representada pela variável de símbolo **P**. Neste caso, sabemos inerentemente que Florianópolis é de fato uma cidade, então **P = verdadeiro**

Sentenças complexas de lógica proposicional podem ser construídas a partir do uso de conectivos lógicos. Existem vários conectivos lógicos, mas os cinco mais comumente usados e que são base deste trabalho, como descritos em (RUSSELL; NORVIG, 2010), são:

$\neg$ **(negação)** Uma sentença como $\neg P$ é chamada de negação de P. Um **literal** é uma sentença atômica (um **literal positivo**) ou uma sentença atômica negada (um **literal negativo**).

$\wedge$ **(e)** Uma sentença cujo principal conectivo é $\wedge$, como $P \wedge Q$, é chamada de **conjunção**; suas partes são os elementos da **conjunção**.

$\vee$ **(ou)** Uma sentença que utiliza $\vee$, como $(P \wedge Q) \vee R$ é uma **disjunção** dos **disjuntos** $(P \wedge Q)$ e R.

$\Rightarrow$ **(implica)** Uma sentença com $(Q \wedge A) \Rightarrow J$ é dita uma implicação. $(Q \wedge A)$ é a premissa e J é a conclusão.

$\Leftrightarrow$ **(se e somente se)** A sentença que utiliza este conector é uma sentença bicondicional, como em $R \Leftrightarrow \neg U$.

Para construir-se as sentenças complexas com estes conectivos, e a partir de sentenças mais simples, valem as regras da seguinte gramática livre de contexto, apresentada na notação de Backus-Naur.

$$
\begin{array}{rcl}
\langle\ \text{Sentença}\ \rangle & ::= & \langle\ \text{SentençaAtômica}\ \rangle \\
 & & \mid \langle\ \text{SentençaComplexa}\ \rangle \\
\langle\ \text{SentençaAtômica}\ \rangle & ::= & \text{Verdadeiro} \mid \text{Falso} \mid \langle\ \text{Símbolo}\ \rangle \\
\langle\ \text{Símbolo}\ \rangle & ::= & \text{P} \mid \text{Q} \mid \text{R} \mid \cdots \\
\langle\ \text{SentençaComplexa}\ \rangle & ::= & \neg \langle\ \text{Sentença}\ \rangle \\
 & & \mid (\langle\ \text{Sentença}\ \rangle \wedge \langle\ \text{Sentença}\ \rangle) \\
 & & \mid (\langle\ \text{Sentença}\ \rangle \vee \langle\ \text{Sentença}\ \rangle) \\
 & & \mid (\langle\ \text{Sentença}\ \rangle \Rightarrow \langle\ \text{Sentença}\ \rangle) \\
 & & \mid (\langle\ \text{Sentença}\ \rangle \Leftrightarrow \langle\ \text{Sentença}\ \rangle)
\end{array}
$$

Figura 1: Gramática da sintaxe da lógica proposicional

A gramática define um uso bastante estrito dos parênteses a fim de evitar possíveis ambiguidades nas construções sintáticas válidas. Porém para facilitar a leitura, muitas vezes os parênteses são omitidos e por isso é importante saber a ordem de precedência dos conectores em lógica proposicional. A ordem de precedência, do maior para o menor, é: $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$.

Uma teoria é um conjunto de proposições. Uma fórmula em lógica proposicional é um conjunto de variáveis proposicionais que representam, cada uma, uma proposição e, logo, uma fórmula em lógica proposicional representa uma teoria.

2.2 SEMÂNTICA

A semântica da lógica proposicional define uma maneira de se computar o valor verdade de uma sentença respeitando um conjunto de atribuições

para as suas variáveis proposicionais. Este conjunto de atribuições são interpretações possíveis. Se uma interpretação torna a sentença verdadeira, diz-se que é um modelo da sentença. Dada a fórmula $p \wedge q$, são interpretações possíveis [{p=verdadeiro, q=verdadeiro}, {p=verdadeiro, q=falso}, {p=falso, q=verdadeiro}, {p=falso, q=falso}]. O único modelo de $p \wedge q$ é {p=verdadeiro, q=verdadeiro}.

O cálculo dos valores verdade das sentenças é, então, feito recursivamente. Todas as sentenças podem ser construídas a partir de sentenças atômicas e dos cinco conectivos apresentados na seção 2.1 . Portanto precisamos saber como computar o valor verdade para sentenças atômicas e como computar o valor verdade para sentenças construídas com os conectivos. Para sentenças atômicas as regras são simples:

- Valores literais (verdadeiro e falso) já tem valor definido em qualquer interpretação;
- Para as demais variáveis o valor verdade deve ser especificado diretamente na interpretação;

Para as sentenças construídas com os conectivos (sentenças complexas), as regras são um pouco mais numerosas, e por isso estão sumarizadas na tabela verdade abaixo (onde V = verdadeiro e F = falso). É fácil notar que o valor verdade de uma sentença complexa pode ser facilmente computado a partir dos valores verdade se suas partes componentes (sentenças simples), e isso é feito recursivamente reduzindo a complexidade da sentenças.

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
F	F	V	F	F	V	V
F	V	V	F	V	V	F
V	F	F	F	V	F	F
V	V	F	V	V	F	V

Tabela 1: Tabela verdade para os conectivos da lógica proposicional

2.3 EQUIVALÊNCIA, VALIDADE E SATISFAZIBILIDADE

Três conceitos importantes para se conhecer melhor a lógica proposicional são os de equivalência, validade e satisfazibilidade.

O primeiro conceito, o de equivalência lógica diz: Duas sentenças α e β são logicamente equivalentes se elas são verdade no mesmo conjunto de interpretações. Por exemplo, podemos facilmente provar através da tabela

verdade apresentada anteriormente que (P $\wedge$ Q) e (Q $\wedge$ P) são logicamente equivalentes.

O segundo conceito é o de validade. Dizemos que uma sentença é válida se ela for verdade em todas as interpretações possíveis. Por exemplo, a sentença $P \vee \neg P$ é válida. As sentenças válidas também são conhecidas como tautologias e são logicamente equivalentes a verdadeiro ($P \vee \neg P \Leftrightarrow$ *Verdadeiro*).

Finalmente chegamos ao terceiro e final conceito, o de satisfazibilidade. Uma sentença α é satisfazível se ela é verdade em alguma interpretação. Se α é verdade em um modelo m, então dizemos que m satisfaz α ou que m é um modelo de α.

2.4 FORMAS NORMAIS CANÔNICAS - CNF E DNF

Toda sentença da lógica proposicional é logicamente equivalente a uma conjunção de disjunções de literais (RUSSELL; NORVIG, 2010). Quando uma dada fórmula F é uma conjunção de disjunções de literais, ela tem a forma:

$$F = C_1 \wedge C_2 \wedge \cdots \wedge C_n$$

onde:

$$C_i = L_1 \vee L_2 \vee \cdots \vee L_m$$

e L_i é um literal. Diz-se que esta sentença está na **Forma normal conjuntiva** ou **CNF**. A forma normal conjuntiva é uma forma normal canônica. Um procedimento simples para a conversão de qualquer fórmula da lógica proposicional em uma fórmula equivalente na forma normal conjuntiva é apresentado em Russell e Norvig (2010) página 209. Também é bastante importante o fato de que existem famílias de sentenças restritas a K-CNF. Uma sentença K-CNF tem exatamente K literais por cláusula. Todos os conjuntos de testes utilizados no capítulo 8 estão em 3-CNF. Toda sentença pode ser transformada em uma sentença em 3-CNF que tem um conjunto equivalente de modelos (RUSSELL; NORVIG, 2010).

A forma normal conjuntiva é bastante usada como entrada por algoritmos de resolução de fórmulas da lógica proposicional por conta das seguintes propriedades (BOAVA, 2005):

1. F terá valor “verdadeiro” se, e somente se, todas as suas cláusulas tiverem valor “verdadeiro”.

2. Se C for uma cláusula de F, então C terá valor "verdadeiro" se pelo menos um de seus literais tiver valor "verdadeiro".

Outra forma normal canônica é a **forma normal disjuntiva** ou **DNF**. Uma fórmula F está na forma normal disjuntiva se ela tem a forma:

$$F = C_1 \vee C_2 \vee \cdots \vee C_n$$

onde:

$$C_i = L_1 \wedge L_2 \wedge \cdots \wedge L_m$$

Também toda fórmula da lógica proposicional tem uma fórmula equivalente na forma normal disjuntiva (ROSEN, 2010). Um pequeno procedimento para a transformação de uma fórmula qualquer da lógica proposicional em uma fórmula na forma normal disjuntiva é mostrado em Boava (2005) página 13.

Assim como a CNF, a DNF de uma fórmula F possui algumas propriedades que tornam o seu uso em resolvedores SAT bastante útil. São elas:

1. F terá valor verdadeiro se pelo menos uma de suas cláusulas tiver valor verdadeiro.

2. Se C for uma cláusula de F, C terá valor verdadeiro se, e somente se, todos os seus literais tiverem valor verdadeiro.

Assim, para uma fórmula W qualquer da lógica proposicional, temos uma fórmula equivalente W_c em CNF, e uma outra W_d em DNF.

$$W \Leftrightarrow W_c \Leftrightarrow W_d$$

Para ambas as formas normais canônicas, se uma dada cláusula C é composta por apenas um literal, diz-se que C é uma cláusula unitária.

2.5 FÓRMULA DUAL E CLÁUSULAS DUAIS

Uma fórmula na forma normal disjuntiva é uma fórmula dual de uma fórmula na forma normal conjuntiva se ambas são logicamente equivalentes (BITTENCOURT; MARCHI; PADILHA, 2003). Da mesma maneira uma fórmula na forma normal conjuntiva é a fórmula dual de uma fórmula na forma normal disjuntiva se elas são logicamente equivalentes (reflexividade). De maneira genérica, a forma normal disjuntiva é dita **forma dual** da forma normal conjuntiva (e vice-versa). As cláusulas de fórmulas duais, são ditas cláusulas duais.

O Algoritmo de Transformação Dual, objeto de estudo deste trabalho é, em essência, um algoritmo de transformação de uma fórmula em CNF para uma fórmula em DNF.

3 COMPLEXIDADE COMPUTACIONAL

Intuitivamente, analisar a complexidade de um algoritmo significa analisar a quantidade de tempo e espaço necessários para que o mesmo calcule a saída para uma entrada qualquer e a relação que esta quantidade de tempo e espaço tem com o tamanho da entrada (VORONKOV, 2012). Em computação, o padrão mais usual para realizar tal tipo de análise e para classificar os algoritmos, vem da definição da Máquina de Turing. Para entendermos a Máquina de Turing e a maneira como, através desta, classificaremos a complexidade dos algoritmos, vamos primeiramente definir linguagem.

3.1 LINGUAGEM

Um alfabeto é um conjunto não vazio de símbolos. Uma *string* sobre este alfabeto é uma sequência de símbolos do alfabeto. Por sua vez, uma linguagem é um conjunto de *strings* sobre um determinado alfabeto. (SIPSER, 1996)

O tamanho da sequência de símbolos que compõe uma *string* s qualquer, é denotado por $|s|$. Por definição, toda linguagem possui uma *string* de tamanho zero denotado por ε.

Como exemplo, seja $\Sigma = \{a,b\}$ um alfabeto, são *strings* sobre Σ: $\varepsilon, a, b, ab, aaaaba, ...$

3.2 MÁQUINA DE TURING

A máquina de Turing, é um modelo matemático de dispositivo computacional, proposto em 1936 por Alan Turing. É até hoje o modelo computacional mais expressivo e problemas que não são solucionáveis por uma Máquina de Turing, são ditos incomputáveis. Todos os demais problemas, em computação, podem ser reduzidos ao problema que a máquina de Turing resolve. As definições seguintes são baseadas em Sipser (1996)

O objetivo da máquina é, de fato, reconhecer se uma determinada *string* pertence ao alfabeto de uma linguagem ou não. Para tanto, esta usa uma fita infinita como memória e possui um cabeçote que pode ler e escrever símbolos nesta fita. Inicialmente, a *string* que está sendo testada está na posição mais a esquerda da fita, como entrada. Todo o resto da fita, que é infinita, está preenchida com espaços vazios. A máquina de Turing possui um conjunto pré definido de estados internos. Ela inicia em um estado inicial, e

vai mudando de estado conforme avança na leitura da fita. Dois estados da Máquina são finais: o estado no qual a Máquina aceita a *string* (reconhece que a mesma pertence ao alfabeto) e o estado no qual a Máquina rejeita a *string*. Ao alcançar qualquer um dos estados finais a Máquina para. Cada vez que um símbolo da fita é lido, uma função de transição é aplicada para determinar qual o próximo estado para o qual a Máquina vai e para onde o cabeçote deve deslocar-se (esquerda ou direita). Esta função de transição leva em conta o estado atual da Máquina. Após a execução da função de transição, diz-se que a Máquina assume uma nova configuração, que é formada pelo estado atual da Máquina e pelo conteúdo da fita. Uma definição formal da Máquina de Turing, pode ser encontrada em Sipser (1996).

Existem vários modelos equivalentes a Máquina de Turing em poder de reconhecimento, sendo a Máquina de Turing Não-Determinística um deles. A diferença desta, em comparação com a Máquina anteriormente apresentada é que a ela pode, pela função de transição, assumir vários estados a partir de uma configuração qualquer. A computação de tal Máquina Não-Determinística forma uma árvore, cujos ramos representam as possibilidades dadas pela função de transição. Nesta Máquina, se algum ramo encontra o estado de aceitação, a Máquina aceita a entrada. Todos os ramos devem falhar para que a Máquina rejeite a entrada.

3.3 CLASSES DE COMPLEXIDADE

A partir das definições das Máquinas de Turing Determinística e Não-Determinística, derivam as classes de complexidade que classificam as linguagens reconhecidas pelas mesmas, segundo tais definições.De maneira resumida, existem duas classes:

Classe P A classe P é a classe das linguagens que são decidíveis em tempo polinomial numa **Máquina de Turing determinística**.

Classe NP A classe NP é a classe das linguagens que são decidíveis em tempo polinomial numa **Máquina de Turing Não-Determinística**.

Existem maneiras alternativas de definir estas classes de complexidade. Como todos os demais problemas podem ser reduzidos a problemas de linguagem, esta classificação se aplica para problemas em geral.

Como uma Máquina Não-Determinística pode se comportar de maneira determinística, é fácil ver que $P \subseteq NP$. Por outro lado, saber se $NP \subseteq P$, ou seja, se $P = NP$ é um problema aberto, muito importante em teoria da computação.

3.4 NP-COMPLETUDE

Dentro da classe NP, existem linguagens que são ditas NP-Completas. Uma linguagem é NP-Completa se todas as demais linguagens NP forem reduzíveis, em tempo polinomial, a ela. Uma redução é uma transformação de um problema em outro ou, neste caso, de uma linguagem em outra. Um exemplo de redução foi apresentado em Rauber (2011).

A primeira linguagem a ser provada NP-Completa foi SAT, linguagem que modela o problema da satisfazibilidade Booleana, central a este trabalho. A prova pode ser encontrada em Sipser (1996). Várias outras linguagens foram provadas NP-Completas a partir da prova da NP-Completude de SAT. A importância teórica das linguagens NP-Completas é grande, porque melhorar a maneira como lidamos com estas linguagens significa melhorar a maneira como lidamos com a complexidade em geral.

4 PROBLEMA SAT

A partir dos conceitos básicos da lógica proposicional, podemos definir o problema da satisfazibilidade booleana ou problema SAT, como o problema de se encontrar os modelos possíveis para uma dada fórmula da lógica proposicional, ou determinar se tais modelos não existem.

Problema SAT - Seja F uma fórmula da lógica proposicional expressa em CNF:

$$F = C_1 \wedge C_2 \wedge \cdots \wedge C_n$$

onde:

$$C_i = L_1 \vee L_2 \vee \cdots \vee L_m$$

e $L_{i,j}$ seu literais. Sejam $x_1, x_2, ..., x_n$ as variáveis proposicionais de F e x o vetor binário $x = [x_1, x_2, ..., x_n]$ que fixa valores verdade para as variáveis da fórmula. Diz-se que F é satisfazível se existir um $x_0 \in \{Verdadeiro, Falso\}^n$ tal que F tenha valor verdade verdadeiro quando $x = x_0$. Em outras palavras H é satisfazível se pode ser tornada verdadeira mediante determinados valores de suas variáveis. Por outro lado, H é instatisfazível, se não existe $x_0 \in \{Verdadeiro, Falso\}^n$ tal que $x = x_0 \Rightarrow F = Verdadeiro$. O problema da satisfazibilidade booleana, ou problema SAT, é o problema de encontrar x_0 ou determinar quando x_0 não existe. Note-se que podem existir mais de um x_0 para uma determinada fórmula.

4.1 K-SAT

Sabe-se que toda fórmula da lógia proposicional tem uma fórmula equivalente na forma normal conjuntiva- CNF (ver capítulo 2). Sabe-se ainda que a representação em CNF pode ter cláusulas de mesmo comprimento k. Quando todas as cláusulas de uma determinada fórmula em CNF, instância de SAT, tem o mesmo comprimento k, diz-se que esta é uma instância k-SAT do problema. Os problemas 1-SAT e 2-SAT não fazem parte da classe de problemas NP (ROSEN, 2010). O problema é NP-Completo para todo k-SAT com $k > 2$. Todo k-SAT com $k > 3$ pode ser reduzido para um problema 3-SAT. Esta análise de complexidade é baseada na análise de pior caso. Contudo é interessante levar em conta os valores médios para cada tipo de instância. A escolha dos pacotes de teste para este trabalho foi influenciada pela análise

e classificação das instâncias de SAT em função da razão entre o número de cláusulas (m) e o número de variáveis (n) das mesmas, conhecida como razão m/n apresentada por Crawford e Auton (1996).

4.2 CLASSES DE RESOLVEDORES

Existem duas classes de resolvedores de SAT: Os baseados em métodos estocásticos e os baseados em métodos completos.

Os algoritmos baseados em métodos estocásticos são bastante eficientes para encontrar rapidamente soluções para determinados tipos de instâncias SAT. Contudo, eles não conseguem dizer se uma determinada instância é insatisfazível e nem são ótimos no caso médio. Existem diversos tipos de algoritmos estocásticos criados a partir das mais diferentes técnicas como redes neurais, algoritmos genéticos, algoritmos de busca local, etc. (BOAVA, 2005). Em sua maioria, os algoritmos baseados em métdos estocásticos não têm uma fundamentação formal, sendo baseados em heurísticas nem sempre demonstradas.

Os algoritmos completos, por outro lado, são capazes de dizer se uma determinada instância é instatisfazível e de retornar uma solução para essa instância, se ela existir. O Algoritmo de Transformação Dual, objeto de estudo deste trabalho, é um algoritmo completo para a resolução de SAT.

5 ALGORITMO DE TRANSFORMAÇÃO DUAL

O Algoritmo de Transformação Dual é um algoritmo completo para a resolução do problema SAT. É também objeto de estudo e proposta de melhorias deste trabalho, e foi idealizado no Departamento de Automação e Sistemas da Universidade Federal de Santa Catarina e apresentado no *"4th International Workshop on the Implementation of Logic"* (BITTENCOURT; MARCHI; PADILHA, 2003). No artigo de apresentação nenhum nome foi proposto ao mesmo. Ainda assim o denominaremos de "Algoritmo de Transformacão Dual", em função da abordagem que este emprega para a resolução de SAT. Tal abordagem objetiva calcular o conjunto de cláusulas da representacão DNF para uma teoria representada inicialmente em CNF, ou seja, calcular a fórmula dual da fórmula de entrada. Se o Algoritmo conseguir calcular pelo menos uma cláusula dual, então a instância SAT é satisfazível.

A ideia por trás do Algoritmo se aproveita da definição, apresentada na seção 2.4, sobre uma fórmula F em CNF:

1. F terá valor "verdadeiro" se, e somente se, todas as suas cláusulas tiverem valor "verdadeiro".
2. Se C for uma cláusula de F, então C terá valor "verdadeiro" se pelo menos um de seus literais tiver valor "verdadeiro".

A partir desta definição é fácil perceber que um modelo mínimo de F é aquele que tem atribuição apenas ao mínimo possível de variáveis proposicionais de maneira a tornar pelo menos um literal de cada cláusula de F verdadeiro. Se, pelo menos um literal de cada cláusula de F for verdadeiro, então F será satisfazível. Como exemplo, tomemos uma teoria κ cujas cláusulas da representação em CNF sejam (a represetação em CNF pode ser referenciada com W_c):

$$
\begin{array}{lcllcl}
0 & : & [\neg P_4, P_2, \neg P_3] & 5 & : & [\neg P_3, P_2, \neg P_1] \\
1 & : & [P_5, \neg P_2, \neg P_3] & 6 & : & [\neg P_2, \neg P_1, P_5] \\
2 & : & [P_5, \neg P_3, P_1] & 7 & : & [P_4, P_5, P_2] \\
3 & : & [\neg P_5, \neg P_2, P_3] & 8 & : & [\neg P_1, P_4, P_3] \\
4 & : & [\neg P_2, \neg P_3, \neg P_1] & 9 & : & [\neg P_3, \neg P_1, \neg P_4]
\end{array}
$$

Um possível modelo mínimo para esta teoria seria $m = [P_5 = Falso, P_2 = Verdadeiro, P_1 = Falso, P_3 = Falso]$. Esse modelo pode ser representado pelo conjunto dos literais que fazem cada cláusula da teoria em CNF se tornar verdadeira, ficando $m = [\neg P_5, P_2, \neg P_1, \neg P_3]$. Diz-se que esta abordagem

de construção de modelos explora a fórmula em CNF de maneira sintática, pois busca um *caminho* pela *forma* da fórmula, pela sua representação. A construção deste "caminho" gera um modelo.

$$
\begin{array}{lllllll}
0 & : & [\neg P_4, P_2, \neg P_3] & 5 & : & [\neg P_3, P_2, \neg P_1] \\
1 & : & [P_5, \neg P_2, \neg P_3] & 6 & : & [\neg P_2, \neg P_1, P_5] \\
2 & : & [P_5, \neg P_3, P_1] & 7 & : & [P_4, P_5, P_2] \\
3 & : & [\neg P_5, \neg P_2, P_3] & 8 & : & [\neg P_1, P_4, P_3] \\
4 & : & [\neg P_2, \neg P_3, \neg P_1] & 9 & : & [\neg P_3, \neg P_1, \neg P_4]
\end{array}
$$

Figura 2: Ilustração de um caminho gerado pelo Algoritmo de Transformação Dual.

A ideia do algoritmo, portanto, é encontrar estes "caminhos", gerando modelos para a teoria em CNF (BITTENCOURT; MARCHI; PADILHA, 2003).

Os caminhos encontrados pelo Algoritmo, são chamados de cláusulas duais mínimas. O conjunto de todas as cláusulas duais mínimas, define a representação em DNF da teoria. Para a teoria κ, por exemplo, as cláusulas duais mínimas são:

$$
\begin{array}{lll}
0 & : & [\neg P_4, P_5, \neg P_1, \neg P_2] \\
1 & : & [\neg P_1, P_5, \neg P_3, \neg P_2] \\
2 & : & [P_4, \neg P_3, \neg P_2] \\
3 & : & [P_2, P_3, P_5, \neg P_1] \\
4 & : & [\neg P_4, P_3, P_5, \neg P_1] \\
5 & : & [P_4, \neg P_5, \neg P_1, \neg P_3] \\
6 & : & [\neg P_5, P_2, \neg P_1, \neg P_3]
\end{array}
$$

Nota-se que cada cláusula dual mínima é também um modelo para a representação em CNF e uma cláusula da representação DNF da teoria.

Essa abordagem é bastante diferente da abordagem proposta pelo algoritmo DPLL, que é notoriamente o mais famoso resolvedor SAT e que deu origem a maioria dos resolvedores SAT completos modernos. O DPLL explora as fórmulas, puramente sobre um prisma semântico, como descrito no apêndice B.

5.1 CLÁUSULA DUAL MÍNIMA

Uma cláusula dual mínima é uma cláusula que faz parte da representação DNF da teoria, denominada W_c, e está associada a um conjunto de atribuições de valores verdade que satisfazem esta representação W_c (CNF da teoria). Um conjunto de literais construído pelo Algoritmo de Transformação Dual, representa uma cláusula dual mínima quando:

1. Contém pelo menos um literal que pertence a cada uma das cláusulas de Wc.

2. Não contém literais contraditórios (ϕ e $\neg\phi$ ao mesmo tempo)

3. Permite que cada literal represente sozinho pelo menos uma cláusula de Wc.

5.2 NOTAÇÃO QUANTUM

Para construir um caminho pela fórmula CNF de uma dada teoria, o Algoritmo de Transformação Dual precisa saber em que cláusulas de tal fórmula cada literal aparece. Para tanto, o mesmo introduz uma notação especial, chamada notação Quantum. Nesta notação, encontra-se a definição de Quantum (BITTENCOURT; MARCHI; PADILHA, 2003).

Um quantum é um par (ϕ, F) onde ϕ é um literal, pertencente ao conjunto de literais que aparecem em Wc, e F é um conjunto de coordenadas para as cláusulas de Wc onde ϕ aparece (BITTENCOURT; MARCHI; PADILHA, 2003). Um quantum pode também ser representado como ϕ^F. O coletivo de quantum (um conjunto deles), chama-se *quanta*.

O *mirror* de um quantum (ϕ, F) é o quantum associado ao literal $\neg\phi$ e pode ser denotado $\overline{\phi^F}$.

Os quanta associados a teoria κ são: $\{P_1^{\{2\}}, \neg P_1^{\{4,5,6,8,9\}}, P_2^{\{0,5,7\}}, \neg P_2^{\{1,3,4,6\}}, P_3^{\{3,8\}}, \neg P_3^{\{0,1,2,4,5,9\}}, P_4^{\{7,8\}}, \neg P_4^{\{0,9\}}, P_5^{\{1,2,6,7\}}, \neg P_5^{\{3\}}\}$

5.3 BUSCA

Conhecida a notação Quantum, encontrar as cláusulas duais mínimas para uma dada teoria κ a partir de sua representação CNF pode ser encarado como um problema de busca em um epaço de estados, em que cado estado corresponde a um caminho para uma possível cláusula dual mínima.

Cada estado de busca Φ, de fato, representa um conjunto incompleto de quanta associados a uma cláusula dual mínima incompleta. Além disso cada estado tem a ele associado um conjunto de quanta proibidos X_Φ e um Gap G_Φ, definido como o conjunto de cláusulas da teoria em CNF nas quais nenhum dos literais associados aos quanta de Φ, aparece. O conjunto de literais associados aos quanta de um estado Φ é denotado L_Φ,

$$\Phi = \{\phi_1^{F_1}, ..., \phi_k^{F_k}\} \qquad L_\Phi = \{\phi_1, ..., \phi_k\}$$

Para resolver a busca, o algoritmo A* é usado (ver seção 7.2). Este algoritmo precisa de três funções de interface para acessar o problema: Uma para definir os estados iniciais da busca, uma para saber quem são os estados vizinhos de um dado estado e outra para saber quais estados são finais (RUSSELL; NORVIG, 2010).

5.3.1 Estados iniciais

A escolha dos estados iniciais é uma decisão heurística. Contudo, para teorias aleatórias escolher os literais que são mais frequentes nas cláusulas da representação CNF da teoria parece ser uma boa escolha(BITTENCOURT; MARCHI; PADILHA, 2003). A implementação deste trabalho considerou como estados iniciais, estados gerados a partir dos literais de uma dada Cláusula C. Esta cláusula C é a cláusula cuja união das coordenadas dos quanta associadas aos seus literais, cobre o maior número de cláusulas na teoria. A ideia por trás desta decisão é a de escolher como estados iniciais, estados cujos quanta cubram um grande número de cláusulas, pois isto reduz o número de cláusulas não cobertas e consequentemente o espaço de busca.

5.3.2 Geração de estados vizinhos

Saber quais estados são vizinhos a um dado estado, implica saber quais quanta podem estender um dado estado e reduzir o seu Gap, ainda respeitando as três condições básicas para que um estado possa representar uma cláusula dual mínima (apresentadas na seção 5.1). Na realidade tais estados vizinhos tem de ser gerados em tempo de execução e é na geração deles de maneira eficiente que residem os esforços deste algoritmo.

Para gerar os estados vizinhos, porém, é necessário que se conheça alguns novos conceitos antes de o algoritmo em si poder ser apresentado.

5.3.2.1 Critério de qualidade $\succ$

Um dado estado de busca Φ poderá ser estendido por muitos quanta diferentes. Para manter a busca disjunta, é necessário que se ordenem os quanta que poderiam estender Φ segundo algum critério de qualidade e que para cada novo estado gerado a partir das extensões, os quanta já usados para estender Φ sejam proibidos.Esta estratégia evita a geração de estados duplicados (BITTENCOURT; MARCHI; PADILHA, 2003).

Exemplo: Dado um estado de busca Φ qualquer e uma lista de quanta que podem estendê-lo $\{\phi_1^{F_1}, \phi_2^{F_2}, \phi_3^{F_3}\}$ já ordenados segundo algum critério de qualidade, o estado originado a partir de $\Phi \cup \phi_1^{F_1}$ pode ser estendido por $\phi_2^{F_2}$ ou $\phi_3^{F_3}$ ou ambos. Já o estado originado a partir de $\Phi \cup \phi_2^{F_2}$ pode ser estendido apenas por $\phi_3^{F_3}$, e o estado $\Phi \cup \phi_3^{F_3}$ não pode ser estendido por $\phi_1^{F_1}$ ou $\phi_2^{F_2}$

É possível encontrar critérios de qualidade para ordernar um conjunto S_Φ de quanta que podem estender Φ de maneira a evitar a geração de estados que seriam, eles ou seus sucessores, cortados da busca futuramente (BITTENCOURT; MARCHI; PADILHA, 2003). As informações necessárias para construir tal ordem seriam: O Gap de Φ, G_Φ, as coordenadas dos quanta de S_Φ e as coordenadas dos *mirror* dos quanta de S_Φ. Sejam $F_i^G = F_i \cap G_\Phi$ e $\overline{F}_i^G = \overline{F}_i \cap G_\Phi$ a intersecção das coordenadas dos quanta com o Gap de S_Φ, e seja $F_{ij} = F_i^G \cap F_j^G$, o critério de qualidade que constrói tal ordem, pode ser definido pelas regras (BITTENCOURT; MARCHI; PADILHA, 2003):

- *se* $|F_i^G - F_{ij}| > |F_j^G - F_{ij}|$ *entao* $\phi_i \succ \phi_j$ *senao* $\phi_j \succ \phi_i$
- *se* $|F_i^G - F_{ij}| = |F_j^G - F_{ij}|$ *entao, se* $|\overline{F}_i^G - \overline{F}_{ij}| > |\overline{F}_j^G - \overline{F}_{ij}|$ *entao* $\phi_i \succ \phi_j$ *senao* $\phi_j \succ \phi_i$

A ideia por trás destas regras do critério de qualidade é que os literais que cobrirem sozinhos mais cláusulas em Wc, devem ser tentados primeiro e, no caso de haver dois que cobrem o mesmo número de cláusulas, aquele cujo *mirror* cobrir sozinho mais cláusulas deve ser tentado primeiro.

5.3.2.2 Gap Conditions

Incluir um quantum ϕ^F a um estado de busca Φ e continuar respeitando a segunda condição para uma cláusula dual mínima (não incluir $\neg\phi^F$) nos traz algumas restrições quanto as cláusulas em G_Φ.

- Se existe uma cláusula $C \in G_\Phi$ tal que $C \subseteq \overline{L_\Phi}$, onde $\overline{L_\Phi}$ é o conjunto de *mirror* quantum dos quanta do estado de busca Φ, então Φ contradiz uma das cláusulas de G_Φ e não pode representar uma cláusula dual mínima.

- Se existe uma cláusula $C \in G_\Phi$ tal que $|C - \overline{L_\Phi}| = 1$, ou seja, L_Φ contradiz todos os literais em C exceto um, então este literal tem necessariamente que estar presente em L_Φ e deve, portanto, estar presente entre os seus possíveis sucessores. Caso o literal não esteja entre os possíveis sucessores de Φ, este estado não poderá representar uma cláusula dual mínima.

- Analogamente, se existe uma cláusula $C \in G_\Phi$ tal que $|C| > |C - \overline{L_\Phi}| > 1$, pelo menos um dos literais restantes de $C - \overline{L_\Phi}$ deve estar presente entre os possíveis sucessores de Φ, pelo mesmo motivo exposto anteriormente.

Estas restrições são conhecidas como *Gap conditions* e podem ser descritas como uma teoria em CNF (BITTENCOURT; MARCHI; PADILHA, 2003).

$$R_\Phi = \{C - \overline{L_\Phi} \mid C \in G_\Phi \; e \; C \cap \overline{L_\Phi} \neq \emptyset\}$$

Se a primeira restrição for verificada, então uma cláusula vazia fará parte de R_Φ. Se a segunda ou a terceira restrições forem verificadas, então R_Φ conterá pelo menos um par de cláusulas unitárias contraditórias.

5.3.2.3 Coordenadas exclusivas

São ditas coordenadas exclusivas de um determinado quantum as que apenas ele dentre os quanta do estado de busca atual possui (BITTENCOURT; MARCHI; PADILHA, 2003). A terceira condição para que uma cláusula seja uma cláusula dual mínima diz que os quanta associados aos literais da cláusula devem representar sozinhos (estar presente sem que nenhum outro da cláusula esteja) pelo menos uma cláusula de Wc. São justamente estas coordenadas para as cláusulas na qual cada quantum aparece sozinho em relação aos demais quanta do estado de busca, que formam as coordenadas exclusivas daquele quantum. As coordenadas exclusivas de um quantum ϕ^F denominam-se F^*_ϕ

$$\forall i \in \{1,...,k\}, F_i^* = F_i - \cup_{j=1, i \neq j}^{k} F_j \neq \emptyset$$

5.3.2.4 Propagação de falha

Uma maneira eficiente de propagar falhas e diminuir o tamanho do espaço de busca é evitar que os quanta que sejam recusados como extensão de um dado estado Φ, sejam testados como possíveis extensões dos estados gerados a partir de Φ (BITTENCOURT; MARCHI; PADILHA, 2003). Seja Φ um estado de busca qualquer e S_Φ o conjunto de quanta que podem estender Φ. Seja S'_Φ o conjunto de todos os quanta que adicionados a Φ gerão novos estados, então os quanta em $S_\Phi - S'_\Phi$ foram recusados para extensão de Φ e também serão recusados para a extensão de todos os estados gerados a partir de Φ. Assim, podemos adicionar estes quanta que não estendem Φ a lista de proibidos de todos os sucessores de Φ.

5.3.2.5 Algoritmo

Uma vez conhecidos os conceitos básicos para a geração de estados vizinhos, o algoritmo de geração de tais estados, em si, apresenta-se em (BITTENCOURT; MARCHI; PADILHA, 2003), como mostrado na figura 3. A notação utilizada pelo mesmo (significado dos símbolos), é apresentada na tabela 2.

Símbolo	Significado
Φ	Estado para o qual gerar-se-ão os sucessores.
$\emptyset$	Conjunto vazio.
Ω	Lista de estados sucessores de Φ.
Θ	Lista de Quanta que podem estender Φ.
C	Uma cláusula da representação em CNF (W_c) da teoria.
$\succ$	Heurística de ordenamento de Θ (Ver 5.3.2.1)
G_Φ	Gap de Φ (Ver 5.3).
X_Φ	Lista de Quanta proíbidos de estender o estado Φ (Ver 5.3).
R_Φ	Gap conditions do estado Φ (Ver 5.3.2.2).
ϕ^F	Quantum (Ver 5.2)
Φ^+	Possível estado sucessor de Φ.
	É o estado Φ com a adição de um dos quanta de Θ.
R_{Φ^+}	Gap conditions de um possível estado sucessor de Φ (Ver 5.3.2.2).
F_i^*	Coordenadas exclusivas de um Quantum (Ver 5.3.2.3).

Tabela 2: Tabela verdade para os conectivos da lógica proposicional

Sucessores(Φ)

1. Inicializar a lista de sucessores: $\Omega \leftarrow \emptyset$
2. Determinar o conjunto de possíveis estensões: $\Theta \leftarrow \{\phi^F \mid \phi \in C \; e \; C \in G_\Phi\} - X_\Phi$
3. Ordenar Θ de acordo com o critério de qualidade $\succ$
4. Verificar se Θ pode cobrir todas as cláusulas de G_Φ: Se $\exists C \in R_\Phi, \Theta \cap C = \emptyset$, então retorna $\emptyset$
5. $\forall \phi^F \in \Theta$
   $\Phi^+ \leftarrow \Phi \cup \{\phi^F\}$
   se $\forall \phi_i^{F_i} \in \Phi, F_i^* \not\subset F$
   e $\emptyset \notin R_\Phi$
   e $\forall C \in R_{\Phi^+}, C \not\subset X_\Phi$
   entao $\Omega \leftarrow \Omega \cup \{\Phi^+\}$
6. Retorna Ω

Figura 3: Algoritmo de geração de estados vizinhos

É preciso notar, porém, que como citado pelos próprios autores em nota de rodapé da página de apresentação do algoritmo, este não inclui um passo importante da geração de um novo estado de busca, que é a geração dos quanta proibidos de estendê-lo X_Φ, que é necessária para :

- Evitar a inclusão dos quanta associados a literais contraditórios, a fim de respeitar a primeira condição de uma cláusula dual (Ver seção 5.1).
- Manter a busca disjunta conforme o critério de qualidade $\succ$ (Ver seção 5.3.2.1)
- Evitar que os quanta que não conseguiram estender estados que deram origem ao estado atual, sejam testados (Ver seção 5.3.2.4)

A versão deste algoritmo usada como base para a implementação do Algoritmo de transformação dual sofreu, portanto, os ajustes necessários para que incorporasse este passo, apresentando-se em sua forma final como observado no quadro seguinte.

Sucessores(Φ)

1. Inicializar a lista de sucessores: $\Omega \leftarrow \emptyset$
2. Determinar o conjunto de possíveis estensões: $\Theta \leftarrow \{\phi^F \mid \phi \in C \, e \, C \in G_\Phi\} - X_\Phi$
3. Ordenar Θ de acordo com o critério de qualidade $\succ$
4. Verificar se Θ pode cobrir todas as cláusulas de G_Φ: Se $\exists C \in R_\Phi, \Theta \cap C = \emptyset$, então retorna $\emptyset$
5. Inicializar lista de quanta usados para estender Φ : $\alpha \leftarrow \emptyset$
6. Inicializar a lista de quanta recusados como sucessores de Φ: $\beta \leftarrow \emptyset$
7. $\forall \phi^F \in \Theta$
   $\Phi^+ \leftarrow \Phi \cup \{\phi^F\}$
   $X_{\Phi^+} \leftarrow X_\Phi \cup \overline{\phi^F}$
   se $\forall \phi_i^{F_i} \in \Phi, F_i^* \not\subset F$
   e $\emptyset \notin R_\Phi$
   e $\forall C \in R_{\Phi^+}, C \not\subset X_\Phi$
   entao $\forall \phi_i^{F_i} \in \alpha, X_{\Phi^+} \leftarrow X_{\Phi^+} \cup \phi_i^{F_i}$
   $\alpha \leftarrow \alpha \cup \{\phi^F\}$
   $\Omega \leftarrow \Omega \cup \{\Phi^+\}$
   senao $\beta \leftarrow \beta \cup \{\phi^F\}$
8. $\forall \Phi^+ \in \Omega, X_{\Phi^+} \leftarrow X_{\Phi^+} \cup \beta$
9. Retorna Ω

5.3.3 Estados finais

Um estado Φ é um estado final se as coordenadas dos literais de Φ cobrem o conjunto Wc. Em outras palavras um estado Φ é um estado final se $G_\Phi = \emptyset$.

É interessante observar que o Algoritmo de Transformação Dual pode ser usado para calcular a satisfazibilidade de uma dada teoria, e não o conjunto de cláusulas duas mínimas completo, se o mesmo interromper a sua execução ao encontrar o primeiro estado final.

6 MELHORAMENTOS AO ALG. DE TRANS. DUAL

Um olhar atento sobre as condições de corte nos caminhos de busca do Algoritmo de Transformação Dual, permite identificar algumas possibilidades de melhoria, principalmente ligadas às restrições impostas sobre quais quanta podem ser incluídos em um dado estado de busca para gerar um novo, em função dos já existentes e da lista de quanta proibidos X_Φ. Tais melhorias, por diminuirem o tamanho do espaço de estados, tem impacto tanto sobre a quantidade de memória necessária para armazenar a busca quanto no número de ciclos de processamento necessários para concluir a busca. Três destas possíveis melhorias foram implementadas, testadas e demonstraram-se efetivas no escopo deste trabalho. Algumas outras aparecem na conclusão deste trabalho, como sugestão para trabalhos futuros. As descrições das três melhorias implementadas encontram-se a seguir, enquanto os resultados dos testes estão no capítulo 8 - Testes e análise dos resultados.

6.1 CORTE PREMATURO

Durante a geração de estados sucessores, os estados gerados a partir de Φ tem uma série de quanta adicionados à sua lista de quanta proibidos por conta das restrições descritas em 5.1, 5.3.2.1 e 5.3.2.4. Após a adição destes quanta, a lista de quanta proibidos X_{Φ^+} possui valiosas informações que podem dizer, de antemão, se este estado tem ou não chances de gerar estados sucessores. Sabendo que um dado estado Φ^+ não gerará sucessores, podemos removê-lo de Ω e cortar prematuramente, caminhos infrutíferos.

Para tanto, precisamos calcular as *Gap conditions* de Φ^+. Estas *Gap conditions* dizem com que quanta as cláusulas do gap podem ser cobertas sem quebrar as restrições. Se a teoria CNF que as representa contiver a cláusula vazia, ou duas cláusulas unitárias contraditórias, sabemos que este estado Φ^+ é infrutífero e podemos descartá-lo.

Assim, o algoritmo de cálculo de estados vizinhos deve incluir um novo passo (9).

Sucessores(Φ)

1. Inicializar a lista de sucessores: $\Omega \leftarrow \emptyset$
2. Determinar o conjunto de possíveis estensões: $\Theta \leftarrow \{\phi^F \mid \phi \in C \, e \, C \in G_\Phi\} - X_\Phi$
3. Ordenar Θ de acordo com o critério de qualidade $\succ$
4. Verificar se Θ pode cobrir todas as cláusulas de G_Φ: Se $\exists C \in R_\Phi, \Theta \cap C = \emptyset$, então retorna $\emptyset$
5. Inicializar lista de quanta usados para estender Φ : $\alpha \leftarrow \emptyset$
6. Inicializar a lista de quanta recusados como sucessores de Φ: $\beta \leftarrow \emptyset$
7. $\forall \phi^F \in \Theta$
   $\Phi^+ \leftarrow \Phi \cup \{\phi^F\}$
   $X_{\Phi^+} \leftarrow X_\Phi \cup \overline{\phi^F}$
   se $\forall \phi_i^{F_i} \in \Phi, F_i^* \not\subset F$
   e $\emptyset \notin R_\Phi$
   e $\forall C \in R_{\Phi^+}, C \not\subset X_\Phi$
   entao $\forall \phi_i^{F_i} \in \alpha, X_{\Phi^+} \leftarrow X_{\Phi^+} \cup \phi_i^{F_i}$
   $\alpha \leftarrow \alpha \cup \{\phi^F\}$
   $\Omega \leftarrow \Omega \cup \{\Phi^+\}$
   senao $\beta \leftarrow \beta \cup \{\phi^F\}$
8. $\forall \Phi^+ \in \Omega, X_{\Phi^+} \leftarrow \beta$
9. $\forall C \in R_{\Phi^+}$ *de todo* $\Phi^+ \in \Omega$, *se* $C = \emptyset$
   ou $|C| = 1$ *e* $\exists D \in R_{\Phi^+}$ *e* $D \equiv \neg C$
   entao $\Omega \leftarrow \Omega - \{\Phi^+\}$
10. Retorna Ω

6.2 EXPANSÃO DE *GAP CONDITIONS*

A ideia geral por trás do cálculo de *Gap conditions* é expressar, na forma de uma teoria em CNF, através de quais quanta cada cláusula do gap de um determinado estado Φ pode ser "coberta", ou seja, quais quanta adicionados a Φ farão cada cláusula sair de G_Φ. Sabemos que esta teoria é gerada segundo a equação:

$$R_\Phi = \{C - \overline{L_\Phi} \mid C \in G_\Phi \, e \, C \cap \overline{L_\Phi} \neq \emptyset\}$$

A ideia da construção desta teoria é a de que se uma determinada cláusula de G_Φ tem intersecção com $\overline{L_\Phi}$, nós devemos remover da mesma os literais da intersecção pois estes por pertencerem a $\overline{L_\Phi}$ não podem estar presentes em Φ e portanto não será através deles que esta cláusula será "coberta". Sobram portanto, nas cláusuas de G_Φ aqueles literais que podem estar em Φ.

A melhoria proposta neste cálculo é simples e toma vantagem dos próprios mecanismos introduzidos pela representação Quantum. A ideia é remover das cláusulas de G_Φ não apenas os literais de $\overline{L_\Phi}$, mas todos aqueles pertencentes a X_Φ, uma vez $\overline{L_\Phi} \subseteq X_\Phi$ e X_Φ ainda contém quanta proibidos por outras razões.

$$R_\Phi = \{C - X_\Phi \mid C \in G_\Phi \; e \; C \cap X_\Phi \neq \emptyset\}$$

Esta melhoria certamente reduz o espaço de busca e torna a melhoria anteriormente proposta ainda mais efetiva.

6.3 *GAP CONDITIONS* SEM INTERESECÇÃO

É fácil perceber, olhando para a equação que demonstra a geração das *Gap Conditions* na seção 6.2 que se uma determinada cláusula $C \in G_\Phi$ não possui intersecção com X_Φ ela não aparecerá em R_Φ. Se ela aparecesse, no entanto, todos os seus literais apareceriam pois justamente por não ter intersecção com X_Φ ela não se alteraria. Consideremos:

- Uma cláusula que está em G_Φ não pode ser uma cláusula vazia.
- Se existirem cláusulas unitárias contraditórias em G_Φ elas existem na representação CNF (W_c) da teoria, fazendo a mesma insatisfazível.

Tendo isso em mente, infere-se que a inclusão das cláusulas de G_Φ em R_Φ que não tem intersecção com X_Φ não altera o seu valor semântico, pois não inclui a cláusula vazia e se incluir alguma contradição, estará apenas cortando a busca para teorias inerentemente insatisfazíveis. Dessa maneira, podemos reescrever a equação de geração de R_Φ da seguinte maneira:

$$R_\Phi = \{C - X_\Phi \mid C \in G_\Phi \}$$

O grande benefício introduzido por tal melhoria é a eliminação da necessidade de se calcular a intersecção $C \cap X_\Phi$, pois esta é uma das operações mais custosas no cálculo das *Gap Conditions*, que é fundamental para a

eliminação de caminhos infrutíferos da busca.

7 IMPLEMENTAÇÃO DO ALG. DE TRANS. DUAL

Para implementar o Algoritmo de Transformação Dual, a linguagem Java foi escolhida. Inicialmente, a não necessidade do gerenciamento de memória e as ferramentas para análise de execução disponíveis para a máquina virtual de Java foram os principais fatores para tal escolha, pois ambos contribuem para que o foco no momento da implementação esteja exclusivamente na lógica algorítmica. O Algoritmo de Transformação Dual introduz vários elementos de representação e cálculos bastante capciosos e quanto mais foco se tiver na lógica dos mesmos na hora da implementação, menores as chances de erro e maior a possibilidade de, em compreendendo seu funcionamento e seus objetivos por completo, propor-se a ele melhorias.

A implementação foi feita seguindo-se o paradigma de orientação a objetos (OO). Mais uma vez, o intuito de tal escolha foi o de manter o foco no algoritmo, simplificando a representação de conceitos do domínio do problema. Contudo, a busca seguindo o algoritmo A* e as funções que este necessita para acessar o problema, que são o foco do Algoritmo de Transformação Dual, estão todos presentes em uma mesma classe, chamada de DualSolver. O código fonte da implementação desta classe encontra-se no apêndice C. A estrutura interna desta classe lembra, por vezes, um programa que segue o paradigma procedimental. Além das classes que modelam o domínio do problema e da classe DualSolver, a implementação conta ainda com uma classe separada para a leitura e carregamento dos arquivos contendo as teorias, chamada DimacsParser, com uma classe para análise da memória alocada (necessária para se saber o máximo de memória alocada durante uma busca), chamada *MemoryAnalyzer*, e com uma classe auxiliar para operações com bits e bytes, necessária por conta das estruturas de dados que representam alguns conceitos do domínio do problema. Estas classes estão representadas na figura 4.

As estruturas de dados que representam os conceitos do domínio do problema estão assim implementadas:

Cláusula Uma cláusula é representada por um objeto da classe *Clause*, que possui como único atributo uma lista de inteiros, onde cada inteiro representa um literal. Esta representação é usual, pois a representação de cláusulas do formato DIMACS (ver seção 7.1), formato das teorias de teste utilizadas, representa cada variável da teoria através de um número natural. Quando a forma não negada de uma variável aparece na cláusula, o próprio número natural que a representa lá aparecerá. Já quando a forma negada de uma variável aparece, o seu inverso aditivo aparecerá na cláusula.

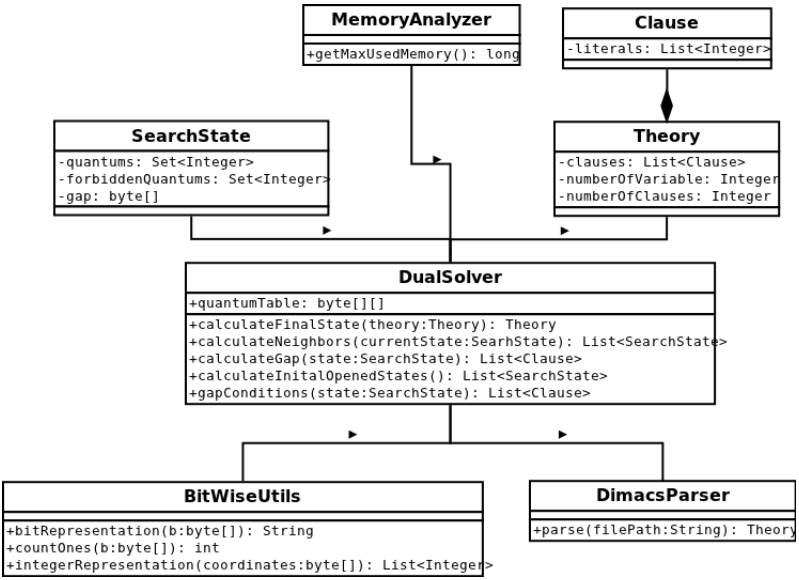

Figura 4: Diagrama de classes simplificado da implementação do Algoritmo de Transformação Dual.

Teoria Uma teoria é um objeto da classe *Theory* e possui três atributos, *clauses*, *numberOfVariables* e *numberOfClauses*. O primeiro é uma lista de cláusulas. O segundo é um inteiro e representa o número de variáveis da teoria. O terceiro também é um inteiro e representa o número de cláusulas na teoria.

Tabela de Quanta Os quanta, parte central da representação Quantum, são representados por uma estrutura chamada de *quantumTable* que é um atributo da classe *DualSolver*. Este atributo é um vetor bidimensional de bytes, onde cada entrada da primeira dimensão do vetor representa um Quantum diferente. O número de entradas na segunda dimensão, ou seja, o número de bytes necessários para representar se cada cláusula da teoria CNF contém ou não o literal deste quantum depende do número de cláusulas na teoria. Cada *bit* de cada *byte* deste vetor representa se um dado literal aparece em uma determinada cláusula ou não, formando assim as coordenadas do quantum. Os indíces das cláusulas na teoria CNF aparecem ordenados do bit menos significativo ao mais significativo em cada byte, e do byte de menor indíce no vetor para o de

maior indíce.

Exemplo: Para exemplificar o funcionamento da tabela de quanta, vamos considerar a mesma teoria usada como exemplo no capítulo 5, cuja representação CNF pode ser:

0	:	[-4, 2, -3]	5	:	[-3, 2, -1]
1	:	[5, -2, -3]	6	:	[-2, -1, 5]
2	:	[5, -3, 1]	7	:	[4, 5, 2]
3	:	[-5, -2, 3]	8	:	[-1, 4, 3]
4	:	[-2, -3, -1]	9	:	[-3, -1, -4]

Tabela 3: Teoria de exemplo

Para esta teoria, temos o conjunto Φ de quanta associados a seus literais definidos por: $\Phi = \{1^{\{2\}}, -1^{\{4,5,6,8,9\}}, 2^{\{0,5,7\}}, -2^{\{1,3,4,6\}}, 3^{\{3,8\}}, -3^{\{0,1,2,4,5,9\}}, 4^{\{7,8\}}, -4^{\{0,9\}}, 5^{\{1,2,6,7\}}, -5^{\{3\}}\}$

E teríamos, neste caso, uma *quantumTable* como a que segue:

Literais	Indíces vetor	byte 2 1	byte 1 0
-5	0	[00000000]	[00001000]
-4	1	[00000010]	[00000001]
-3	2	[00000010]	[00110111]
-2	3	[00000000]	[01011010]
-1	4	[00000011]	[01110000]
1	5	[00000000]	[00000100]
2	6	[00000000]	[10100001]
3	7	[00000001]	[00001000]
4	8	[00000001]	[10000000]
5	9	[00000000]	[11000110]

O literal -5 aparece apenas na cláusula de indíce 3 da representação CNF. Por isso, apenas o bit que representa tal cláusula (o quarto da direita para a esquerda, pois a contagem de cláusulas inicia com a cláusula 0) tem o valor "1" no vetor de bytes que representa as coordenadas de tal literal.

Estado de busca Um estado de busca é representado por um objeto da classe *SearchState* e possui três atributos: um conjunto de inteiros, chamado

quanta que representa os quanta atualmente neste estado; um segundo conjunto de inteiros, chamado *forbiddenQuanta* que representa os quanta que não podem ser adicionados ao estado atual (X_Φ); e um vetor de *bytes* chamado gap, que representa, de forma análoga a representação dos quanta na Quantum Table, quais cláusulas de W_c estão no Gap do estado de busca (estão em G_Φ).

7.1 FORMATO DOS ARQUIVOS DE ENTRADA

O formato DIMACS é o formato padrão dos arquivos dos conjuntos de teste da SATLib (SATLIB, 1999). Os conjuntos de testes utilizados para testarem a implementação são conjuntos da SATLib, portanto o formato padrão de entradas de dados não poderia ser outro senão o formato DIMACS. Tal formato foi descrito em Dimacs-Challenge (1993).

Neste formato, as cláusulas da teoria estão dispostas uma em cada linha. Nestas linhas, números indicam a presença de determinados literais. A ideia básica é que cada variável da teoria esteja associada a um número natural. Quando a forma não negada desta variável aparece em uma cláusula, o próprio número natural aparecerá na linha que descreve esta cláusula. Já quando a forma negada desta variável aparecer em uma cláusula, o inverso aditivo do número que representa a variável aparecerá na linha que descreve a cláusula. Os números, representando literais, são separados por espaços. Toda cláusula termina com o número zero, que não tem valor semântico, e uma quebra de linha. O arquivo de entrada pode conter ainda linhas com comentários, que iniciam com o caracter “c” e não tem valor semântico, e devem conter uma linha de preâmbulo, que inicia sempre com o caracter “p” e segue o formato

p FORMATO VARIÁVEIS CLÁUSULAS

onde “FORMATO” é o formato em que se descreve a teoria (CNF, DNF), “VARIÁVEIS” representa o número de variáveis na fórmula, e “CLÁUSULAS” o número de cláusulas da mesma. Como exemplo, a representação em um arquivo no formato DIMACS da teoria apresentada na tabela 3 seria:

```
c Teoria de exemplo
p CNF 5 10
-4, 2, -3 0
5, -2, -3 0
5, -3, 1 0
-5, -2, 3 0
```

-2, -3, -1 0
-3, 2, -1 0
-2, -1, 5 0
4, 5, 2 0
-1, 4, 3 0
-3, -1, -4 0

7.2 IMPLEMENTAÇÃO DA BUSCA

Uma das formas mais conhecidas de busca com informação eficiente é a chamada busca A* (RUSSELL; NORVIG, 2010). Nesta busca, cada estado é avaliado combinando-se o custo para alcançar este estado, a partir do estado inicial $g(n)$, e o custo para ir deste estado até um estado final $h(n)$. Assim, tem-se uma estimativa do custo da solução de menor custo que passe por um determinado estado, $f(n)$. É a partir desta estimativa que o algorimo A* escolhe quais devem ser os próximos estados a serem estendidos.

$$f(n) = g(n) + h(n)$$

Na busca A* realizada pelo Algoritmo de Transformação Dual, a função $g(n)$ é mapeada para o número de quanta já adicionados ao estado atual, em comparação com o estado inicial da busca, enquanto $h(n)$ é mapeada para uma função que determina o número de cláusulas no Gap do estado. Assim, o próximo caminho a ser tomado será sempre o que tiver a menor soma do resultado das duas funções, ou seja, o menor resultado para $f(n)$. Estas três definições encontram-se na figura 5.

$g(\Phi)$ = número de quanta já adicionados ao estado Φ em comparação com o estado inicial
$h(\Phi)$ = número de cláusulad no Gap de Φ.
$f(\Phi) = g(\Phi) + h(\Phi)$

Figura 5: Definições da função de custo

É interessante notar que a função $h(\Phi)$ é uma função heurística, uma vez que o número de cláusulas no gap de um estado não tem ligação com o número de quanta que precisam ser adicionados ao mesmo para que o seu gap seja vazio.

A busca A* sempre inicia com alguns estados iniciais, em um conjunto de estados denominados estados abertos. Cria-se ainda um conjunto de estados fechados, que são aqueles estados por onde a busca já passou. No caso

da busca do Algoritmo de Transformação Dual, existe ainda um conjunto de estados para armazenar os estados finais, isto é, aqueles que representam uma cláusula dual mínima. Enquanto existirem estados abertos a busca prossegue, selecionando um estado, gerando seus vizinhos e adicionando-os ao conjunto de abertos.

Um algoritmo que descreve a busca por estados finais é o seguinte:

```
1: Algoritmo A*
2: abertos ← gerar estados iniciais (Ver seção 5.3.1)
3: fechados ← ∅
4: finais ← ∅
5: while abertos ≠ ∅ do
6:    Φ ← estado com menor resultado para f(Φ)
7:    if G_Φ = ∅ then
8:       finais ← finais ∪ {Φ}
9:       abertos = abertos − {Φ}
10:   else
11:      fechados ← fechados ∪ {Φ}
12:      abertos = abertos − {Φ}
13:      vizinhos = gerar vizinhos de Φ (Ver seção 5.3.2)
14:      for vizinho in vizinhos do
15:         abertos ← abertos ∪ vizinho
16:      end for
17:   end if
18: end while
```

Depois de executado o algoritmo acima, os estados que estiverem no conjunto denominado "finais", representam as cláusulas duais mínimas geradas e são retornados pela implementação.

7.3 FORMATO DA SAÍDA

A saída da implementação feita do Algoritmo de Transformação Dual, foi pensada de modo a facilitar os testes e as comparações entre diferentes versões da implementação que seriam necessárias a este trabalho. Deste modo, a saída é composta por duas linhas.

A primeira é um resumo do desempenho do algoritmo para a entrada fornecida. Ela contém o nome do arquivo da teoria (nome), o tempo total (em

centésimos de segundos) gasto para se encontrar a primeira (TP) e todas (TT) cláusula duais, a contagem de iterações sobre o *loop* principal da busca A* para se encontrar a primeira (IP) e todas (IT) as cláusulas duais, o máximo de memória alocada (Max. mem.), em Megabytes, durante a busca e o número de cláusulas duais geradas ($|W_d|$). Os valores da primeira linha são separados por "|" e estão dispostos na seguinte ordem:

$$\text{nome} \mid \text{TP} \mid \text{IP} \mid \text{TT} \mid \text{IT} \mid \text{Max. Mem.} \mid |W_d|$$

Nesta linha, todas as informações básicas para suportar as análises e conclusões do capítulo 8 estão presentes.

A segunda linha contém uma representação das cláusulas duais mínimas geradas. Elas estão representadas na forma de uma lista, em que cada cláusula contém os literais que nela aparece. Uma entrada que gerasse duas cláusulas duais mínimas, poderia ter uma segunda linha de saída como:

$$[\{1,3,-5,8,-10\},\{1,-36,7,9\}]$$

8 TESTES E ANÁLISES DOS RESULTADOS

Para tomar ciência da dimensão do melhoramento ocasionado pelas melhorias propostas, o Algoritmo de transformação Dual foi testado em versões com e sem as mesmas. Neste capítulo apresentam-se apenas os resultados dos testes para a versão sem melhorias e a versão com todas as três melhorias. Detalhes sobre os testes de versões parciais podem ser encontrados no apêndice A.

Um conjunto de teorias de teste padrão conhecido como SATLIB, disponível em *http://www.cs.ubc.ca/ hoos/SATLIB/benchm.html*, foi usado para os testes. Os pacotes de teorias selecionados contém instâncias SAT em CNF com 3 literais randômicos por cláusula, e são eles:

- *uf20-91*: teorias com 20 variáveis e 91 cláusulas, todas satisfazíveis.
- *uf50-218*: teorias com 50 variáveis e 218 cláusulas, todas satisfazíveis.
- *uf75-325*: teorias com 75 variáveis e 325 cláusulas, todas satisfazíveis.
- *uf100-430*: teorias com 100 variáveis e 430 cláusulas, todas satisfazíveis.

Estes mesmos pacotes e as teorias neles contidas são usados por vários outros trabalhos relacionados a pesquisa em SAT (HOOS; STUTZLE, 2000). Alguns testes também foram realizados para instâncias não satisfazíveis, porém, apresentaremos aqui apenas os resultados para instâncias satisfazíveis.

Todos os testes foram executados em uma máquina com processador Intel Core i5-2410M com 2,3GHz e 2 GiB. Alguns dos resultados obtidos são apresentados nas tabelas 4, 5, 6 e 7 onde "Tempo total" é o tempo total em centésimos de segundo (0,01s), "Chamadas" é o número de iterações sobre o *loop* principal do algoritmo A*, "Max. mem." é o máximo de memória alocada durante a execução, em MB, e $|W_d|$ é o número de cláusulas duais mínimas geradas.

A versão sem melhorias do algoritmo excedeu o limite de memória disponível (2GiB) quando executada com os pacotes de teste "uf75-325" e "uf100-430". Por esse motivo, os resultados para estes testes estão marcados com "?". Também por isso, no apêndice A, não se apresentam os resultados parciais para estes pacotes de teste.

Os resultados obtidos para os pacotes "*uf20-91*" e "*uf50-218*" estão descritos nas tabelas 4 e 5, e indicam que as três melhorias propostas contribuem positivamente para a redução do espaço de busca e, consequentemente, da memória necessária e do tempo total de busca. Verifica-se isto através da redução média de 64% do tempo total de busca e de 89% na quantidade de

Teoria	Tempo total		Chamadas		Max. mem.		$\|W_d\|$
	Sem	Com	Sem	Com	Sem	Com	
uf20-0110	69	46	2212	204	20	9	21
uf20-0111	24	19	1583	95	39	9	5
uf20-0112	23	11	1415	62	65	9	3
uf20-0113	32	19	1893	78	77	9	1
uf20-0114	29	13	1707	93	73	8	6
uf20-0115	32	21	2107	97	94	8	4
uf20-0116	22	14	1402	60	67	7	2
uf20-0117	30	17	1916	77	88	7	1
uf20-0118	31	17	2152	188	79	6	14
uf20-0119	21	17	1378	71	67	6	2

Tabela 4: Pacote de teste uf20-91

Teoria	Tempo total		Chamadas		Max. mem.		$\|W_d\|$
	Sem	Com	Sem	Com	Sem	Com	
uf50-0110	18916	1644	137668	1311	495	38	27
uf50-0111	17108	1356	131643	592	488	33	1
uf50-0112	13587	1221	94016	564	384	26	4
uf50-0113	13983	1141	92886	630	389	34	9
uf50-0114	35614	4139	266851	5631	919	52	244
uf50-0115	14559	1345	111694	1608	574	19	40
uf50-0116	14759	1302	110290	795	531	27	15
uf50-0117	33357	3489	268012	4301	918	38	191
uf50-0118	9549	793	69720	449	413	17	7
uf50-0119	19787	2013	141276	1435	597	29	24

Tabela 5: Pacote de teste uf50-218

Teoria	Tempo total		Chamadas		Max. mem.		$\|W_d\|$
	Sem	Com	Sem	Com	Sem	Com	
uf75-010	?	84188	?	41917	?	237	1025
uf75-011	?	38921	?	6027	?	71	9
uf75-012	?	46518	?	5985	?	87	2
uf75-013	?	25052	?	6545	?	92	132
uf75-014	?	33056	?	8362	?	109	157
uf75-015	?	50720	?	7783	?	87	16
uf75-016	?	134676	?	2126	?	63	6
uf75-017	?	55576	?	15246	?	131	177
uf75-018	?	38742	?	18579	?	161	925
uf75-019	?	15098	?	2136	?	63	2

Tabela 6: Pacote de teste uf75-325

Teoria	Tempo total		Chamadas		Max. mem.		$\|W_d\|$
	Sem	Com	Sem	Com	Sem	Com	
uf100-0110	?	512070	?	30832	?	174	18
uf100-0111	?	494565	?	33024	?	167	140
uf100-0112	?	109984	?	8604	?	199	46
uf100-0113	?	747005	?	45384	?	171	3
uf100-0114	?	415859	?	23202	?	158	2
uf100-0115	?	182862	?	19507	?	161	424
uf100-0116	?	588479	?	40536	?	168	66
uf100-0117	?	501414	?	33280	?	188	47
uf100-0118	?	283759	?	28180	?	178	433
uf100-0119	?	582873	?	90136	?	512	1835

Tabela 7: Pacote de teste uf100-430

memória necessária para a resolução de cada teoria destes pacotes. Os resultados alcançados com os pacotes *uf75-325* e *uf100-430*, que são exibidos nas tabelas 6 e 7, também indicam uma melhora significativa no algoritmo. Para estes resutados, não é possível calcular diretamente a média das reduções de tempo total e de quantidade de memória necessária uma vez que, sem as melhorias propostas, o algoritmo nem mesmo conseguia resolver as teorias destes pacotes. Este resultado, então, reforça a contribuição das melhorias propostas.

A versão com melhorias do Algoritmo de Transformação Dual foi comparada, a título de curiosidade, com a versão do algoritmo DPLL descrita no apêndice B. Os resultados de tal comparação encontram-se neste mesmo apêndice. Esta comparação serviu apenas para demonstrar que o Algoritmo de trasnformação Dual é, aparentemente, mais eficiente na resolução de SAT que o DPLL, sem influenciar, porém, as conclusões obtidas com este trabalho. Tais conclusões, são baseadas apenas nas comparações descritas anteriormente neste capítulo, da versão com e sem melhorias do Algoritmo de Transformação Dual.

9 CONCLUSÃO

O Algoritmo de Transformação Dual foi apresentado no capítulo 5, enquanto as melhorias a este propostas foram apresentadas no capítulo 6 e os testes e resultados obtidos no capítulo 8.

Através da comparação das versões com e sem melhorias do Algoritmo, verificou-se que as três melhorias propostas contribuiram significativamente para a redução do tempo total de execução do Algoritmo e também para a redução da quantidade de memória utilizada. Em especial a primeira melhoria apresentada (seção 6.1) reduz significativamente o espaço de busca, resltuando em um menor número de chamadas a função de busca. Quando as três melhorias são aplicadas em conjunto, contudo, o desempenho se torna melhor. Apesar de os testes terem sido realizados com um conjunto limitado de teorias, os resultados parecem promissores.

A implementação em Java foi de grande ajuda para simplificar os esforços de implementação e garantir mais foco ao problema em si. Implementações anteriores do Algoritmo haviam sido feitas em LISP e C++. Comparações entre as versões anteriores de implementações e a versão implementada neste trabalho não foram feitas pela diferença de linguagens. Neste caso, implementar o Algoritmo de Transformação Dual com melhorias em Lisp seria uma alternativa melhor. No entanto, a implementação feita em Java sem os melhoramentos segue estritamente as definições conceituais da descrição do algoritmo e portanto, a comparação feita, entre esta versão e a com melhorias, é valida.

Existem ainda, duas possibilidades de melhorias que, embora tenham surgido dentro deste trabalho e estejam perfeitamente alinhadas com os objetivos aqui perseguidos, não foram implementadas por falta de tempo hábil. Ficam aqui então, como sugestões para trabalhos futuros:

Verificação completa da satisfazibilidade de gapConditions As melhorias apresentadas nas seções 6.2 e 6.3 envolvem ambas o cálculo de *Gap Conditions*. Tal cálculo tem como resultado uma teoria da lógica proposicional em CNF (subconjunto da teoria de entrada do algoritmo). Da maneira como o algoritmo está descrito na seção 5.3.2.5, as três condições relativas as *Gap Conditions* descritas em 5.3.2.2 são verificadas individualmente. Porém, como as *Gap Conditions* estão descritas como uma teoria, a mesma poderia ser verificada através da checagem de sua satisfazibilidade. A ideia por trás disso é que se a teoria que descreve as *Gap conditions* for satisfazível nenhumas das três condições estava presente.

Paralelização Através de alguns mecanismos descritos pelo Algoritmo de transformação Dual, mas principalmente pela critério de qualidade descrito em 5.3.2.1, verifica-se que a busca por estados vizinhos e por modelos, a cada novo passo de execução do algoritmo A* é totalmente disjunta. Isso significa que a partir da escolha dos estados iniciais do algoritmo, a busca poderia ser executada em paralelo, para todos os ramos ao mesmo tempo. Esta técnica poderia ser aplicada recursivamente dentro do espaço de busca, onde cada passo geraria uma série de estados vizinhos que poderiam ser verificados paralelamente. Esta proposta de melhoria não é teórica, apenas toma vantagem de características do Algoritmo para tentar uma implementação que resulte em um menor tempo de processamento.

REFERÊNCIAS BIBLIOGRÁFICAS

BITTENCOURT, G.; MARCHI, J.; PADILHA, R. S. A syntatical approach to satisfaction. In: UNIVERSITY OF LIVERPOOL AND UNIVERSITY OF MANCHESTER. *4th International Workshop on the Implementation of Logic (Konev, B. and Schimidt, R. eds.)*. Almaty - Kazakhstan, 2003. p. 18–32.

BOAVA, G. *Algoritmos para Satisfazibilidade: Implementação e Testes*. Dissertação (TCC) — Universidade Federal de Santa Catarina, 2005.

CRAWFORD, J. M.; AUTON, L. D. Experimental results on the crossover point in random 3sat. *Artificial Intelligence*, v. 81, p. 31–57, 1996.

DARWICHE, A.; PIPATSRISAWAT, K. Complete algorithms. In: ____. [S.l.]: IOS Press, 2009. (Frontiers in Artificial Intelligence and Applications, v. 185), cap. 3, p. 99–130. ISBN 978-1-58603-929-5.

DAVIS, M.; LOGEMANN, G.; LOVELAND, D. A machine program for theorem-proving. *Commun. ACM*, ACM, New York, NY, USA, v. 5, n. 7, p. 394–397, jul. 1962. ISSN 0001-0782. <http://doi.acm.org/10.1145/368273.368557>.

DAVIS, M.; PUTNAM, H. A computing procedure for quantification theory. *J. ACM*, ACM, New York, NY, USA, v. 7, n. 3, p. 201–215, jul. 1960. ISSN 0004-5411. <http://doi.acm.org/10.1145/321033.321034>.

DIMACS-CHALLENGE. *Satisfiability: Suggested Format*. [S.l.], Maio 1993. <http://www.cs.ubc.ca/ hoos/SATLIB/Benchmarks/SAT/satformat.ps>.

HOOS, H. H.; STUTZLE, T. Satlib: An online resource for research on sat. In: . [S.l.]: IOS Press, 2000. p. 283–292.

MARIć, F. Formalization and implementation of modern sat solvers. *J. Autom. Reason.*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 43, n. 1, p. 81–119, jun. 2009. ISSN 0168-7433. <http://dx.doi.org/10.1007/s10817-009-9127-8>.

RAUBER, P. E. *Análise da solução do problema do caminho Hamiltoniano Através de Redução para Problema da Satisfazibilidade Booleana*. Dissertação (TCC) — Universidade Federal de Santa Catarina, 2011.

ROSEN, K. H. *Handbook of Discrete and Combinatorial Mathematics, Second Edition*. 2nd. ed. [S.l.]: Chapman & Hall/CRC, 2010. ISBN 158488780X, 9781584887805.

RUSSELL, S. J.; NORVIG, P. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. [S.l.]: Pearson Education, 2010. I-XVIII, 1-1132 p. ISBN 978-0-13-207148-2.

SATLIB. *SATLIB - Submission Guidelines*. Dezembro 1999. Internet. <http://www.cs.ubc.ca/ hoos/SATLIB/submission.html>.

SIPSER, M. *Introduction to the Theory of Computation*. 1st. ed. [S.l.]: International Thomson Publishing, 1996. ISBN 053494728X.

VORONKOV, A. A. Logic and modeling 2011. Outubro 2012. <http://www.voronkov.com/lics.cgi>.

APÊNDICE A – Resultados de testes para implementações parciais das melhorias

Teoria	Tempo total	Chamadas	Max. mem.	$\|W_d\|$
uf20-0110	45	215	9	21
uf20-0111	17	112	19	5
uf20-0112	12	82	38	3
uf20-0113	16	91	36	1
uf20-0114	13	117	35	6
uf20-0115	17	124	34	4
uf20-0116	11	78	32	2
uf20-0117	15	95	31	1
uf20-0118	16	202	30	14
uf20-0119	11	83	29	2

Tabela 8: Algoritmo de Transformação Dual com a melhoria 1 - uf20-091

Teoria	Tempo total	Chamadas	Max. mem.	$\|W_d\|$
uf20-0110	61	1850	21	21
uf20-0111	26	1032	39	5
uf20-0112	14	756	36	3
uf20-0113	22	860	57	1
uf20-0114	18	1042	42	6
uf20-0115	32	1230	55	4
uf20-0116	17	918	44	2
uf20-0117	23	1074	56	1
uf20-0118	24	1492	50	14
uf20-0119	16	814	41	2

Tabela 9: Algoritmo de Transformação Dual com a melhoria 2 - uf20-091

Teoria	Tempo total	Chamadas	Max. mem.	$\|W_d\|$
uf20-0110	72	2390	21	21
uf20-0111	41	1844	39	5
uf20-0112	28	1588	56	3
uf20-0113	40	2193	77	1
uf20-0114	33	1897	65	6
uf20-0115	40	2355	82	4
uf20-0116	31	1758	64	2
uf20-0117	40	2373	82	1
uf20-0118	40	2537	76	14
uf20-0119	26	1553	57	2

Tabela 10: Algoritmo de Transformação Dual com a melhoria 3 - uf20-091

Teoria	Tempo total	Chamadas	Max. mem.	$\|W_d\|$
uf50-0110	4426	3045	35	27
uf50-0111	4522	2528	28	1
uf50-0112	3102	1719	24	4
uf50-0113	3088	1685	20	9
uf50-0114	8442	8461	49	244
uf50-0115	3515	2990	28	40
uf50-0116	3673	2292	34	15
uf50-0117	8304	7696	59	191
uf50-0118	2347	1363	28	7
uf50-0119	4634	3071	26	24

Tabela 11: Algoritmo de Transformação Dual com a melhoria 1 - uf50-218

Teoria	Tempo total	Chamadas	Max. mem.	$\|W_d\|$
uf50-0110	5867	35074	390	27
uf50-0111	4736	23375	342	1
uf50-0112	4716	23542	342	4
uf50-0113	4588	23523	342	9
uf50-0114	15262	106378	550	244
uf50-0115	4902	35860	330	40
uf50-0116	4772	26944	343	15
uf50-0117	12867	89834	519	191
uf50-0118	2824	15741	314	7
uf50-0119	7535	43177	384	24

Tabela 12: Algoritmo de Transformação Dual com a melhoria 2 - uf50-218

Teoria	Tempo total	Chamadas	Max. mem.	$\|W_d\|$
uf50-0110	12796	91897	513	27
uf50-0111	9913	69754	458	1
uf50-0112	8856	58315	438	4
uf50-0113	9794	64127	441	9
uf50-0114	23066	181522	738	244
uf50-0115	8139	64611	472	40
uf50-0116	8795	61705	430	15
uf50-0117	20922	164002	698	191
uf50-0118	4992	33149	350	7
uf50-0119	13217	89754	497	24

Tabela 13: Algoritmo de Transformação Dual com a melhoria 3 - uf50-218

APÊNDICE B – O Algoritmo DPLL

O algoritmo conhecido como DPLL é um algoritmo completo para a resolução de SAT e leva este nome por causa de seus autores - Davis, Logemann e Loveland. Foi pela primeira vez descrito em Davis, Logemann e Loveland (1962). Ele é uma evolução de um procedimento conhecido como DP, descrito pela primeira vez em Davis e Putnam (1960). Apesar de sua já longa data de proposição, o DPLL continua sendo a base de muitos algoritmos atuais para resolução de instâncias SAT, quando implementado com melhorias. Uma implementação conhecida como CHAFF, por exemplo, é usada para resolver problemas de verificação de hardware com um milhão de variáveis (RUSSELL; NORVIG, 2010).

B.1 DESCRIÇÃO

O objetivo básico do algoritmo DPLL é, dada uma fórmula da lógica proposicional em CNF, dizer se esta é satisfazível ou não. Ele faz isso tentando construir um modelo através de atribuições sucessivas de valores verdade as variáveis da fórmula. Grosso modo, a cada atribuição de valor verdade o algoritmo testa a fórmula para verificar se, com as atribuições feitas até o momento, pode-se concluir se a fórmula é ou não satisfazível. Por isso diz-se que ele é, em essência, uma enumeração recursiva, em profundidade, de modelos possíveis para uma dada fórmula. Isto o torna, em contraposição com o Algoritmo de Transformação Dual, de entendimento bastante simples. Todos os seus conceitos estão relacionados aos conceitos de resolução por tentativa e erro ou força bruta, que são o caminho lógico natural para o entendimento da construção de modelos para uma dada instância SAT. Note-se ainda, que esta abordagem parte de uma visão totalmente semântica da fórmula.

Ao DPLL incorporam-se algumas melhorias propostas ao longo dos anos, como descrito em Russell e Norvig (2010), são elas:

Término prematuro
O algoritmo detecta se a sentença tem de ser verdadeira ou falsa, mesmo no caso de um modelo parcialmente concluído (com apenas parte das variáveis tendo valor verdade atribuído). Uma cláusula é verdadeira se qualquer literal é verdadeiro, mesmo que os outros literais não tenham valores-verdade. Consequentemente, a sentença como um todo pode ser considerada verdadeira, mesmo antes de o modelo estar completo. Por exemplo, a sentença $(A \vee B) \wedge (A \vee C)$ é verdadeira se A é verdadeiro, independentemente dos valores de A e C. De modo semelhante, uma sentença é falsa se qualquer cláusula é falsa, o que ocorre quando cada um de seus literais é falso. Mais

uma vez, isso pode ocorrer bem antes de o modelo estar completo. O término prematuro, evita que subárvores inteiras no espaço de busca sejam examinadas.

Heurística de símbolo puro
Um símbolo puro é um símbolo que sempre aparece com o mesmo "sinal" em todas as cláusulas. Por exemplo, nas três cláusulas $(A \vee \neg B)$, $(\neg B \vee \neg C)$ *e* $(C \vee A)$, o símbolo A é puro porque só aparece o literal positivo, B é puro porque só aparece o literal negativo e C é impuro. É fácil ver que, se uma sentença tem um modelo, então ela tem um modelo com os símbolos puros atribuídos de forma a tornar seus literais verdadeiros, porque isso nunca poderá tornar uma cláusulas falsa. Na determinação da pureza de um símbolo, o algoritmo pode ignorar cláusulas que já são reconhecidas como verdadeiras no modelo construído até o momento. Por exemplo, se o modelo contém B = falso, a cláusula $(\neg B \vee \neg C)$ já é verdadeira, e C se torna puro porque só aparece em $(C \vee A)$.

Heurística de cláusula unitária
Uma cláusula unitária é uma cláusula com apenas um literal. No contexto de DPLL, essa expressão também significa cláusula em que todos os literais com exceção de um já têm o valor falso atribuído pelo modelo. Por exemplo, se o modelo contém B = falso, então $(B \vee \neg C)$ se torna uma cláusula unitária, porque é equivalente a $(Falso \vee \neg C)$, ou simplesmente $\neg C$. É óbivio que, para que esta cláusula seja verdadeira, C deve ser definido como falso. A heurística de cláusula unitária atribui todos esse símbolos antes de efetuar a ramificação sobre o restante. A atribuição de uma cláusula unitária pode tornar outra cláusula unitária - por exemplo, quando C é definido como falso, $(C \vee A)$ se torna uma cláusula unitária, fazendo com que o valor verdadeiro seja atribuído a A. Essa "cascata" de atribuições forçadas é chamada propagação unitária.

Com isso, a forma básica deste algoritmo, como descrita em Davis, Logemann e Loveland (1962) e adaptada em Russell e Norvig (2010) (p. 216), é apresentada na figura 6. Esta versão apresenta as três melhorias descritas anteriormente. Uma versão mais simples, que não incorpora todas as melhorias também é descrita em Voronkov (2012) e apresentada na figura 7.

1: **função** SATISFATÍVEL-DPLL?(s) **retorna** verdadeiro ou falso
2: **entrada:** s, uma sentença em lógica proposicional
3: cláusulas ← o conjunto de cláusulas na representação em CNF de s
4: símbolos ← uma lista de símbolos proposicionais em s
5: **retornar** DPLL(cláusulas, símbolos, [])

1: **função** DPLL(cláusulas, símbolos, modelo) **retorna** verdadeiro ou falso
2: **se** toda cláusula em cláusulas é verdadeira em modelo **então retornar** verdadeiro
3: **se** alguma cláusula em cláusulas é falsa em modelo **então retornar** falso
4: P, valor ← ENCONTRAR-SÍMBOLO-PURO(símbolos, cláusulas, modelo)
5: **se** P é não-nulo **então retornar** DPLL(cláusulas, símbolos - P, ESTENDER(p, valor, modelo))
6: P, valor ← ENCONTRAR-CLÁUSULA-UNITÁRIA(cláusulas, modelo)
7: **se** P é não-nulo **então retornar** DPLL(cláusulas, símbolos - P, ESTENDER(p, valor, modelo))
8: P ← PRIMEIRO(símbolos); restantes ← RESTO(símbolos)
9: **retornar** DPLL(cláusulas, restantes, ESTENDER(P, verdadeiro, modelo)) **ou** DPLL(cláusulas, restantes, ESTENDER(P, falso, modelo))

Figura 6: Algorimo DPLL apresentado em Russell e Norvig (2010)

```
1: função DPLL(S)
2: entrada: conjunto de cláusulas S
3: saída: satisfazível ou insatisfazível
4: parâmetros: função selecionar_literal
5: início
6:    S := propagar(S)
7:    se S é vazio então retorne satisfazível
8:    se S contêm a cláusula vazia então retorne insatisfazível
9:    L := selecionar_literal(S)
10:   se DPLL(S ∪ {L}) = satisfazível
11:      então retorne satisfazível
12:   senão retorne DPLL(S ∪ {¬L})
13: fim
```

Figura 7: Algorimo DPLL como apresentado em Voronkov (2012)

Nesta segunda versão, a função *propagar* implementa a melhoria definida pela heurística de cláusula unitária. Através dela, cláusulas unitárias são identificadas e a atribuição de valor a variável proposicional que torna o seu único literal verdadeiro é adicionada ao modelo atual. Caso novas cláusulas unitárias sejam geradas em função desta atribuição, o procedimento se repete. A função termina quando nenhuma cláusula unitária for encontrada.

A função *selecionar_literal*, por sua vez, escolhe um dos literais da teoria descrita por *s* cuja variável proposicional ainda não tenha valor atribuído no modelo atual. Esta variável ocasionará o que se conhece por "splitting", pois a checagem do modelo prossegue primeiro atribuindo-se valor a variável proposicional de forma a tornar o literal verdadeiro e, se a busca por esse caminho falhar, tenta-se o oposto. A decisão sobre qual será o literal escolhido e consequentemente qual será a próxima variável proposicional a ter o seu valor atribuído no modelo, é inerentemente uma decição heurística. Contudo, esta decisão pode ter um efeito bastante grande sobre o tempo total necessário para o DPLL resolver instâncias SAT (DARWICHE; PIPATSRISAWAT, 2009).

B.2 EXTENSÃO

É possível estender o algoritmo DPLL como descrito por Voronkov (2012) para que o mesmo retorne todos os modelos possíveis para a fórmula de entrada. Para tanto algumas alterações são necessárias. O algoritmo em

si, porém, é estruturalmente semelhante ao apresentado na Figura 7. A esta versão extendida, dá-se o nome de DPLL-all.

```
1:  função DPLL-all(S)
2:  entrada: conjunto de cláusulas S
3:  saída:  conjunto de modelos de S
4:  início
5:     Φ  :=  ∅
6:     S,F  :=  propagar(S)
7:     se S é vazio então retorne {F}
8:     se S contêm a cláusula vazia então retorne {}
9:     Ω  :=  literais_restantes(S)
10:    para todo L em Ω
11:       modelos = DPLL-all(S ∪ {L})
12:       para todo modelo em modelos
13:          Φ = Φ ∪ {F ∪ modelo}
14:    retorne Φ
15: fim
```

Figura 8: Algorimo DPLL-all

A primeira alteração necessária é a alteração da função *propagar*. Esta função, além de retornar *s* com os efeitos da propagação já aplicados a mesma, passa a também retornar uma lista com todos os literais das cláusulas unitárias que foram encontradas. Esta lista representa uma série de atribuições de valores verdade. Com efeito, se *s* estiver vazia depois da propagação unitária, esta lista será um modelo de *s*. A esta lista dá-se o nome de F.

A segunda alteração necessária é a eliminação da escolha de um literal para *splitting*. Como queremos retornar todos os modelos para uma determinada teoria, passamos a considerar todos os literais que continuam na teoria (sem valor atribuido a variável proposicional associada ao literal) para a continuação da busca por modelos. Para isso introduz-se a função *literais_restantes* que tem justamente a função de retornar todos os literais de *s* cuja variável associada ainda não teve valor verdade atribuído. Unindo-se estes literais a *s*, em forma de cláusula unitária e, depois, chamando a função DPLL-all recursivamente sobre o resultado desta união, se obtém os modelos para todas as sub-árvores de busca. Por fim, basta unir-se a estes modelos os valores de F, que são as propagações já feitas anteriormente, e obtém-se os modelos de *s*.

B.2.1 Geração de cláusulas duais mínimas

Sabemos, por definição, que a fórmula de entrada do algoritmo DPLL-all que está em CNF, tem uma fórmula dual em DNF. Dentre os modelos gerados como saída do algoritmo DPLL-all, alguns representam cláusulas desta fórmula dual em DNF. São aqueles modelos que tem valor verdade atribuído para exatamente as mesmas variáveis que aparecem em uma das cláusulas da representação DNF. Geralmente não se conhece a fórmula dual em DNF da fórmula de entrada, mas é possível verificar a relação entre um modelo e uma cláusula dual através da definição de cláusula dual mínima, apresentada na seção 5.1. Se as atribuições de valor verdade do modelo satisfazem tal definição, então o modelo representa uma cláusula dual mínima, ou seja, representa uma cláusula da fórmula dual da fórmula de entrada.

Portanto, o algoritmo DPLL-all em conjunto com a verificação acima descrita é capaz de retornar as cláusulas duais mínimas para uma dada teoria s. Em outras palavras, esta combinação é capaz de calcular a fórmula dual a fórmula de entrada. O Algoritmo de Transformação Dual tem este mesmo objetivo.

B.3 COMPARAÇÃO COM O ALGORITMO DE TRANSFORMAÇÃO DUAL

A versão descrita neste apêndice do Algoritmo DPLL, foi comparada com a versão com melhorias do Algoritmo de Transformação Dual. Esta comparação foi feita apenas para demonstrar que o segundo é, aparentemente, mais eficiente que o primeiro na resolução de instâncias SAT. Para tal comparação, foram geradas dez teorias aleatórias, com três literais por cláusula, doze variáveis proposicionais e trinta cláusulas cada uma. Tais teorias encontram-se disponíveis em *https://github.com/murilloflores/ sat-solvers/ blob/new_data_structure/examples/uf-12.zip*.

Todos os testes foram executados em uma máquina com processador Intel Core i5-2410M com 2,3GHz e 2 GiB. Os resultados obtidos são apresentados na tabela 14, onde "TT" é o tempo total em centésimos de segundo (0,01s), "Mem." é o máximo de memória alocada durante a execução, em MB, e $|W_d|$ é o número de cláusulas duais mínimas geradas.

Os resultados indicam claramente que o Algoritmo de Transformação Dual é bastante superior ao Algoritmo DPLL em ambos os critérios avaliados.

Teoria	DPLL		Dual		$\|W_d\|$
	TT	Mem	TT	Mem	
problema-0.cnf	2080	88	7	2	15
problema-1.cnf	147	56	2	1	10
problema-2.cnf	1115	77	2	1	30
problema-3.cnf	130	57	1	1	14
problema-4.cnf	119	57	1	1	17
problema-5.cnf	2060	61	1	1	15
problema-6.cnf	1575	66	2	2	25
problema-7.cnf	108	48	1	1	18
problema-8.cnf	135	61	1	1	8
problema-9.cnf	1204	67	1	1	27

Tabela 14: Comparação entre DPLL e Algoritmo de Transformação Dual

APÊNDICE C – Código fonte

Apresenta-se a seguir o código fonte da implementação final do Algoritmo de Transformação Dual, já contendo as três melhorias propostas neste trabalho.

```
package solvers;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import representation.Clause;
import representation.SearchState;
import representation.Theory;
import util.BitWiseUtils;
import core.MemoryAnalyzer;

public class DualSolver implements Solver {

  private Theory theory;
  private byte[][] quantumTable;
  private int coordinatesArraySize;

  public boolean isSatisfiable(Theory theory){
    List<SearchState> finalStates =
        calculateFinalStates(theory, true);
    return !finalStates.isEmpty();
  }

  @Override
  public List<Clause> toMinimalDualClauses(Theory
      theory) {

    //Start memory analyzer
    MemoryAnalyzer memoryAnalyzer = new
        MemoryAnalyzer();
    Thread thread = new Thread(memoryAnalyzer);
    thread.start();

    List<Clause> minimalDualClauses = new ArrayList
        <Clause>();
```

```
    List<SearchState> finalStates =
        calculateFinalStates(theory, false);
    for(SearchState state : finalStates){
      List<Integer> literals = new ArrayList<
          Integer>();
      Set<Integer> quantums = state.getQuantums();
      for(Integer quantum: quantums){
        literals.add(quantum);
      }

      minimalDualClauses.add(new Clause(literals));
    }

    //stop memory analyzer
    memoryAnalyzer.stop();
    thread.interrupt();

    System.out.println("␣|␣"+((memoryAnalyzer.
        maxUsedMemory()/1024) / 1024)+"␣|␣"+
        minimalDualClauses.size());
    System.out.println(minimalDualClauses);

    return minimalDualClauses;

  }

  public List<SearchState> calculateFinalStates(
      Theory theory, boolean returnFirst) {

    long loops = 0;
    long begin = System.currentTimeMillis();
    long loopsFirst = 0;
    long timeFirst = 0;

    this.coordinatesArraySize = (theory.
        getNumberOfClauses() + 7) / 8;
    this.theory = theory;
    buildQuantumTable();

    List<SearchState> openedStates =
        calculateInitialOpenedStates();
```

```
List<SearchState> closedStates = new ArrayList<
    SearchState>();
List<SearchState> finalStates = new ArrayList<
    SearchState>();

while(!openedStates.isEmpty()){
  loops++;
  SearchState currentState =
      getStateWithEstimatedLowestCost(
      openedStates);

  if(isFinalState(currentState)){
    if(loopsFirst == 0){
      loopsFirst = loops;
      timeFirst = System.currentTimeMillis();
    }

    openedStates.remove(currentState);
    finalStates.add(currentState);

    if(returnFirst){
      return finalStates;
    }

    continue;
  }

  openedStates.remove(currentState);
  closedStates.add(currentState);

  List<SearchState> neighbors =
      calculateNeighbors(currentState);

  for(SearchState state: neighbors){
    openedStates.add(state);
  }

}

long end = System.currentTimeMillis();
System.out.print(((timeFirst-begin)/10));
```

```
    System.out.print("␣|␣"+(loopsFirst));
    System.out.print("␣|␣"+((end-begin) / 10));
    System.out.print("␣|␣"+(loops));

    return finalStates;
  }

  private boolean isFinalState(SearchState state) {
    return BitWiseUtils.countOnes(state.getGap())
        == 0;
  }

  private SearchState
      getStateWithEstimatedLowestCost(List<
      SearchState> openedStates) {

    int firstStateIndex = 0;
    SearchState firstState = openedStates.get(
        firstStateIndex);
    int firstStateGapSize = BitWiseUtils.countOnes(
        firstState.getGap()); // h(x)
    int firstStateSizeFromTheBeggining = firstState
        .getQuantums().size(); // g(x)
    int firstStateEstimatedCost =
        firstStateSizeFromTheBeggining +
        firstStateGapSize; //f(x)

    int lowestEstimatedCost =
        firstStateEstimatedCost;
    int stateWithLowestEstimatedCostIndex =
        firstStateIndex;

    for(int i=0; i<openedStates.size(); i++){

      SearchState currentState = openedStates.get(i
          );
      int gapSize = BitWiseUtils.countOnes(
          currentState.getGap()); // h(x)
      int sizeFromTheBeginnig = currentState.
          getQuantums().size(); // g(x)
```

```
      int currentStateEstimatedCost =
          sizeFromTheBeginnig + gapSize; // f(x)


      if(currentStateEstimatedCost <
          lowestEstimatedCost){
        lowestEstimatedCost =
            currentStateEstimatedCost;
        stateWithLowestEstimatedCostIndex = i;
      }

    }

    return openedStates.get(
        stateWithLowestEstimatedCostIndex);

  }

  // Quantum table operations

  private byte[] getCoordinates(Integer literal){
    int tableIndex = getQuantumTableIndex(literal);
    return quantumTable[tableIndex];
  }

  private int getQuantumTableIndex(Integer literal)
      {
    int tableIndex;

    if(literal < 0){
      tableIndex = literal + theory.
          getNumberOfVariables();
    }else{
      tableIndex = literal + (theory.
          getNumberOfVariables() - 1);
    }

    return tableIndex;
  }

  private void buildQuantumTable(){
```

```
    int numberOfVariables = theory.
        getNumberOfVariables();
    this.quantumTable = new byte[numberOfVariables
        * 2][coordinatesArraySize];

    for(int literal=numberOfVariables; literal>0;
        literal--){
      fillTableForLiteral(literal);
      fillTableForLiteral(literal * -1);
    }


  }

  private void fillTableForLiteral(int literal) {

    int quantumTableIndex = getQuantumTableIndex(
        literal);

    List<Integer> clausesWithQuantum =
        getClausesWithQuantum(literal);
    for(Integer clause: clausesWithQuantum){

      int pos = (coordinatesArraySize -1) - (clause
          / 8);
      int exp = clause % 8;

      byte b = (byte) Math.pow(2, exp);

      quantumTable[quantumTableIndex][pos] = (byte)
          (quantumTable[quantumTableIndex][pos] |
          b);

    }

  }

  private List<Integer> getClausesWithQuantum(int
      literal) {
```

```
    List<Clause> clauses = theory.getClauses();

    List<Integer> positions = new ArrayList<Integer
        >();

    for(int i=0; i<clauses.size(); i++){
      if(clauses.get(i).getLiterals().contains(
          literal)){
        positions.add(i);
      }
    }

    return positions;

  }

  // Initial state calculus

  private List<SearchState>
      calculateInitialOpenedStates() {

    Clause clause = getBestCnfClauseToStart();
    List<Integer> clauseLiterals =
        sortInitialQuantums(clause.getLiterals());

    List<SearchState> initialOpenedStates = new
        ArrayList<SearchState>();

    for(int i=0; i<clauseLiterals.size(); i++){

      SearchState state = new SearchState();
      state.addQuantum(clauseLiterals.get(i));
      state.addForbiddenQuantum(clauseLiterals.get(
          i) * -1);

      for(int j=i-1; j>=0; j--){
        state.addForbiddenQuantum(clauseLiterals.
            get(j));
      }

      byte[] calculatedGap = calculateGap(state);
```

```
        state.setGap(calculatedGap);

        initialOpenedStates.add(state);
      }

      return initialOpenedStates;

    }

    private byte[] calculateGap(SearchState
        searchState) {

      byte[] gap = completeGap();

      for(Integer quantum: searchState.getQuantums())
          {
        byte[] quantumCoordinates = getCoordinates(
            quantum);
        gap = byteArrayXor(gap, quantumCoordinates);
      }

      return gap;

    }

    private List<Integer> sortInitialQuantums(List<
        Integer> clauseLiterals) {
      SearchState dummyState = new SearchState();
      dummyState.setGap(completeGap());

      sortQuantumsAccordingToHeuristic(clauseLiterals
          , dummyState);
      return clauseLiterals;
    }

    private void sortQuantumsAccordingToHeuristic(
        List<Integer> possibleExtensions, SearchState
         currentState) {

      byte[] gap = currentState.getGap();
```

```
for(int i=0; i<possibleExtensions.size(); i++){
  for(int j=i+1; j<possibleExtensions.size(); j
      ++){

    Integer quantumI = possibleExtensions.get(i
        );
    byte[] coordinatesQuantumI = getCoordinates
        (quantumI);

    Integer quantumJ = possibleExtensions.get(j
        );
    byte[] coordinatesQuantumJ = getCoordinates
        (quantumJ);

    byte[] interGapQuantumI = byteArrayAnd(
        coordinatesQuantumI, gap);
    byte[] interGapQuantumJ = byteArrayAnd(
        coordinatesQuantumJ, gap);

    byte[] interGapIJ = byteArrayAnd(
        interGapQuantumI, interGapQuantumJ);

    byte[] interGapIMinusIJ = subtract(
        interGapQuantumI, interGapIJ);
    int counterInterGapIMinusIJ = BitWiseUtils.
        countOnes(interGapIMinusIJ);

    byte[] inteGapJMinusIJ = subtract(
        interGapQuantumJ, interGapIJ);
    int counterInterGapJMinusIJ = BitWiseUtils.
        countOnes(inteGapJMinusIJ);

    if(counterInterGapIMinusIJ ==
        counterInterGapJMinusIJ){

      Integer mirrorQuantumI = quantumI * -1;
      byte[] mirrorICoordinates =
          getCoordinates(mirrorQuantumI);

      Integer mirrorQuantumJ = quantumJ * -1;
```

```
            byte[] mirrorJCoordinates =
                getCoordinates(mirrorQuantumJ);

            byte[] interGapMirrorI = byteArrayAnd(
                mirrorICoordinates, gap);
            byte[] interGapMirrorJ = byteArrayAnd(
                mirrorJCoordinates, gap);

            byte[] interGapMirrorIJ = byteArrayAnd(
                interGapMirrorI, interGapMirrorJ);

            int counterInterGapMirrorIMinusIJ =
                BitWiseUtils.countOnes(subtract(
                interGapMirrorI, interGapMirrorIJ));
            int counterInterGapMirrorJMinusIJ =
                BitWiseUtils.countOnes(subtract(
                interGapMirrorJ, interGapMirrorIJ));

            if(!(counterInterGapMirrorIMinusIJ >
                counterInterGapMirrorJMinusIJ)){
              possibleExtensions.set(i, quantumJ);
              possibleExtensions.set(j, quantumI);
            }

          } else {
            if(!(counterInterGapIMinusIJ >
                counterInterGapJMinusIJ)){
              possibleExtensions.set(i, quantumJ);
              possibleExtensions.set(j, quantumI);
            }
          }

        }
      }

    }

    private byte[] subtract(byte[] b1, byte[] b2) {

      byte[] sub = byteArrayAnd(b1, b2);
      sub = byteArrayXor(sub, b1);
```

```
    return sub;
  }


  private Clause getBestCnfClauseToStart() {

    //Trying heuristic to determine the first
        clause to be used, i.e. the initial state (
        pg 25)

    int bestClauseIndex = 0;
    int maxCoverage = 0;

    for(int i=0; i<theory.getClauses().size(); i++)
        {

      Clause clause = theory.getClauses().get(i);

      byte[] allCoordinates = new byte[
          coordinatesArraySize];
      for(Integer literal: clause.getLiterals()){
        byte[] literalCoordinates = getCoordinates(
            literal);
        allCoordinates = byteArrayOr(allCoordinates
            , literalCoordinates);
      }

      int coverage = BitWiseUtils.countOnes(
          allCoordinates);
      if(coverage > maxCoverage){
        bestClauseIndex = i;
        maxCoverage = coverage;
      }

    }

    return theory.getClauses().get(bestClauseIndex)
        ;

  }
```

```
// Neighbor calculus operatinos

private List<SearchState> calculateNeighbors(
    SearchState currentState) {

  // Sucessors function implemented from pg 25

  List<SearchState> sucessors = new ArrayList<
      SearchState>();

  //step 1
  List<Integer> possibleExtensions =
      determinePossibleExtensions(currentState);

  //step 2
  sortQuantumsAccordingToHeuristic(
      possibleExtensions, currentState);


  //step 3
  List<Clause> gapConditions = gapConditions(
      currentState);
  for(Clause clause: gapConditions){
    if(!intersects(clause, possibleExtensions)){
      return new ArrayList<SearchState>();
    }
  }

  List<Integer> usedQuantums = new ArrayList<
      Integer>();
  List<Integer> refused = new ArrayList<Integer
      >();

  //step 4
  for(Integer quantum: possibleExtensions){

    SearchState possibleNextState = new
        SearchState(currentState);
    possibleNextState.addQuantum(quantum);
    Integer mirrorQuantum = quantum * -1;
```

```
        possibleNextState.addForbiddenQuantum(
            mirrorQuantum);
        removeFromGapClausesOfQuantum(
            possibleNextState, quantum);

        if(isExclusiveCoordinateCompatible(
            currentState, quantum)
            && gapConditionsAreSatisfied(
                gapConditions(possibleNextState),
                currentState)
            ){

          for(Integer forbiddenQuantum:usedQuantums){
            possibleNextState.addForbiddenQuantum(
                forbiddenQuantum);
          }

          usedQuantums.add(quantum);

          sucessors.add(possibleNextState);

        }else{
          refused.add(quantum);
        }

      }

      for(SearchState sucessor: sucessors){
        for(Integer quantum: refused){
          sucessor.addForbiddenQuantum(quantum);
        }
      }

      // MELHORIA 1
      // List<SearchState> sucessorsWithFuture = new
          ArrayList<SearchState>();
      // for(SearchState sucessor: sucessors){
      //   List<Clause> gapConditionsSucessor =
          gapConditions(sucessor);
      //   if(haveFuture(gapConditionsSucessor,
          sucessor)){
```

```
    //      sucessorsWithFuture.add(sucessor);
    //    }
    //  }
    //  return sucessorsWithFuture;

    return sucessors;
  }


  private void removeFromGapClausesOfQuantum(
      SearchState possibleNextState, Integer
      quantum) {

    byte[] gap = possibleNextState.getGap();
    byte[] coordinates = getCoordinates(quantum);

    byte[] newGap = subtract(gap, coordinates);
    possibleNextState.setGap(newGap);

  }

  private List<Clause> gapConditions(SearchState
      state) {

    List<Clause> gapConditions = new ArrayList<
        Clause>();

    Set<Integer> mirrorQuantums = calculateMirror(
        state.getQuantums());
    // MELHORIA 2
    // Set<Integer> mirrorQuantums = state.
        getForbiddenQuantums();
    List<Integer> clausesInGap = getClausesFromGap(
        state.getGap());

    for (Integer clause : clausesInGap) {
      // MELHORIA 3
      if (!intersects(clause, mirrorQuantums))
        continue;
```

```
        Clause clone = new Clause(theory.getClauses()
            .get(clause));
        removeLiteralsOfQuantumsFromClause(
            mirrorQuantums, clone);
        gapConditions.add(clone);

      }

      return gapConditions;
    }

    private void removeLiteralsOfQuantumsFromClause(
        Set<Integer> mirrorQuantumLiterals, Clause
        clause) {
      for (Integer literal : mirrorQuantumLiterals) {
        clause.removeLiteral(literal);
      }
    }


    private boolean intersects(Integer clause,
        Collection<Integer> quantums) {

      int pos = (coordinatesArraySize -1) - (clause /
          8);
      int exp = clause % 8;

      byte b = (byte) Math.pow(2, exp);

      for(Integer quantum: quantums){

        byte[] quantumCoordinates = getCoordinates(
            quantum);
        byte intersec = (byte) (quantumCoordinates[
            pos] & b);
        if(BitWiseUtils.countOnes(intersec) > 0)
            return true;

      }

      return false;
```

```
  }

  private boolean intersects(Clause clause, List<
      Integer> possibleExtensions) {

    for(Integer literal: clause.getLiterals()){
      if(possibleExtensions.contains(literal))
          return true;
    }
    return false;
  }


  private List<Integer> getClausesFromGap(byte[]
      gap) {

    List<Clause> clauses = theory.getClauses();
    List<Integer> clausesInGap = new ArrayList<
        Integer>();

    for(int i=0; i<clauses.size(); i++){

      int pos = (coordinatesArraySize -1) - (i / 8)
          ;
      int exp = i % 8;

      byte b = (byte) Math.pow(2, exp);

      byte b2 = (byte) (gap[pos] & b);

      if(b2 == b){
        clausesInGap.add(i);
      }

    }

    return clausesInGap;

  }
```

```
private Set<Integer> calculateMirror(Set<Integer>
    quantums) {

  Set<Integer> mirror = new HashSet<Integer>();
  for (Integer quantum : quantums) {
    mirror.add(quantum*-1);
  }

  return mirror;

}

private List<Integer> determinePossibleExtensions
   (SearchState currentState) {

  List<Integer> possibleExtensions =
      getLiteralsFromGapOf(currentState);
  possibleExtensions.removeAll(currentState.
      getForbiddenQuantums());
  return possibleExtensions;

}

private List<Integer> getLiteralsFromGapOf(
    SearchState currentState) {

  byte[] gap = currentState.getGap();

  List<Integer> literalsInTheClausesOfGap = new
      ArrayList<Integer>();
  for(int i=theory.getNumberOfVariables()*-1; i<=
      theory.getNumberOfVariables(); i++){
    if(i==0) continue;

    byte[] coordinates = getCoordinates(i);
    byte[] intersection = byteArrayAnd(gap,
        coordinates);
    if(BitWiseUtils.countOnes(intersection) > 0){
      literalsInTheClausesOfGap.add(i);
    }
```

```
    }

    return literalsInTheClausesOfGap;
  }

  @SuppressWarnings("unused")
  private boolean haveFuture(List<Clause>
      nextStateGapConditions, SearchState state){

    for(Clause clause: nextStateGapConditions){
      if(state.getForbiddenQuantums().containsAll(
          clause.getLiterals())) return false;
    }

    return true;
  }


  private boolean gapConditionsAreSatisfied(List<
      Clause> nextStateGapConditions, SearchState
      currentState){

    Set<Integer> forbiddenQuantums = currentState.
        getForbiddenQuantums();

    for(Clause clause: nextStateGapConditions){

      if(clause.isEmpty()) return false; // Old
          isNewRestrictionsContradictory function

      if(forbiddenQuantums.containsAll(clause.
          getLiterals())){ // Old
          isNewRestrictionsCompatibleWithFobiddenList
           function
        return false;
      }

      if(clause.isUnit()){
        for(Clause otherClause:
            nextStateGapConditions){
```

```
            if(otherClause.isUnit() && otherClause.
                getLiterals().get(0).equals(clause.
                getLiterals().get(0) * -1)){
              return false;
            }
          }
        }

      }

      return true;

    }

    private boolean isExclusiveCoordinateCompatible(
        SearchState currentState, Integer
        quantumBeingAdded) {

      byte[] quantumBeingAddedCoordinates =
          getCoordinates(quantumBeingAdded);

      for(Integer quantum: currentState.getQuantums()
          ){

        byte[] exclusiveCoordinatesOfQuantum =
            getExclusiveCoordinatesFor(currentState,
            quantum);

        boolean equal = true;
        for(int i=0; i<coordinatesArraySize; i++){
          byte comparison = (byte) (
              exclusiveCoordinatesOfQuantum[i] &
              quantumBeingAddedCoordinates[i]);
          if (!(comparison ==
              exclusiveCoordinatesOfQuantum[i])){
            equal = false;
            break;
          }
        }

        if(equal) return false;
```

```
    }

    return true;
  }


  private byte[] getExclusiveCoordinatesFor(
      SearchState currentState, Integer quantum) {

    byte[] allCoordinates = new byte[
        coordinatesArraySize];
    for(Integer currentStateQuantum: currentState.
        getQuantums()){
      if(!currentStateQuantum.equals(quantum)){
        allCoordinates = byteArrayOr(allCoordinates
            , getCoordinates(currentStateQuantum));
      }

    }

    byte[] exclusiveCoordinates = byteArrayXor(
        getCoordinates(quantum), allCoordinates);
    exclusiveCoordinates = byteArrayAnd(
        exclusiveCoordinates, getCoordinates(
        quantum));

    return exclusiveCoordinates;
  }

  // ByteArray operations

  private byte[] byteArrayOr(byte[] allCoordinates,
      byte[] literalCoordinates) {

    byte[] result  = new byte[coordinatesArraySize
        ];
    for(int i=0; i<coordinatesArraySize; i++){
      result[i] = (byte) (allCoordinates[i] |
          literalCoordinates[i]);
    }
```

```
    return result;
  }

  private byte[] byteArrayXor(byte[] b1, byte[] b2)
      {

    byte[] result  = new byte[coordinatesArraySize
        ];
    for(int i=0; i<coordinatesArraySize; i++){
      result[i] = (byte) (b1[i] ^ b2[i]);
    }
    return result;
  }

  private byte[] byteArrayAnd(byte[] b1, byte[] b2)
      {

    byte[] result  = new byte[coordinatesArraySize
        ];
    for(int i=0; i<coordinatesArraySize; i++){
      result[i] = (byte) (b1[i] & b2[i]);
    }
    return result;
  }

  private byte[] completeGap() {

    byte[] gap = new byte[coordinatesArraySize];
    int pos = theory.getClauses().size() / 8;
    int exp = theory.getClauses().size() % 8;

    // fill begin with zeros
    for(int i=0; i<pos; i++){
      gap[coordinatesArraySize-1-i] = (byte) 255;
    }

    //fill last one
    for(int i=0; i<exp; i++){

      byte b = (byte) Math.pow(2, i);
```

```
            gap[coordinatesArraySize-1-pos] = (byte) (gap
                [coordinatesArraySize-1-pos] | b);

        }

        return gap;
    }

}
```

APÊNDICE D – Artigo

Melhoramentos para o Algoritmo de Transformação Dual

Murillo Lagranha Flores[1], Jerusa Marchi[1]

[1] Departamento de Informática e Estatística – Universidade Federal de Santa catarina
Campus Universiário João David Ferreira Lima – Florianópolis – SC – Brazil

{murillo.lf,jerusa}@inf.ufsc.br

Abstract. *The problem of Boolean satisfiability, or SAT problem is a classic problem in computing. It was the first proven NP-Complete problem and improve the way we deal with it means improving the way we deal with complexity in general. This report presents the ``Dual Transformation Algorithm'', which is an algorithm for solving the SAT problem, proposing improvements to it, and demonstrating that these improvements reduce the computational cost associated with solving SAT by this algorithm.*

Resumo. *O problema da satisfazibilidade Booleana, ou problema SAT, é um problema clássico em computação. Foi o primeiro problema provado NP-Completo e melhorar a forma como trabalhamos com ele significa melhorar a forma como trabalhamos com complexidade em geral. Este trabalho apresenta o Algoritmo de Transformação Dual, que é um algoritmo de resolução do problema SAT, pronpondo-lhe melhorias, e demonstrando que tais melhorias reduzem o custo computacional associado à resolução de SAT por tal algoritmo.*

1. Introdução

O problema da satisfazibilidade Booleana, ou problema SAT, é um problema NP-Completo, pertencendo portanto à classe de problemas de resolução custosa. Desde há muito tempo, algoritmos e métodos para a resolução de tal problema vêm sendo propostos e aprimorados para diferentes contextos/usos específicos, tornando possível inclusive a classificação de tais estudos de acordo com uma série de características. Uma destas características que permite a classificação é a completude do algoritmo - se consegue retornar a resposta, ao invés de apenas dizer que ela existe. O Algoritmo de Transformação Dual é um algoritmo completo para a resolução do problema SAT.

Este trabalho tem por objetivo propor três melhorias ao Algoritmo de Transformação Dual e comprovar que as mesmas reduzem o tempo de resposta e a quantidade de memória utlizada pelo mesmo, permitindo assim que instâncias maiores (com mais cláusulas e variáveis proposicionais) sejam resolvidas com menos recursos computacionais. Como resultado é esperada a comprovação desta hipótese de trabalho através da análise dos resultados obtidos nos testes. Alcançar tal objetivo significa descrever as melhorias propostas, implementá-las, executar os testes e comparar os dados obtidos, comprovando a hipótese.

2. Lógica proposicional e Complexidade computacional

O problema da satisfazibilidade booleana, que é o problema central atacado pelo algoritmo ao qual se propõe melhorias neste trabalho, fundamenta-se principalmente sobre coceitos da lógica proposicional e complexidade computacional.

2.1 Lógica proposicional

A lógica proposicional, ou lógica Booleana, estuda as proposições e a combinação destas através de conectivos [Rosen, 2010]. Como toda lógica, ela pode ser definida em função da sua sintaxe e da sua semântica [Russel; Norvig, 2010].

A sintaxe de uma lógica define quais sentenças são permitidas [Russel; Norvig, 2010]. Em lógica proposicional, a menor sentença permitida é a sentença atômica, que consiste em um único símbolo proposicional. Cada símbolo proposicional representa uma proposição que pode ser verdadeira ou falsa. Um símbolo proposicional representa uma variável proposicional. Cada variável proposicional é geralmente representada por uma letra [Russel; Norvig, 2010].

Sentenças complexas de lógica proposicional podem ser construídas a partir do uso de conectivos lógicos. Existem vários conectivos lógicos, mas os cinco mais comumente usados e que são base deste trabalho, como descritos em [Russel; Norvig, 2010], são:

¬ (negacao) Uma sentenca como ¬P e chamada de negação de P. Um literal é uma sentenca atômica (um literal positivo) ou uma sentença atômica negada (um literal negativo).

∧ (e) Uma sentenca cujo principal conectivo é ∧, como P ∧ Q, é chamada de conjunção e suas partes são os elementos da conjunção.

∨ (ou) Uma sentenca que utiliza ∨, como (P ∧ Q) ∨ R é uma disjunção dos disjuntos (P ∧ Q) e R.

⇒ (implica) Uma sentenca com (Q ∧ A) ⇒ J é dita uma implicação. (Q ∧ A) é a premissa e J é a conclusão.

⇔ (se e somente se) A sentença que utiliza este conector é uma sentença bicondicional, como em R ⇔ ¬U.

Para construir-se as sentenças complexas com estes conectivos, e a partir de sentenças mais simples, valem as regras da gramática livre de contexto, apresentada em [RAUBER, 2011].

Uma teoria é um conjunto de proposições. Uma fórmula em lógica proposicional é um conjunto de variáveis proposicionais que representam, cada uma, uma proposição e, logo, uma fórmula em lógica proposicional representa uma teoria.

A semântica da lógica proposicional define uma maneira de se computar o valor verdade de uma sentença respeitando um conjunto de atribuições para as suas variáveis proposicionais. Este conjunto de atribuições são interpretações possíveis. Se uma interpretação torna a sentença verdadeira, diz-se que é um modelo da sentença.

O cálculo dos valores verdade das sentenças é, então, feito recursivamente. Todas as sentenças podem ser construídas a partir de sentenças atômicas e dos cinco conectivos apresentados anteriormente. Portanto precisamos saber como computar o valor verdade para sentenças atômicas e como computar o valor verdade para sentenças construídas com os conectivos. Para sentenças atômicas valem os valores verdade dos literais que compõe a sentença. Para as demais, valem os valores verdade estipulados pela tabela 1.

Tabela 1. Tabela verdade para os conectivos da lógica proposicional

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
F	F	V	F	F	V	V
F	V	V	F	V	V	F
V	F	F	F	V	F	F
V	V	F	V	V	F	V

Em lógica proposicional, uma sentença **a** é satisfazível se ela é verdade em alguma interpretação. Se **a** é verdade em um modelo m, então dizemos que m satisfaz **a** ou que m é um modelo de **a**.

Toda sentença da lógica proposicional é logicamente equivalente a uma conjunção de disjunções de literais [Russel; Norvig, 2010]. Quando uma dada fórmula **F** é uma conjunção de disjunções de literais, diz-se que ela está na **forma normal conjuntiva** ou **CNF**. A forma normal conjuntiva é uma forma normal canônica. Outra forma normal canônica é a **forma normal disjuntiva** ou **DNF**. Uma fórmula **F** está na forma normal disjuntiva se ela for uma disjunção de conjunções de literais. Diz-se que a forma normal conjuntiva é uma forma dual da forma normal disjuntiva, e vice-versa.

2.2 Complexidade computacional

Intuitivamente, analisar a complexidade de um algoritmo significa analisar a quantidade de tempo e espaço necessários para que o mesmo calcule a saída para uma entrada qualquer e a relação que esta quantidade de tempo e espaço tem com o tamanho da entrada [Voronkov, 2012].

Em computação, o padrão mais usual para realizar tal tipo de análise e para classificar os algoritmos, vem da definição da Máquina de Turing. A partir das definições das Máquinas de Turing Determinística e Não-Determinística, derivam as classes de complexidade que classificam as linguagens reconhecidas pelas mesmas, segundo tais definições. De maneira resumida, existem duas classes:

- **Classe P**: A classe P é a classe das linguagens que são decidíveis em tempo polinomial numa Máquina de Turing determinística.
- **Classe NP**: A classe NP é a classe das linguagens que são decidíveis em tempo polinomial numa Máquina de Turing Não-Determinística.

Existem maneiras alternativas de definir estas classes de complexidade. Como todos os demais problemas podem ser reduzidos a problemas de linguagem, esta classificação se aplica para problemas em geral.

Dentro da classe NP, existem linguagens que são ditas NP-Completas. Uma linguagem é NP-Completa se todas as demais linguagens NP forem reduzíveis, em tempo polinomial, a ela. Uma redução é uma transformação de um problema em outro ou, neste caso, de uma linguagem em outra.

A primeira linguagem a ser provada NP-Completa foi SAT, linguagem que modela o problema da satisfazibilidade Booleana, central a este trabalho. A prova pode ser encontrada em [Sisper, 1996]. Várias outras linguagens foram provadas NP-Completas a partir da prova da NP-Completude de SAT. A importância teórica das linguagens NP-Completas é grande, porque melhorar a maneira como lidamos com estas linguagens significa melhorar a maneira como lidamos com a complexidade em geral.

3. Problema SAT

A partir dos conceitos básicos da lógica proposicional, podemos definir o problema da satisfazibilidade booleana ou problema SAT, como o problema de se encontrar os modelos possíveis para uma dada fórmula da lógica proposicional, ou determinar se tais modelos não existem. Se existem modelos que satisfazem uma determinada fórmula da lógica proposicional, diz-se que esta fórmula é satisfazível. Em caso contrário, diz-se que a fórmula é instatisfazível.

4. Algoritmo de Transformação Dual

O Algoritmo de Transformação Dual é um algoritmo completo para a resolução do problema SAT. É também objeto de estudo e pr oposta de melhorias deste trabalho, e foi idealizado no Departamento de Automação e Sistemas da Universidade Federal de Santa Catarina e apresentado no *4th International Workshop on the Implementation of Logic* [Bittencourt; Marchi; Padilha, 2003]. No artigo de apresentação nenhum nome foi proposto ao mesmo. Ainda assim o denominaremos de “Algoritmo de Transformacão Dual”, em função da abordagem que este emprega para a resolução de SAT. Tal

abordagem objetiva calcular o conjunto de cláusulas da representacão DNF para uma teoria representada inicialmente em CNF, ou seja, calcular a fórmula dual da fórmula de entrada. Se o Algoritmo conseguir calcular pelo menos uma cláusula dual, então a instância SAT é satisfazível.

A ideia por trás do Algoritmo se aproveita da definição, sobre uma fórmula **F** em CNF:

- **F** terá valor “verdadeiro” se, e somente se, todas as suas cláusulas tiverem valor “verdadeiro”.
- Se **C** for uma cláusula de **F**, então **C** terá valor “verdadeiro” se pelo menos um de seus literais tiver valor “verdadeiro”.

A partir desta definição é fácil perceber que um modelo mínimo de **F** é aquele que tem atribuição apenas ao mínimo possível de variáveis proposicionais de maneira a tornar pelo menos um literal de cada cláusula de **F** verdadeiro. Se, pelo menos um literal de cada cláusula de **F** for verdadeiro, então **F** será satisfazível. Como exemplo, tomemos uma teoria **K** cujas cláusulas da representação em CNF sejam:

$$
\begin{array}{lcllcl}
0 & : & [\neg P_4, P_2, \neg P_3] & 5 & : & [\neg P_3, P_2, \neg P_1] \\
1 & : & [P_5, \neg P_2, \neg P_3] & 6 & : & [\neg P_2, \neg P_1, P_5] \\
2 & : & [P_5, \neg P_3, P_1] & 7 & : & [P_4, P_5, P_2] \\
3 & : & [\neg P_5, \neg P_2, P_3] & 8 & : & [\neg P_1, P_4, P_3] \\
4 & : & [\neg P_2, \neg P_3, \neg P_1] & 9 & : & [\neg P_3, \neg P_1, \neg P_4]
\end{array}
$$

Um possível modelo mínimo para esta teoria seria m=[P_5=Falso, P_2=Verdadeiro, P_1=Falso, P_3=Falso]. Esse modelo pode ser representado pelo conjunto dos literais que fazem cada cláusula da teoria em CNF se tornar verdadeira, ficando m=[$\neg P_5$, P_2, $\neg P_1$, $\neg P_3$]. Diz-se que esta abordagem de construção de modelos explora a fórmula em CNF de maneira sintática, pois busca um *caminho* pela *forma* da fórmula, pela sua representação. A construção deste “caminho” gera um modelo.

$$
\begin{array}{lcllcl}
0 & : & [\neg P_4, P_2, \neg P_3] & 5 & : & [\neg P_3, P_2, \neg P_1] \\
1 & : & [P_5, \neg P_2, \neg P_3] & 6 & : & [\neg P_2, \neg P_1, P_5] \\
2 & : & [P_5, \neg P_3, P_1] & 7 & : & [P_4, P_5, P_2] \\
3 & : & [\neg P_5, \neg P_2, P_3] & 8 & : & [\neg P_1, P_4, P_3] \\
4 & : & [\neg P_2, \neg P_3, \neg P_1] & 9 & : & [\neg P_3, \neg P_1, \neg P_4]
\end{array}
$$

A ideia do algoritmo, portanto, é encontrar estes “caminhos”, gerando modelos para a teoria em CNF [Bittencourt; Marchi; Padilha, 2003].

Os caminhos encontrados pelo Algoritmo, são chamados de cláusulas duais mínimas. O conjunto de todas as cláusulas duais mínimas, define a representação em DNF da teoria. Para a teoria **K**, por exemplo, as cláusulas duais mínimas são:

$$
\begin{array}{lcl}
0 & : & [\neg P_4, P_5, \neg P_1, \neg P_2] \\
1 & : & [\neg P_1, P_5, \neg P_3, \neg P_2] \\
2 & : & [P_4, \neg P_3, \neg P_2] \\
3 & : & [P_2, P_3, P_5, \neg P_1] \\
4 & : & [\neg P_4, P_3, P_5, \neg P_1] \\
5 & : & [P_4, \neg P_5, \neg P_1, \neg P_3] \\
6 & : & [\neg P_5, P_2, \neg P_1, \neg P_3]
\end{array}
$$

Nota-se que cada cláusula dual mínima é também um modelo para a representação em CNF e uma cláusula da representação DNF da teoria.

Essa abordagem é bastante diferente da abordagem proposta pelo algoritmo DPLL, que é notoriamente o mais famoso resolvedor SAT e que deu origem a maioria dos resolvedores SAT completos modernos. O DPLL explora as fórmulas, puramente sobre um prisma semântico.

4.1. Clúsula Dual Mínima

Uma cláusula dual mínima é uma cláusula que faz parte da representação DNF da teoria, denominada Wc, e está associada a um conjunto de atribuições de valores verdade que satisfazem esta representação Wc (CNF da teoria).

Um conjunto de literais construído pelo Algoritmo de Transformação Dual, representa uma cláusula dual mínima quando:

- Contém pelo menos um literal que pertence a cada uma das cláusulas de Wc.
- Não contém literais contraditórios (F e ¬F ao mesmo tempo).
- Permite que cada literal represente sozinho pelo menos uma cláusula de Wc.

4.2 Notação Quantum

Para construir um caminho pela fórmula CNF de uma dada teoria, o Algoritmo de Transformação Dual precisa saber em que cláusulas de tal fórmula cada literal aparece. Para tanto, o mesmo introduz uma notação especial, chamada notação Quantum. Nesta notação, encontra-se a definição de Quantum [Bittencourt; Marchi; Padilha, 2003].

Um quantum é um par (φ, F) onde φ é um literal, pertencente ao conjunto de literais que aparecem em Wc, e F é um conjunto de coordenadas para as cláusulas de

Wc onde φ aparece [Bittencourt; Marchi; Padilha, 2003]. Um quantum pode também ser representado como φ^F. O coletivo de quantum (um conjunto deles), chama-se *quanta*.

O *mirror* de um quantum (φ, F) é o quantum associado ao literal ¬φ e pode ser denotado φ^F.

Os quanta associados a teoria **K** são: {$P_1^{\{2\}}$, $\neg P_1^{\{4,5,6,8,9\}}$, $P_2^{\{0,5,7\}}$, $\neg P_2^{\{1,3,4,6\}}$, $P_3^{\{3,8\}}$, $\neg P_3^{\{0,1,2,4,5,9\}}$, $P_4^{\{7,8\}}$, $\neg P_4^{\{0,9\}}$, $P_5^{\{1,2,6,7\}}$, $\neg P_5^{\{3\}}$ }.

4.3 Busca

Conhecida a notação Quantum, encontrar as cláusulas duais mínimas para uma dada teoria **K** a partir de sua representação CNF pode ser encarado como um problema de busca em um epaço de estados, em que cado estado corresponde a um caminho para uma possível cláusula dual mínima.

Cada estado de busca Φ, de fato, representa um conjunto incompleto de quanta associados a uma cláusula dual mínima incompleta. Além disso cada estado tem a ele associado um conjunto de quanta proibidos X_Φ e um Gap G_Φ, definido como o conjunto de cláusulas da teoria em CNF nas quais nenhum dos literais associados aos quanta de Φ aparece. O conjunto de literais associados aos quanta de um estado Φ é denotado L_Φ.

Para resolver a busca, o algoritmo A* é usado. Este algoritmo precisa de três funções de interface para acessar o problema: Uma para definir os estados iniciais da busca, uma para saber quem são os estados vizinhos de um dado estado e outra para saber quais estados são finais [Russel; Norvig, 2010].

5. Melhoramentos ao algoritmo de transformação Dual

Um olhar atento sobre as condições de corte nos caminhos de busca do Algoritmo de Transformação Dual, permite identificar algumas possibilidades de melhoria, principalmente ligadas às restrições impostas sobre quais quanta podem ser incluídos em um dado estado de busca para gerar um novo, em função dos já existentes e da lista de quanta proibidos. Tais melhorias, por diminuirem o tamanho do espaço de estados, tem impacto tanto sobre a quantidade de memória necessária para armazenar a busca quanto no número de ciclos de processamento necessários para concluir a busca. Três destas possíveis melhorias foram implementadas, testadas e demonstraram-se efetivas no escopo deste trabalho. As descrições das três melhorias implementadas encontram-se a seguir.

5.1 Corte prematuro

Durante a geração de estados sucessores, os estados gerados a partir de Φ tem uma série de quanta adicionados à sua lista de quanta proibidos por conta das restrições descritas no algoritmo de geração de estados sucessores. Após a adição destes quanta, a lista de quanta proibidos XΦ+ possui valiosas informações que podem dizer, de antemão, se este estado tem ou não chances de gerar estados sucessores. Sabendo que um dado estado Φ+ $ não gerará sucessores, podemos removê-lo da lista de estados sucessores e cortar prematuramente, caminhos infrutíferos.

Para tanto, precisamos calcular as *Gap Condition*s de Φ+. Estas *Gap Conditions* dizem com que quanta as cláusulas do gap podem ser cobertas sem quebrar as restrições. Se a teoria CNF que as representa contiver a cláusula vazia, ou duas cláusulas unitárias contraditórias, sabemos que este estado Φ+ é infrutífero e podemos descartá-lo.

5.2 Expansão de Gap Conditions

A ideia geral por trás do cálculo de *Gap Conditions* é expressar, na forma de uma teoria em CNF, através de quais quanta cada cláusula do gap de um determinado estado Φ pode ser "coberta", ou seja, quais quanta adicionados a Φ farão cada cláusula sair de seu *Gap*. Sabemos que esta teoria é gerada segundo a equação:

$$R_\Phi = \{C - \overline{L_\Phi} \mid C \in G_\Phi \; e \; C \cap \overline{L_\Phi} \neq \emptyset\}$$

A ideia da construção desta teoria é a de que se uma determinada cláusula de Gap tem intersecção com a lista de mirror quanta, nós devemos remover da mesma os literais da intersecção pois estes por pertencerem a lista de mirror quanta (negação dos quanta da fórmula) não podem estar presentes em Φ e portanto não será através deles que esta cláusula será "coberta". Sobram portanto, nas cláusuas de do Gao de Φ aqueles literais que podem estar em Φ.

A melhoria proposta neste cálculo é simples e toma vantagem dos próprios mecanismos introduzidos pela representação Quantum. A ideia é remover das cláusulas do Gap de Φ não apenas os literais da lista de mirror quanta, mas todos aqueles pertencentes ao conjunto de literais proíbidos em Φ, uma vez que a lista de mirror quanta é um subconjunto desta última e ainda contém quanta proibidos por outras razões. O novo cálculo de *Gap Conditions* seria:

$$R_\Phi = \{C - X_\Phi \mid C \in G_\Phi \, e \, C \cap X_\Phi \neq \emptyset\}$$

5.3 Gap Conditions sem intersecção

A ideia por trás desta melhoria é a de eliminar a intersecção do cálculo de quanta proíbidos pois esta é uma das operações mais custosas, e a eliminação desta operação não altera o valor semântico da lista de proibidos. A fórmula, sem a intersecção, fica:

$$R_\Phi = \{C - X_\Phi \mid C \in G_\Phi \}$$

6. Teste e análise dos resultados

Para tomar ciência da dimensão do melhoramento ocasionado pelas melhorias propostas, o Algoritmo de transformação Dual foi testado em versões com e sem as mesmas. Neste capítulo apresentam-se apenas os resultados dos testes para a versão sem melhorias e a versão com todas as três melhorias.

Um conjunto de teorias de teste padrão conhecido como SATLIB, disponível em *http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html*, foi usado para os testes. Os pacotes de teorias selecionados contém instâncias SAT em CNF com 3 literais randômicos por cláusula, e são eles:

- **uf20-91**: teorias com 20 variáveis e 91 cláusulas, todas satisfazíveis.
- **uf50-218**: teorias com 50 variáveis e 218 cláusulas, todas satisfazíveis.
- **uf75-325**: teorias com 75 variáveis e 325 cláusulas, todas satisfazíveis.
- **uf100-430** : teorias com 100 variáveis e 430 cláusulas, todas satisfazíveis.

Estes mesmos pacotes e as teorias neles contidas são usados por vários outros trabalhos relacionados a pesquisa em SAT [Hoos; Stutzle, 2000]. Alguns testes também foram realizados para instâncias não satisfazíveis, porém, apresentaremos aqui apenas os resultados para instâncias satisfazíveis.

Todos os testes foram executados em uma máquina com processador Intel Core i5-2410M com 2,3GHz e 2 GiB. Alguns dos resultados obtidos são apresentados nas tabelas 2,3,4 e 5 onde “Tempo total” é o tempo total em centésimos de segundo (0,01s), “Chamadas” é o número de iterações sobre o *loop* principal do algoritmo A*, “Max. Mem.” é o máximo de memória alocada durante a execução, em MB, e |Wd | é o número de cláusulas duais mínimas geradas.

Teoria	Tempo total		Chamadas		Max. mem.		$\|W_d\|$
	Sem	Com	Sem	Com	Sem	Com	
uf20-0110	69	46	2212	204	20	9	21
uf20-0111	24	19	1583	95	39	9	5
uf20-0112	23	11	1415	62	65	9	3
uf20-0113	32	19	1893	78	77	9	1
uf20-0114	29	13	1707	93	73	8	6
uf20-0115	32	21	2107	97	94	8	4
uf20-0116	22	14	1402	60	67	7	2
uf20-0117	30	17	1916	77	88	7	1
uf20-0118	31	17	2152	188	79	6	14
uf20-0119	21	17	1378	71	67	6	2

Tabela 2: Pacote de teste uf20-91

Teoria	Tempo total		Chamadas		Max. mem.		$\|W_d\|$
	Sem	Com	Sem	Com	Sem	Com	
uf50-0110	18916	1644	137668	1311	495	38	27
uf50-0111	17108	1356	131643	592	488	33	1
uf50-0112	13587	1221	94016	564	384	26	4
uf50-0113	13983	1141	92886	630	389	34	9
uf50-0114	35614	4139	266851	5631	919	52	244
uf50-0115	14559	1345	111694	1608	574	19	40
uf50-0116	14759	1302	110290	795	531	27	15
uf50-0117	33357	3489	268012	4301	918	38	191
uf50-0118	9549	793	69720	449	413	17	7
uf50-0119	19787	2013	141276	1435	597	29	24

Tabela 3: Pacote de teste uf50-218

Teoria	Tempo total		Chamadas		Max. mem.		$\|W_d\|$
	Sem	Com	Sem	Com	Sem	Com	
uf75-010	?	84188	?	41917	?	237	1025
uf75-011	?	38921	?	6027	?	71	9
uf75-012	?	46518	?	5985	?	87	2
uf75-013	?	25052	?	6545	?	92	132
uf75-014	?	33056	?	8362	?	109	157
uf75-015	?	50720	?	7783	?	87	16
uf75-016	?	134676	?	2126	?	63	6
uf75-017	?	55576	?	15246	?	131	177
uf75-018	?	38742	?	18579	?	161	925
uf75-019	?	15098	?	2136	?	63	2

Tabela 4: Pacote de teste uf75-325

Teoria	Tempo total		Chamadas		Max. mem.		$\|W_d\|$
	Sem	Com	Sem	Com	Sem	Com	
uf100-0110	?	512070	?	30832	?	174	18
uf100-0111	?	494565	?	33024	?	167	140
uf100-0112	?	109984	?	8604	?	199	46
uf100-0113	?	747005	?	45384	?	171	3
uf100-0114	?	415859	?	23202	?	158	2
uf100-0115	?	182862	?	19507	?	161	424
uf100-0116	?	588479	?	40536	?	168	66
uf100-0117	?	501414	?	33280	?	188	47
uf100-0118	?	283759	?	28180	?	178	433
uf100-0119	?	582873	?	90136	?	512	1835

Tabela 5: Pacote de teste uf100-430

A versão sem melhorias do algoritmo excedeu o limite de memória disponível (2GiB) quando executada com os pacotes de teste “uf75-325” e “uf100-430”. Por esse motivo, os resultados para estes testes estão marcados com “?”.

Os resultados obtidos para os pacotes “uf20-91” e “uf50-218” estão descritos nas tabelas 2 e 3, e indicam que as três melhorias propostas contribuem positivamente para a redução do espaço de busca e, consequentemente, da memória necessária e do tempo total de busca. Verifica-se isto através da redução média de 64\% do tempo total de busca e de 89\% na quantidade de memória necessária para a resolução de cada teoria destes pacotes. Os resultados alcançados com os pacotes “uf75-325” e “uf100-

430”, que são exibidos nas tabelas 4 e 5, também indicam uma melhora significativa no algoritmo. Para estes resutados, não é possível calcular diretamente a média das reduções de tempo total e de quantidade de memória necessária uma vez que, sem as melhorias propostas, o algoritmo nem mesmo conseguia resolver as teorias destes pacotes. Este resultado, então, reforça a contribuição das melhorias propostas.

7. Conclusões

Através da comparação das versões com e sem melhorias do Algoritmo, verificou-se que as três melhorias propostas contribuiram significativamente para a redução do tempo total de execução do Algoritmo e também para a redução da quantidade de memória utilizada. Em especial a primeira melhoria apresentada reduz significativamente o espaço de busca, resultando em um menor número de chamadas a função de busca. Quando as três melhorias são aplicadas em conjunto, contudo, o desempenho se torna melhor. Apesar de os testes terem sido realizados com um conjunto limitado de teorias, os resultados parecem promissores.

A implementação em Java foi de grande ajuda para simplificar os esforços de implementação e garantir mais foco ao problema em si. Implementações anteriores do Algoritmo haviam sido feitas em LISP e C++. Comparações entre as versões anteriores de implementações e a versão implementada neste trabalho não foram feitas pela diferença de linguagens. Neste caso, implementar o Algoritmo de Transformação Dual com melhorias em Lisp seria uma alternativa melhor. No entanto, a implementação feita em Java sem os melhoramentos segue estritamente as definições conceituais da descrição do algoritmo e portanto, a comparação feita, entre esta versão e a com melhorias, é valida.

Existem ainda, duas possibilidades de melhorias que, embora tenham surgido dentro deste trabalho e estejam perfeitamente alinhadas com os objetivos aqui perseguidos, não foram implementadas por falta de tempo hábil. Ficam aqui então, como sugestões para trabalhos futuros:

- **Verificação completa da satisfazibilidade de gapConditions:** As melhorias apresentadas nas seções 5.1 e 5.2 envolvem ambas o cálculo de *Gap Conditions*. Tal cálculo tem como resultado uma teoria da lógica proposicional em CNF (subconjunto da teoria de entrada do algoritmo). Da maneira como o algoritmo está descrito, as três condições relativas as *Gap Conditions* são verificadas individualmente. Porém, como as *Gap Conditions* estão descritas como uma teoria, a mesma poderia ser verificada através da checagem de sua satisfazibilidade. A ideia por trás disso é que se a teoria que descreve as *Gap conditions* for satisfazível nenhumas das três condições estava presente.
- **Paralelização:** Através de alguns mecanismos descritos pelo Algoritmo de transformação Dual, mas principalmente pelo critério de qualidade verifica-se que a busca por estados vizinhos e por modelos, a cada novo passo de execução do algoritmo A* é totalmente disjunta. Isso significa que a partir da escolha dos estados iniciais do algoritmo, a busca poderia ser executada em paralelo, para todos os ramos ao mesmo tempo. Esta técnica poderia ser aplicada recursivamente dentro do espaço de busca, onde cada passo geraria uma série de

estados vizinhos que poderiam ser verificados paralelamente. Esta proposta de melhoria não é teórica, apenas toma vantagem de características do Algoritmo para tentar uma implementação que resulte em um menor tempo de processamento.

Referências

BITTENCOURT, G.; MARCHI, J.; PADILHA, R. S. A syntatical approach to satisfaction. In: UNIVERSITY OF LIVERPOOL AND UNIVERSITY OF MANCHESTER. 4th International Workshop on the Implementation of Logic (Konev, B. and Schimidt, R. eds.). Almaty - Kazakhstan, 2003. p. 18–32.

BOAVA, G. Algoritmos para Satisfazibilidade: Implementacao e Testes. Dissertação (TCC) — Universidade Federal de Santa Catarina, 2005.

CRAWFORD, J. M.; AUTON, L. D. Experimental results on the crossover point in random 3sat. Artificial Intelligence, v. 81, p. 31–57, 1996.

DARWICHE, A.; PIPATSRISAWAT, K. Complete algorithms. In: [S.l.]: IOS Press, 2009. (Frontiers in Artificial Intelligence and Applications, v. 185), cap. 3, p. 99–130. ISBN 978-1-58603-929-5.

DAVIS, M.; LOGEMANN, G.; LOVELAND, D. A machine program for theorem-proving. Commun. ACM, ACM, New York, NY, USA, v. 5, n. 7, p. 394–397, jul. 1962. ISSN 0001-0782. <http://doi.acm.org/10.1145/368273.368557>.

DAVIS, M.; PUTNAM, H. A computing procedure for quantification theory. J. ACM, ACM, New York, NY, USA, v. 7, n. 3, p. 201–215, jul. 1960. ISSN 0004-5411. <http://doi.acm.org/10.1145/321033.321034>.

DIMACS-CHALLENGE. Satisfiability: Suggested Format. [S.l.], Maio 1993. <http://www.cs.ubc.ca/ hoos/SATLIB/Benchmarks/SAT/satformat.ps>.

HOOS, H. H.; STUTZLE, T. Satlib: An online resource for research on sat. In: . [S.l.]: IOS Press, 2000. p. 283–292.

MARI , F. Formalization and implementation of modern sat solvers. J. Autom. Reason., Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 43, n. 1, p. 81–119, jun. 2009. ISSN 0168-7433. <http://dx.doi.org/10.1007/s10817-009-9127-8>.

RAUBER, P. E. Análise da solucao do problema do caminho Hamiltoniano Através de Redução para Problema da Satisfazibilidade Booleana. Dissertção (TCC) — Universidade Federal de Santa Catarina, 2011.

ROSEN, K. H. Handbook of Discrete and Combinatorial Mathematics, Second Edition. 2nd. ed. [S.l.]: Chapman & Hall/CRC, 2010. ISBN 158488780X, 9781584887805.

RUSSELL, S. J.; NORVIG, P. Artificial Intelligence - A Modern Approach (3. internat. ed.). [S.l.]: Pearson Education, 2010. I-XVIII, 1-1132 p. ISBN 978-0-13-207148-2.

SATLIB. SATLIB - Submission Guidelines. Dezembro 1999. Internet. <http://www.cs.ubc.ca/ hoos/SATLIB/submission.html>.

SIPSER, M. Introduction to the Theory of Computation. 1st. ed. [S.l.]: International Thomson Publishing, 1996. ISBN 053494728X.

VORONKOV, A. A. Logic and modeling 2011. Outubro 2012. <http://www.voronkov.com/lics.cgi>.