ROMANELLI LODRON ZUIM

UMA HEURÍSTICA DE DECISÃO BASEADA NA SUBTRAÇÃO DE CUBOS PARA SOLUCIONADORES DPLL DO PROBLEMA DA SATISFABILIDADE

Belo Horizonte
22 de novembro de 2007

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

UMA HEURÍSTICA DE DECISÃO BASEADA NA SUBTRAÇÃO DE CUBOS PARA SOLUCIONADORES DPLL DO PROBLEMA DA SATISFABILIDADE

Tese apresentada ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

ROMANELLI LODRON ZUIM

Belo Horizonte

22 de novembro de 2007

UNIVERSIDADE FEDERAL DE MINAS GERAIS

FOLHA DE APROVAÇÃO

Uma heurística de decisão baseada na subtração de cubos para solucionadores DPLL do problema da satisfabilidade

ROMANELLI LODRON ZUIM

Tese defendida e aprovada pela banca examinadora constituída por:

Ph. D. Claudionor Nunes Coelho Júnior – Orientador
Universidade Federal de Minas Gerais

Ph. D. José João Henriques Teixeira de Sousa – Co-orientador
INESC-ID - Lisboa, Instituto Superior Técnico-IST Lisboa

Ph. D. Henrique Pacca Loureiro Luna
Universidade Federal de Alagoas

Ph. D. Marco Túlio de Oliveira Valente
Pontifícia Universidade Católica de Minas Gerais

Ph. D. Antônio Otávio Fernandes
Universidade Federal de Minas Gerais

Ph. D. Newton José Vieira
Universidade Federal de Minas Gerais

Belo Horizonte, 22 de novembro de 2007

Resumo Estendido

Este trabalho propõe uma nova heurística de decisão para solucionadores do problema da satisfabilidade (SAT) baseados no algoritmo de Davis Putnam, Logemann e Loveland (DPLL). Essa heurística se baseia na subtração de cubos. Cada cláusula negada é visualizada como um cubo no espaço de procura booleano n-dimensional, denotando um subespaço onde nenhuma atribuição de valores às variáveis, que satisfaça a instância do problema, possa ser encontrada. A subtração dos cubos, sistematicamente subtrai todas as cláusulas-cubo do cubo universal, que representa todo o espaço booleano de procura. Se o resultado for um cubo vazio o problema não admite uma solução que o torne satisfazível, caso contrário, o problema é satisfazível. Esse algoritmo pode ser implementado modificando-se o mecanismo de decisão de solucionadores do problema da satisfabilidade baseados no algoritmo DPLL. Essas modificações restringem a escolha da próxima variável de decisão após um retrocesso cronológico. A intuição do algoritmo é confinar a procura a uma cláusula ou a um grupo de cláusulas, com o objetivo de escapar o mais rapidamente possível de regiões onde a solução não possa ser encontrada, ou uma resposta "sim" ou "não" possa ser dada para a instância, ou permitir o aprendizado de cláusulas mais úteis à solução do problema. Inicialmente foram utilizadas duas versões básicas de um solucionador DPLL, sem quaisquer das melhorias atualmente encontradas na literatura e posteriormente em um solucionador no estado-da-arte, o zChaff. Para o teste foram utilizados 1252 instâncias de problemas do DIMACS, IBM CNF BMC e resultados de verificação formal de microprocessadores. Observa-se uma melhoria significativa no tempo de execução e uma redução do

número de instâncias não resolvidas dentro de um tempo limite, em todos os casos. Considerando a avaliação experimental, podemos concluir que a subtração de cubos é um algoritmo efetivo para a melhoria do desempenho de solucionadores do problema da satisfabilidade baseados no algoritmo DPLL.

Abstract

This work proposes a new decision heuristic for Davis Putnam, Loveland and Logemann algorithm (DPLL)-based satisfiability (SAT) solvers based on cube subtraction. Each negated clause is viewed as a cube in the n-dimensional Boolean search space denoting a subspace where no satisfying assignments can be found. Cube subtraction, systematically subtracts all clause-cubes from the universal cube that represents the entire n-dimensional Boolean search space. If the result is an empty cube, then the problem is unsatisfiable; else the problem is satisfiable. This algorithm can be implemented by modifying the decision engine of a DPLL-based SAT solver. This modification restricts the choice of the next decision variable after a chronological backtrack step. Intuitively, the idea is to confine the search to a clause or a group of clauses, in the hope of getting out of non-solution regions, or regions where we can not find a "yes" or "no" answer to the instance, faster and/or learning more useful clauses. This idea is implemented in the well-known zChaff solver. The test suite includes 1252 instances from the DIMACs, IBM-CNF bounded model checking and microprocessor formal verification benchmarks. Significant improvements in execution time and number of timed-out instances have been observed in all cases. Given the breadth of the experimental evaluation, the disjoint cube subtraction search is claimed to be an effective algorithm for improving the performance of DPLL-based SAT solvers.

Aos meus pais, José e Leila, à minha esposa Elenice e ao Lucca.

Agradecimentos

Agradeço inicialmente a Deus, por me ensinar a realizar todas as atividades com paciência, tolerância e força de vontade.

À Universidade Federal de Minas Gerais, especialmente ao Programa de Pós-Graduação do Departamento de Ciência da Computação, pela oportunidade de realização do doutorado e por compreender e permitir minhas inúmeras solicitações.

À Pontifícia Universidade Católica de Minas Gerais, que me ajudou financeiramente nos três primeiros anos e permitiu o meu afastamento em 2005.

Ao INESC-ID Lisboa, em particular ao professor José Henriques Teixeira de Sousa, que me recebeu e ajudou de uma forma indescritível, acho que sem a sua ajuda eu não teria concluído essa tese e nem publicado os trabalhos.

Ao professor Newton José Vieira, que pacientemente muito me auxiliou no texto.

Ao meu orientador professor Claudionor Nunes Coelho Júnior, que me aceitou como seu orientando, mesmo me conhecendo.

Aos professores do Programa de Pós-Graduação em Ciência da Computação da UFMG, pelo profissionalismo apresentado na condução das disciplinas do curso.

Aos funcionários da secretaria do Programa de Pós-Graduação, pelo suporte durante toda a minha permanência no curso.

A todos aqueles que, de forma direta ou indireta, contribuíram para construção e conclusão desse trabalho.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

No processo de se atingir alta produtividade na utilização dos computadores, diversos problemas computacionais foram identificados como centrais em ciência da computação. Um desses problemas é o da satisfabilidade. Instâncias desse problema são relativamente simples de serem formuladas mediante codificação em lógica proposicional. Mas elas são, em muitos casos, de difícil resolução. Fundamentalmente, o problema da satisfabilidade é conhecido pela sua importância teórica, já que foi o primeiro problema a ser provado como NP-Completo (por Cook em 1971 [Coo71]). Nesta última década, a pesquisa em satisfabilidade tem sido intensa, em função, principalmente, das mais variadas áreas onde o problema pode ser aplicado: engenharia, ciência da computação, biologia, entretenimento e muitas outras. Os solucionadores mais recentes do problema da satisfabilidade são capazes de solucionar instâncias do problema com centenas de milhares de variáveis e milhões de restrições.

1.1 O que é o problema da satisfabilidade

Para definirmos o problema da satisfabilidade, algumas considerações iniciais são necessárias. Seja $x_1, x_2, \ldots, x_n$ um conjunto de variáveis proposicionais que podem assumir, cada uma, o valor "verdadeiro" ou "falso". Fórmulas, também chamadas de expressões booleanas, são construídas com variáveis proposicionais e os operadores $\wedge$ (e), $\vee$ (ou) e

$\neg$ (não).[1] Informalmente, afirmar a veracidade de uma fórmula $x_1 \vee x_2$ significa afirmar que x_1 ou x_2, ou ambas, são verdadeiras, enquanto que afirmar que a fórmula $x_1 \wedge x_2$ é verdadeira significa dizer que ambas as variáveis são verdadeiras; já dizer que $\neg x_1$ é verdadeira, significa que x_1 deve ser falsa. Um literal é definido como uma variável proposicional, x_i, ou a sua negação, $\neg x_i$. Definimos a disjunção dos literais $x_1, x_2, \ldots, x_n$ como sendo $x_1 \vee x_2 \vee \ldots \vee x_n$, ao que damos o nome de cláusula; assim, n é o número de literais da cláusula. O comprimento de uma cláusula é a quantidade de literais diferentes da mesma. Uma cláusula com um só literal é uma cláusula unitária. Uma fórmula na forma normal conjuntiva (FNC) é uma conjunção de cláusulas. Assim, se cada w_i for uma cláusula, a conjunção $w_1 \wedge w_2 \wedge \ldots \wedge w_m$ é uma fórmula na FNC; neste caso, m é a quantidade de cláusulas.

Se existir uma atribuição para as variáveis proposicionais que torne uma fórmula verdadeira, diz-se que a fórmula é satisfazível (é SAT). Se tal atribuição não existir, diz-se que a fórmula é insatisfazível (não-SAT). Podemos agora definir o que é o problema da satisfabilidade.

O problema da satisfabilidade (problema SAT) consiste em determinar se uma fórmula na FNC é satisfazível ou não.

O problema SAT pertence à classe dos denominados problemas de decisão, isto é, problemas que admitem dois tipos de solução: sim ou não. Uma instância do problema SAT é dada por um conjunto de cláusulas específicas; o termo instância é um neologismo do inglês que, no contexto, significa um exemplo, uma amostra.

Um algoritmo resolve um determinado problema de decisão se, ao receber qualquer das instâncias do problema, devolve uma solução correta que, no caso, é "sim" ou "não". No caso do problema SAT, o algoritmo deve devolver "sim" se a instância for dada por um conjunto de cláusulas satisfazível e "não" se for dada por um conjunto de cláusulas insatisfazível.

Um solucionador para o problema SAT, tipicamente tenta encontrar uma atribui-

[1] Outros operadores, como $\rightarrow$ (implica), podem ser definidos a partir desses.

ção para as variáveis da instância do problema que tornem o conjunto de cláusulas respectivo satisfazível. Encontrando tal atribuição, ele devolve "sim". Determinando que tal atribuição não existe, ele devolve "não".

O problema SAT tem uma longa história, assim como os solucionadores desenvolvidos. O algoritmo de Davis e Putnam [Dav60], publicado em 1960, é tipicamente citado como a primeira proposta real de solução para o problema de satisfabilidade. Um dos problemas do algoritmo de Davis-Putnam é que, se o número de variáveis aumenta, existe o potencial de se produzir um crescimento exponencial da fórmula, resultando em um consumo explosivo de memória. Por esta razão, uma variação desse algoritmo inicial foi desenvolvida por Davis, Logemann e Loveland [Dav62], que é preferencialmente utilizada, em que o espaço adicional criado durante a resolução cresce de forma linear com o número de variáveis. A sigla referenciando esse algoritmo em todo este trabalho será DPLL, que traz claramente a real origem do algoritmo.

1.1.1 Um exemplo simples

Para dar uma noção de como um problema pode ser formulado como uma instância de um problema SAT, apresentaremos um exemplo simples. Nesse exemplo, nós temos um grupo de professores que pretendem montar cursos preparatórios para um exame final de curso. Foram identificadas quatro disciplinas como as que mais trazem problemas: Matemática, Português, Ciências e História. Existem diversos professores de cada uma destas disciplinas. O objetivo é dividir esses professores em vários grupos de estudos menores para montar diversas atividades para os alunos. A divisão dos professores nos respectivos grupos de estudos deverá satisfazer certas restrições. Sejam os conjuntos de professores $M = \{m_1, m_2, \ldots\}$, de matemática, $P = \{p_1, p_2, \ldots\}$, de português, $C = \{c_1, c_2, \ldots\}$, de ciências e $H = \{h_1, h_2, \ldots\}$, de história. Seja $G = \{g_1, g_2, \ldots\}$ o conjunto dos grupos de estudos. Vamos introduzir uma variável $x_{p,g}$ para indicar que o professor p faz parte do grupo g.

A seguir, algumas restrições que podemos encontrar nesta situação, cada uma ex-

pressa por um tipo de cláusula:[2]

1. Cada professor p é membro de pelo menos um grupo de estudos:

$$x_{p,g_1} \vee x_{p,g_2} \vee \ldots \vee x_{p,g_{|G|}}$$

São, então, $|M| + |P| + |C| + |H|$ cláusulas, no máximo, cada uma com $|G|$ literais.

2. Um professor p não pode estar em dois grupos de estudo g e g', $g \neq g'$:

$$\neg x_{p,g} \vee \neg x_{p,g'}$$

São $(|M| + |P| + |C| + |H|)|G|(|G| - 1)/2$ cláusulas, no máximo, cada uma com dois literais.

3. Em cada grupo g há pelo menos um professor de matemática:

$$x_{m_1,g} \vee x_{m_2,g} \vee \ldots \vee x_{m_{|M|},g}$$

São $|G|$ cláusulas, cada uma com $|M|$ literais.

4. Se existe um professor de português p no grupo g, deve existir no mínimo um professor de história no grupo g:

$$\neg x_{p,g} \vee x_{h_1,g} \vee x_{h_2,g} \vee \ldots \vee x_{h_{|H|},g}$$

São $|P||G|$ cláusulas, cada uma com $|H|$ literais.

Da mesma forma como adicionamos essas quatro restrições para a composição dos grupos de estudo, poderíamos ter adicionado dezenas, centenas ou a quantidade necessária para melhor modelar o problema.

A instância do problema de satisfabilidade aqui é a de determinar se existe uma divisão de professores nos grupos de estudos que satisfaz às restrições impostas. De uma maneira mais resumida, se existe uma atribuição para as variáveis $x_{p,g}$, $p \in M \cup$

[2] O número de elementos do conjunto X é aqui denotado por $|X|$.

$P \cup C \cup H$ e $g \in G$, que satisfaça todas as cláusulas.

O problema SAT restrito a cláusulas com no máximo k literais é dito k-SAT.

1.1.2 Importância e aplicações do problema da satisfabilidade

O problema da satisfabilidade encontra uma vasta gama de aplicações nas mais diferentes áreas da ciência da computação, engenharia, planejamento, inteligência artificial, dentre outras. Alguns exemplos são aqui destacados que motivam o estudo e a pesquisa nesta área.

Teoria da Complexidade

A prova de Cook [Coo71], que problemas da classe k-SAT onde $k \geq 3$ são NP-completos, pode ser utilizada como uma base teórica para a análise da complexidade.

A complexidade de um problema é dada pelo consumo de tempo (ou memória) de um algoritmo ótimo para o mesmo. Assim, a complexidade é uma medida da dificuldade computacional intrínseca do problema. Um problema é polinomial se existe um algoritmo polinomial para o mesmo e é não-polinomial se não existe algoritmo polinomial que o resolva. Problemas não-polinomiais são considerados computacionalmente intratáveis.

A classe P de problemas é o conjunto de todos os problemas de decisão polinomiais, ou seja, o conjunto dos problemas de decisão que podem ser resolvidos por algoritmos polinomiais. Verificou-se recentemente que o "problema do número primo"está em P [Agr04].

A classe NP de problemas[3] é o conjunto de todos os problemas de decisão para os quais existe um verificador polinomial. Ou seja, um problema está em NP se toda instância positiva do problema admite um certificado que é aceito por um verificador polinomial. Um algoritmo verificador para um problema de decisão é aquele que recebe dois objetos: uma instância do problema e um certificado. Podemos entender o certifi-

[3]Em inglês, *nondeterministic polynomial.*

cado como uma "prova"de que a instância é positiva (ou verdadeira). Ao receber esses dois objetos, o verificador pode responder sim, responder não, ou não parar. O verificador é polinomial se para cada instância positiva do problema, existe um certificado que leva o verificador a responder sim em tempo limitado por uma função polinomial do tamanho da instância e para cada instância negativa do problema, não existe certificado que leve o verificador a responder sim. Para uma instância de um problema da satisfabilidade satisfazível, existe este certificado.

A classe NP contém P, ou seja, todo problema polinomial está em NP. Intuitivamente P é apenas uma pequena parte de NP, mas ninguém conseguiu ainda exibir um problema de NP que não esteja em P, isto é, um problema de NP para o qual não existe um algoritmo polinomial. Essa situação abre caminho para a suspeita de que P seja, talvez, igual a NP. Mas a maioria dos especialistas não acredita nessa possibilidade [Url01].

Um problema de decisão A é completo em NP (ou NP-completo) se A está em NP e qualquer outro problema em NP pode ser polinomialmente reduzido a A. Um problema de decisão é redutível a outro problema de decisão se existe um algoritmo que transforma qualquer instância x do primeiro em uma instância y do segundo de tal forma que a resposta para y fornece a resposta para x. Em outras palavras, um problema é redutível a A se for um "subproblema", ou "caso particular", de A.

Com a prova de Cook (que o problema SAT é NP-completo) e a noção da redutibilidade, podemos determinar se um problema é NP-completo através do seguinte resultado: um problema X é NP-completo se (1) X pertence a NP, e (2) Y é redutível polinomialmente a X, para algum problema Y que é NP-Completo.

Como existe redução polinomial de todo problema em NP a um problema NP-completo, se fosse encontrado um algoritmo polinomial para resolver o problema da satisfabilidade, ter-se-ia provado que P = NP.

Garey e Johnson apresentam inúmeros problemas NP-completos em [Gar79].

Automação de projetos eletrônicos de sistemas digitais (*EDA*)[4]

Duas importantes áreas da automação de projetos digitais são a de otimização e a de validação. A otimização procura dirigir os esforços para a construção de circuitos que sejam rápidos, pequenos e eficientes em termos de energia. Na validação procura-se circuitos que funcionem corretamente. A técnica de raciocínio lógico automatizado[5] é o elemento central nestas atividades. Métodos de raciocínio lógico têm sido largamente utilizados em diversas áreas no processo de automação de projetos, tais como otimização lógica, geração de padrões de teste, verificação formal e simulação funcional. Tendo em vista a sua importância, muitas técnicas tem sido estudadas e publicadas. Encontramos diversos trabalhos referentes a diagramas de decisão binários (BDD) [Bry92] e aprendizado recursivo [Kun94]. Dentre os métodos existentes, um dos mais importantes e que vem sendo cada vez mais utilizado tendo em vista os avanços atualmente alcançados é a utilização de solucionadores de instâncias de problemas de satisfabilidade.

Resolvendo o problema SAT, ferramentas automáticas podem implementar raciocínio lógico em fórmulas booleanas e em circuitos digitais. A satisfabilidade proposicional booleana (SAT) já tem sido largamente utilizada pela comunidade EDA e ganhado cada vez mais espaço tendo em vista os recentes avanços da comunidade SAT.

A quantidade de situações e problemas que podem ser reduzidos ao problema SAT em EDA mostra a sua importância. Aplicações como geração automática de padrões de teste (ATPG) [Lar92] [Ste96], verificação de equivalência em circuitos combinacionais [Sil99a], verificação de microprocessadores [Vel01], análise de alcance[6] [Gup00] [Abd00], verificação de modelos[7] [Bie99] [Mil02], remoção de redundâncias [Sil97], análises temporais [Sil94] e roteamento [Nam99] exemplificam alguns destes problemas.

Observa-se, empiricamente, que instâncias SAT geradas a partir de problemas EDA, ou mais especificamente problemas de origem industrial, recaem numa classe de proble-

[4] Em inglês, *Electronic Design Automation.*

[5] Em inglês, *automated logic reasoning.*

[6] Em inglês, *reachability analysis.*

[7] Em inglês, *model checking.*

mas relativamente "fáceis" [Pra96]. Contrariamente, problemas aleatoriamente gerados se mostram de extrema dificuldade [Mit92] [Sel97] [Min00] [Dub01].

Mesmo assim, aplicações reais podem ser consideradas complexas, pois ferramentas EDA em geral lidam com projetos em que encontramos milhares, ou mesmo milhões, de portas lógicas. Instâncias SAT geradas a partir desses projetos contêm a mesma proporção de variáveis. Tal situação faz com que grandes esforços sejam necessários para a construção de solucionadores SAT suficientemente eficientes para solucionar esses tipos de problemas.

Na verdade, ferramentas comerciais de verificação normalmente gastam a maior parte do tempo de execução na resolução SAT, que pode facilmente atingir horas ou mesmo dias de execução. Tornar um solucionador SAT o mais eficiente possível é hoje de extrema importância prática, já que a verificação tem se tornado a parte que mais tempo consome no projeto de circuitos integrados.

Arquiteturas reconfiguráveis

Apesar do conceito de computação reconfigurável ter sido proposto no início de 1960, só recentemente a tecnologia permitiu a sua utilização prática, quando as densidades de portas lógicas das FPGAs romperam a barreira das 10k portas [Guc01]. Desde então, a computação reconfigurável tem sido alvo de uma intensa pesquisa [Skl04a].

Para algumas classes de aplicações, os sistemas reconfiguráveis mostraram um desempenho similar aos dos processadores de uso geral. Para outros tipos de aplicações o mapeamento em arquiteturas reconfiguráveis ofereceu novas oportunidades a serem exploradas. Essas aplicações podem ser divididas, basicamente, em três categorias: emulação e prototipação rápida em hardware, algoritmos evolutivos em hardware e aceleração de tarefas computacionalmente intensivas. A última categoria prevaleu até o momento atual. Mais recentemente, tentativas foram feitas para a aceleração de aplicações que envolvessem um fluxo de controle mais complexo. Neste contexto se encontram certos problemas de otimização combinatória e particularmente o problema

SAT.

Diversos grupos têm explorado diferentes abordagens para a resolução do problema SAT com o auxílio de computação reconfigurável. Skliarova e Ferrari apresentam uma coletânea de arquiteturas em [Skl04b], que podem ser vistas em [Yok96] [Suy01] [Zho99] [Men99] [Pla98] [Abr97] [Abr00] [Dan02] [Red00] [Chu99] [Yun99] [Leo04] [Sou01] [Rei02] [Rip01] [Skl02] [Yap03] [Ham97] [Ras98] [Boy00].

Planejamento e escalonamento[8]

A utilização de métodos de satisfabilidade em problemas de planejamento e escalonamento tem recebido uma grande atenção. Podemos dizer que a pesquisa teve início com uma publicação de Kautz e Selman [Kau92]. Ernst, Millstein e Weld [Ern97] apresentaram um método de geração automática de codificação SAT para problemas de planejamento. Outros exemplos da utilização de resolução SAT aplicada a planejamento e escalonamento podem ser encontradas em [Cas03] [Kam00] [Rin04] [Wol01].

Pesquisa Operacional

Podemos encontrar publicações ilustrando o uso de satisfabilidade para resolver problemas de pesquisa operacional, até mesmo livros introdutórios em pesquisa operacional, como Hillier and Liebermann [Hil05], que mostram idéias combinadas utilizando satisfabilidade e programação inteira. O livro de Hooker [Hoo00] mostra que lógica e satisfabilidade podem ser utilizadas para o projeto de algoritmos de otimização. O campo de pesquisa em programação com restrições[9] usa idéias de satisfabilidade. Em Warners [War98] temos um método de transformar um problema de programação linear 0-1 com coeficientes inteiros em um problema de satisfabilidade, resolvido em períodos de tempo similares aos de técnicas de pesquisa operacional conhecidas para solução de problemas de programação inteira. Em [Man98] Manquinho, Marques Silva, Oliveira e Sakallah apresentam algumas propostas para o projeto de solucionadores de satisfabi-

[8] Em inglês, *Planning and scheduling*.

[9] Em inglês, *Constraint Programming*.

lidade para programas de programação inteira 0-1.

Análise Combinatória

Herwig, Heule, Lambalgen e Maaren [Her05] apresentaram um método baseado em satisfabilidade para computar os números de Van der Waerden em um tempo inferior aos algoritmos anteriormente existentes. Os números de Van der Waerden são números inteiros positivos N tais que toda partição dos números $\{1, ..., N\}$ em k partes possui no mínimo uma progressão aritmética de tamanho t.

Criptografia

Massacci and Marraro [Mas00] usaram solucionadores do problema da satisfabilidade para testar as propriedades de algoritmos de criptografia e chaves. Eles apresentam um método que pode ser utilizado para modelar e verificar algoritmos criptográficos no estado-da-arte como o largamente utilizado DES[10]. Encontrar uma atribuição de valores para uma fórmula SAT é equivalente à recuperação de uma chave em um ataque criptoanalítico. Na época da pesquisa as instâncias geradas pelos algoritmos criptográficos eram muito complexas para os solucionadores SAT. Fiorini, Martinelli and Massacci [Fio03] estudaram formulações para chaves-públicas. A forma de modelagem sugerida é a seguinte: dados três números a, b e c, encontrar um número d, tal que $c = d^a \bmod b$. As fórmulas geradas constituem instâncias que podem ser utilizadas para testar os solucionadores SAT.

Alocação de registradores

Em arquitetura de computadores, podemos encontrar exemplos utilizando algoritmos de satisfabilidade para a alocação de registradores, como proposto por Potlapally [Pot03]. O problema da alocação de registradores é o de associar variáveis em um programa de computador com os registradores de um processador, de tal forma que o

[10] *Data Encryption Standard.*

tempo de execução dos programas seja mínimo. Em geral encontramos mais variáveis do que registradores e as variáveis devem ser associadas aos registradores de tal forma que um mínimo de variáveis compartilhando registradores seja encontrado ao mesmo tempo. Associar duas variáveis ativas a um registrador requer a movimentação de conteúdos de variáveis para locais intermediários, o que custa tempo e conseqüentemente leva a uma degradação de desempenho.

Outras aplicações

Mesmo em áreas de pesquisa não tão próximas à ciência da computação, podemos encontrar soluções baseadas em satisfabilidade. É o caso de um exemplo apresentado por Di Giacomo, Felici, Maceratini e Truemper [Gia01], em que a satisfabilidade é utilizada na redução dos custos associados às variáveis para o diagnóstico de carcinomas. Baseado em algumas características, o método é capaz de separar pacientes com e sem carcinoma hepatocelular, podendo ser utilizado para detectar a doença em estágios iniciais. Outro exemplo na área médica é apresentado por Inês Lynce e João M. Silva em [Ine06b], onde é realizada uma análise de DNA modelada em SAT para tratamento de doenças.

Podemos até mesmo encontrar a utilização de algoritmos baseados em satisfabilidade para a resolução do popular quebra-cabeça Sudoku, como mostrado por Inês Lynce em [Ine06a].

1.2 Objetivos

O uso de solucionadores para o problema SAT tem se tornado uma forte alternativa para um número cada vez maior de diferentes domínios de problemas. Como cada instância de um problema NP-Completo pode ser traduzida para uma instância de um problema SAT em um tempo polinomial, isto torna o desenvolvimento de solucionadores rápidos do problema SAT de extrema importância, assim como o desenvolvimento de algoritmos e técnicas de aceleração destes solucionadores.

O principal objetivo deste trabalho é desenvolver um solucionador para o problema SAT baseado no algoritmo clássico DPLL que seja eficiente, robusto e competitivo com os solucionadores no estado-da-arte, principalmente na resolução de problemas industriais reais. Esse solucionador do problema de satisfabilidade deve ainda possuir uma estrutura de dados mais adaptada à sua implementação em uma arquitetura de hardware, abrindo perspectivas para novos desenvolvimentos nessa área.

O novo algoritmo deve ainda ser facilmente portável para qualquer solucionador do problema de satisfabilidade que possua como base o algoritmo clássico DPLL e em que o momento de utilização do algoritmo seja determinado pela estratégia proposta pelo solucionador.

1.3 Contribuições

Apesar de muitos terem estudado o algoritmo DPLL nos últimos anos, tem havido uma grande quantidade de novas intuições advinda da observação cuidadosa do algoritmo. A velocidade dos solucionadores tem aumentado e não existe um sinal claro de que o fim esteja próximo. Novos mecanismos de análise de cláusulas, como proposto por Eén e Biere [Nik05] no solucionador SAT vencedor da última competição mundial de SAT, a análise do potencial do processo de aprendizado de cláusulas, de acordo com a proposta de Beame, Kautz e Sabharwal [Bea04], e mesmo a utilização de hardware dedicado à solução do problema da satisfabilidade, como a proposta por Skliarova e Ferrari em [Skl04a], são intuições que têm surgido e sido integradas aos solucionadores.

Na verdade, o trabalho aqui descrito é uma conseqüência do estudo cuidadoso e minucioso do algoritmo DPLL procurando novas intuições teóricas e práticas, aliadas a técnicas de simplificação booleanas, que resultaram em um solucionador robusto e eficiente com operações que podem ser mais facilmente adaptadas aos solucionadores em hardware.

Em síntese, esse trabalho propõe uma nova forma de organização do algoritmo de procura em solucionadores baseados na abordagem clássica DPLL.

Cada cláusula negada é visualizada como um cubo no espaço booleano n-dimensional denotando um subespaço onde nenhuma atribuição que satisfaça a cláusula pode ser encontrada. O algoritmo sistematicamente subtrai todas estas cláusulas-cubo de um cubo universal que representa todo o espaço booleano n-dimensional. Se o resultado final é um cubo vazio então a instância do problema é não–satisfazível (não-SAT), caso contrário, a instância do problema admite uma solução (SAT).

O algoritmo pode ser implementado modificando-se, de forma relativamente fácil, o mecanismo de decisão em solucionadores SAT baseados no algoritmo DPLL, reutilizando a informação extraída do mecanismo de análise de conflitos.

Intuitivamente este algoritmo é mais determinístico, explora melhor o conceito de localidade e consegue aprender cláusulas mais úteis. Isto foi corroborado pelos resultados obtidos em instâncias de problemas industriais complexos que foram exaustivamente executados em um solucionador estado-da-arte.

Considerando-se a estrutura de dados basicamente matricial (construída ao nível de bits) e as operações sobre ela executadas, o algoritmo se mostra indicado à implementação em arquiteturas reconfiguráveis. A estrutura matricial, mais adaptável aos dispositivos reconfiguráveis, permite que as atuais técnicas dos solucionadores do problema da satisfabilidade em software possam ser traduzidas para o hardware de uma forma mais simples.

O desenvolvimento do trabalho propiciou a geração de um solucionador de instâncias de problemas de satisfabilidade baseado no DPLL e em cinco publicações diretamente relacionadas [Zui06a] [Zui06b] [Zui06c] [Zui06d] [Zui06e].

Em [Zui06a] e [Zui06b], apenas a idéia geral do trabalho é apresentada. Os resultados apresentados não incorporam um pacote significativo de *benchmarks*. São apresentados os principais conceitos utilizados na subtração de cubos para a solução do problema da satisfabilidade e resultados preliminares que demonstram que a metodologia é mais eficiente no tempo de solução de instâncias SAT quando comparada a um solucionador que utiliza o algoritmo clássico DPLL.

Em [Zui06c], são apresentados alguns solucionadores do problema da satisfabilidade em arquiteturas reconfiguráveis e a proposta da implementação da subtração de cubos como mais uma possível estratégia, tendo em vista a estrutura de dados mais voltada para a implementação a nível de vetores.

Em [Zui06d], detalhes do algoritmo inserido em um solucionador no estado da arte são apresentados, assim como os resultados referentes a um conjunto de *benchmarks* que incorpora as principais instâncias de problemas da satisfabilidade oriundas de problemas reais industriais de BMC da IBM e verificação formal de microprocessadores.

Em [Zui06e] são apresentados todos os resultados de simulações referentes a quatro meses de execução contínua do algoritmo em diferentes instâncias de problemas da satisfabilidade (industriais, aleatórios e computacionalmente construídos ou sintéticos). É apresentado o conceito básico da subtração de cubos, assim como indica as principais alterações a serem realizadas nas estruturas de dados e no mecanismo de decisão dos atuais solucionadores SAT baseados no algoritmo DPLL de forma a permitir a reprodução da estratégia. O artigo fornece as tabelas, gráficos com os resultados e os programas executáveis em diferentes sistemas operacionais para a reprodução dos experimentos. Esses dados estão disponíveis através de um endereço na *WEB*[11]. Apresenta ainda uma análise comparativa com o solucionador vencedor da competição SAT(2006) onde, em algumas instâncias de problemas SAT, a estratégia é superior.

1.4 Organização

Os Capítulos 2 e 3 apresentam uma revisão bibliográfica. O Capítulo 2 fornece os conceitos básicos associados ao problema da satisfabilidade e descreve diferentes abordagens para a solução do problema SAT. O Capítulo 3 trata mais especificamente dos solucionadores baseados no algoritmo clássico DPLL, partindo do algoritmo inicial DP e identificando as principais melhorias que foram incorporadas a esse algoritmo até os dias atuais.

[11] http://homepages.dcc.ufmg.br/∼rzuim/

O Capítulo 4 introduz a noção de cubos, uma abordagem matricial do problema da satisfabilidade e a operação DSharp, ilustrando como se resolver o problema da satisfabilidade através de um solucionador baseado nessa técnica.

O Capítulo 5 mostra como a operação DSharp pode ser introduzida em um solucionador clássico DPLL, qual a intuição por trás dessa inserção e analisa os resultados obtidos dessa combinação. A parte inicial do capítulo apenas mostra uma avaliação do conceito, visto que o solucionador utilizado é simples e não incorpora as principais inovações dos solucionadores no estado-da-arte. Em seguida, mostra como um solucionador no estado-da-arte pode ser transformado para receber a operação DSharp, ilustrando todas as alterações necessárias.

O Capítulo 6 mostra os resultados obtidos pela inserção da operação DSharp em um solucionador estado-da-arte através da comparação de medidas em *benchmarks* industriais, aleatórios e sintéticos, os mesmos *benchmarks* utilizados nas competições mundiais de solucionadores SAT. A análise comparativa é realizada entre o solucionador DPLL+DSharp e um solucionador DPLL atual, o Zchaff, na sua última versão disponível.

O Capítulo 7 apresenta as principais conclusões desse trabalho e propõe novas idéias para a solução de problemas atualmente pesquisados onde o algoritmo DSharp pode, de alguma forma, auxiliar.

Capítulo 2

Conceitos e algoritmos básicos

Nos últimos anos, diferentes algoritmos para a solução do problema da satisfabilidade foram desenvolvidos e implementados. Esse capítulo mostra os principais algoritmos divulgados. Anualmente, a comunidade de pesquisadores de problemas SAT realiza uma competição de solucionadores do problema da satisfabilidade cujo objetivo é comparar o desempenho dos solucionadores em diferentes tipos de instâncias de problemas. Podemos classificar esses solucionadores de uma forma geral como aqueles que são consistentes (corretos)[1], caso em que toda instância para a qual o solucionador responde "sim" é realmente satisfazível, e aqueles que são completos, caso em que o solucionador responde "sim" para toda instância satisfazível. O Capítulo descreve, de forma não exaustiva, diferentes algoritmos consistentes, completos e não completos, atualmente adotados pelos solucionadores do problema da satisfabilidade e, ao final do capítulo, uma pequena descrição dos principais solucionadores encontrados na atualidade. Esses solucionadores, que exemplificam os algoritmos apresentados, obtiveram bons resultados nas mais recentes competições mundiais de SAT.

[1] *Sound*.

2.1 Conceitos básicos

Uma fórmula, ou expressão booleana, denota uma função que envolve variáveis booleanas. Tal função mapeia B^n em B, onde n é o número de variáveis da fórmula e $B = \{verdadeiro, falso\}$ é o espaço booleano.

Recapitulando da seção 1.1:

- Um literal é definido como uma variável proposicional x_i ou a sua negação $\neg x_i$. Se a variável proposicional x_i for verdadeira, o literal x_i será verdadeiro e o literal $\neg x_i$ será falso. Se a variável proposicional x_i for falsa, o literal x_i será falso e o literal $\neg x_i$ será verdadeiro. O complemento do literal x_i é $\neg x_i$ e, vice-versa, o complemento de $\neg x_i$ é x_i. Se a variável proposicional x_i não possuir um valor atribuído, ela será denominada variável livre.

- Uma cláusula é uma disjunção de literais, $(x_1 \vee x_2 \vee \ldots \vee x_n)$, onde n representa o número de literais da cláusula.

- Uma cláusula é dita vazia se não possuir nenhum literal.

- Uma cláusula com um só literal é uma cláusula unitária. Denotaremos uma cláusula unitária com a variável x_i como (x_i) ou, quando o literal aparecer negado, como $(\neg x_i)$.

- Uma cláusula é dita satisfeita se as atribuições já feitas fizeram com que um de seus literais assumisse o valor verdadeiro. Uma cláusula é dita não resolvida se possui variável livre e ainda não foi satisfeita.

- Uma fórmula na forma normal conjuntiva (FNC) ϕ é uma conjunção de cláusulas, $w_1 \wedge w_2 \wedge \ldots \wedge w_m$, onde m representa a quantidade de cláusulas.

Um exemplo de fórmula na FNC é:

$$\phi = (a \vee b \vee \neg c) \wedge (a \vee \neg a \vee \neg c) \wedge (\neg a \vee b \vee \neg c) \wedge (a \vee b) \wedge (c) \wedge (\neg c)$$

A fórmula exemplificada contém seis cláusulas. As três primeiras cláusulas contêm três literais cada uma, a quarta cláusula contém dois literais e as duas últimas contêm um literal cada uma. A fórmula ϕ é avaliada como 0 (é sempre falsa) devido às duas últimas cláusulas.

Uma cláusula é redundante em uma fórmula na FNC se, ao ser retirada, a função booleana representada pela fórmula não se altera; de outra forma, a cláusula é dita não redundante. No exemplo, as quatro primeiras cláusulas são redundantes, já que, se retiradas, não afetam o valor representado pela fórmula ϕ ; por outro, lado as duas últimas cláusulas são não redundantes. Uma cláusula w_1 é subjugada por uma cláusula w_2 se todos os literais que aparecem em w_2 também aparecem em w_1. Se uma cláusula w em uma fórmula na FNC é subjugada por outra, então a cláusula w é redundante. No exemplo, as três primeiras cláusulas são subjugadas pela última cláusula. Uma cláusula w é dita uma tautologia se, tanto o literal, quanto o seu complemento, aparecem na cláusula; a segunda cláusula no exemplo é uma tautologia.

Alternativamente, também é possível representarmos uma fórmula na forma normal disjuntiva (FND), também conhecida como soma de produtos. Cada termo é a conjunção (e) de um ou mais literais.

Um exemplo de uma fórmula na FND é:

$$\varphi = (a \wedge b \wedge \neg c) \vee (a \wedge \neg a \wedge \neg c) \vee (\neg a \wedge b \wedge \neg c) \vee (a \wedge b) \vee (c) \vee (\neg c)$$

A fórmula φ contém seis termos:$(a \wedge b \wedge \neg c)$, $(a \wedge \neg a \wedge \neg c)$, $(\neg a \wedge b \wedge \neg c)$, $(a \wedge b)$, (c) e $(\neg c)$. A função denotada pela fórmula é avaliada como verdadeira (1) para quaisquer atribuições às variáveis proposicionais, tendo em vista os dois últimos termos. Um termo é vazio se possuir tanto o literal como o seu complemento; o segundo termo da fórmula φ por exemplo. Um termo é redundante em uma fórmula se, ao ser retirado, o valor representado pela função não sofre alteração; os quatro primeiros termos da fórmula φ são redundantes. Um termo é não redundante caso o contrário ocorra; os dois últimos termos da fórmula φ por exemplo.

Um termo c_1 está contido em um termo c_2 se qualquer literal em c_2 também estiver em c_1 (por exemplo, em φ, o termo $(\neg a \wedge b \wedge \neg c)$ está contido no termo $(\neg c)$) . Um termo contido é também redundante. Um termo sem literais é dito uma tautologia.

Um termo vazio é redundante em uma fórmula na FND, assim como cláusulas tautológicas em uma fórmula na FNC. O Capítulo 4 irá entrar em mais detalhes sobre a representação na FND, quando apresentar e definir cubos.

2.2 Algoritmos para solução do problema SAT

A história de alguns algoritmos utilizados na resolução do problema SAT é curiosa. Em 1960 Davis e Putnam publicaram um algoritmo para verificar se uma fórmula proposicional na FNC é não satisfazível [Dav60]. Eles não usaram o termo "resolução", que foi introduzido por Robinson em 1965 [Rob65]. A literatura subseqüente reconheceu que, na verdade, o trabalho de Davis e Putnam era uma política particular de resolução proposicional. Em 1962, Davis, Logemann e Loveland publicaram um outro algoritmo bastante conhecido e utilizado até os dias de hoje [Dav62], descrevendo-o como uma otimização do algoritmo de 1960, procurando preservar a memória daquele. Esse algoritmo de pesquisa passou a ser denominado "Algoritmo de Davis-Putnam" ou simplesmente "DP". Apesar de ser um algoritmo diferente, a denominação DP foi utilizada para o mesmo, já que o trabalho de 1960 era continuamente citado como a fonte do algoritmo. Essa situação perdurou até os anos 90, quando o trabalho de 1962 foi redescoberto e o acrônimo DLL passou a ser utilizado. Finalmente, o acrônimo DPLL foi adotado amplamente para o algoritmo de procura de Davis, Logemann e Loveland, agora, reconhecendo a contribuição dos quatro autores. Essa será a notação utilizada nesse trabalho.

2.2.1 Algoritmo DP (Davis-Putnam)

O algoritmo original para a solução de problemas SAT é atribuído a Davis e Putnam após a divulgação de uma solução baseada em resolução em 1960 [Dav60]. Para um algoritmo baseado em resolução, podemos gerar cláusulas redundantes a partir de duas cláusulas se certas condições estiverem presentes [Hac96]. Se L for um conjunto de literais, seja ΣL a disjunção de todos os literais em L. Então: se L_1 e L_2 são conjuntos de literais e x uma variável proposicional[2], $(x \vee \Sigma L_1) \wedge (\neg x \vee \Sigma L_2) \equiv (x \vee \Sigma L_1) \wedge (\neg x \vee \Sigma L_2) \wedge (\Sigma L_1 \vee \Sigma L_2)$ ou, exemplificando, se a, b e c são literais, então:

$$(a \vee b) \wedge (\neg a \vee c) \equiv (a \vee b) \wedge (\neg a \vee c) \wedge (b \vee c)$$

À operação de se gerar a cláusula $(\Sigma L_1 \vee \Sigma L_2)$ a partir das cláusulas $(x \vee \Sigma L_1)$ e $(\neg x \vee \Sigma L_2)$ chamamos resolução. A cláusula resultante $(\Sigma L_1 \vee \Sigma L_2)$ é chamada resolvente das cláusulas $(x \vee \Sigma L_1)$ e $(\neg x \vee \Sigma L_2)$.

Vê-se, então, que o resolvente de duas cláusulas é redundante em relação às duas cláusulas originais.

Além da regra de resolução, o procedimento de Davis-Putnam aplica duas outras, a regra da cláusula unitária e a regra do literal puro. A regra da cláusula unitária, na verdade, é a aplicação da regra da resolução quando uma das premissas é uma cláusula unitária. Se uma cláusula é unitária, a única variável existente na cláusula deverá receber um valor que torne o literal verdadeiro; neste caso, o valor do literal e, conseqüentemente, da variável são ditos implicados. A contínua aplicação dessa regra de implicação, ou cláusula unitária, é freqüentemente chamada Propagação de Restrição Boolena, ou BCP[3] [Zab88]. Um literal é puro se está presente na fórmula, mas não o seu complemento. Conseqüentemente, a variável deve receber um valor que torne o literal verdadeiro e, portanto, todas as cláusulas que o contêm. Essas duas novas regras significaram melhorias importantes à regra da resolução. Primeiramente,

[2] $\varphi \equiv \psi$ significa que as fórmulas φ e ψ são logicamente equivalentes.

[3] *Boolean Constraint Propagation.*

procuramos todas as implicações e, em seguida, removemos as cláusulas com literais puros.

A seguir o algoritmo Davis-Putnam, sendo a entrada do algoritmo uma fórmula φ na FNC e a saída a resposta SAT (a fórmula é satisfazível) ou não-SAT (a fórmula não é satisfazível):

1. Se φ possuir uma cláusula vazia retorne não-SAT.

2. Remova de φ todas as cláusulas que contenham literais complementares. Se φ for vazia, retorne SAT.

3. Se φ contiver uma cláusula unitária (x) e outra $(\neg x)$, retorne não-SAT.

4. Se φ contiver uma cláusula unitária (u), remova de φ todas as cláusulas que contenham u , e remova o complemento de u de todas as cláusulas em que ele esteja presente. Se φ for vazia, retorne SAT, senão, se φ contiver uma cláusula unitária retorne ao passo 1.

5. Enquanto existir uma cláusula com um literal puro em φ, remova todas as cláusulas em que o literal puro estiver presente. Se φ for vazia retorne SAT.

6. Selecione uma variável x qualquer. Seja L_1 a conjunção de todas as cláusulas de φ que contenham x. Seja L_2 a conjunção de todas as cláusulas de φ que contenham $\neg x$. Seja L_3 a conjunção de todas as cláusulas de φ que não contenham x nem $\neg x$. Remova x de L_1 produzindo $L_1^{'}$ e remova $\neg x$ de L_2 produzindo $L_2^{'}$. Substitua φ por $(L_1^{'} \vee L_2^{'}) \wedge L_3$. Use a propriedade distributiva para transformar φ em uma conjunção de cláusulas. Retorne ao passo 2.

O passo 1 apenas identifica a existência de uma cláusula vazia, o que significa que o problema é não-SAT. O passo 2 é uma simplificação: remove as cláusulas que são trivialmente verdadeiras. Se todas as cláusulas da fórmula φ forem desse tipo, a fórmula é satisfazível.

Os passos 3 e 4 representam a regra da cláusula unitária ou BCP. Em 3, verifica-se na fórmula a existência de cláusulas unitárias contraditórias. A presença da cláusula (x) implica que o valor da variável x deve ser verdadeiro e a presença do literal $(\neg x)$ implica que o valor da variável x deve ser falso. Se ambas estiverem presentes, a fórmula é não satisfazível. No passo 4, a ocorrência de uma variável em uma cláusula unitária é o que denominamos uma variável implicada. Assim, se o valor falso for atribuído à variável x, a cláusula unitária (x) se torna não satisfazível, portanto a atribuição do valor verdadeiro a x é uma conseqüência necessária para tornar a fórmula φ satisfazível, analogamente uma cláusula unitária $(\neg x)$ implica na atribuição do valor falso. Nessa situação podemos remover todas as cláusulas com ocorrências de x pois elas são subjugadas por (x): se uma cláusula possui um literal u, ela se torna satisfazível e portanto pode ser retirada da fórmula φ. Todas as ocorrências do complemento de u são eliminadas através da resolução unitária: desde que o complemento de u seja falso, ele não pode contribuir para tornar a cláusula onde se encontra satisfazível; dessa forma ele deve ser retirado da cláusula e esta deverá se tornar satisfazível através de outro literal. Se após a aplicação da regra da cláusula unitária, φ não possuir mais cláusulas, a fórmula é satisfazível e, conseqüentemente, a fórmula originalmente fornecida ao algoritmo. Caso contrário, os passos 3 e 4 são novamente executados até que não restem mais cláusulas unitárias.

O passo 5 é responsável pela regra do literal puro, que retira da fórmula todas as cláusulas com literais puros existentes.

Quando a execução atingir o passo 6, nenhum literal em φ é susceptível de ser retirado através da regra da cláusula unitária ou do literal puro. Neste passo é necessário a escolha de uma variável para continuar o processo. A esse mecanismo damos o nome de estratégia de decisão; ele é um dos fatores fundamentais na eficiência dos atuais solucionadores do problema da Satisfabilidade.

O passo 6 é também o responsável por manipular a fórmula φ para excluir a variável x escolhida. A fórmula φ é logicamente equivalente a $L_1 \wedge L_2 \wedge L_3$. Substituindo L_1 pela

expressão logicamente equivalente $\left(L_1^{`} \vee x\right)$ e $L_2$ pela expressão logicamente equivalente $\left(L_2^{`} \vee \neg x\right)$, temos que $\varphi \equiv (L_1^{`} \vee x) \wedge (L_2^{`} \vee \neg x) \wedge L_3$ e φ é satisfazível se, e somente se, $(L_1^{`} \vee L_2^{`}) \wedge L_3$ é satisfazível. E nessa última expressão, a variável x não mais aparece. Essa expressão, entretanto, não é , necessariamente, uma conjunção de cláusulas. Uma conversão é necessária para que o algoritmo retorne ao passo 2. É exatamente nessa conversão que o algoritmo de Davis-Putnam se mostra ineficiente. Se $L_1^{`}$ possuir $m_1$ cláusulas e $L_2^{`}$ possuir m_2 cláusulas, a distribuição de um sobre o outro pode resultar em $m_1 m_2$ cláusulas. O que pode ocasionar uma expansão quadrática na fórmula a cada vez que uma variável for removida. Uma outra maneira de compreendermos isto é perceber que no passo 6, o que realmente está acontecendo é que a fórmula φ está sendo formatada como $L_4 \wedge L_3$, onde L_4 é a conjunção das cláusulas produzidas pela resolução em x de cada cláusula em L_1 por cada uma das cláusulas em L_2.

Segue um exemplo; seja a fórmula na FNC $\varphi = w_1 \wedge w_2 \wedge w_3 \wedge w_4 \wedge w_5$ onde $w_1 = (a \vee c)$, $w_2 = (b \vee c)$, $w_3 = (c \vee d)$, $w_4 = (\neg a \vee \neg b \vee \neg c \vee \neg e)$ e $w_5 = (e)$. Ao aplicarmos o algoritmo de Davis-Putnam, primeiramente a variável e será implicada como verdadeira para resolver a cláusula w_5, considerando a regra da cláusula unitária no passo 4; ainda no passo 4, a ocorrência do literal $\neg e$ será removida da cláusula w_4. O problema agora possui apenas quatro cláusulas. Em seguida, vamos aplicar a regra da resolução no passo 6. Supondo a escolha para a eliminação da variável c, o problema terá então três cláusulas; usaremos como índices dessas cláusulas os índices das cláusulas originais utilizadas na sua determinação:

w_{1-4} será o resolvente entre w_1 e $w_4 = (a \vee \neg a \vee \neg b)$
w_{2-4} será o resolvente entre w_2 e $w_4 = (b \vee \neg a \vee \neg b)$
w_{3-4} será o resolvente entre w_3 e $w_4 = (d \vee \neg a \vee \neg b)$

As cláusulas w_{1-4} e w_{2-4} são tautologias (possuem um literal e o seu complemento), portanto serão retiradas no passo 2 do algoritmo e a cláusula restante, w_{3-4}, possui apenas literais puros. Ao serem removidos, o algoritmo retorna SAT para o problema.

Observamos, em geral, um uso exponencial de espaço; a cada variável eliminada podemos ter uma expansão quadrática da fórmula e inúmeras variáveis serão eliminadas. Uma proposta posteriormente apresentada por Davis, Longemann e Loveland [Dav62] substituiu a regra da resolução por uma regra de separação que divide o problema em dois subproblemas menores. Esse novo algoritmo, que é utilizado pela grande maioria dos solucionadores SAT atuais, será descrito aqui resumidamente e em detalhes no próximo capítulo.

2.2.2 Algoritmo Davis, Logemann e Loveland (DLL)

Como visto, existe para o algoritmo de Davis-Putnam, uma possibilidade da fórmula crescer exponencialmente com relação ao número de variáveis do problema. Na prática, essa situação ocorre freqüentemente. Por essa razão o algoritmo de Davis, Logemann e Loveland é geralmente utilizado. O algoritmo DPLL utiliza um espaço adicional apenas linear no número de variáveis, o que o torna mais eficiente que seu predecessor, DP. Um algoritmo de procura com retrocesso[4] é utilizado através de uma técnica que, implicitamente, enumera o espaço de 2^n possíveis atribuições binárias às n variáveis do problema.

A seguir o algoritmo; a entrada consiste em uma fórmula φ na FNC e a saída em uma resposta SAT ou não-SAT:

1. Se não existir nenhuma cláusula unitária em φ vá para o passo 3.

2. Se existir uma cláusula unitária (u) em φ, remova de φ todas as cláusulas que contenham o literal u e remova o complemento de u de todas as cláusulas em que ele apareça. Retorne ao passo 1.

3. Se φ contiver uma cláusula vazia retorne não-SAT, senão, enquanto existir um literal puro em φ, retire todas as cláusulas em que esse literal puro estiver. Se agora φ for vazia, retorne SAT.

[4] *Backtracking.*

4. Selecione alguma variável x de φ.

5. Execute recursivamente o algoritmo para $\varphi \wedge (x)$. Se a chamada recursiva retornar SAT, retorne SAT.

6. Execute recursivamente o algoritmo para $\varphi \wedge (\neg x)$. Se a chamada recursiva retornar SAT, retorne SAT, senão retorne não-SAT.

Os passos 1 e 2 executam o BCP, de forma similar ao algoritmo DP. O passo 3 verifica a existência de alguma cláusula vazia, caso em que a instância é insatisfazível. Se não for encontrada alguma cláusula vazia, o algoritmo executará a eliminação dos literais puros. O passo 4 executa uma estratégia de decisão que, assim como no algoritmo DP, tem um impacto decisivo na eficiência global do algoritmo. Escolhendo-se uma variável x, existem duas possibilidades para a atribuição de valor a x, uma que torna x verdadeira e outra que torna x falsa. Os passos 5 e 6 avaliam essas duas possibilidades. Executando-se o algoritmo com a entrada $\varphi \wedge (x)$ no passo 5, o BCP força a variável x a ter um valor verdadeiro e com a entrada $\varphi \wedge (\neg x)$, força a variável x a ter um valor falso. φ é satisfazível se uma das duas possibilidades retornar SAT e insatisfazível caso contrário.

Existem três mecanismos importantes que influenciam o comportamento e o desempenho do algoritmo, um primeiro mecanismo que iremos denominar dedução, um segundo, denominado análise de conflito e um terceiro, que iremos denominar decisão. O Capítulo 3 entrará em detalhes nesses três mecanismos, apresentando as principais inovações em cada um deles. Uma descrição sucinta deles é:

- O mecanismo de dedução é o responsável pelas atribuições dos valores às variáveis decorrentes das implicações, isto é, é o responsável pela aplicação da regra da cláusula unitária e a regra do literal puro. Esse mecanismo corresponde aos passos 1, 2 e 3 do algoritmo. O passo 3 é que determina um resultado final da dedução. Se ao removermos as cláusulas satisfazíveis com a atribuição de um valor à variável x e retirarmos todas as ocorrências do complemento de x,

conforme o passo 2, chegarmos a uma cláusula vazia, o problema é não-SAT, caso contrário é SAT.

- O mecanismo de análise de conflito é aquele que procura identificar uma cláusula vazia durante o mecanismo de dedução. Pode ser identificado como a primeira linha no passo 3. Na verdade, indica que as duas tentativas de atribuição de valor a uma variável retornaram não-SAT, ou que não é possível nenhum desses dois valores para uma determinada variável já que ambos levam a uma, ou mais, cláusulas vazias. A essa contradição, iremos denominar uma situação de conflito. É o mecanismo básico que permite abandonar regiões do espaço de procura em que não existe uma atribuição de valores às variáveis que satisfaça a instância. Na próxima seção, veremos que o mecanismo de análise de conflito nos solucionadores atuais é utilizado não apenas para identificar um conflito mas, também, identificar as causas que geraram uma cláusula vazia.

- O mecanismo de decisão é aquele que seleciona uma nova variável a cada estágio da procura. Podemos identificá-lo como o passo 4 para a escolha da variável e dos passos 5 e 6 para a escolha inicial do valor a ser atribuído à variável escolhida. Uma decisão sempre irá procurar uma variável livre de acordo com alguma estratégia de decisão, o mesmo acontecendo para o valor a ser atribuído a essa variável. Esse é o mecanismo básico que garante que novas regiões do espaço de procura serão exploradas. O algoritmo original DPLL não especifica um critério específico para a escolha da variável nem do valor inicial.

Na prática, os solucionadores modernos baseados no algoritmo DPLL possuem algumas simplificações: uma primeira simplificação se refere à não aplicação da regra do literal puro. Se considerarmos que deveremos percorrer todo o conjunto de cláusulas para a identificação de cada literal puro, o tempo necessário para essa atividade pode ser longo. Essa característica está associada às estruturas de dados atualmente utilizadas nos solucionadores SAT no estado da arte e será descrita no próximo capítulo.

Uma segunda simplificação é que, uma estratégia eficiente de escolha de variáveis, preferencialmente, indicará uma variável livre presente na fórmula como um literal puro. Os solucionadores modernos gastam a maior parte do tempo executando a regra da cláusula unitária, ou BCP, que também será descrita em detalhes no próximo capítulo.

A enumeração do espaço de procura pode ser realizada através de uma árvore de decisão. Cada nó da árvore de decisão representa uma variável com um valor ainda não atribuído e, portanto, uma candidata a ser escolhida para uma atribuição de valor. À escolha damos o nome de decisão de atribuição e ao nível da árvore onde essa decisão ocorreu, de nível de decisão. A primeira decisão de atribuição está associada à raiz da árvore de decisão, que é considerada como estando no nível um. Atribuições realizadas antes da primeira decisão, em geral através de um pré-processamento, são consideradas como estando no nível zero. Associado à árvore de decisão existe uma pilha de decisões. Todas as atribuições feitas durante a execução do algoritmo são armazenadas nessa pilha e, quando um retrocesso acontece, são retiradas todas as atribuições desfeitas. Podemos observar que essa pilha de decisões é, na verdade, a pilha de recursão do algoritmo.

Solucionadores modernos procuram manter essa pilha de atribuições, assim como o caminhamento na árvore de atribuições, de uma forma explícita, contrariamente à versão recursiva onde tanto o caminhamento quanto a árvore estão implícitos no algoritmo. O pseudocódigo apresentado na Figura 2.1 ilustra essa versão não recursiva do mesmo algoritmo.

No pseudocódigo da Figura 2.1 os três mecanismos podem ser identificados. O mecanismo de decisão é identificado pela escolha aleatória de uma variável livre, já que o procedimento DPLL original não especifica a variável a ser escolhida.

O mecanismo de dedução implementa a propagação binária de restrições ou BCP.

O mecanismo de análise dos conflitos implementa um retrocesso simples, também chamado de retrocesso cronológico[5]. O algoritmo mantém um controle de qual atri-

[5] *Chronological Backtracking*

```
DPLL( )
início
    enquanto (verdadeiro)
        se ( para a variável x de decisão do último conflito encontrado, o valor atribuído a
             torna verdadeira)
            então atribua para a variável x o complemento do seu valor e execute o BCP( );
            senão escolha uma nova variável y, atribua um valor que a faça verdadeira e
                execute o BCP( );
        se (for encontrado um conflito)
            enquanto existirem variáveis atribuídas e este laço não for rompido
                se ( não foi atribuído o complemento do valor da variável de decisão y
                     desfaça a atribuição do valor da variável y e todas as atribuições que
                     foram implicadas como resultado do BCP após esta atribuição de y)
                    marque que existe uma variável a ser atribuída;
                    rompa este laço de enquanto;
                fimse
                se (foi atribuído o complemento da última variável de decisão y)
                    desfaça a atribuição da variável y e todas as atribuições que foram
                    implicadas como resultado do BCP após esta atribuição de y;
                    continue o laço de enquanto;
                fimse
            fimenquanto
        se (existe atribuição a todas as variáveis)
            retorne SAT;
        se (não existem variáveis atribuídas e não existem variáveis marcadas para serem
             atribuídas)
            retorne não-SAT;
    fimenquanto
fim
```

Figura 2.1: Pseudocódigo do algoritmo DPLL

buição de decisão foi realizada. Quando ocorrer um conflito no nível de decisão d, o algoritmo verifica se a variável de decisão x do nível correspondente já foi avaliada para os dois literais possíveis. Se não foi, ele desfaz as atribuições que foram implicadas pela atribuição de x, incluindo a atribuição de x, e atribui o literal com o complemento para x. Caso o valor do complemento já tenha sido anteriormente avaliado, acontece o retrocesso para o nível mais recente de uma variável de decisão de um literal ainda não complementado. A pilha de atribuições é utilizada para identificar e restaurar os valores originais das variáveis durante o retrocesso.

As Figuras 2.2 e 2.3 ilustram a utilização desses três mecanismos. Na Figura 2.2 podemos acompanhar a evolução da fórmula original exemplificada até a solução, os três mecanismos estão indicados. Na Figura 2.3, podemos ver a árvore binária de decisão com as variáveis da mesma instância.

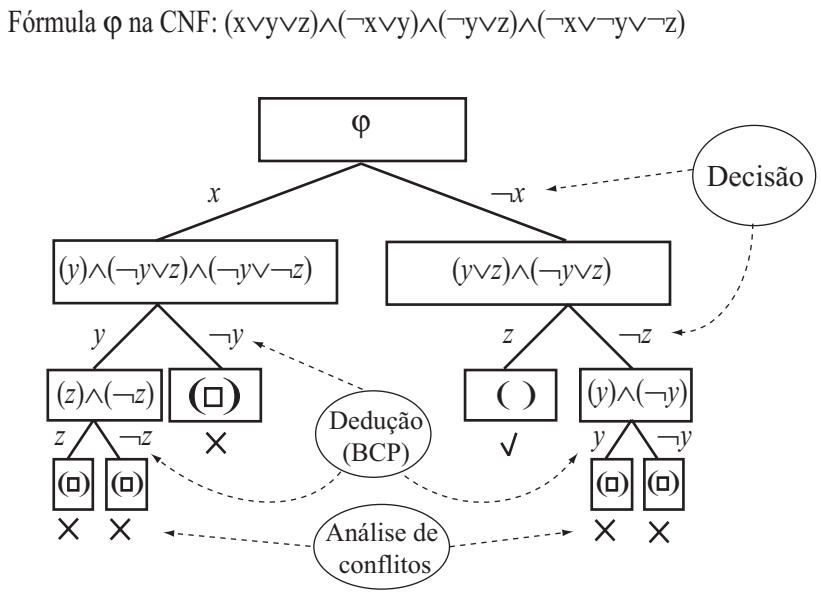

Figura 2.2: Os três mecanismos principais do DPLL

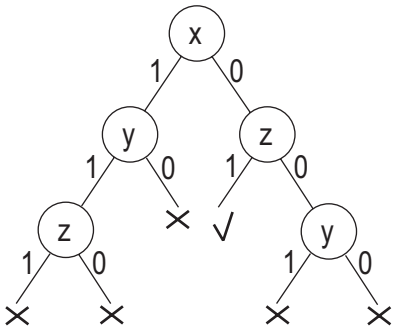

Figura 2.3: Árvore binária de decisão para as variáveis x,y e z.

A resolução e o DPLL

Toda árvore gerada pelo algoritmo DPLL é linearmente equivalente a uma dedução por resolução [Gel01]. Cada decisão corresponde à resolução de duas cláusulas.

2.2.3 Diagramas de Decisão Binários (BDD)

Podemos encontrar na literatura diversos artigos referentes a diagramas de decisão binários (BDDs) [Bry86] [Bry92] [Lee59] [Ake78] [Bra90] [Mei98]. O que aqui se segue é uma descrição sucinta desse método e como podemos relacioná-lo com o problema da satisfabilidade.

Um diagrama de decisão binário representa uma função boolena através de um grafo quase fortemente conexo direcionado acíclico com raiz. As folhas desse grafo são identificadas como verdadeiro ou falso indicando os valores possíveis da função subjacente. Cada nó interno representa uma variável booleana e possui duas arestas apontando para os filhos, uma para uma atribuição de falso para a variável representada pelo nó pai e outra para uma atribuição de verdadeiro. Para uma dada atribuição às variáveis, o valor da função pode ser determinado através do caminhamento no grafo da raiz até uma folha.

A Figura 2.4 ilustra dois BDDs representando as funções booleanas $(p \vee q) \wedge r$, em (a), e $(p \wedge q \wedge \neg r)$, em (b). A aresta pontilhada representa a variável atribuída como falsa e a aresta cheia representa a variável atribuída verdadeira.

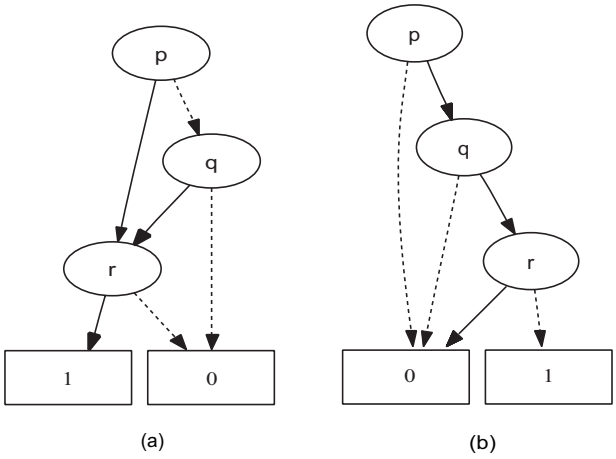

Figura 2.4: Ilustração de dois BDDs

Os primeiros conceitos da representação de funções boolenas através de BDDs foram apresentados por Lee em 1959 [Lee59], mas a utilização como uma estrutura de dados para a manipulação booleana teve início após a apresentação por Bryant, em 1986 [Bry86], com restrições na ordenação das variáveis de decisão de um BDD. Essas restrições permitiram o desenvolvimento de algoritmos mais eficientes para a manipulação de BDDs.

Se α for uma ordem total sobre um conjunto de variáveis booleanas, um BDD é dito ordenado (OBDD) em relação a α se as variáveis em todo caminho da raiz até

uma folha estão em ordem ascendente em respeito a α.

Os BDDs ilustrados na Figura 2.4 são exemplos de OBDDs de acordo com a ordenação de variáveis $p\ \alpha\ q\ \alpha\ r$. A ordenação pode ser arbitrariamente escolhida, mas uma boa escolha é crucial para um melhor desempenho, o que não será tratado aqui nesse trabalho.

Apesar da ordenação apresentar vantagens, algumas redundâncias podem ser encontradas:

- Para um nó v qualquer, se seus dois predecessores forem um único nó, ele não acrescenta nenhuma nova informação.
- Dentro de um diagrama de decisão podemos encontrar múltiplos sub-diagramas que ocorrem diversas vezes. Isto implica na múltipla representação da mesma informação sobre a função.

Para remover esses tipos de redundâncias, as seguintes regras podem ser aplicadas:

- Se as duas arestas de um nó v apontam para o mesmo nó u, v é retirado e todas as arestas que apontam para v passam a apontar para u.
- Se os nós internos u e v representam a mesma variável booleana e apontam para o mesmo nó destino, remove-se um dos nós, u ou v, e redireciona-se todas as arestas que chegam ao nó retirado para o que permaneceu.

Ao aplicarmos essas duas regras descritas, obtemos um OBDD reduzido R(O)BDD[6].

Para uma ordenação fixa de variáveis, um ROBDD é uma representação canônica de uma função booleana: duas funções booleanas são idênticas, ou seja, as fórmulas respectivas são logicamente equivalentes, se e somente se as suas representações ROBDD são isomorfas.

Para verificarmos se uma fórmula booleana é satisfazível, basta construir uma representação BDD da fórmula e verificar se ela é isomorfa ao grafo que representa falso. Se não for, a fórmula é SAT, caso contrário é não-SAT.

[6] *Reduced (and Ordered) Binary Decision Diagram*

Nas implementações modernas de BDDs [Bra90], temos um tempo constante para a verificação se uma representação é isomorfa a outra. Entretanto, o tamanho do BDD gerado é sensível à ordenação das variáveis na construção do BDD. Essa situação pode ser exemplificada quando representamos um circuito somador em BDD: esse pode ser linear ou exponencial com relação ao número de bits da largura do somador, dependendo da forma como as variáveis forem ordenadas. Drechsler em [Rol98] mostra que encontrar uma boa ordenação é um problema NP-completo, apresenta um algoritmo de ordenação assim como resultados que comprovam a eficiência de algoritmo comparativamente às heurísticas existentes. Existem ainda os casos em que a representação em BDD é exponencial no espaço, independentemente da forma de ordenação [Bry86]; por exemplo, a representação do bit mais significativo de um multiplicador em função dos bits da sua entrada. Por essa razão, a representação prática por meio de BDDs de funções booleanas é, freqüentemente, limitada a algumas centenas de variáveis. Nota-se, entretanto, que uma vez construída a representação BDD de uma instância, podemos extrair muito mais informações do que a satisfabilidade para essa mesma instância.

2.2.4 Algoritmo de Stålmarck

O algoritmo de Stålmarck proposto por Gunnar Stålmarck em 1980 [Sta89] é um algoritmo patenteado. Existem solucionadores comerciais baseados nesse algoritmo disponibilizados por *Prover Technology* [Url02]. O algoritmo de Stålmarck executa uma procura em largura, contrariamente ao algoritmo DPLL, que realiza uma procura em profundidade.

O algoritmo é basicamente destinado à verificação de tautologia, o que conseqüentemente, o credencia para a solução do problema da satisfabilidade. Para a verificação se uma fórmula booleana φ é satisfazível, basta verificar se a sua negação, $\neg\varphi$, é tautologia; se for, φ é insatisfazível, se não for φ é SAT. Um tutorial sobre o método é apresentado por Sheeran e Stålmarck em [She00].

O método original trabalha com fórmulas contendo os conectivos lógicos usuais:

negação ($\neg$), conjunção ($\wedge$), disjunção ($\vee$) e implicação ($\rightarrow$). Versões mais recentes do algoritmo procuram reduzir a fórmula aos conectivos: negação e conjunção.

Todas as fórmulas lógicas podem ser escritas como : $\varphi \equiv (q \rightarrow r)$. Fica claro que qualquer função lógica pode ser representada através desse formato e do valor lógico falso. Exemplificando:

$(\neg a) \equiv a \rightarrow \bot$;
$(a \vee b) \equiv \neg a \rightarrow b \equiv (a \rightarrow \bot) \rightarrow b$;
$(a \wedge b) \equiv \neg(a \rightarrow \neg b) \equiv \neg(a \rightarrow (b \rightarrow \bot)) \equiv (a \rightarrow (b \rightarrow \bot) \rightarrow \bot)$.

O método adota a tripla (p, q, r) para representar $p \equiv (q \rightarrow r)$.

Inicialmente, operações simples são aplicadas à fórmula, para utilizarmos apenas os conectivos $\rightarrow$ e $\bot$. As etapas seguintes introduzirão novas variáveis proposicionais construindo as triplas, para representar cada subfórmula. Podemos, dessa forma, utilizar um conjunto de triplas de literais, os valores 0 e 1 e adicionar novas variáveis à formula original. Exemplificando: seja a fórmula $a \rightarrow (b \rightarrow a)$, podemos introduzir as variáveis v_1 e v_2 , onde $v_1 \equiv b \rightarrow a$ e $v_2 \equiv a \rightarrow v_1$, e que será representada através do conjunto de triplas $\{(v_1, b, a), (v_2, a, v_1)\}$. O objetivo é provar que a fórmula é uma tautologia. Assim, podemos assumi-la como falsa e derivar uma contradição. Uma série de sete regras de redução podem ser aplicadas às triplas e a estratégia consiste na aplicação das regras até atingirmos uma tripla terminal. As regras básicas são:

$(0, y, z)$, deduz-se que y=1 e z=0;

$(x, y, 1)$, deduz-se que x=1;

$(x, 0, z)$, deduz-se que x=1;

$(x, 1, z)$, deduz-se que $x = z$;

$(x, y, 0)$, deduz-se que $x = \neg y$;

(x, x, z), deduz-se que x=1 e z=1;

(x, y, y), deduz-se que x=1.

Uma tripla é dita terminal se ela é contraditória. Por exemplo, através da tripla $(1, 1, 0)$ deduz-se que $1 \rightarrow 0$, uma contradição. As triplas terminais são: $(1, 1, 0)$, $(0, x, 1)$, $(0, 0, x)$.

A aplicação das regras irá procurar remover triplas e substituir esses literais em outras triplas. Um exemplo completo: seja a fórmula $a \rightarrow (b \rightarrow a)$, como ela possui os dois conectivos, construímos as triplas: $v_1 \equiv b \rightarrow a$ e $v_2 \equiv a \rightarrow (b \rightarrow a)$. A seguir vamos assumir a fórmula como falsa e verificar se chegamos a uma contradição: $\{(v_1, b, a), (v_2, a, v_1)\}$ e assumindo que $v_2 = 0$. Substituindo vem: $\{(v_1, b, a), (0, a, v_1)\}$, de $(0, a, v_1)$ deduz-se que $a = 1$ e $v_1 = 0$. Substituindo em $\{(v_1, b, a)\}$, vem: $\{(0, b, 1)\}$, que é uma tripla terminal ou uma contradição e, portanto, $a \rightarrow (b \rightarrow a)$ é uma tautologia.

Ao processo de se deduzir tantas triplas quantas forem possíveis, através da aplicação das regras simples, o algoritmo dá o nome de saturação-0. De acordo com Groote em [Gro00], na prática, o conjunto de triplas geradas não é tão grande como teoricamente poderíamos imaginar.

Se, com a saturação-0, não chegarmos a uma tripla terminal, o algoritmo utiliza ainda uma outra regra, denominada dilema. Se, por exemplo, o seguinte conjunto de triplas for encontrado: $\{(1, \neg a, a), (1, a, \neg a)\}$, nenhuma das regras básicas pode ser aplicada. Essa nova regra consiste na escolha de uma variável de separação. Considerando que a saturação-0 gerou um conjunto de triplas Ψ, e a variável de separação escolhida foi v. Novamente iremos executar a saturação-0 sobre os dois conjuntos: $\Psi \cup \{v = 1\}$ e $\Psi \cup \{v = 0\}$, produzindo novos conjuntos de equações: $\Upsilon_{verdadeiro}$ e Υ_{falso} respectivamente. Mesmo que uma contradição não seja alcançada, novas informações serão geradas através da separação. A esse mecanismo, o algoritmo dá o nome de saturação-1.

Exemplificando para o conjunto de triplas $\{(1, \neg a, a), (1, a, \neg a)\}$:

- fazendo $a = 1$: $\{(1, 0, 1), (1, 1, 0)\}$, onde $(1, 1, 0)$ é uma tripla terminal (ou falso) e esse desvio é uma contradição.

- fazendo $a = 0$: $\{(1, 1, 0), (1, 0, 1)\}$, onde $(1, 1, 0)$ é uma tripla terminal (ou falso) e esse desvio também é uma contradição.

- Ambos os desvios são contradições, portanto a fórmula é refutada.

Se, após a execução da saturação-1 sobre todas as variáveis, uma contradição não for alcançada, o algoritmo executa uma saturação-2, similar à saturação-1, porém são escolhidas duas variáveis. Para o conjunto Ψ, e as variáveis de separação v e w, teremos os novos conjuntos: $\Psi \cup \{v = 1, w = 1\}$, $\Psi \cup \{v = 1, w = 0\}$, $\Psi \cup \{v = 0, w = 1\}$, $\Psi \cup \{v = 0, w = 0\}$ e tomando a interseção dos resultados enquanto novas informações forem geradas. Similarmente o algoritmo pode executar uma saturação-n, onde n é um número natural. Para uma fórmula com n variáveis, o algoritmo necessita de no máximo saturação-n para determinar se ela é satisfazível ou não.

Devido à sua característica comercial, solucionadores SAT baseados no algoritmo de Stålmarck não são disponíveis em domínio público, dessa forma, poucas avaliações comparativas com solucionadores baseados em Davis-Putnam podem ser encontradas. Uma implementação apresentada por J. F. Groote, denominada HeerHugo [Gro00], sugere que verificadores de fórmulas proposicionais baseados no algoritmo de Stålmarck se mostram superiores aos baseados no algoritmo DPLL. Uma análise teórica do algoritmo de Stålmarck versus resolução, apresentada por Nordstrom, pode ser encontrada em [Nor01].

2.2.5 Algoritmos estocásticos

Todos os algoritmos descritos até o momento para a resolução de problemas SAT são completos, isto é, se tanto o tempo quanto o espaço (memória) forem suficientes, um algoritmo completo sempre encontrará uma atribuição às variáveis de uma instância se ela for satisfazível (SAT), ou provará que essa atribuição não existe se ela for insatisfazível (não-SAT).

Algoritmos estocásticos pertencem a uma classe diferente de solucionadores SAT baseados em otimização matemática. Métodos estocásticos não podem provar que um

determinado problema é não-SAT, mas dada uma instância satisfazível, solucionadores baseados nesses métodos podem encontrar uma atribuição às variáveis da instância. Algumas aplicações cujos resultados são, na grande maioria SAT, são, por exemplo, Planejamento em Inteligência Artificial [Kau99] [Pat04] [Kau96] ou roteamento de FPGAs [Nam99] [Pdu06] [Xia04]. Para algumas classes de problemas difíceis e aleatórios, métodos estocásticos podem ser mais rápidos que algoritmos baseados em uma pesquisa sistemática (ou baseados no algoritmo de DPLL), a principal razão é a própria aleatoriedade da instância.

A principal idéia de solução dos métodos estocásticos é considerar o número de cláusulas não satisfeitas como uma função objetivo e minimizar esse objetivo assinalando valores às variáveis. Diversos algoritmos e métodos utilizados para otimização combinatória em geral podem então ser utilizados nos solucionadores SAT. As principais contribuições vêm de: *simulated annealing* [Wil93] [Soh96], algoritmos genéticos [Ele99] [Kib06] [Fre06] [Are04], redes neurais [Wil96] [Ant00] e busca local [Jun93] [Bar92] [Dav06]. Dentre esses métodos, os algoritmos baseados na busca local mostraram-se de melhor desempenho para solucionadores SAT.

Trabalhos iniciais foram propostos por Selman, Levesque and Mitchell [Sel92] e Gu [Jun93] e, desde então, inúmeras variações teóricas [Pap91] e práticas [Hol99] [Dal01] podem ser encontradas.

As heurísticas são geralmente baseadas na crença de que não é necessário percorrer todo o espaço de busca até encontrarmos uma solução ótima. Podem existir pistas na vizinhança do espaço de busca atual onde poderemos encontrar uma solução.

As técnicas de pesquisa local para solução do problema da satisfabilidade usam uma estratégia de melhoria local, o algoritmo inicia com uma atribuição de valores às variáveis e essa atribuição é iterativamente alterada. Seja α um vetor coluna de tamanho n denotando as atribuições de valores às variáveis da instância $x_1, x_2, x_3, \ldots, x_n$ e $x_k = \{0, 1\}$. O algoritmo é iniciado com uma atribuição de valores α_0 e essa atribuição é iterativamente alterada. Uma função objetivo $C(\alpha)$ é utilizada para medir o valor de

uma atribuição α.

Como instâncias de problemas de satisfabilidade são representadas por um conjunto de cláusulas, ou restrições, sem nenhuma função explícita que possa ser otimizada, a função objetivo $C(\alpha)$ deve ser criada, para permitir a busca local. Existem diferentes funções objetivos, mas a mais comum e utilizada é aquela que utiliza o número de cláusulas não satisfeitas para uma determinada atribuição de valores às variáveis.

Seja n o número de variáveis em uma instância do problema, m o número de cláusulas e a função $U_i(\alpha) = 0$ se a cláusula c_i é satisfazível com uma atribuição α de valores às variáveis e $U_i(\alpha) = 1$ se a cláusula c_i não é satisfazível para a mesma atribuição. Consideraremos $U(\alpha)$ um vetor coluna com dimensão m que contém como coluna i o valor de $U_i(\alpha)$. Então a função objetivo C é:

$$C(\alpha) = \sum_{i=1}^{m} U_i(\alpha)$$

Através desta função objetivo, o problema SAT pode ser formulado como um problema de minimização, ou seja:

minimizar: $\sum_{i=1}^{m} U_i(\alpha)$

sujeito a : $\alpha_k \in \{0, 1\}$ para $k = 1, 2, \ldots, n$

Observa-se que podemos avaliar se uma atribuição das variáveis torna o problema satisfazível, testando se $C(\alpha) = 0$, já que isto significa que nenhuma cláusula é não satisfazível.

O passo seguinte em um método de busca local é decidir sobre a função vizinhança. O objetivo dessa função é fornecer o subconjunto de atribuições que serão os candidatos na iteração seguinte. Um pseudocódigo do método de pesquisa local pode ser visto na Figura 2.5.

Uma função vizinhança V recebe uma atribuição α e retorna um conjunto de novas atribuições. O método mais simples para essa função vizinhança é:

$$V(\alpha) = \{inverte(\alpha, k) \,|k \in \{1, \ldots, n\}\}$$

```
inicio
    inicializa α_0 ;
    k = 0;
    enquanto ( C(α_k) > 0 )
        V = V(α_k);
        α_{k+1} = seleciona(α_k, V);
        k = k + 1;
    fim enquanto
fim
```

Figura 2.5: Pseudocódigo para uma pesquisa local.

que representa o conjunto de atribuições α_k que diferem de α de exatamente uma variável. A função $inverte(\alpha, k)$, inverte a linha k no vetor de atribuições α, ou simplesmente inverte α_k.

Similarmente, duas atribuições α e β são ditas adjacentes se e só se elas diferem de exatamente uma variável: $adjacente(\alpha, \beta) \equiv \exists k \in \{1, \ldots, n\}$: $\beta = inverte(\alpha, k)$.

Uma outra maneira de expressar essa relação de adjacência é dizer que a distancia *Hamming* entre as atribuições de variáveis α e β é exatamente um, se α e β forem representadas como vetores de bits de tamanho n.

As transições realizadas em cada iteração do algoritmo podem ser descritas como: $\alpha_{k+1} = seleciona(\alpha_k, V(\alpha_k))$ onde a função $seleciona()$ encontra uma atribuição α_{k+1} da vizinhança $V(\alpha_k)$. A função $seleciona()$ pode basicamente trabalhar através de duas estratégias: usando uma estratégia gulosa ou uma estratégia de subida de montanha[7].

Em uma estratégia gulosa tomamos como a melhor atribuição sobre a anterior, aquela que fornece o menor valor da função objetivo, se Ψ representar todo o conjunto de atribuições adjacentes a α vem:

$$seleciona_{Gulosa}(\alpha, \Psi) = \alpha^*$$

onde, $\alpha^* \equiv (\alpha^* \in \Psi) \wedge (C(\alpha^*) \leq C(\alpha)) \wedge (\forall \beta \in \Psi : C(\beta) \geq C(\alpha^*))$

Se diversas atribuições possuírem um mesmo valor, a estratégia de escolha influencia o desempenho da procura. Estratégias mais comuns seguem uma ordem predefinida

[7] Hill climbing.

de escolha como, por exemplo, escolher a variável de menor índice ou escolher aleatoriamente.

A Figura 2.26 ilustra o algoritmo utilizado pelo solucionador GSAT, proposto por Selman, Lavesque e Mitchell [Sel92] e que utiliza uma estratégia gulosa. Valores iniciais são fornecidos às constantes Max-Flips e Max-Tries. Se, após Max-flips tentativas não chegarmos a uma solução satisfazível para a instância, reinicia-se o algoritmo com uma nova atribuição aleatória. A quantidade de reinicializações é dada por Max-Tries.

```
inicio
    r = 0;
    enquanto ( r < Max − Tries )
        inicializa α0 com uma atribuição aleatória de valores;
        k = 0;
        enquanto ( k < Max − Flips )
            se ( C(αk) = 0 )
                retorne ( SAT com αk );
            V = V(αk);
            αk+1 = seleciona_Gulosa(αk, V);
            k = k + 1;
        fim enquanto
        r = r + 1;
    fim enquanto
    retorne ( não-SAT );
fim
```

Figura 2.6: Pseudocódigo do solucionador GSAT.

Usando a estratégia de subida da montanha, tomamos a primeira variável em qualquer ordem que dê qualquer melhoramento. Shang e Wah [Sha98], alegam que essa estratégia oferece melhor resultado que a gulosa:

$$seleciona_{sobe-mon}(\alpha, \Psi) = \alpha^* \equiv (\alpha^* \in \Psi) \wedge (C(\alpha^*) < C(\alpha))$$

Freqüentemente, a trajetória da procura por um melhor valor para a função objetivo, pode estacionar em áreas onde não podemos encontrar um valor inferior ao já existente. Desde que o algoritmo se baseia na contínua procura por melhorias locais, estacionar em alguma região dessa natureza, ou um vale localmente ótimo, pode não

representar uma solução global para o problema [Fra97]; sair destas áreas é um desafio para os algoritmos de solucionadores SAT baseados em procura local.

Definimos um plano[8] como um conjunto P onde todas as atribuições possuem o mesmo valor para a função objetivo: $plano(P) \equiv \forall \alpha, \beta \in P : C(\alpha) = C(\beta)$. O nível de um plano P é o valor da função objetivo para uma atribuição arbitrária em P. A borda B de um plano P é o conjunto de atribuições que não estão no plano, mas que são adjacentes a uma atribuição que está no plano: $B(P) = \left(\bigcup_{p \in P} V(p) \right) - P$.

Um mínimo é um plano onde todas as atribuições na borda do plano possuem um valor para a função objetivo maior que o valor da função objetivo do plano: $minimo(P) \equiv \forall \alpha \in B(P) : C(\alpha) < C(B(P))$.

Um mínimo local é definido como um mínimo M_1 onde existe um mínimo M_2 com um nível inferior a M_1, $C(M_2) < C(M_1)$. Se um mínimo não é um mínimo local, ele é um mínimo global. Mínimos globais são fáceis de detectar já que $C(\alpha) = 0$ se e somente se α for um mínimo global.

Mínimos locais são mais freqüentemente encontrados na transição de fase de instâncias de problemas da satisfabilidade aleatórios difíceis [Lar93]. A transição de fase ocorre quando o número de cláusulas dividido pelo número de variáveis é aproximadamente 4.3.

Os principais representantes de solucionadores SAT baseados em pesquisa local são o GSAT [Sel92], WalkSAT [Dav97], UnitWalk [Hir05], QingTing [Xyl04], MFSAT [Mal03], Novelty [Dav04], UBCSAT [Dav05].

2.2.6 Principais solucionadores SAT modernos

Essa lista de solucionadores de problemas SAT foi obtida diretamente dos resultados das competições nos últimos cinco anos [Url03]. A maioria dos solucionadores aqui apresentados são de domínio público. Existem três modalidades de avaliação de acordo com o tipo de instâncias dos problemas apresentados: industriais, sint´ticos

[8] Plateau.

ou aleatórios. Essas categorias estão ainda distribuídas em instâncias exclusivamente satisfazíveis, não satisfazíveis e ambas. A lista a seguir não é exaustiva, apresenta os solucionadores que obtiveram os melhores resultados de cada ano e está dividida naqueles que são completos e incompletos. As referências apresentadas se referem aos locais onde poderemos encontrar os solucionadores. Alguns novos conceitos e termos são aqui mencionados; eles serão detalhadamente descritos no próximo capítulo, que trata das principais inovações no algoritmo DPLL.

Solucionadores completos

GRASP [Url04]

Apesar de não participar das últimas competições de SAT, é um dos primeiros solucionadores a apresentar as principais inovações existentes nos atuais solucionadores e é continuamente mencionado. O solucionador é baseado no algoritmo DPLL e foi concluído em 1996. Introduziu o conceito de grafo de implicações e a estratégia de aprendizado de cláusulas, que permite traduzir as atribuições de variáveis que levam a um conflito através da geração de uma nova cláusula. Essa nova cláusula, quando combinada com as cláusulas originais do problema, evita percorrer todo um caminho de implicações já anteriormente percorrido. Essa idéia se relaciona diretamente com o conceito de retrocesso cronológico e não cronológico. Apresentou uma heurística de decisão estática para as variáveis baseada na contagem de ocorrência de literais, e que muito contribuiu para as heurísticas atuais.

zChaff [Url05]

O solucionador zChaff é baseado no algoritmo de DPLL. Introduziu o conceito de literais observados, uma estrutura de dados que permite identificar mais rapidamente quando uma cláusula se torna unitária acelerando o BCP. Apresentou uma heurística de decisão baseada na atividade de uma variável. A atividade de uma variável reflete a quantidade de vezes em que uma determinada variável está envolvida em um conflito.

Um terceiro aspecto foi uma nova técnica de aprendizado de cláusulas, conforme o solucionador GRASP, porém, gerando cláusulas menores durante a análise de conflito. O solucionador zChaff, inicialmente em 2001, apresentou uma a duas ordens de grandeza na redução do tempo necessário para a solução de instâncias de problemas. Venceu a competição de SAT em 2002 e em 2004, na categoria industrial e em 2005 ficou classificado em terceiro lugar. A versão utilizada em 2004 incorporou algumas características mencionadas no solucionador Berkmin.

OKSolver [Url06]

O solucionador OKSolver recebeu o título de melhor solucionador na competição de SAT de 2002 na categoria de solucionador de instâncias aleatórias. É um algoritmo baseado no DPLL e a principal contribuição foi o conceito de autarquia. Uma autarquia é um conjunto de literais tais que, toda cláusula que possui a negação de um literal desse conjunto, também possui algum outro literal desse conjunto. A heurística de decisão seleciona uma variável que, após a aplicação do BCP, produzirá a maior quantidade de cláusulas com dois literais quanto for possível. O resolvedor ainda tem um desempenho satisfatório nas outras categorias de instâncias, conforme observado nos resultados da competição.

Kcnfs [Url07]

O solucionador kcnfs é um solucionador baseado em DPLL. Introduziu uma heurística de procura onde a decisão sobre uma variável é baseada no conceito de espinha dorsal de uma fórmula CNF. Uma variável pertence à espinha dorsal da fórmula CNF se possuir o mesmo valor em todas as atribuições que tornam satisfazíveis um maior número de cláusulas. O solucionador recebeu o prêmio de melhor solucionador na competição de SAT em 2003 e 2005 para a categoria de instâncias aleatórias e, de acordo com os criadores, foi otimizado para ter um bom desempenho nessa categoria.

March [Url08]

Esse solucionador também é baseado no algoritmo DPLL. Foi o melhor classificado na categoria de problemas construídos em 2004 e o terceiro em 2005. Uma das principais contribuições é o uso de uma heurística que olha para frente[9] para a escolha da variável de decisão. Essa heurística seleciona um pequeno grupo de variáveis através de um pré-processamento. Possui uma estrutura de dados que permite a tradução de cláusulas maiores para um formato contendo três variáveis por cláusula, o que permite uma uniformização das estruturas de dados.

Berkmin [Url09]

É um solucionador baseado em DPLL. O solucionador venceu a competição de SAT na categoria de instâncias sintéticas em 2002 e em 2003 venceu a categoria industrial competindo com o nome de Forklift, este último de código não disponível para a comunidade. O solucionador incorpora a idéia de reinicialização, ou seja, permite abandonar certas regiões de procura que, através de diferentes métodos de avaliação, estão demandando um tempo maior para a sua computação. Apresenta ainda uma nova idéia no armazenamento e na administração das cláusulas introduzindo e identificando as cláusulas que foram criadas a partir do aprendizado de cláusulas. Criou o conceito de atividade e idade de uma cláusula aprendida. Esses conceitos permitem que as mesmas possam ser removidas de acordo com a sua utilização, mantendo um banco de dados de cláusulas reduzido e mais dinâmico.

Satzoo [Url10]

Esse solucionador foi construído basicamente a partir do solucionador zChaff e venceu a competição de SAT, na categoria de instâncias construídas, em 2003.Utiliza a idéia dos literais observados, para o BCP mais rápido, e incorpora idéias do Berkmin no que se refere à escolha de variáveis oriundas de cláusulas geradas pelo aprendizado de cláusulas. Também utiliza o conceito de atividade da cláusula. O algoritmo de procura permite incorporar as heurísticas para a escolha de variáveis do Zchaff e Berkmin

[9] Lookahead.

simultaneamente. Implementa uma ordenação inicial estática das variáveis, isso altera a ordem inicial das cláusulas interferindo no tempo de solução final do algoritmo. Essa ordenação procura colocar variáveis encontradas mais freqüentemente juntas em diferentes cláusulas próximas umas das outras.

SatElliteGTI (Minisat) [Url11]

Esse solucionador é uma evolução do Satzoo. Na competição de SAT de 2005 foi o melhor classificado na categoria industrial. Os ajustes realizados nessa versão incluem uma alteração no controle de variáveis; periodicamente a atividade de uma variável é reduzida, esse solucionador procurou atingir um valor nessa redução melhor adaptado aos problemas submetidos. Introduziu uma lista separada de cláusulas de tamanho dois, para tornar o BCP mais rápido. Introduziu uma estratégia de remoção de cláusulas aprendidas capaz de retirar uma quantidade maior de cláusulas que os antecessores; a estratégia atua em problemas pequenos, mas difíceis, e se mostrou eficiente. A maior contribuição do solucionador é uma estratégia de simplificação do problema que procura cláusulas que possam ser subjugadas por outras e, conseqüentemente, diminuir o tamanho do problema. Esse mecanismo de simplificação pode, ainda, ser importado para outros solucionadores.

Solucionadores incompletos

UnitWalk [Url12]

Esse solucionador combina procura local e eliminação de cláusulas unitárias. O algoritmo consiste de períodos onde atribuições ocorrem. Em cada período uma permutação aleatória é implementada e se cláusulas unitárias forem encontradas, uma delas é selecionada e ao literal correspondente da cláusula, é atribuído o valor que torne a variável verdadeira. Se nenhuma cláusula for encontrada, escolhe-se a primeira variável livre da permutação construída. O solucionador obteve o melhor resultado na competição de SAT de 2003 na categoria de problemas aleatórios.

Adaptnovelty+ [Url13]

Esse solucionador obteve a melhor classificação na categoria de problemas aleatórios em 2004. Ele apresenta duas principais contribuições para os algoritmos de procura local. O algoritmo aleatoriamente escolhe um literal. Dentre as cláusulas onde esse literal aparece complementado, isto é, em que ele não irá contribuir para tornar a cláusula satisfazível, ele irá escolher outro literal. Para essa escolha, o algoritmo cria o conceito de escore para uma variável. Esse escore representa a diferença da quantidade de cláusulas que podem se tornar satisfazíveis se o valor de um determinado literal for complementado ou não. Esse será o novo literal a ser escolhido. Outra contribuição refere-se a um parâmetro denominado ruído. Esse parâmetro é o responsável por controlar o grau de aleatoriedade no processo da procura local e interfere diretamente no desempenho do solucionador. O solucionador dinamicamente altera esse parâmetro, de acordo com a evolução da procura e a estagnação em algum mínimo local, usando como avaliação, a quantidade de cláusulas que se tornaram satisfazíveis.

Capítulo 3

Principais avanços no algoritmo DPLL

Um dos aspectos surpreendentes dos progressos em termos práticos dos algoritmos completos atualmente encontrados para a solução do problema da satisfabilidade é que a maioria é, de alguma forma, descendente do algoritmo de Davis-Logemann-Loveland ou DPLL, de 1962.

Como visto no capítulo anterior, a principal idéia do algoritmo, responsável pela sua eficácia, é a capacidade de reduzir o espaço de procura. O algoritmo basicamente possui três pontos que determinam a sua eficiência: a determinação de qual a variável a ser escolhida a cada nível de decisão, como executar o retrocesso caso uma cláusula vazia seja encontrada e quais as regras a serem aplicadas para a dedução das atribuições em um determinado nível. Desde a sua introdução, o algoritmo tem sofrido melhoramentos nesses três principais mecanismos, tanto na sua versão original quanto nas versões mais atuais:

- o mecanismo de análise dos conflitos;
- o mecanismo de decisão;
- o mecanismo de dedução.

Esse capítulo é um resumo das principais inovações em cada um desses mecanismos.

3.1 A análise de conflitos

Um dos mais poderosos melhoramentos introduzidos no DPLL básico foi a capacidade de analisar e identificar o conjunto de atribuições que gerou um conflito. Esse conjunto irá propiciar a geração de uma nova cláusula e que será incorporada às cláusulas originais. A esse processo foi dado o nome de aprendizado de cláusulas e, às cláusulas geradas a partir desse conjunto, de cláusulas aprendidas.

De acordo com Beame em [Bea04], o aprendizado de cláusulas é considerado uma das características mais importantes para permitir que o algoritmo DPLL trate problemas reais. Diferentes pesquisadores [Bay97] [Sil98] [Zha97] [Zha01] [Mos01] [Een03] demonstraram que o aprendizado de cláusulas pode ser eficientemente implementado e utilizado na solução de problemas difíceis os quais, dificilmente, poderiam ser resolvidos através de outras técnicas.

De uma forma geral, o aprendizado de cláusulas pode ser incorporado ao algoritmo DPLL conforme ilustrado na Figura 3.1, onde, através de uma simplificação do algoritmo DPLL, podemos visualizar como o aprendizado de cláusulas será inserido.

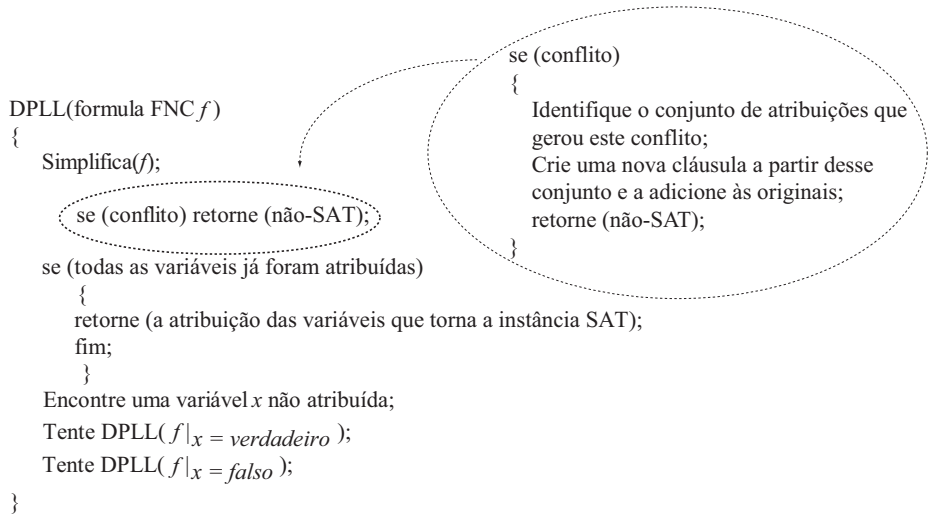

Figura 3.1: DPLL com e sem inserção do aprendizado de cláusulas

Para melhor compreender os diferentes mecanismos de análise de conflitos e aprendizado de cláusulas, alguns conceitos sobre grafo de implicação serão fornecidos. Para um detalhamento maior sobre o assunto o leitor é convidado a consultar a tese de

doutoramento de Marques e Silva, apresentada resumidamente em [Sil96]; as mesmas notações serão aqui utilizadas.

Um grafo de implicação expressa os relacionamentos entre as variáveis durante a execução de um algoritmo solucionador de uma instância do problema da satisfabilidade. Tipicamente um grafo de implicação I é definido como:

- Cada vértice em I corresponde à atribuição de um valor a uma variável em um determinado nível, $x = v@n$, onde x representa a variável, $v \in \{0, 1\}$ representa o valor e $n \geq 1$ representa o nível em que a variável x recebeu o valor.

- Os predecessores do vértice $x = v@n$ em I, são os relativos às atribuições que causaram a implicação do valor de x, produzindo a cláusula unitária (x) ou $(\neg x)$. As arestas direcionadas partindo de um antecedente de x até o vértice $x = v@n$ são rotuladas com a cláusula que originou a implicação do valor de x. Vértices que não possuem predecessores correspondem às atribuições de decisão.

- Um conflito, ou uma contradição, é identificado quando os dois valores são atribuídos à mesma variável. Nesse caso, a variável em questão é denominada variável de conflito.

A Figura 3.2 ilustra a instância ϕ, composta de quatro cláusulas, e o seu grafo de implicação. Observa-se que foram necessárias duas decisões, representadas pelos dois níveis na árvore de decisão, cujas variáveis de decisão são identificadas no grafo de implicação como os vértices mais à esquerda, sem preenchimento. Através das duas decisões $x_1 = 1@1$ e $x_4 = 1@2$, os valores das demais variáveis da instância que tornam a fórmula ϕ verdadeira (SAT) foram determinados através de implicação (BCP).

A Figura 3.3 ilustra a instância φ, composta de dez cláusulas. Para o terceiro nível de decisão, onde a variável de decisão é x_7, com o vértice de atribuição $x_7 = 1@3$, um conflito foi identificado. A variável de conflito é x_{12}, identificada pela necessidade das atribuições $x_{12} = 1@3$ e $x_{12} = 0@3$ através das cláusulas w_{10} e w_8 respectivamente. Esse vértice está sendo ilustrado na Figura 3.3 através da coloração mais clara.

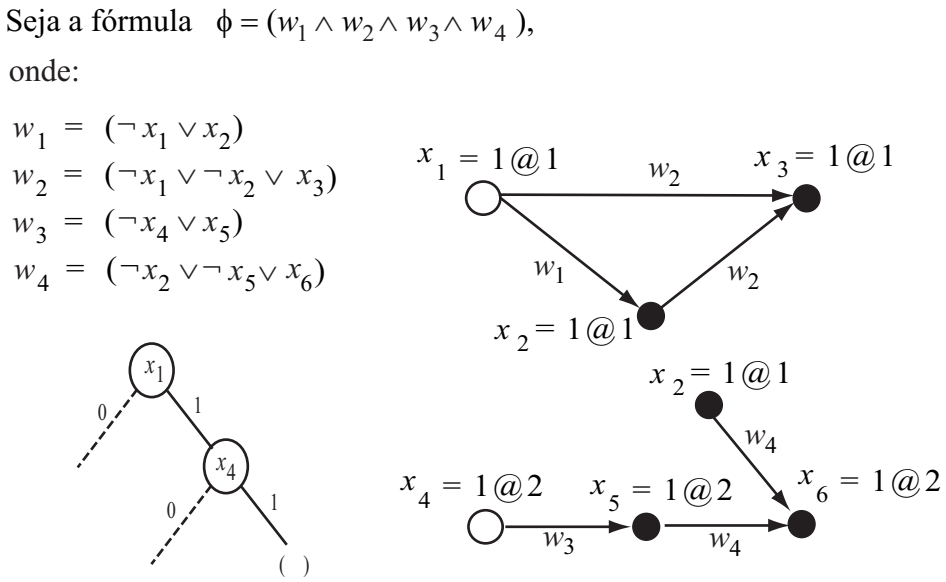

Figura 3.2: Instância de um problema SAT e seu grafo de implicação

Em um grafo de implicação, um vértice a é dito dominar um vértice b em um nível n, se qualquer caminho a partir da variável de decisão do nível n até b passar, obrigatoriamente, por a. Exemplificando para cada nível na Figura 3.3 vem:

- no nível 1, o vértice $x_1 = 1@1$ domina $x_3 = 1@1$ e $x_2 = 1@1$;
- no nível 2, o vértice $x_4 = 1@2$ domina $x_5 = 1@2$ e $x_6 = 1@2$, e o vértice $x_5 = 1@2$ domina $x_6 = 1@2$;
- no nível 3, o vértice $x_7 = 1@3$ domina $x_8 = 1@3$, $x_9 = 1@3$, $x_{10} = 1@3$, $x_{11} = 1@3$ e o conflito ($x_{12} = 1@3$ e $x_{12} = 0@3$). O vértice $x_{10} = 1@3$ domina $x_{11} = 1@3$ e o conflito.

Um ponto de implicação único (UIP)[1] [Sil97a] é um vértice no nível de decisão corrente que domina o vértice correspondente à variável de conflito. Na Figura 3.3, os UIPs são $x_{10} = 1@3$ e $x_7 = 1@3$, pois ambos dominam a variável de conflito.

A variável de decisão é sempre um UIP e podemos ter mais do que um UIP para um determinado conflito. Intuitivamente, um UIP é uma causa simples do conflito no

[1] *Unique Implication Point.*

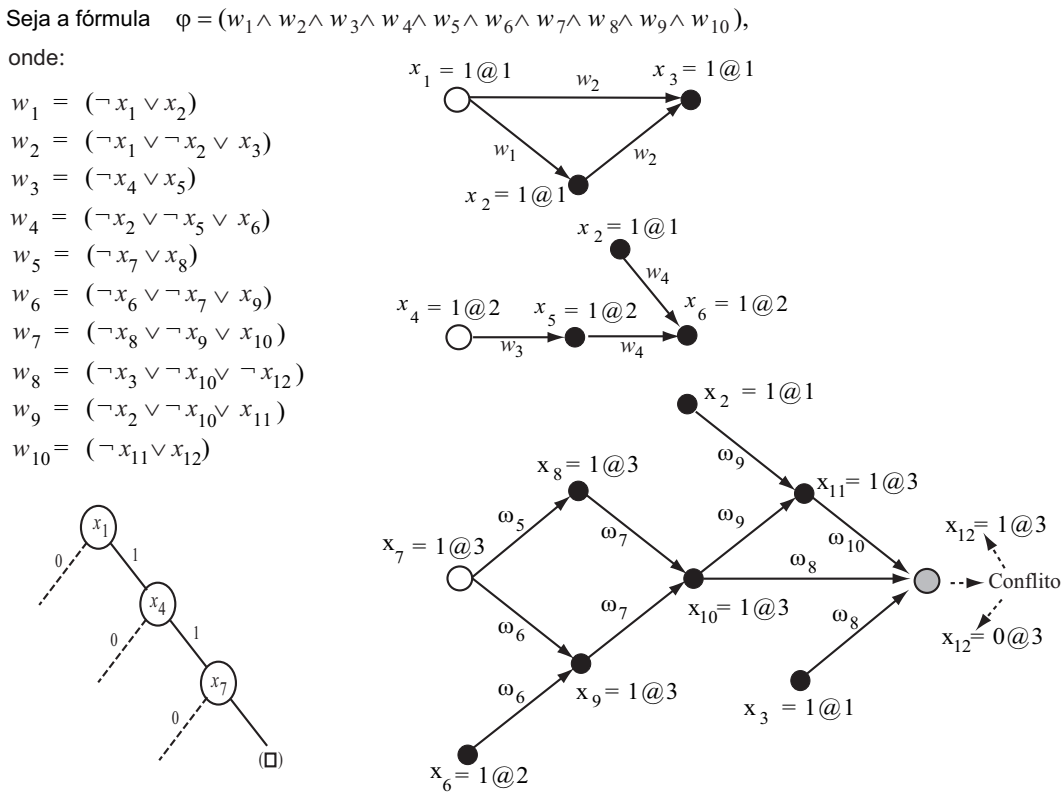

Figura 3.3: Instância de um problema SAT e seu grafo de implicação em um nível com conflito

nível de decisão. Os UIPs são ordenados a partir da variável de conflito, assim para os dois UIPs da Figura 3.3, o vértice $x_{10} = 1@3$ é o p*rimeiro UIP* (ou 1 UIP).

A análise de conflitos é o mecanismo que encontra causas de um conflito e identifica que naquele determinado espaço de procura não existe uma atribuição que torne a cláusula satisfazível. Além disto, procura indicar um novo espaço de procura.

O algoritmo DPLL original apresenta um método de análise simples de conflitos. Para cada variável de decisão existe um indicador informando se os dois valores possíveis (verdadeiro ou falso) já foram atribuídos à variável de decisão. Quando ocorre um conflito, o mecanismo de análise de conflitos verifica qual a variável de decisão com o nível mais alto de decisão à qual não foram atribuídos os dois possíveis valores. Marca esta variável, desfaz todas as atribuições realizadas entre esse nível e o nível onde o conflito ocorreu e constrói novas atribuições a partir do novo valor. A esse método,

damos o nome de retrocesso cronológico (CB)[2] . Se a instância não possuir nenhum tipo de estrutura, por exemplo, for completamente aleatória, intuitivamente pode-se concluir que o aprendizado de cláusulas terá pouca influência e esse tipo mais simples de retrocesso terá resultados satisfatórios.

As técnicas mais avançadas para a análise de conflitos fazem uso do grafo de implicações para a determinação do motivo do conflito. Essas técnicas, através do caminhamento dentro do grafo de implicações, constroem novas cláusulas a serem adicionadas às cláusulas originais. O retrocesso proporcionado permite que um nível anterior ao retrocesso cronológico seja realizado. A esse retrocesso damos o nome de retrocesso não cronológico (NCB)[3] [Bay97]. A Figura 3.4 ilustra esses dois tipos de retrocesso em uma árvore de decisão.

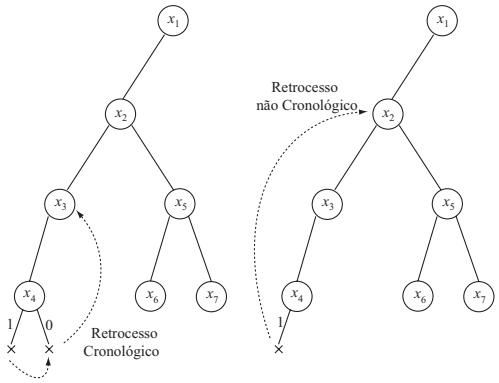

Figura 3.4: Retrocesso Cronológico e não Cronológico

Ao processo de criação de cláusulas a partir do grafo de implicações damos o nome de *aprendizado de cláusulas* e, coerentemente, às cláusulas que forem adicionadas às cláusulas originais damos o nome de *cláusulas aprendidas*. Essas cláusulas evitam que os mesmos erros sejam cometidos em uma procura posterior. Elas indicam que uma determinada combinação de atribuições de valores às variáveis não é válida, uma vez que irá forçar a variável de conflito a assumir dois valores (verdadeiro e falso) e, conseqüentemente, gerar um conflito no grafo de implicações.

[2] *Chronological Backtracking.*

[3] *Nonchronological Backtracking.*

Uma cláusula aprendida pode ser gerada a partir da bipartição do grafo de implicações. Esta partição irá colocar todos os vértices de atribuições de decisão em um lado (lado razão) e os vértices com atribuições das variáveis de conflito no outro lado (lado conflito). Todos os vértices do lado razão possuem no mínimo uma aresta para o lado conflito. Diferentes bipartições representam diferentes técnicas de análise de conflitos e, conseqüentemente, diferentes cláusulas aprendidas. Um corte deve ser selecionado de forma a tornar efetivo o aprendizado das cláusulas, melhorando o desempenho do algoritmo. A cláusula aprendida pode ser criada usando-se os complementos das variáveis do lado razão com as arestas interceptando o corte. Em adição ao aprendizado da cláusula, o solucionador deverá, através da cláusula aprendida, executar um retrocesso não cronológico. O nível de decisão destino para esse retrocesso será escolhido dentre os níveis de atribuição das variáveis dessa cláusula aprendida. Dessa forma, para um retrocesso mais longo, escolhemos como destino o nível de atribuição mais distante do nível atual dentre todas as variáveis presentes na cláusula aprendida.

Uma característica importante na cláusula aprendida é que ela possua apenas uma variável livre, isto é, todas as variáveis já possuam valores anteriormente atribuídos com exceção de uma. Essa situação faz com que, ao realizarmos o retrocesso, essa cláusula se torne uma cláusula unitária e seja automaticamente implicada no BCP.

Tomemos o mesmo grafo da Figura 3.3, agora reapresentado na Figura 3.5, onde três partições foram realizadas e identificadas pelos cortes 1, 2 e 3. As identificações das arestas, que correspondem às cláusulas do problema, foram retiradas; entretanto, elas podem ser reconstruídas a partir do restante do grafo de implicações, basta verificarmos quais foram os valores atribuídos para cada variável. Por exemplo, na Figura 3.5, o vértice $x_9 = 1@3$, possui duas arestas incidentes e que são resultantes de duas atribuições $x_7 = 1@3$ e $x_6 = 1@2$, portanto a cláusula antecedente para x_9 será: $(\neg x_6 \vee \neg x_7 \vee x_9)$, que corresponde a w_6, se retornarmos ao conjunto de cláusulas da Figura 3.3.

Podemos então, a partir de diferentes cortes, criar mais cláusulas. Se considerarmos o corte 1, mais próximo do conflito, identificamos os seguintes vértices do lado razão:

$x_2 = 1@1$, $x_{10} = 1@3$ e $x_3 = 1@1$. Isso significa que a veracidade de $(x_2 \wedge x_{10} \wedge x_3)$ leva a um conflito; assim, para evitar o conflito, deve-se fazer $\neg(x_2 \wedge x_{10} \wedge x_3)$, o que pode ser realizado adicionando-se a cláusula aprendida: $(\neg x_2 \vee \neg x_3 \vee \neg x_{10})$. De maneira similar, o corte 2 gera a seguinte cláusula aprendida: $(\neg x_2 \vee \neg x_3 \vee \neg x_6 \vee \neg x_7)$. O corte 3 não envolve o conflito, não possui um lado razão e nem um lado conflito.

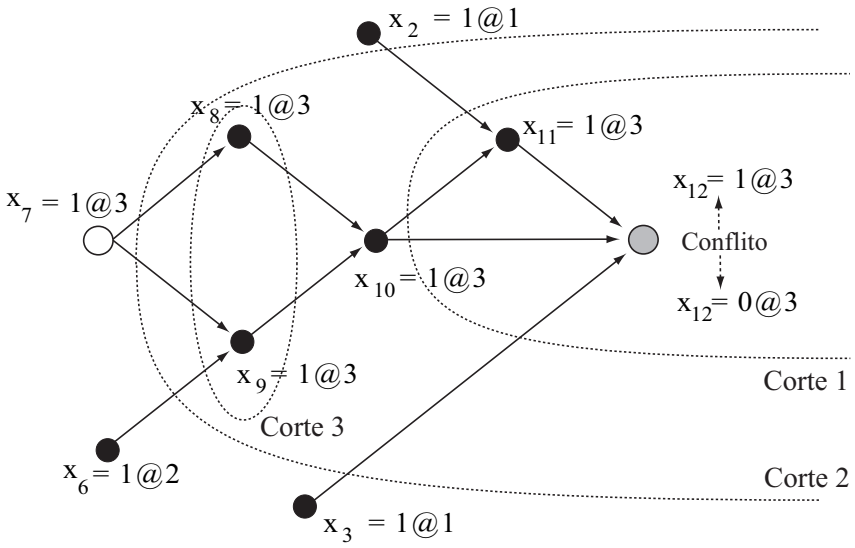

Figura 3.5: Grafo de implicação com três bipartições diferentes.

A Figura 3.6 ilustra o efeito proporcionado pela adição da cláusula $w_{11} = (\neg x_2 \vee \neg x_3 \vee \neg x_6 \vee \neg x_7)$, resultante do corte 2 da Figura 3.5, às cláusulas originais, após a análise de conflitos. O algoritmo, após a identificação de um conflito, desfaz as atribuições no nível 3 e insere a nova cláusula. Esse é o momento do retrocesso, ou seja, retornamos ao nível 2, porém com a adição de uma nova cláusula ao grupo de cláusulas originais. Utilizando a Figura 3.3, vemos que até nesse nível 2, as variáveis x_1, x_2, x_3, x_4, x_5 e x_6 já foram atribuídas, o que leva a nova cláusula adicionada a ser uma cláusula unitária e o literal livre é exatamente a fase contrária da variável de decisão identificada no conflito anterior, no caso x_7. Uma vez que é uma cláusula unitária, ela é automaticamente implicada pelo BCP. Essa situação corresponde ao comportamento do algoritmo DPLL original verificando as duas fases da variável, o que denominamos retrocesso cronológico.

Seja a fórmula $\varphi = (w_1 \wedge w_2 \wedge w_3 \wedge w_4 \wedge w_5 \wedge w_6 \wedge w_7 \wedge w_8 \wedge w_9 \wedge w_{10})$,
onde:

$w_1 = (\neg x_1 \vee x_2)$
$w_2 = (\neg x_1 \vee \neg x_2 \vee x_3)$
$w_3 = (\neg x_4 \vee x_5)$
$w_4 = (\neg x_2 \vee x_5 \vee x_6)$
$w_5 = (\neg x_7 \vee x_8)$
$w_6 = (\neg x_6 \vee \neg x_7 \vee x_9)$
$w_7 = (\neg x_8 \vee \neg x_9 \vee x_{10})$
$w_8 = (\neg x_3 \vee \neg x_{10} \vee \neg x_{12})$
$w_9 = (\neg x_2 \vee \neg x_{10} \vee x_{11})$
$w_{10} = (\neg x_{11} \vee x_{12})$

Cláusula aprendida:
$w_{11} = (\neg x_2 \vee \neg x_3 \vee \neg x_6 \vee \neg x_7)$

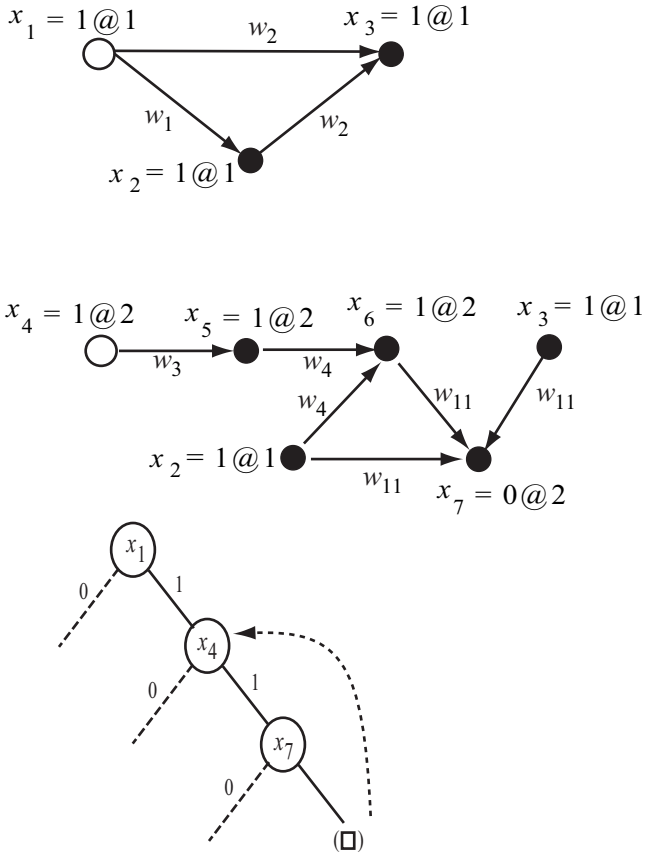

Figura 3.6: Cláusula aprendida resultando em um retrocesso cronológico

Se utilizarmos o corte 1 da Figura 3.5, que resultou na cláusula aprendida $(\neg x_2 \vee \neg x_3 \vee \neg x_{10})$, o efeito proporcionado é similar. Analisando o grafo de implicações, vemos que x_2 foi atribuída no nível 1, x_3 foi atribuída no nível 1 e x_{10} foi atribuída no nível 3. Como o conflito ocorreu no nível 3 e, dentre as variáveis da cláusula aprendida, o nível de atribuição mais distante do atual é o nível 1, o retrocesso será para esse nível. Esse mecanismo, denominado retrocesso não cronológico, é ilustrado na Figura 3.7. Ao retornarmos ao nível 1, identificamos as atribuições das variáveis x_1, x_2 e x_3. Com a cláusula aprendida adicionada ao conjunto de cláusulas, vemos que a variável x_{10} será automaticamente implicada no BCP. A Figura 3.7 ilustra esse retrocesso ao nível 1 e a implicação de x_{10}, observa-se que com a fase oposta àquela detectada no grafo de implicações do conflito.

Através de diferentes cortes cláusulas adicionais, consistentes com as cláusulas ori-

Seja a fórmula $\varphi = (w_1 \wedge w_2 \wedge w_3 \wedge w_4 \wedge w_5 \wedge w_6 \wedge w_7 \wedge w_8 \wedge w_9 \wedge w_{10})$,
onde:

$w_1 = (\neg x_1 \vee x_2)$
$w_2 = (\neg x_1 \vee \neg x_2 \vee x_3)$
$w_3 = (\neg x_4 \vee x_5)$
$w_4 = (\neg x_2 \vee x_5 \vee x_6)$
$w_5 = (\neg x_7 \vee x_8)$
$w_6 = (\neg x_6 \vee \neg x_7 \vee x_9)$
$w_7 = (\neg x_8 \vee \neg x_9 \vee x_{10})$
$w_8 = (\neg x_3 \vee \neg x_{10} \vee \neg x_{12})$
$w_9 = (\neg x_2 \vee \neg x_{10} \vee x_{11})$
$w_{10} = (\neg x_{11} \vee x_{12})$

Cláusula aprendida:
$w_{11} = (\neg x_2 \vee \neg x_3 \vee \neg x_{10})$

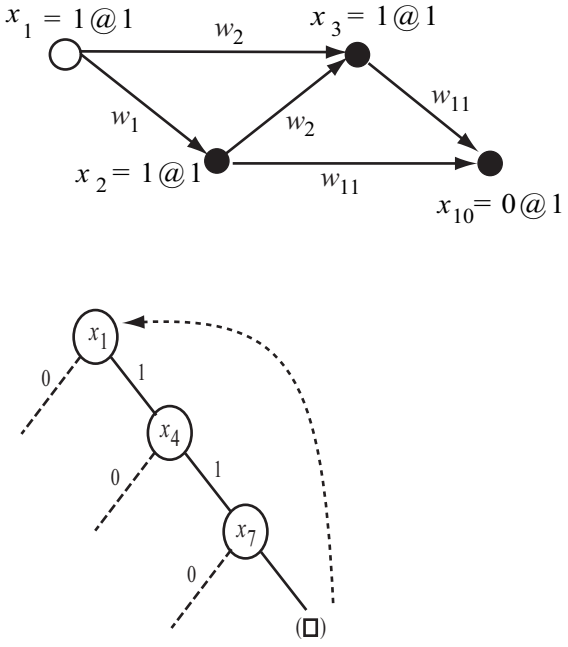

Figura 3.7: Cláusula aprendida resultando em um retrocesso não cronológico

ginais, também podem ser criadas. De uma forma mais clara isto é ilustrado pelo corte 3 na Figura 3.5 que envolve as variáveis x_8 e x_9. Com esse procedimento obtemos a cláusula: $(\neg x_6 \vee \neg x_7 \vee x_{10})$. Esta cláusula também pode ser utilizada como uma cláusula aprendida e adicionada às cláusulas originais. Entretanto, para que essa cláusula tenha alguma utilidade prática, deve existir algum tipo de repetição desta parte do grafo envolvida neste aprendizado. Se essa parte do grafo for apenas uma estrutura de árvore, então toda a informação dessa cláusula aprendida já estará presente no conjunto de cláusulas originais e ela será praticamente inútil. Se, no entanto, essa parte do grafo corresponder, por exemplo, a alguma estrutura presente em diversas partes da fórmula original, a cláusula aprendida será eficiente.

Uma prova que o aprendizado de cláusulas mantém o algoritmo DPLL completo, pode ser encontrada em [Bap00].

3.1.1 Diferentes estratégias de aprendizado de cláusulas

Corte último UIP

Esse corte é implementado no solucionador SAT Rel-SAT [Bay97]. Nesta estratégia, todas as atribuições de variáveis no nível de decisão analisado são colocadas no lado conflito e a variável de decisão e as atribuições de níveis anteriores no lado razão. Na Figura 3.5 é ilustrado como Corte 2. Através deste método, o retrocesso ocorrerá para o nível de decisão, dentre as variáveis da cláusula aprendida, mais próximo do nível atual. Desta forma, ao efetuarmos um retrocesso, a cláusula aprendida possuirá apenas a variável de decisão do nível onde o conflito foi detectado, como variável livre para a atribuição e, conseqüentemente, será atribuída com a fase invertida, conforme vimos.

Esta é uma importante característica para os vértices UIP, ela influencia o mecanismo de decisão descrito nesse trabalho e será detalhada nas próximas seções. Quando executamos a partição de forma que um UIP do nível de decisão analisado fique do lado razão e todos os vértices posteriormente assinalados no lado conflito, a cláusula aprendida gerada possuirá a variável deste vértice UIP como a única variável livre e será automaticamente implicada com a fase oposta àquela anteriormente atribuída.

Mecanismos GRASP de aprendizado de cláusulas

Um outro mecanismo de aprendizado de cláusulas, proposto por Marques-Silva [Sil96] no solucionador GRASP, procura aprender o máximo possível de um conflito. Uma primeira estratégia é procurar padrões semelhantes ao corte 3 da Figura 3.5 entre os UIPs do nível de decisão onde o conflito foi detectado, construir as cláusulas de acordo com esses cortes e adicioná-las às cláusulas originais. Na Figura 3.5, o corte 3 irá adicionar a cláusula $(\neg x_6 \vee \neg x_7 \vee x_{10})$. Além disto um segundo corte é realizado correspondendo ao primeiro UIP (Corte 1 da Figura 3.5), onde todas as variáveis do nível de decisão após o 1UIP são colocadas do lado conflito, o restante do lado razão. Como vimos, a cláusula adicionada é $(\neg x_2 \vee \neg x_3 \vee \neg x_{10})$. Ao ser realizado o retrocesso, o literal $\neg x_{10}$ será automaticamente implicado e o que se observa é que a variável x_{10}

não é uma variável real de decisão. A esse caso o GRASP dá o nome de "*decisão falsa*", o nível de decisão de x_{10} permanece inalterado, uma vez que ele ainda não foi concluído e o literal $\neg x_{10}$ é implicado.

Se novamente um conflito for encontrado, então um segundo modo de operação é executado e, além das cláusulas aprendidas no primeiro modo, uma nova cláusula é adicionada às cláusulas originais. O corte para esta cláusula procura ver a variável de decisão do nível como uma "*decisão falsa*" ou seja, vamos supor que no exemplo da Figura 3.5 uma cláusula antecedente de um nível de decisão anterior $(\neg x_7 \vee x_{14} \vee x_{15})$ estivesse presente, a nova cláusula gerada seria $(x_{14} \vee x_{15} \vee \neg x_2 \vee \neg x_3 \vee \neg x_6)$, que contém apenas variáveis atribuídas em níveis anteriores ao nível atual, esse mecanismo propicia um retorno não cronológico a níveis anteriores ao nível atual.

Mecanismo Primeiro UIP

Esta é a estratégia adotada pelo solucionador SAT zChaff [Mos01]. O autor procura relacionar a relevância da cláusula aprendida com o conflito, executando o corte o mais próximo do conflito quanto se possa aliado ao fato que, estando mais próximo do conflito, cláusulas menores serão aprendidas. Como a cláusula aprendida deverá possuir apenas uma variável livre que será implicada, um UIP deverá estar presente no lado razão. Somando-se estas duas características obtemos o corte 1UIP ilustrado na Figura 3.5. Todas as variáveis atribuídas após o primeiro UIP e com caminhos para o conflito estão no lado conflito , o restante no lado razão.

Outras técnicas de corte

- **Técnica do corte Decisão**: esse é um esquema simples onde a cláusula aprendida possui apenas as variáveis de decisão envolvidas no conflito, o que corresponde a um corte onde apenas essas variáveis de decisão estejam do lado razão e todas as outras do lado conflito. Observa-se que, se colocarmos todas as variáveis de decisão da árvore de procura na cláusula aprendida, esta cláusula dificilmente será útil para simplificar a procura, uma vez que esta combinação de decisões

dificilmente ocorrerá novamente no futuro, a não ser que uma reinicialização aconteça.

- **1UIP, 2UIP, 3UIP, etc**: A idéia aqui é similar ao primeiro UIP. Nela os UIPs subseqüentes são os responsáveis pela geração das cláusulas. De acordo com esta ordenação, a variável de decisão é o último UIP.

- **Técnica primeiro novo corte (*FirstNewCut*)**: Esta é uma técnica proposta por Beam em [Bea04] e utilizada para mostrar a efetividade do aprendizado de cláusulas em problemas específicos. O corte é realizado o mais próximo possível do conflito, na figura 3.5 o corte teria início englobando as atribuições $x_{12} = 0@3$ e $x_{12} = 1@3$, e, de forma iterativa, vai se movendo no sentido da variável de decisão. Existem diversas maneiras de se achar novos literais para a realização do corte, o que torna o método não redundante já que sempre irá aprender uma cláusula ainda não conhecida. O trabalho apresenta alguns resultados sem, no entanto, executar o algoritmo sobre um conjunto de instâncias industrial do problema da satisfabilidade.

3.1.2 Conclusão

O trabalho apresentado por Zhang em [Zha01] mostra diferentes técnicas de particionamento que foram implementadas no solucionador zChaff . Contrariamente à crença geral de que técnicas de particionamento que geram cláusulas aprendidas pequenas geralmente são superiores a outros mecanismos, a técnica do 1UIP se mostrou a mais eficiente. A técnica original GRASP, que procura aprender o máximo possível do conflito criando diversas cláusulas aprendidas a cada conflito, consegue aprender mais cláusulas e o solucionador SAT consegue um número menor de desvios para se chegar à solução, entretanto, uma vez que mais cláusulas são aprendidas, o processo de BCP requer mais tempo, o que piora o tempo total de solução. Aprender cláusulas indiscriminadamente pode, eventualmente, aumentar o tempo de BCP, além de aumentar

significativamente o conjunto de cláusulas originais.

Com os problemas atuais de SAT, o número de literais das cláusulas aprendidas tem crescido. Para alguns exemplos encontramos cláusulas aprendidas com milhares de literais e conseqüentemente também aumentando a quantidade de cláusulas originais. Isto faz com que a utilização da memória limite o processo de solução do problema.

Outra característica interessante é que diferentes técnicas são mais ou menos efetivas de acordo com o tipo de problema a que são submetidas. Além disso, a eficiência do aprendizado de cláusulas está também relacionada com as heurísticas aplicadas ao BCP.

Uma técnica que se mostrou eficiente é a reinicialização [Bap00] [Bap01] que, aliada ao aprendizado de cláusulas, é geralmente eficiente. Uma questão presente, e que motiva investigações, é a dificuldade quanto à implementação prática do algoritmo de reinicialização, tendo-se em conta o não determinismo relativo à decisão de "quando" ocorrer a reinicialização assim como "a qual" algoritmo desviar após a reinicialização.

Entende-se por reinicialização, o mecanismo através do qual todo o algoritmo é reiniciado em algum instante da procura, desfazendo-se todas as atribuições realizadas até o momento. A idéia central é evitar permanecer um longo período de tempo em regiões do espaço de procura que levarão a um ponto sem solução.

Esta estratégia foi proposta inicialmente em [Sil97b] e tem sido largamente utilizada pelos solucionadores SAT atuais. Inicialmente, a proposta da reinicialização foi aplicada após um número constante de passos do algoritmo. Uma vez que caminhos podem ser abandonados, um dos problemas da reinicialização é um comprometimento da completude do algoritmo DPLL.

Para solucionar esse problema, algumas idéias foram introduzidas:

1. Guardar as cláusulas aprendidas registrando-as em um banco de dados.

2. Após cada reinicialização, aumentar o número de passos necessários para a reinicialização seguinte.

3. Manter apenas parte das cláusulas aprendidas no banco de dados, mas aumentar o número de cláusulas aprendidas a cada reinicialização.

4. Manter uma assinatura, ou um registro individual, de cada procura executada.

3.2 O mecanismo de decisão

O mecanismo de decisão é aquele que, a cada nó na árvore de procura DPLL, decide qual deve ser a variável a ser atribuída como verdadeira ou falsa e, conseqüentemente, gerar um filho para aquele nó [Hoo95]. Uma das formas mais simples de se realizar esse procedimento é escolher aleatoriamente uma variável ainda não atribuída e, também aleatoriamente, atribuir um valor para essa variável. Entretanto, as heurísticas mais efetivas utilizam, de alguma forma, as informações dinamicamente geradas pelo algoritmo de procura. Exemplos destas informações são: o número de literais de cada variável em cláusulas com variáveis livres, o tamanho das cláusulas com variáveis livres que contêm o literal de uma determinada variável etc.

A principal estratégia para o mecanismo de decisão é procurar, através de algum cálculo ou heurística, propriedades estruturais na instância que possam restringir o espaço de procura. A estratégia deve: rapidamente encontrar uma atribuição que torne a cláusula satisfazível, que irá provocar um grande número de implicações e encaminhar a procura para esse objetivo; ou encontrar uma contradição, e desviar daquele espaço de procura o mais rápido possível, sempre levando em conta o custo computacional exigido para o cálculo dessa decisão.

Podemos encontrar heurísticas de desvio que não usam as informações geradas pela análise de conflitos ou aquelas, mais recentes, que apresentam um baixo custo computacional e utilizam as informações extraídas da análise de conflitos.

Os principais exemplos destas heurísticas são: solucionador SAT Bohm [Bur92], Jeroslow-Wang [Jer90], MOM [Fre95], Bayardo [Bay97] [LiA97], VSIDS [Mos01] [Nov02] [Rya04].

O resultado principal desse trabalho é a geração de uma nova heurística baseada na subtração de cubos, que será descrita em detalhes nas seções subseqüentes.

3.2.1 Heurísticas não relacionadas com a análise de conflito

Heurística Bohm

A heurística Bohm está descrita de forma resumida em [Bur92], que apresenta um solucionador SAT baseado em procura com retrocesso, competitivo na época em que foi apresentado.

A cada etapa do algoritmo de procura a heurística de Bohm seleciona uma variável x com o seguinte vetor: $[H_1(x)H_2(x)\ldots H_n(x)]$, onde cada $H_i(x)$ é computado da seguinte forma: $H_i(x) = \alpha \max(h_i(x), h_i(\neg x)) + \beta \min(h_i(x), h_i(\neg x))$, onde $h_i(x)$ é o número de cláusulas não SAT que possuam a variável x e com variáveis livres, i representa a quantidade de variáveis na cláusula. O vetor é utilizado de forma a selecionar uma variável dando preferência a tornar satisfazível pequenas cláusulas, quando atribuído um valor verdade, ou reduzir o tamanho das menores cláusulas, quando um valor falso for atribuído. Os valores de α e β são escolhidos de forma heurística. O trabalho sugere os valores 1 e 2 respectivamente.

Heurística MOM

A heurística MOM (*Maximum Occurrence on clauses of Minimum size*), que leva em conta a existência de variáveis considerando o tamanho das cláusulas, é descrita detalhadamente em [Fre95].

A heurística procura maximizar a equação $M(x) = [f^*(x)+f^*(\neg x)]*2^k+f^*(x).f^*(\neg x)$, onde $f^*(x)$ representa o número de ocorrências da variável x na menor cláusula ainda não satisfazível e que possua variáveis livres. Intuitivamente a preferência é dada às variáveis com um grande número de ocorrências em x ou $\neg x$ (se k for um valor grande) e variáveis em um grande número de clausulas que aparecem ambos x e $\neg x$. O solucionador Satz [Lia97] utiliza esta heurística.

Heurística de Jeroslow-Wang

Essa heurística procura maximizar a equação: $J(x) = \sum_{x \in C_i} 2^{-n_i}$ onde C_i é uma cláusula que contenha a variável x, possui outras variáveis livres e ainda não é satisfazível, n_i é o número de variáveis livres na cláusula C_i. Deve-se atribuir o valor verdadeiro a x. Conclui-se que a preferência é dada à variável que ocorra o maior número de vezes em cláusulas menores.

Heurísticas baseadas na contagem de literais

O trabalho de Silva [Sil99b] apresenta três heurísticas similares que levam em conta a quantidade de literais em cláusulas ainda não satisfeitas durante o mecanismo de decisão: DLCS (Maior Soma Combinada Dinâmica), DLIS (Maior Soma Individual Dinâmica) e RDLIS (Maior Soma Individual Dinâmica Aleatória).

Essas heurísticas levam em conta o número de cláusulas não resolvidas onde uma determinada variável x aparece como um literal na sua forma positiva ou negativa. Essa quantidade de literais é representada por C_p e C_n, para as ocorrências positivas e negativas respectivamente. Esses dois números podem ser considerados de forma combinada ou individual. O termo dinâmico aparece porque o cálculo é dinamicamente realizado durante o processo de procura.

Para DLCS, seleciona-se a variável com maior $C_p + C_n$, a atribui-se verdadeiro se $C_p \geq C_n$, e falso caso contrário. Para DLIS, C_p e C_n são considerados separadamente e seleciona-se a variável com maior valor individual, novamente atribui-se verdadeiro se $C_p \geq C_n$, e falso caso contrário. Para RDLIS, seleciona-se a variável de acordo com DLIS e aleatoriamente o valor é atribuído.

Conclusão

Apesar dessas heurísticas parecerem extremamente gulosas, como relatado por Silva em [Sil99b], existem problemas onde elas se mostram eficientes [Rya04] [Lia97], entretanto elas possuem um valor computacional alto a ser pago para o cálculo.

De acordo com a tese de doutoramento de Zhang em [Zha99], estas heurísticas são eficientes para problemas aleatórios ou aqueles que não possuem algum tipo de estrutura como as encontradas nos problemas industriais, gerados a partir de um problema real.

Vamos supor uma determinada região de uma árvore de procura, os cálculos efetuados sobre quantidades de literais e ou variáveis podem redirecionar a procura para outras regiões. Escolhendo uma variável não relacionada com aquela região que estamos, pode significar a perda de todo o esforço realizado em uma determinada região e um direcionamento para outro lugar do espaço de procura completamente não relacionado. Se o problema é aleatório, deveríamos nos preocupar apenas com os custos computacionais dessa decisão, entretanto, se o problema é de alguma forma industrial ou possui algum tipo de estrutura isto é, não é completamente aleatório, essas heurísticas não levam em conta esta localidade, que é refletida pelo aprendizado de cláusulas. As heurísticas que de alguma forma se relacionam com o mecanismo de aprendizado de cláusulas se mostraram superiores, como será descrito a seguir.

3.2.2 Heurísticas relacionadas com a análise de conflito

VSIDS (Variable State Independent Decaying Sum)

Essa estratégia de decisão talvez tenha sido uma das maiores contribuições do solucionador zChaff [Mos01]. É uma heurística de decisão mais poderosa que DLIS e que permite ao solucionador Chaff resolver problemas industriais difíceis em tempos inferiores aos solucionadores até agora mencionados.

A idéia principal dessa heurística é manter um escore $s(l)$ para cada literal l, e um contador de ocorrências $r(l)$. O escore de um literal inicialmente representa o número de ocorrências de um literal na fórmula inicial. Quando o aprendizado de cláusulas adiciona uma cláusula ao problema original contendo um literal l, a heurística incrementa o escore do literal l. Periodicamente todos os escores são divididos por uma constante de acordo com a fórmula $s(l) = r(l) + s(l)/2$ e $r(l) = 0$. Avaliando-se os esco-

res, o algoritmo escolhe a variável livre que corresponde ao literal com o maior escore. Uma vez que os escores não são dependentes das atribuições das variáveis, os custos computacionais são menores que os custos das heurísticas anteriormente mencionadas. Os resultados apresentados pelo solucionador Chaff, além do tempo de solução do problema, mostram ainda que a quantidade de decisões proporcionada por esta heurística é competitivo com outras heurísticas.

De acordo com Ryan em [Ray04], VSIDS difere de DLIS sobre dois aspectos: primeiro é a não distinção entre cláusulas satisfeitas e não satisfeitas e segundo, a influência de cada ocorrência é escalonada de acordo com a novidade da ocorrência. As decisões são baseadas nos escores e não na contagem das ocorrências. Podemos classificar a heurística VSIDS como uma heurística de aprendizado de cláusulas que guia o solucionador a gerar grupos de cláusulas resolvíveis relacionadas.

Duas estratégias relacionadas com VSIDS são encontradas nos solucionadores SAT Siege [Rya04] e Berkmin [Nov02]. O solucionador Siege apresenta a heurística VMTF[4], a justificativa desta técnica é que VSIDS pode não ser uma técnica eficiente para a escolha de literais se considerarmos as cláusulas mais recentemente aprendidas. A razão é que existe um atraso no uso da estatística, ou seja, até que os literais sejam atualizados, as cláusulas aprendidas mais recentes são ignoradas. Considerando-se a organização de busca em profundidade da procura DPLL, estas são as cláusulas produzidas nas folhas da árvore de pesquisa e que compartilham um longo prefixo com o atual espaço de procura.

A solução proposta, em ambos os solucionadores, é a existência de uma lista ou uma pilha com as variáveis do problema. Quando uma cláusula é aprendida, essa estrutura é atualizada considerando-se os literais presentes na cláusula aprendida. O mecanismo de decisão procura uma variável livre que esteja o mais próximo possível do início desta lista ou pilha.

[4] *Variable Move to Front.*

3.3 O mecanismo de dedução

O mecanismo de dedução é o responsável por executar a propagação da restrição booleana (BCP), ou seja, implementar a regra da cláusula unitária continuamente até que não mais existam cláusulas unitárias ou um conflito seja alcançado. De acordo com Moskewicz em [Mos01], um solucionador do problema da Satisfabilidade baseado no DPLL gasta até 90% do tempo de execução no BCP. Existem duas razões para isso. Primeiro, porque o procedimento BCP é realizado todas as vezes que uma variável recebe uma atribuição, já que é o principal mecanismo que permite o caminhamento na árvore de implicações. Segundo, porque opera continuamente sobre uma estrutura de dados cujo tamanho é maior do que a memória cachê da máquina, promovendo acessos a diferentes, muitas vezes não seqüenciais, locais dessa mesma estrutura. Esse acesso difuso, freqüentemente não encontra o dado necessário na memória cachê e provoca acessos à memória principal. Considerando o tempo de CPU versus o tempo de acesso à memória, essa é a principal restrição do mecanismo.

Em particular devido a esta característica, diversas estruturas de dados para armazenar as cláusulas, variáveis e literais foram propostas e ainda são motivo de intenso estudo conforme descrito por Lynce em [Lyn02] e [Lyn04b]. As primeiras e mais tradicionais estruturas de dados são as mais ricas em termos de informação, procurando armazenar todas as informações sobre a situação dos literais nas suas respectivas cláusulas instantaneamente, isto é, as informações são atualizadas conforme acontecem durante o caminhamento na árvore de decisão. Nesta categoria encontramos as listas de adjacências, utilizadas por diversos solucionadores. Os solucionadores mais atuais procuram mecanismos que evitam a contínua atualização destas informações utilizando estruturas de dados denominadas atrasadas[5].

[5] *Lazy data structures.*

3.3.1 Estruturas de dados baseadas em lista de adjacências

O termo lista de adjacência vem do formato da estrutura de dados. Procura-se representar cada cláusula como uma lista de literais, além de associar cada variável x às cláusulas que contenham x. As listas criadas para cada variável contêm o que foi denominado cláusulas adjacentes à variável, aquelas cláusulas que possuem a variável. A Figura 3.8 ilustra o algoritmo para a propagação unitária considerando uma lista de cláusulas associada a cada variável.

```
BCP(x)
inicio
    se (x = verdadeiro)
        então C_x = lista de cláusulas onde o literal ¬x aparece;
        senão C_x = lista de cláusulas onde o literal x aparece;
    para (i = 1 até comprimento(C_x))
        se (C_x[i] é uma cláusula unitária)
            x_1 = variável livre em C_x[i];
            se ( o literal aparece como (x_1) )
                x_1 =verdadeiro;
            se ( o literal aparece como (¬x_1) )
                x_1 =falso;
            se (BCP(x_1)=falso)
                retorne falso;
        fimse
        se (C_x[i] apresentar um conflito)
            retorne falso;
    fimpara
    retorne verdadeiro;
fim
```

Figura 3.8: O algoritmo de BCP usando uma lista de cláusulas

As estruturas de dados diferem, principalmente, no que se refere à quantidade de cláusulas que deverão ser investigadas a cada atribuição e como a lista de cláusulas C_x no algoritmo da Figura 3.8 é construída. Em geral, as estruturas de dados baseadas em listas de adjacências procuram verificar todas as cláusulas que contém o literal para executar o BCP.

As primeiras abordagens procuravam verificar todos os literais em uma cláusula para identificá-la como unitária. Métodos mais modernos utilizam uma abordagem baseada em contadores. Esse método possui uma estrutura de dados onde, cada cláu-

sula armazena um contador para a ocorrência da fase positiva e outro para a fase negativa de uma variável, indicando a quantidade de variáveis da cláusula atribuídas como verdadeiras e falsas. A quantidade de variáveis não atribuídas é deduzida desses contadores.

A Figura 3.9 ilustra o algoritmo para esse método. Quando uma variável x é atribuída, todas as cláusulas que possuem os literais x e $\neg x$ atualizam os seus respectivos contadores de acordo com a escolha para a variável, verdadeiro ou falso. Se a escolha para a variável for verdadeiro, isto significa que o contador N^V é incrementado, caso contrário, se a escolha para a variável for falso, o contador N^F é incrementado. Citando um exemplo para ilustrar o algoritmo vem: sejam as cláusulas $w_1 = \neg x_1 \vee x_2 \vee x_3$, $w_2 = \neg x_1 \vee x_2$ e $w_3 = x_1 \vee x_2$, partes de uma instância φ qualquer. O número total de literais em w_1 é $N_1 = 3$, em w_2 é $N_2 = 2$, e em w_3 é $N_3 = 2$. Inicialmente as variáveis x_1, x_2, x_3 são variáveis livres, o que significa que $N_1^V = N_1^F = N_2^V = N_2^F = N_3^V = N_3^F = 0$. Vamos supor que, através da propagação unitária, alguma outra cláusula em φ, levou à atribuição de $x_1 = verdadeiro$. Desde que w_1 e w_2 possuem o literal $\neg x_1$, os contadores $N_1^F = 1$ e $N_2^F = 1$ são atualizados. Como w_3 possui o literal x_1, $N_3^V = 1$ é atualizado. Supondo agora que $x_2 = falso$, através de outra cláusula em φ, os contadores $N_1^F = 2$, $N_2^F = 2$ e $N_3^F = 1$ são atualizados. O algoritmo avalia os contadores de cada literal em relação ao número total de literais na cláusula. Para a cláusula w_3, $N_3^V \neq 0$, portanto a cláusula é satisfazível. Para w_1, $N_1^V = 0$ e $N_1 - N_1^F = 1$, o que significa uma cláusula unitária, já para w_2, $N_2^V = 0$ e $N_2 - N_2^F = 0$, o que significa um conflito, a cláusula não é satisfazível e nem possui variáveis livres respectivamente.

Quando for necessário desfazer uma atribuição, devido a um retrocesso, todas as cláusulas contendo os literais envolvidos no retrocesso, também deverão ser atualizadas e, conseqüentemente, todos os contadores. Se, no exemplo dado, as atribuições das variáveis x_1 e x_2 forem desfeitas, todos os contadores deverão ser passados para zero nas três cláusulas. O trabalho necessário de manutenção desses contadores é praticamente o mesmo tanto para a atribuição, quanto para desfazer uma atribuição [Zha01].

```
BCP(x)
inicio
    se (x = verdadeiro)
        então
            C_x^F = lista de cláusulas onde o literal ¬x aparece;
            C_x^V = lista de cláusulas onde o literal x aparece;
        senão
            C_x^F = lista de cláusulas onde o literal x aparece;
            C_x^V = lista de cláusulas onde o literal ¬x aparece;
    fimse
    nenhumconflito=verdadeiro;
    para (i = 1 até comprimento(C_x^F))
        incremente o número de literais falsos em C_x^F[i] de 1;
        N_i^V = número de literais verdadeiros em C_x^F[i];
        N_i^F = número de literais falsos em C_x^F[i];
        N_i = número total de literais em C_x^F[i];
        se (N_i^V = 0 e N_i^F = N_i - 1)
            x_1 = variável livre em C_x^F[i];
            se ( o literal aparece na cláusula como x_1 )
                então x_1 = verdadeiro;
            se ( o literal aparece como ¬x_1 )
                então x_1 = falso;
            se (BCP(x_1)=falso)
                então nenhumconflito = falso;
        fimse
        se (N_i^V = 0 e N_i^F = N_i)
            então nenhumconflito = falso;
    fimpara
    para (j = 1 até comprimento(C_x^V))
        incremente o número de literais verdadeiros em C_x^V[i] de 1;
    fimpara
    retorne (nenhumconflito);
fim
```

Figura 3.9: O algoritmo de BCP usando uma lista de cláusulas com contadores

3.3.2 Estruturas de dados baseadas em listas início-fim (Head-tail)

Uma alternativa proposta por Zhang em [Zha96] e no seu solucionador Sato em [Zha97] utiliza uma estrutura de dados baseada em listas com dois apontadores. A principal contribuição dessa proposta é a intuição de que, para a identificação de cláusulas unitárias ou conflitos, não é necessário o algoritmo percorrer toda a cláusula contendo uma variável recentemente atribuída. A estrutura de dados proposta, mantém dois apontadores para cada cláusula contendo dois ou mais literais. Um apontador para

o primeiro literal, o apontador início, e outro para o último literal, o apontador fim. Como um conflito não pode existir se a cláusula possuir duas ou mais variáveis livres, independentemente das atribuições nas demais variáveis da cláusula, se os dois literais apontados permanecerem livres, significa que a cláusula não gera um conflito e nem é uma cláusula unitária.

A Figura 3.10 ilustra o algoritmo. Se consideramos, por exemplo, a cláusula $w_1 = x_1 \vee \neg x_2 \vee x_3 \vee \neg x_4$, parte de uma instância φ qualquer. Vamos considerar o apontador início para o literal x_1 e o apontador fim para o literal $\neg x_4$. Uma cláusula não pode ser unitária ou conflituosa se os literais início e fim não possuírem um valor atribuído. Por esse motivo, a cláusula w_1 só necessita ser visitada se acontecer uma atribuição às variáveis x_1 ou x_4, já que ambos estão, no momento, sendo apontados. Mais uma conclusão é que: apenas quando os literais forem atribuídos como falsos é que deveremos visitar a cláusula, já que, se tanto x_1 quanto $\neg x_4$ forem avaliados como verdadeiros, w_1 é verdadeira e não uma cláusula unitária ou conflituosa. No algoritmo da Figura 3.10, se atribuirmos um valor à variável x, e, considerando que as cláusulas C_p possuem algum literal $\neg x$ ou x , apenas as cláusulas $C_p[i]$ onde esses literais foram avaliados como falsos deverão ser consideradas. Se um dos literais u^{incio} ou u^{fim} em $C_p[i]$ forem falsos, o algoritmo procura um novo literal não atribuído para assumir a posição inicio ou fim. Se, nessa procura, encontrar um literal verdadeiro, a cláusula $C_p[i]$ está satisfeita e não necessita mais de início ou fim. Portanto dizemos que é uma estrutura atrasada, já que não atualiza os valores imediatamente quando os mesmos são atribuídos, mas apenas quando são visitados em procuras posteriores. No nosso exemplo, se atribuirmos um valor verdadeiro à variável x_2, como não temos x_2 ou $\neg x_2$ apontados por início ou fim em w_1, ela não será avaliada. Se em uma atribuição posterior atribuirmos $x_1 = falso$, o algoritmo irá procurar um novo literal a ser apontado pelo início. Como o literal $\neg x_2$, em w_1, já foi anteriormente atribuído e o seu valor na cláusula é falso, a procura caminha para x_3. A variável x_3 é uma variável livre, portanto, é a nova candidata a ser apontada pelo início na cláusula w_1. Se, entretanto, tivéssemos atribuído $x_2 = falso$

anteriormente, o literal $\neg x_2$ seria verdadeiro, a cláusula w_1 seria verdadeira e não seria necessário encontrar x_3.

```
BCP(x)
inicio
    se (x = verdadeiro)
        então C_x = lista de cláusulas onde o literal ¬x aparece apontado pelo inicio ou fim;
        senão C_x = lista de cláusulas onde o literal x aparece apontado pelo início ou fim;
    fimse
    nenhumconflito = verdadeiro;
    para (i = 1 até comprimento(C_x))
        u^inicio = literal apontado pelo início em C_x[i];
        u^fim = literal apontado pelo fim em C_x[i];
        para (cada literal u entre u^inicio e u^fim iniciando em u^inicio e terminando em u^fim)
            se (u = verdadeiro)
                então saia do laço;
            se( u é um literal livre e u ≠ u^inicio e u ≠ u^fim)
                então u é um novo literal apontado pelo início ou fim;
                insira C_x[i] na lista de literais inicio/fim de u;
                saia do laço;
            fimse
            se ( u é um literal livre e (u = u^inicio ou u = u^fim))
                então
                    se o literal aparece na cláusula como u
                        então variável u = verdadeiro;
                        senão variável u = falso;
                    se BCP(u) = falso
                        então nenhumconflito = falso;
            fimse
            se ( u = falso e (u = u^inicio ou u = u^fim))
                então nenhumconflito = falso;
        fimpara
    fimpara
    retorne (nenhumconflito);
fim
```

Figura 3.10: O algoritmo de BCP usando apontadores inicio/fim

Podemos declarar a cláusula $C_p[i]$ uma cláusula unitária se os apontadores inicio e fim indicam o mesmo literal de um variável livre. $C_p[i]$ é uma cláusula conflito se ambos indicam o mesmo literal e a variável é falsa. Continuando o exemplo, se alguma cláusula em φ atribuir $x_4 = verdadeiro$, como o ponteiro pra fim aponta para $\neg x_4$ em w_1, o algoritmo irá procurar por um novo literal a ser apontado. Neste caso, irá encontrar o literal x_3, que é o literal apontado pelo início. Desde que x_3 é uma variável livre, w_1 é uma cláusula unitária. Se tivéssemos atribuído $x_4 = falso$, w_1 seria uma

cláusula conflito.

Não existe a necessidade de percorrermos a cláusula para verificarmos x_1 e x_2. Os apontadores início e fim indicam a primeira e a última variável livre. Não são alterados se forem atribuídos valores verdadeiros e todos os literais que não estiverem entre o início e o fim são avaliados como falsos.

Durante um retrocesso, os apontadores devem ser reconstruídos. Se, no nosso exemplo, efetuarmos um retrocesso e todas as variáveis de w_1 se tornarem variáveis livres, como o início atual indica x_3 e o fim indica x_4, deveremos reverter o apontador do início para x_1, já que é uma variável livre anterior a x_3. O trabalho necessário para as atualizações dos apontadores para o início ou fim, durante um retrocesso, é o mesmo que o realizado durante as atribuições [Zha97].

3.3.3 Estruturas de dados baseadas em literais observados (watched literals)

Esta estrutura de dados foi inicialmente proposta por Moskewicz , Madigan, Zhao e Zhang e implementada no solucionador zChaff [Mos01]. A sua eficácia é comprovada através dos resultados obtidos nas competições de solucionadores SAT [Url03] e é utilizada, com pequenas variações, nos principais solucionadores modernos. Essa é a estrutura de dados que o solucionador DPLL combinado com DSharp e que será mostrado na seção cinco utiliza, razão pela qual mais em detalhes estão aqui presentes.

Assim como na estrutura baseada nas listas com apontadores para literais início e fim, mencionadas na subseção anterior, esta estrutura também possui e avalia apenas dois apontadores para os literais de uma cláusula, os literais observados, e também é uma estrutura que atualiza as variáveis de uma forma atrasada.

As principais contribuições são que os literais observados podem ser quaisquer literais e inclusive os literais já avaliados como falsos da cláusula, não necessariamente o primeiro e o último e principalmente, quando o retrocesso acontece, não existe a necessidade de se refazer os apontadores, o que se mostra como uma grande vantagem

sobre a abordagem anterior, no que se refere ao tempo necessário para esse procedimento. Se nos revertermos ao exemplo da subseção anterior, onde para a cláusula $w_1 = x_1 \vee \neg x_2 \vee x_3 \vee \neg x_4$, os atuais inicio e fim eram x_3 e $\neg x_4$ respectivamente, ao realizarmos o retrocesso seria necessário reverter o início para x_1. Nos literais observados, os apontadores podem permanecer em x_3 e $\neg x_4$ mesmo após o retrocesso e com todas as atribuições das variáveis de w_1 desfeitas.

O algoritmo é ilustrado na Figura 3.11. Quando atribuímos o valor falso a um literal observado, o algoritmo irá procurar por algum outro literal não observado, não falso, existente na cláusula. O literal observado permanecerá como falso apenas se um literal substituto que satisfaça a condição mencionada não for encontrado. Continuando o exemplo anterior, vamos considerar $w_1 = x_1 \vee \neg x_2 \vee x_3 \vee \neg x_4$ uma das cláusulas de uma instância φ qualquer, todos literais de w_1 não possuem nenhuma atribuição e os literais observados são x_3 e $\neg x_4$. Se um BCP anteriormente executado chegar a uma atribuição $x_4 = verdadeiro$, então o literal observado $\neg x_4$ se torna falso. O algoritmo irá procurar por um novo literal não falso a ser observado, neste caso x_1 seria o novo literal observado. Vamos assumir agora uma atribuição $x_2 = verdadeiro$. Como x_2 não é um literal observado, a cláusula w_1 não será visitada após esta atribuição. Agora vamos assumir a atribuição $x_3 = falso$. ocasionada pelo BCP em alguma cláusula em φ. Novamente o algoritmo irá procurar por um novo literal não observado e não falso em w_1. Como todos os literais não observados em w_1 já estão atribuídos, o literal x_3 permanece observado, mesmo tendo um valor atribuído como falso.

Se um dos literais observados possui o valor falso e o outro não possui uma atribuição temos uma cláusula unitária. Se ambos os literais observados possuem o valor falso temos uma cláusula contraditória. Retornando ao exemplo, como o literal observado $x_3 = falso$ e o literal observado x_1 não possui uma atribuição, w_1 é uma cláusula unitária. Se, após o BCP, alguma outra cláusula em φ levar à atribuição $x_1 = falso$, então, ambos os literais observados em w_1, x_3 e x_1, terão o valor falso, o que é um conflito. Em geral, localizar um novo literal na estrutura de literais observados pode

requerer mais tempo que na lista anterior com apontadores para o início e fim, já que na segunda levamos em conta apenas os literais que estão entre os dois apontadores para a escolha de um novo início ou fim, enquanto que na primeira devemos levar em conta todos os literais para a escolha do novo a ser observado.

Observa-se, entretanto, que certos cuidados deverão ser tomados durante o retrocesso. Existem condições nas quais os literais observados podem se apresentar erradamente apontados se esses cuidados não forem observados. Durante o retrocesso, os últimos literais atribuídos devem ser os primeiros a ter as atribuições desfeitas. Se esta condição não for observada, poderemos ter literais observados com uma atribuição falso e ao mesmo tempo literais livres em uma cláusula. Retornando ao exemplo para ilustrar esse efeito: para a cláusula $w_1 = x_1 \vee \neg x_2 \vee x_3 \vee \neg x_4$, fizemos as seguintes atribuições na ordem em que foram realizadas, inicialmente $x_4 = verdadeiro$, a seguir $x_2 = verdadeiro$, $x_3 = falso$ e por fim $x_1 = falso$, lembrando também que os literais observados eram x_1 e x_3. Se, durante um retrocesso, as atribuições que devem ser desfeitas forem x_1, x_3 e x_4, e esse processo for realizado nessa ordem, os literais observados estarão livres e teremos um literal atribuído como falso, o literal $\neg x_2$. Essa condição está correta, os literais observados estão livres e existe um literal atribuído na cláusula. Se, no entanto, desfizermos as atribuições de x_3 seguida de $\neg x_4$ e não a de x_1, como os literais observados são x_1 e x_3, estaremos observando x_3 livre e x_1 com um valor atribuído de falso, enquanto que $\neg x_4$ está livre.

Se realizarmos o retrocesso na ordem reversa, estaremos eliminando esse problema já que: primeiro, se os literais observados estiverem originalmente como verdadeiros ou livres eles não poderiam se tornar falso e, portanto, não existiria a necessidade da atualização; segundo, se o literal estiver originalmente falso, então ele deve ser atribuído após todos os literais observados da cláusula, já que ele poderia permanecer como um literal observado apenas se todos os literais não observados da cláusula estiverem atribuídos como falso. Dessa forma, se o retrocesso desfizer as atribuições em ordem reversa, literais observados falsos se tornarão não atribuídos antes dos literais não

```
BCP(x)
inicio
    se (x = verdadeiro)
        então C_x = lista de cláusulas onde o literal assistido ¬x aparece;
        senão C_x = lista de cláusulas onde o literal assistido x aparece;
    fimse
    nenhumconflito = verdadeiro;
    para (i = 1 até comprimento(C_x))
        u^1 = literal assistido ¬x ou x em C_x[i];
        u^2 = algum outro literal assistido em C_x[i];
        se possível, substitua u^1 por algum outro literal não assistido, não falso;
        se ( u^1 não puder ser substituído e u^2 é um literal livre)
            então
                se o literal aparece na cláusula como u^2
                    então a variável u^2 = verdadeiro;
                    senão a variável u^2 = falso;
                se BCP(u^2) = falso
                    então nenhumconflito = falso;
        fimse
        se ( u^1 não puder ser substituído e u^2 = falso )
            então nenhumconflito = falso;
    fimpara
    retorne (nenhumconflito);
fim
```

Figura 3.11: O algoritmo de BCP usando literais observados

observados e, portanto, não necessitarão ser atualizados.

Os resultados apresentados pelos solucionadores que se baseiam nesse método são superiores aos métodos anteriores, fato comprovado nas competições de SAT. O principal motivo sobre a abordagem baseada em contadores está no fato de visitarmos apenas um pequeno conjunto de cláusulas associadas à variável mais recentemente atribuída, e não todas as cláusulas que contém aquela variável. Essa mesma vantagem é compartilhada pela abordagem baseada em apontadores início e fim, porém, enquanto que essa necessita sempre atualizar os apontadores início e fim, a abordagem baseada em literais observados pode manter os apontadores mesmo após os retrocessos, não necessitando atualiza-los. Aparentemente, a desvantagem da abordagem baseada nos literais observados sobre os apontadores início fim, e que é o trabalho de se verificar todos os literais da cláusula e não apenas os que estão entre os apontadores, é menor do que a vantagem anteriormente descrita, conforme se vê nos resultados das competições e nas

mais diferentes instâncias do problema da satisfabilidade.

Capítulo 4

Cubos, operador DSharp e solucionador DSharp

Esse capítulo descreve os conceitos básicos sobre cubos. Através desses conceitos, são definidos dois operadores utilizados na subtração de cubos e a operação Sharp modificada para a operação DSharp, mais eficiente para solucionadores do problema da satisfabilidade. O algoritmo e os resultados utilizando um solucionador DSharp são apresentados e comparados com um solucionador clássico DPLL, os testes realizados utilizaram um conjunto de instâncias de problemas da satisfabilidade mundialmente conhecidos e disponibilizados pelo DIMACS. Esses resultados mostram que um solucionador DSharp é competitivo frente a um solucionador clássico DPLL. Finalmente, o capítulo propõe uma combinação de métodos: DSharp+DPLL.

4.1 Conceitos básicos

Seja $N \geq 1$ o número de variáveis de uma fórmula. Um cubo é um subconjunto de $\{b_1, b_2, \ldots, b_N | b_i \in \{0,1\}$ para $1 \leq i \leq N\}$. Ele é utilizado aqui para representar a atribuição de valores booleanos às variáveis de uma fórmula (um conjunto de linhas da tabela verdade) e que podemos também entender como um subespaço booleano.

Além da notação usual de conjuntos, será utilizada a notação $(c_1 c_2 \ldots c_N)$ para

cubos, onde cada $c_i \in \{0, 1, X\}$ para $1 \leq i \leq N$. Neste caso, o cubo denotado por $(c_1 c_2 \ldots c_N)$ é: $(c_1 c_2 \ldots c_N) = \{b_1 b_2 \ldots b_N |$ para cada i de 1 até $N : b_i = c_i$, se $c_i \in \{0, 1\}$, e $b_i \in \{0, 1\}$, se $c_i = X\}$. Será adotado o termo "possui um valor especificado" para c_i se $c_i \in \{0, 1\}$ e "não possui um valor especificado" se $c_i = X$.

A cada cláusula de uma fórmula na FNC pode-se fazer corresponder facilmente um cubo, aqui denominado cláusula cubo, utilizando-se a notação anterior que denota as atribuições que tornam a cláusula falsa. Cada variável presente na cláusula estará representada como especificada. As outras variáveis, não presentes na cláusula, como não especificadas. Por exemplo: seja $\varphi = w_1 \wedge w_2$, sendo $w_1 = x_1 \vee \neg x_2 \vee \neg x_4$ e $w_2 = \neg x_1 \vee x_3$. A negação das cláusulas é: $\neg w_1 = \neg x_1 \wedge x_2 \wedge x_4$ e $\neg w_2 = x_1 \wedge \neg x_3$. As cláusulas-cubo de φ são $c_1 = (01X1)$, correspondendo a w_1 e $c_2 = (1X0X)$, correspondendo a w_2. Usaremos a notação $cc(\varphi)$ para o conjunto das cláusulas cubo de φ, assim temos: $cc(\varphi) = \{(01X1), (1X0X)\}$.

A interseção de dois cubos é sempre um outro cubo ou um conjunto vazio, essa última, se os dois cubos diferem no mínimo de um elemento especificado. Um cubo c_i pode ser subtraído de um cubo c_j e o resultado é um conjunto de cubos que cobrem todos os minitermos de c_j, que não estejam em c_i.

Um cubo booleano típico, de dimensão três, pode ser visualizado na Figura 4.1. A figura ilustra os elementos especificados e não especificados em um cubo.

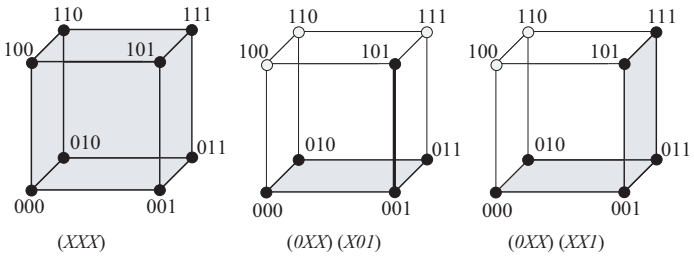

Figura 4.1: Exemplificação de cubos de dimensão três

A subtração de cubos é normalmente denotada pelo operador Sharp (#). Se esta subtração é definida de tal forma que o resultado seja um conjunto de cubos disjuntos, denotamos este operador como Sharp disjunto ou simplesmente DSharp. O algoritmo

utilizado para executar a operação DSharp é similar ao proposto em [Smi79] [Hon79] para determinação de falhas em arranjos lógicos programáveis[1] e por Skliavora em [Skl04a] [Skl04b] para plataformas de solucionadores de instâncias do problema SAT em *hardware* reconfigurável.

Definição 1: Operação de subtração de cubos DSharp (#)

Sejam os cubos $a = (a_1 a_2 \dots a_n)$ e $b = (b_1 b_2 \dots b_n)$. Então $a\#b$ é definido como: $a\#b = C$, onde $C = \{c_1, c_2, \dots, c_n)\}$, um conjunto de cubos, e

$$\text{cada } c_i = \begin{cases} ((a_1 \beta b_1)(a_2 \beta b_2) \dots (a_{j-1} \beta b_{j-1})(a_j \gamma b_j)(a_{j+1}) \dots (a_{n-1})(a_n)), \\ \quad \emptyset \text{ se algum termo } (a_j \gamma b_j) \text{ ou } (a_j \beta b_j) \text{ for indefinido.} \end{cases}$$

Os operadores β e γ são definidos como na Tabela 4.1 e, $c_i = \emptyset$, significa um cubo vazio.

β	0	1	X
0	0	–	0
1	–	1	1
X	0	1	X

γ	0	1	X
0	–	0	–
1	1	–	–
X	1	0	–

Tabela 4.1: Definição dos operadores β e γ

Definição 2: Operação DSharp entre um conjunto de cubos e um cubo

Seja $S = \{s_1, s_2, \dots, s_n\}$ um conjunto de cubos contendo n cubos e c um cubo. Definimos $S\#c$ como:

$$S\#c = \{s_1\#c, s_2\#c, \dots, s_n\#c\}$$

Exemplo 1: Seja $u = (XXXX)$ e $b = (0011)$. Aplicando a **Definição 1** e os operadores β e γ como definidos na **Tabela 1** obtemos:

$(u\#b) = \{c_1, c_2, c_3, c_4\}$, onde:

[1] *Programmable Logic Array (PLA).*

$c_1 = ((u_1\gamma b_1)(u_2)(u_3)(u_4)) = ((X\gamma 0)XXX) = (1XXX)$

$c_2 = ((u_1\beta b_1)(u_2\gamma b_2)(u_3)(u_4)) = ((X\beta 0)(X\gamma 0)XX) = (01XX)$

$c_3 = ((u_1\beta b_1)(u_2\beta b_2)(u_3\gamma b_3)(u_4)) = ((X\beta 0)(X\beta 0)(X\gamma 1)X) = (000X)$

$c_4 = ((u_1\beta b_1)(u_2\beta b_2)(u_3\beta b_3)(u_4\gamma b_4)) = ((X\beta 0)(X\beta 0)(X\beta 1)(X\gamma 1)) = (0010)$ então

$(u\#b) = \{(1XXX), (01XX), (000X), (0010)\}$

Nota-se que todos os cubos resultantes são disjuntos ou ortogonais (quaisquer dois cubos possuem interseção nula). Uma árvore , representando a operação DSharp desse exemplo pode ser vista na Figura 4.2. O cubo $u = (XXXX)$ é o cubo universal, representando todo o espaço booleano para quatro variáveis.

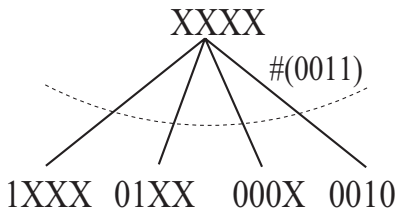

Figura 4.2: Árvore resultante da operação DSharp do exemplo 1

Se removermos um subespaço restrição, o cubo $b = (0011)$, o resultado será um conjunto de subespaços disjuntos que unidos com b, reconstroem u. A Figura 4.3 ilustra graficamente a operação DSharp e a reconstrução de u, em um cubo com dimensão três.

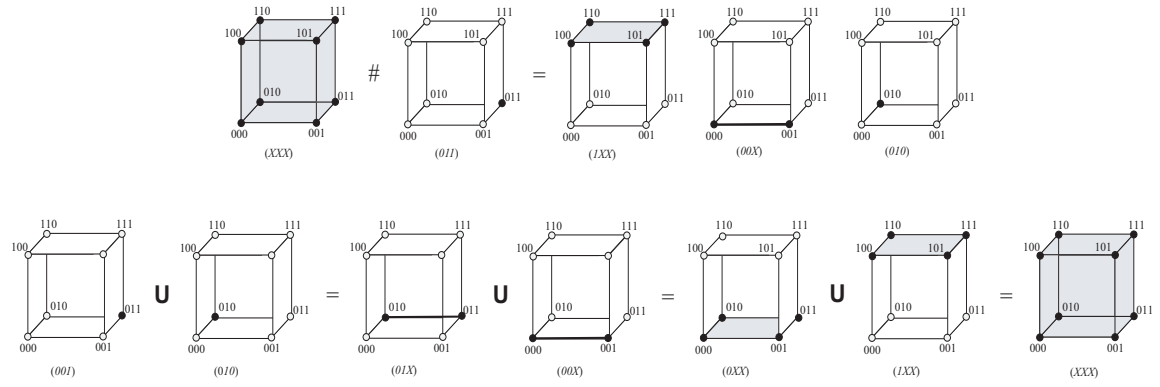

Figura 4.3: Operação DSharp e reconstrução de u em cubos de dimensão três

4.2 Solucionador SAT Dsharp

De forma a perceber a intuição sobre o solucionador SAT DSharp, um exemplo simples com uma descrição similar ao algoritmo DPLL clássico será utilizada, ou seja, nenhuma das técnicas atualmente sobrepostas ao DPLL original, tais como retrocessos não cronológicos, aprendizado de cláusulas e reinicializações, serão inseridas ao solucionador. Posteriormente algumas técnicas de simplificação, análogas a estas técnicas, serão exibidas.

Para uma fórmula $\phi = w_1 \wedge w_2 \wedge \ldots \wedge w_m$ na FNC, inicialmente negamos ambos os lados e obtemos: $\neg\phi = \neg w_1 \vee \neg w_2 \vee \ldots \vee \neg w_m$, e que, conforme vimos, corresponde ao conjunto de cláusulas-cubo $cc(\phi) = c_1 c_2 \ldots c_m$. A disjunção das cláusulas-cubo representa todas as regiões do espaço de procura onde não encontraremos uma atribuição das variáveis capaz de satisfazer a instância ϕ do problema SAT.

Iniciando com um cubo universal $u = (u_1 u_2 \ldots u_m)$, que denota todo o espaço de procura e onde cada elemento u_i possui o valor X, ao sucessivamente aplicarmos a operação DSharp entre o atual espaço de procura e cada cláusula-cubo, chegaremos ao subespaço solução da instância ϕ se ela for satisfazível, ou a um subespaço vazio, se ela não for. O conjunto de cubos solução S_ϕ é dado por:

$$S_\phi = (((((u \# c_1) \# c_2) \ldots) \# c_m)$$

Após cada operação DSharp obtemos um conjunto de subespaços de procura disjuntos. As cláusulas-cubo restantes devem ser subtraídas de cada um desses subespaços. A escolha da subtração disjunta evita que a procura seja realizada de forma repetida em cada subespaço.

O algoritmo básico de um solucionador SAT DSharp é ilustrado a seguir. O solucionador é definido de forma recursiva e é inicialmente chamado a partir da rotina principal, passando como argumentos $u = (XX \ldots X)$ como o atual espaço de procura e o nível como 0, ou nível inicial. O pseudocódigo está descrito na Figura 4.4.

A função Decide( ) é responsável por escolher o próximo cubo a ser subtraído, nenhuma heurística será utilizada nesta etapa, todas as cláusulas-cubo são igualmente tratadas e aleatoriamente escolhidas.

A função Dsharp ( ) executa a operação DSharp entre o espaço corrente de procura e a cláusula-cubo eleita pela função Decide( ). Ela executa exatamente a operação descrita na **Definição 1**.

```
Soluciona (solução_corrente, nível)
inicio
    se (nível = último nível)
        retorne (SAT);
    clausula_cubo_escolhida = Decide ( );
    C=Dsharp (solução_corrente, cláusula_cubo_escolhida)
    para ( cada cubo ci em C )
        retorno_solucionador = Soluciona(ci, nível +1);
        se (retorno_solucionador = SAT)
            retorne (SAT);
    fimpara
    retorne ( não_SAT);
fim
```

Figura 4.4: Pseudo código do solucionador SAT baseado em DSharp

Como em um solucionador baseado em DPLL, existem algumas estratégias para se chegar mais rapidamente à resposta SAT ou não-SAT para uma instância:

- Um conflito é alcançado quando uma cláusula-cubo a ser subtraída, cobre o espaço de procura corrente, nessa situação acontece um retrocesso. A justificativa é simples, essa situação significa que o espaço booleano da cláusula-cubo é maior do que o espaço onde estamos buscando a solução, assim sendo, devemos sair deste espaço imediatamente;

- Se uma cláusula-cubo possui uma interseção nula com o espaço corrente de procura, não existe a necessidade de se aplicar a operação DSharp, o resultado é o próprio espaço de procura. A justificativa é que esta restrição, ou a cláusula cubo,

não se aplica ao espaço onde estamos procurando a solução, conseqüentemente não existe a necessidade de se executar a operação DSharp, mesmo porque o resultado de se aplicar o operador seria o próprio espaço corrente, esta simplificação nos permite deixar de executar uma operação DSharp;

- Finalmente, uma implicação (ou a propagação de restrição booleana-BCP) acontece, quando a operação DSharp produz apenas um cubo como resultado.

O algoritmo pode se melhor entendido através de um exemplo contendo uma fórmula simples codificada em CNF:

$$\phi = (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_3 \vee \neg x_4)$$

Invertendo, iremos obter:

$$\neg\phi = (x_1 \wedge \neg x_2 \wedge x_4) \vee (\neg x_1 \wedge x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge x_3 \wedge x_4)$$

Em uma notação de cubos temos:

$$cc(\phi) = \{(10X1), (010X), (0X11)\}$$

Iniciando-se com $u = (XXXX)$, o que representa a atribuição de um valor não especificado para x_1, x_2 , x_3 e x_4, tomamos aleatoriamente a primeira cláusula-cubo a ser subtraída e, aplicando o algoritmo DSharp, obtemos como resposta: $(XXXX)\#(10X1) = \{(0XXX), (11XX), (10X0)\}$, que representa três subespaços disjuntos de procura.

A partir desse ponto, podemos realizar a procura através de uma busca em largura (BFS) ou uma busca em profundidade (DFS). Se tomarmos a segunda cláusula cubo para subtrair, temos os seguintes resultados:

$(0XXX)\#(010X) = \{(000X), (011X)\}$,
$(11XX)\#(010X) = \{(11XX)\}$ e
$(10X0)\#(010X) = \{(10X0)\}$.

Observa-se que as duas subtrações finais foram desnecessárias, o espaço de procura após a subtração foi idêntico ao espaço de procura anterior à subtração, já que a interseção entre ambos é nula. E finalmente, subtraindo a terceira cláusula-cubo:

$(000X)\#(0X11) = \{(000X)(0010)\}$,
$(011X)\#(0X11) = \{(0110)\}$,
$(11XX)\#(0X11) = \{(11XX)\}$ e
$(10X0)\#(0X11) = \{(10X0)\}$.

O resultado final é $S_\phi = \{(000X), (0010), (0110), (11XX), (10X0)\}$, onde temos todas as atribuições às variáveis que tornam a instância satisfazível, apresentadas de forma disjunta.

O ponto fundamental a observar é que, como todos esses subespaços respostas de subtrações possuem interseção nula, evita-se uma busca desnecessária em regiões de procura sobrepostas, o que representa a principal justificativa para o uso da operação DSharp. A busca em regiões que se sobrepõem significa a aplicação desnecessária do operador e, conseqüentemente, em mais tempo para se chegar a uma resposta.

A árvore completa de procura, para o exemplo ilustrado, pode ser vista na Figura 4.5. Cada nível da árvore corresponde à operação DSharp entre o subespaço corrente de procura e uma cláusula-cubo, neste exemplo, na mesma ordem em que elas aparecem na fórmula original.

Observa-se que a implementação simples desse algoritmo rapidamente poderia consumir recursos de memória para fórmulas grandes, já que um grande número de cláusulas cubo iria proliferar após cada operação DSharp. Conclui-se que uma busca em largura (*BFS*) é ineficiente. Uma abordagem mais eficiente, em termos de memória, seria uma busca em profundidade (*DFS*), subtraindo o cubo seguinte apenas do cubo disjunto previamente obtido pela operação # anterior. Estas duas direções de procura estão indicadas na Figura 4.5.

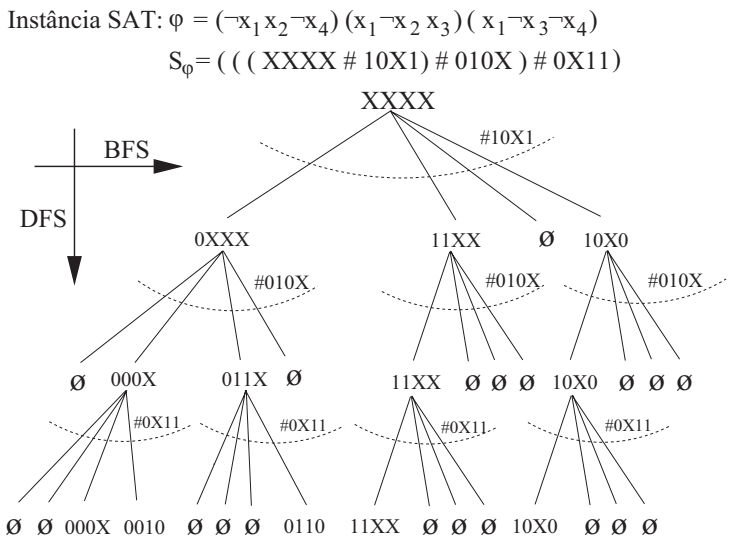

Figura 4.5: Instância de um problema SAT e árvore de procura completa DSharp

4.3 Uma abordagem matricial

O problema da satisfabilidade pode ser especificado através de vários modelos matemáticos. Uma abordagem alternativa é o uso de matrizes discretas, onde a operação DSharp pode ser claramente observada, gerando os mesmos resultados. A justificativa para este tipo de abordagem é a relativa facilidade encontrada na implementação de matrizes no processamento em hardware.

Uma instância SAT pode ser visualizada através de uma matriz $\boldsymbol{U}$, onde cada cláusula w_i, $i = 1, \ldots, m$, será representada por uma linha (u_i) e cada variável x_j, $j = 1, \ldots, n$, uma coluna (u_j). Se uma variável x_j estiver presente na cláusula w_i, o elemento respectivo da matriz u_{ij} será igual a 1, se a variável x_j estiver aparecer complementada na cláusula w_i, o elemento será igual a 0 e, finalmente, se a variável x_j não estiver presente na cláusula w_i, o elemento será representado como '-' (uma variável não especificada).

A figura 4.6 ilustra a representação matricial da seguinte fórmula:

$$\phi = (x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2)$$

$$\mathbf{U} = \begin{array}{c} \begin{matrix} x_1 & x_2 & x_3 \end{matrix} \\ \begin{bmatrix} 1 & - & 0 \\ 0 & 1 & - \\ - & 0 & 0 \\ 1 & 1 & - \end{bmatrix} \end{array} \begin{matrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{matrix}$$

Figura 4.6: Exemplo de uma representação matricial de uma instância do problema SAT

Resolver uma instância do problema SAT é equivalente a encontrarmos um vetor $\boldsymbol{v}$ que seja ortogonal a cada uma das linhas da matriz $\boldsymbol{U}$. Se este vetor não existir, o problema é insatisfazível. Se este vetor for encontrado, basta inverte-lo para chegarmos ao valor das variáveis. Para o exemplo ilustrado na Figura 4.6, a solução é o vetor $v = \begin{bmatrix} - & 0 & 1 \end{bmatrix}$ e que $\overline{v} = \begin{bmatrix} - & 1 & 0 \end{bmatrix}$, o que representa a solução: $x_2 = 1$ e $x_3 = 0$.

4.3.1 Solucionadores SAT em Hardware

Uma característica comum de diversas tarefas de computação intensiva é que elas são bem orientadas para implementações paralelizadas e que podem ser beneficiadas pelas características oferecidas pela computação reconfigurável. Dentre estas tarefas encontramos o problema da satisfabilidade proposicional SAT.

A grande maioria dos solucionadores SAT em computação reconfigurável existentes implementa alguma variação do algoritmo de Davis-Putnam [Dav60] como mostrado em [Skl04a]. As exceções são as arquiteturas propostas por Leong [Leo04], [Yun99], Yap [Yap03], Hamadi [Ham97], que implementam algoritmos incompletos e o solucionador SAT de Abramovici [Abr00] e Rashid [Ras98]. Vários solucionadores SAT em hardware reconfigurável diferem também no formato de entrada que suportam. Por exemplo, as arquiteturas de Suyama [Suy99], Leong [Leo04] e de Sousa [Sou01] são apenas capazes de trabalhar com expressões Booleanas no formato 3-SAT CNF; o solucionador de Zhong [Zho99] e Dandalis [Dan02] são limitados a fórmulas k-SAT CNF; as implementações de Mencer Men99], Skliarova [Skl04a] e Boyd [Boy00] aceitam

qualquer fórmula CNF; e finalmente, a arquitetura de Abramovici [Abr00] é capaz de processar qualquer expressão booleana arbitrária. A estratégia de seleção da variável de decisão também varia entre os solucionadores SAT em hardware. Apesar da seleção dinâmica ser considerada de difícil implementação, podemos encontra-la em algumas arquiteturas [Suy99], [Sou01], [Zho99].

Nos atuais solucionadores SAT em software, novas técnicas e heurísticas podem ser encontradas (assim como retrocessos não cronológicos [Sil98] e adição dinâmica de cláusulas). Contudo, até o momento, estas técnicas foram largamente ignoradas pelos solucionadores SAT em hardware, excetuando-se o solucionador proposto por Zhong [Zho99]. A principal razão é a complexidade das estruturas de dados necessárias a estas novas técnicas.

A estrutura de dados utilizada pelo solucionador DSharp é essencialmente orientada a vetor, o que a torna bem adaptada às implementações intensivamente orientadas a bits como as encontradas em hardware reconfigurável.

4.4 Resultados

Essa subseção apresenta os resultados, em uma versão inicial, do algoritmo descrito na Figura 4.4. O objetivo e a razão da inserção desses resultados é simplesmente avaliar o desempenho prático da subtração de cubos em instâncias do problema SAT. Na verdade, as instâncias de teste utilizadas são relativamente simples, mas nos permitem uma comparação do desempenho entre um solucionador baseado na operação DSharp e o DPLL.

A estrutura de dados utilizada foi basicamente a mesma para os dois solucionadores: uma lista de adjacências conforme descrito no capítulo 3.

Para o solucionador DPLL:

- Para cada cláusula, existe uma lista de literais e a quantidade de literais avaliados como verdadeiros ou falsos;

- Para cada variável, existe uma lista de cláusulas onde ela aparece, o valor atual atribuído à mesma (verdadeiro, falso ou livre), o nível da árvore de decisão onde esta atribuição foi realizada e uma variável de controle identificando se a variável já foi avaliada nas suas duas fases;

- O algoritmo DPLL adotado é o mesmo descrito no capítulo 3. Como se trata de um procedimento não recursivo, existe uma pilha das atribuições para o retrocesso.

Para o solucionador DSharp:

- Para cada cláusula-cubo, existe de uma lista de variáveis, representadas por um valor numérico identificando a variável e outra para o seu valor especificado. Apenas estão presentes na lista as variáveis especificadas, se uma variável é não especificada (ou possui o valor X), ela não aparece na lista da cláusula-cubo;

- Como o procedimento é recursivo, a pilha, com as atribuições de cada nível, está implicitamente determinada. A cada chamada recursiva, caminhamos dentro da lista da cláusula-cubo identificando a próxima variável a ser subtraída. Isso nos permite um caminhamento em profundidade (*DFS*) na árvore de decisões DSharp;

- O cubo resposta inicial é representado por uma lista vazia, todas as variáveis são não especificadas, a cada operação DSharp, as variáveis que recebem alguma atribuição, se tornam especificadas, são inseridas nessa lista.

Nenhuma heurística foi implementada em nenhum solucionador, isto é, para o solucionador DPLL, não existe uma heurística de decisão, as variáveis e os valores atribuídos a cada decisão são aleatoriamente escolhidos e o retrocesso é cronológico. Para o solucionador baseado no DSharp, as cláusulas-cubo são aleatoriamente escolhidas, todas as operações entre os cubos são realizadas e os mecanismos de cobertura ou interseção nula não são avaliados.

As instâncias utilizadas foram obtidas do pacote de instâncias do DIMACS [Url13] e obtidas em [Url18]. O grupo de instâncias consiste no pacote de instâncias AIM. As instâncias AIM, elaboradas por Iwama, Miyano e Asahiro em [Aim96], foram elaboradas através de um gerador aleatório de instâncias 3-SAT. O gerador trabalha de forma completamente aleatória o que diferencia as instâncias geradas daquelas que utilizam uma geração com um certo determinismo ou daquelas instâncias que são traduções de problemas. O pacote de instâncias inclui:

- Instâncias não-SAT com relações cláusulas/variáveis baixos
- Instâncias SAT com relações altas e baixas, mas que possuem apenas uma atribuição de variáveis que torne a instância SAT.

As instâncias são identificadas como *aim-xxx-y_y-zzzz-j*, onde:

- Todos os nomes possuem "aim"no início (referente às iniciais Asahiro, Iwama e Miyano)
- *xxx* mostra o número de variáveis 50, 100 ou 200
- y_y mostra a relação cláusulas/variáveis (1.6, 2.0, 3.4 ou 6.0)
- zzzz significa "*no*"ou "*yes1*", o primeiro denota a não existência de um conjunto de atribuições e o segundo que existe apenas uma atribuição de variáveis que torne o problema satisfazível
- o último *j* indica apenas qual j-ésima instância dentro daquele grupo.

A razão para escolha desse conjunto de instâncias é que os solucionadores são relativamente simples e o tempo necessário para chegarmos a uma resposta, nesse conjunto de instâncias, é relativamente baixo. Para as simulações, foi adotado um tempo limite de 1000 segundos, ou seja, o solucionador tem este tempo limite para encontrar uma atribuição para as variáveis que torne a instância proposta satisfazível ou indicar que a

instância não é satisfazível. Se pacotes de instâncias mais complexas forem utilizados, um número grande de instâncias do conjunto romperá o tempo limite para ambos os solucionadores, prejudicando a análise dos resultados. O objetivo, no momento, é uma análise relativa entre dois solucionadores e não uma análise absoluta do desempenho de um solucionador.

Os solucionadores foram executados em um computador Pentium 4, com uma freqüência de 2.0 GHz e 1GByte de memória. O sistema operacional utilizado foi o Linux. Os resultados estão apresentados na Tabela 4.2. Na tabela, cada linha representa a instância avaliada e as colunas mostram: o número de variáveis da instância (*Var.*), o número de cláusulas (*Clá.*), se a instância é SAT ou não-SAT (sat) e o tempo em segundos gasto pelo solucionador. Foi utilizado o termo TO (*time out*), se o tempo necessário para chegarmos a uma resposta foi superior ao tempo limite. Para o solucionador DSharp, temos uma coluna indicando o número de subtrações realizadas e para o solucionador DPLL uma coluna indicando o número de decisões realizadas. Para as instâncias onde o solucionador DSharp é mais rápido, o tempo de solução está em negrito.

O que podemos observar dos resultados é que existe uma nítida vantagem do processo de subtração de cubos sobre o DPLL clássico. Além de apresentar um tempo de solução inferior, o solucionador é mais robusto dentro do tempo limite especificado. Se desconsiderarmos as instâncias onde um, ou ambos, apresentaram um tempo superior ao tempo limite, o solucionador DSharp se mostra 3,57 vezes mais veloz que o DPLL. Apesar de encontrarmos instâncias onde este ganho é superior a 1000 vezes. Aparentemente não se pode correlacionar a melhoria obtida com o fato da instância ser SAT ou não-SAT e nem com a relação cláusulas/variáveis.

Um grande problema encontrado e que tornou o solucionador DSharp lento, foi a estrutura de dados utilizada para a representação dos cubos. Tendo em mente que, a cada subtração realizada, no mínimo um elemento, ou uma variável especificada, é inserida no cubo resposta, de acordo com a implementação utilizada, isso significou

percorrer a lista de variáveis para a inserção desse elemento. Esse fato tornou o tempo de resposta do solucionador dependente do tamanho da lista e, de uma forma mais cruel, da quantidade de subtrações. Essa característica é a principal desvantagem da estrutura de dados utilizada e que, fatalmente, inviabiliza a sua implementação. Mas, mesmo com essa estrutura de dados inadequada, o solucionador se mostra competitivo frente ao DPLL.

4.5 Conclusões

Apesar das desvantagens mencionadas não indicarem um futuro promissor, aparentemente, a subtração consegue restringir o espaço de procura saindo de regiões onde a solução não pode ser encontrada de uma forma mais rápida. Isso pode ser identificado pela quantidade de subtrações realizadas versus o número de decisões, para aquelas instâncias onde o solucionador DSharp se mostra superior. Podemos considerar que o resultado de uma subtração é equivalente a uma decisão, isto é, ao subtrairmos um cubo de outro, os cubos resultados possuem interceptos nulos e diferem de apenas uma variável entre si. Portanto, a situação é relativamente semelhante à tentativa do DPLL de se verificar uma variável em ambas as fases. A grande intuição é que existe uma ordem nessa escolha de variáveis. E essa ordem é obtida através de uma cláusula. Quando uma cláusula é selecionada para ser subtraída, automaticamente a seqüência de escolha de variáveis adotada estará relacionada com as variáveis presentes na cláusula, é como se um certo determinismo estivesse sendo incorporado às escolhas e, esse determinismo, dado pela presença ou não de uma variável em uma determinada cláusula.

Essa é uma das principais intuições desse trabalho. Principalmente considerando que as melhorias atuais estão relacionadas com a existência de estruturas dentro da formulação CNF de um problema. O capítulo seguinte discutirá mais detalhadamente essa conclusão inicial.

Entretanto, observando a quantidade de subtrações realizadas e sabendo que a cada

Instância	Var.	Clá.	sat	D-Sharp		DPLL	
				tempo	D-Sharp	tempo	decisões
\aim-50-1_6-no-1.cnf	50	80	não	**0,03**	4685	9,55	677573
\aim-50-1_6-no-2.cnf	50	80	não	**0,03**	4432	2,02	154649
\aim-50-1_6-no-3.cnf	50	80	não	**0,08**	9687	37,13	3106549
\aim-50-1_6-no-4.cnf	50	80	não	**0,00**	591	7,47	576398
\aim-50-1_6-yes1-1.cnf	50	80	sim	**0,02**	505	0,72	41106
\aim-50-1_6-yes1-2.cnf	50	80	sim	**0,00**	253	2,00	164442
\aim-50-1_6-yes1-3.cnf	50	80	sim	**0,00**	183	0,02	1003
\aim-50-1_6-yes1-4.cnf	50	80	sim	**0,02**	50	0,22	16043
\aim-50-2_0-no-1.cnf	50	100	não	**0,06**	5972	4,41	268363
\aim-50-2_0-no-2.cnf	50	100	não	**0,08**	8485	0,61	29735
\aim-50-2_0-no-3.cnf	50	100	não	**0,02**	2356	1,31	63517
\aim-50-2_0-no-4.cnf	50	100	não	**0,02**	887	0,45	22771
\aim-50-2_0-yes1-1.cnf	50	100	sim	**0,02**	1005	0,06	3917
\aim-50-2_0-yes1-2.cnf	50	100	sim	0,00	220	0,00	45
\aim-50-2_0-yes1-3.cnf	50	100	sim	**0,02**	66	0,22	11534
\aim-50-2_0-yes1-4.cnf	50	100	sim	**0,00**	412	0,05	1462
\aim-50-3_4-yes1-1.cnf	50	170	sim	0,02	1579	0,02	287
\aim-50-3_4-yes1-2.cnf	50	170	sim	0,34	17461	0,00	118
\aim-50-3_4-yes1-3.cnf	50	170	sim	0,14	7926	0,02	460
\aim-50-3_4-yes1-4.cnf	50	170	sim	0,34	20574	0,00	50
\aim-50-6_0-yes1-1.cnf	50	300	sim	**1,48**	54834	0,00	19
\aim-50-6_0-yes1-2.cnf	50	300	sim	0,03	870	0,00	15
\aim-50-6_0-yes1-3.cnf	50	300	sim	0,30	10626	0,00	7
\aim-50-6_0-yes1-4.cnf	50	300	sim	0,06	2237	0,00	4
\aim-100-1_6-no-1.cnf	100	160	não	**11,81**	637037	TO	47335278
\aim-100-1_6-no-2.cnf	100	160	não	**14,19**	681676	TO	51871191
\aim-100-1_6-no-3.cnf	100	160	não	**195,86**	8776278	TO	60047331
\aim-100-1_6-no-4.cnf	100	160	não	**11,14**	534113	TO	50964667
\aim-100-1_6-yes1-1.cnf	100	160	sim	**0,02**	376	TO	50925975
\aim-100-1_6-yes1-2.cnf	100	160	sim	**0,00**	252	TO	60220109
\aim-100-1_6-yes1-3.cnf	100	160	sim	**0,27**	8362	TO	53788991
\aim-100-1_6-yes1-4.cnf	100	160	sim	**0,00**	342	TO	60716904
\aim-100-2_0-no-1.cnf	100	200	não	**0,02**	939	TO	38526055
\aim-100-2_0-no-2.cnf	100	200	não	**10,31**	375814	TO	38154567
\aim-100-2_0-no-3.cnf	100	200	não	**0,11**	5473	TO	51414836
\aim-100-2_0-no-4.cnf	100	200	não	**0,25**	9820	TO	51254588
\aim-100-2_0-yes1-1.cnf	100	200	sim	**0,09**	2605	80,53	3091054
\aim-100-2_0-yes1-2.cnf	100	200	sim	**0,49**	13427	497,70	19722579
\aim-100-2_0-yes1-3.cnf	100	200	sim	**0,34**	9009	582,95	24884461
\aim-100-2_0-yes1-4.cnf	100	200	sim	**0,39**	11283	83,66	3604840
\aim-100-3_4-yes1-1.cnf	100	340	sim	**0,69**	14629	0,17	3827
\aim-100-3_4-yes1-2.cnf	100	340	sim	**0,17**	3976	1,00	27284
\aim-100-3_4-yes1-3.cnf	100	340	sim	**2,67**	62549	0,31	7108
\aim-100-3_4-yes1-4.cnf	100	340	sim	**4,98**	98781	1,25	26933
\aim-100-6_0-yes1-1.cnf	100	600	sim	**3,80**	48573	0,02	52
\aim-100-6_0-yes1-2.cnf	100	600	sim	329,45	3601997	0,09	1516
\aim-100-6_0-yes1-3.cnf	100	600	sim	16,50	186819	0,03	355
\aim-100-6_0-yes1-4.cnf	100	600	sim	4,20	50062	0,08	1205
\aim-200-1_6-no-1.cnf	200	320	não	TO	25661480	TO	30786950
\aim-200-1_6-no-2.cnf	200	320	não	TO	28783421	TO	49933961
\aim-200-1_6-no-3.cnf	200	320	não	TO	30988732	TO	55053451
\aim-200-1_6-no-4.cnf	200	320	não	**23,44**	538794	TO	45443697
\aim-200-1_6-yes1-1.cnf	200	320	sim	**0,33**	3443	TO	52355974
\aim-200-1_6-yes1-2.cnf	200	320	sim	**0,34**	4179	TO	45480414
\aim-200-1_6-yes1-3.cnf	200	320	sim	**1,17**	12566	TO	50940039
\aim-200-1_6-yes1-4.cnf	200	320	sim	**5,39**	68211	TO	43316685
\aim-200-2_0-no-1.cnf	200	400	não	**461,70**	6753021	TO	41276503
\aim-200-2_0-no-2.cnf	200	400	não	**292,92**	5083387	TO	44556973
\aim-200-2_0-no-3.cnf	200	400	não	**15,42**	311355	TO	34995057
\aim-200-2_0-no-4.cnf	200	400	não	**19,97**	371884	TO	32570134
\aim-200-2_0-yes1-1.cnf	200	400	sim	**8,59**	103898	TO	33546306
\aim-200-2_0-yes1-2.cnf	200	400	sim	**33,33**	348783	TO	28358689
\aim-200-2_0-yes1-3.cnf	200	400	sim	**2,06**	21736	TO	28689764
\aim-200-2_0-yes1-4.cnf	200	400	sim	**76,95**	800985	TO	26254739
\aim-200-3_4-yes1-1.cnf	200	680	sim	TO	23635574	TO	15694537
\aim-200-3_4-yes1-2.cnf	200	680	sim	TO	19827456	TO	16666733
\aim-200-3_4-yes1-3.cnf	200	680	sim	TO	18654321	TO	14611243
\aim-200-3_4-yes1-4.cnf	200	680	sim	TO	19872345	TO	15433739
\aim-200-6_0-yes1-1.cnf	200	1200	sim	**2,41**	13075	TO	18654431
\aim-200-6_0-yes1-2.cnf	200	1200	sim	**11,34**	54793	14,66	112002
\aim-200-6_0-yes1-3.cnf	200	1200	sim	**2,53**	14038	25,77	186499
\aim-200-6_0-yes1-4.cnf	200	1200	sim	**3,72**	16965	19,28	154321

Tabela 4.2: Resultados dos solucionadores DSharp e DPLL

subtração existe uma proliferação de novos cubos, constatamos a ineficiência do processo. Não apenas pelo problema da estrutura de dados mencionada, mas também pelo consumo de memória. Utilizar um solucionador DSharp DFS é ainda ineficiente, já que também não estariam presentes todas as características de solucionadores modernos; heurísticas de decisão, computação das implicações lógicas (BCP), retrocessos não cronológicos (NCB), aprendizado de cláusulas, reinicializações e etc.

Uma questão importante é: Considerando a intuição existente na subtração, poderíamos modificar um solucionador DPLL de tal forma que implementasse a subtração de cláusulas, assim como um solucionador DSharp DFS, sem perder todas as técnicas mais sofisticadas atualmente presentes nos solucionadores mais competitivos? E respondida a questão, viria a segunda: O solucionador orientado dessa forma aos cubos seria mais eficiente?

A resposta é sim e é a idéia principal desse trabalho. O próximo capítulo apresenta a sua implementação em dois solucionadores DPLL simples, para a avaliação do conceito e resposta à primeira questão. Posteriormente, a sua implementação em um solucionador no estado-da-arte para a avaliação do desempenho e resposta à segunda questão.

Capítulo 5

Decisões DSharp em um solucionador DPLL

Esse capítulo mostra os detalhes necessários para a implementação do algoritmo DSharp sobre um solucionador baseado no DPLL. Inicialmente, sobre dois solucionadores sem as melhorias encontradas nos atuais solucionadores, apenas para avaliar o desempenho dos conceitos teóricos. Os resultados obtidos nesses dois solucionadores mostram que a estratégia é eficiente e também motivam a sua implementação em um solucionador no estado da arte. Dessa vez, com todas as estratégias descritas no Capítulo três, incorporadas ao solucionador.

5.1 Árvore de pesquisa DSharp e árvore de pesquisa DPLL

Quando observamos a árvore de pesquisa gerada pela operação DSharp vemos que, diferentemente da árvore de pesquisa gerada por uma pesquisa DPLL, ela não é binária. Entretanto, as folhas geradas pela árvore DSharp, que são os subespaços resultantes da subtração, são os mesmos gerados pela árvore binária do DPLL. Vamos supor, por exemplo, que o cubo (0011) é obtido a partir de uma instância de apenas uma cláusula

$(x_1 \vee x_2 \vee \neg x_3 \vee \neg x_4)$. As atribuições obtidas a partir da subtração do cubo (0011) de (XXXX) podem ser visualizadas também como uma árvore binária de procura, a Figura 5.1 ilustra essa situação, onde, a árvore binária de procura de um solucionador DPLL (à direita) produz, exatamente, as mesmas folhas que a árvore DSharp (à esquerda). Essa equivalência nos permite sobrepor as duas árvores e nos beneficiar de todas as técnicas que já foram desenvolvidas para solucionadores SAT baseados em DPLL: BCPs rápidos, heurísticas de decisão, retrocessos não cronológicos e aprendizado de cláusulas. De forma análoga, todos os solucionadores baseados em DPLL podem utilizar os conceitos da resolução DSharp, em particular no processo de escolha de variáveis, quando um determinismo não encontrado na resolução DPLL pode agora ser inserido.

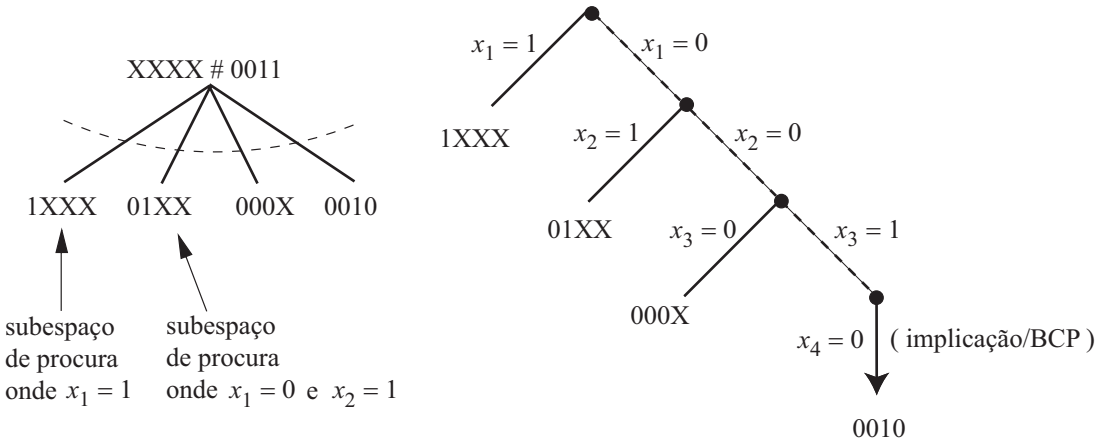

Figura 5.1: Árvore de pesquisa DSharp (esquerda) e árvore de pesquisa DPLL (direita)

5.2 Decisão DSharp no algoritmo DPLL básico

Um dos aspectos mais importantes e que motivou a implementação do algoritmo DSharp em um solucionador SAT baseado em DPLL é que poderíamos inserir a decisão DSharp apenas nos momentos em que um retrocesso não cronológico ocorre. Neste momento retornaríamos à cláusula de conflito e procuraríamos valores que tornassem esta cláusula SAT. Se nenhuma variável estiver disponível nessa cláusula, procederíamos a um retrocesso sobre a primeira variável atribuída dessa cláusula. Esse procedimento

permite abandonar, mais rapidamente, regiões do espaço de procura onde uma atribuição que satisfaça a cláusula não esteja presente, uma vez que estamos refinando a procura através da associação do conflito com uma cláusula. Desde que a ocorrência de retrocessos não cronológicos em instâncias de problemas que possuem algum tipo de estrutura é mais rara nos solucionadores SAT atuais, a aplicação do algoritmo será esporádica, adicionando pouco custo à procura. Outro benefício é estarmos aprendendo mais com o conflito cronológico e, permitindo o aprendizado de cláusulas mais relacionadas com aquela região do espaço de procura que ocasionou o conflito.

Inicialmente, as decisões baseadas em DSharp são introduzidas em dois algoritmos, aqui denominados DPLL1 e DPLL2. Esses algoritmos executam o algoritmo clássico DPLL. Para o DPLL1 as variáveis de decisão são aleatoriamente escolhidas e não existe o aprendizado de cláusulas. Para o DPLL2, o mesmo procedimento para as decisões é adotado, entretanto, existe o aprendizado de cláusulas. Para ambos o retrocesso é sempre cronológico. Essa foi a estratégia de análise adotada, já que o objetivo no momento é verificar apenas a eficiência das decisões baseadas no DSharp. O aprendizado de cláusulas no DPLL2 irá nos dizer se, com as decisões orientadas pelo DSharp, as cláusulas aprendidas podem ser mais úteis, permitindo que o algoritmo saia, mais rapidamente, de regiões do espaço de procura onde um conjunto de atribuições que leve à satisfabilidade da instância não possa ser encontrado, no caso de uma instância SAT, ou que retorne que a instância não é satisfazível, no caso não-SAT.

Para um melhor entendimento do processo de inserção das decisões D-Sharp, o algoritmo básico DPLL é apresentado aqui em um nível de detalhamento maior. A implementação utilizada é uma adaptação do algoritmo disponibilizado por Lintao Zhang em [URL14]. Como mencionado, as inovações descritas no capítulo anterior não estão presentes, entretanto, é o algoritmo base dos principais solucionadores do problema da satisfabilidade atualmente encontrados. A Figura 5.2 ilustra a rotina principal e as figuras que se seguem, as funções internas.

A estrutura de dados é a mesma do solucionador adotado no capítulo anterior

e baseada em uma lista de adjacências, algumas características adicionais são aqui mencionadas para uma melhor compreensão das funções internas:

- Para cada cláusula, existe de um vetor com os literais presentes e existem contadores indicando a quantidade de literais avaliados como verdadeiros ou falsos;
- Para cada variável, existe: uma lista de cláusulas onde ela aparece, o valor da mesma (verdadeiro, falso ou livre), a fase atribuída à variável, o nível da árvore de decisão onde a atribuição de valor foi realizada e um controle, identificando se a variável já foi avaliada nas suas duas fases (invertida ou não);
- Um vetor de cláusulas, com todas as cláusulas da instância;
- Um vetor de variáveis, com todas as variáveis da instância;
- Uma lista de implicações, contendo as variáveis que deverão ser implicadas (BCP);
- Uma matriz de atribuições, onde cada linha representa o nível na árvore de decisões e cada coluna representa cada atribuição realizada.

```
DPLL1 ( )
início
    enquanto ( tempo de solução fornecido não é ultrapassado)
        se (DPLL_decide ( ) = verdadeiro)
            enquanto ( DPLL_deduz ( ) = CONFLITO)
                nível de retrocesso = DPLL_analisa_conflito ( );
                se (nível de retrocesso = 0)
                    retorne não-SAT;
                senão
                    DPLL_backtrack (nível de retrocesso);
        fimse
        senão
            retorne SAT;
    fimenquanto
    retorne (TEMPO-ULTRAPASSADO);
fim
```

Figura 5.2: Algoritmo base DPLL1

A Figura 5.3 ilustra a função DPLL_decide ( ), que é responsável por incrementar o atual nível de decisão e, aleatoriamente, escolher uma fase para uma variável livre.

Após a escolha, a variável é inserida na lista de implicações que é quem administra o BCP. É essa lista de implicações que indica quais serão as variáveis atribuídas.

```
booleano DPLL_decide ( )
início
    nível de decisão=nível de decisão + 1;
    para ( i =1 até total de variáveis da instância)
    se (variáveis[i].valor = variável livre)
        variáveis[i].fase= escolha aleatória da fase;
        variáveis[i].invertida = falso;
        insira variáveis[i] na lista de implicações;
        retorne verdadeiro;
    fimse
    retorne falso;
fim
```

Figura 5.3: Função DPLL decide

A Figura 5.4 ilustra a função DPLL_analisa_conflito ( ), que é responsável por promover o retrocesso, como nesse caso temos apenas o retrocesso cronológico, ela simplesmente identifica qual foi a variável de decisão no nível onde o conflito surgiu e insere, na lista de implicações, essa variável com a fase invertida. Se essa variável já foi invertida, significando que já foi avaliada nas suas duas fases, deveremos subir na árvore de decisões. Se a função retornar zero, significa que o problema não é satisfazível.

```
inteiro DPLL_analisa_conflito ( )
início
    para (i = nível de decisão atual até o nível=0 )
        vetor de atribuições no nível  = vetor de atribuições[i];
        literal de decisão = vetor de atribuições no nível[0];
        se (variáveis[literal de decisão].invertido = falso)
            variáveis[literal de decisão].invertido = verdadeiro;
            insira literal de decisão com a fase invertida na lista de implicações;
            retorne i;
        fimse
    fimpara
    retorne 0;
fim
```

Figura 5.4: Função DPLL para analise de conflito

A Figura 5.5 ilustra a função DPLL_backtrack ( ), é a função responsável por caminhar no vetor de atribuições de cada nível e desfazer todas as atribuições realizadas. Esse processo sobe, na árvore de implicações, a partir no atual nível de decisão até o ponto onde o retrocesso acontece.

```
DPLL_backtrack ( nível do retrocesso)
início
    para ( i = nível de decisão atual até nível de retrocesso)
    |   vetor de atribuições no nível  = vetor de atribuições[i];
    |   para (j=0 até todas as variáveis do vetor de atribuição no nível)
    |   |   x = vetor de atribuição no nível [j];
    |   |   desfaz atribuição (x);
    |   |   variáveis[x].nível de decisão = -1;
    |   fimpara
    fimpara
    nível de decisão atual = nível de retrocesso;
fim
```

Figura 5.5: Função DPLL para o retrocesso ou *backtrack*

A Figura 5.6 ilustra a função DPLL_deduz ( ), é a função que promove o BCP. A cada chamada, irá percorrer a lista de implicações. Durante esse percurso, fará as atribuições de valor de todas as variáveis da lista de implicações, se não for possível executar a atribuição, a variável não é uma variável livre, a função retorna que detectou um conflito.

```
inteiro DPLL_deduz ( )
início
|   enquanto ( lista de implicações não está vazia e conflitos = 0)
|   |   x = retira o primeiro elemento da lista de implicações;
|   |   se ( x.valor = variável livre)
|   |   |   faz atribuição (x, literal associado a x);
|   |   |   x.nível de decisão = nível de decisão atual;
|   |   |   retira x do vetor de atribuições [nível de decisão atual];
|   |   fimse
|   fimenquanto
|   se (lista de implicações não está vazia )
|       retire todos os elementos da lista de implicações;
|   se (conflitos = 0) retorne NÃO_CONFLITO;
|   senão retorne CONFLITO;
fim
```

Figura 5.6: Função DPLL responsável pela dedução

A Figura 5.7 ilustra a função DPLL_faz atribuição ( ), é parte integrante do BCP, é a função que altera o valor da variável e administra os contadores de variáveis verdadeiras e falsas na cláusula. Esse procedimento de avaliar os contadores, conforme descrito no capítulo anterior, identifica se uma nova implicação é originada pela atribuição de uma variável e, é conseqüentemente inserida na lista de implicações, ou se um conflito surgiu.

```
DPLL_faz atribuição (variável x, valor)
início
    x.valor = valor;
    para ( todas as clausulas que possuem a variável x na mesma fase do valor atribuído)
        incrementa o contador de variáveis verdadeiras na cláusula;
    para ( todas as clausulas que possuem a variável x na fase oposta ao valor atribuído)
        incrementa o contador de variáveis falsas na cláusula;
        se (num. de variáveis falsas = num. de literais na cláusula)
            incrementa conflitos;
        senão
            se (num. de variáveis verdadeiras = 0 e num. de variáveis falsas = total de
                variáveis na cláusula –1)
                para (j=0 até última variável da cláusula)
                    se (variáveis[j].valor = variável livre)
                    insira variáveis[i] na lista de implicações;
                    saia do laço;
                fimpara
            fimse
    fimpara
fim
```

Figura 5.7: Função DPLL que faz a atribuição de valores

A Figura 5.8 ilustra a função DPLL_desfaz atribuição ( ), é a função que desfaz uma atribuição e recupera o valor dos contadores de variáveis verdadeiras ou falsas na cláusula.

```
DPLL_desfaz atribuição (variável x)
início
    para ( todas as clausulas que possuem a variável x na mesma fase do valor atribuído)
        decrementa o contador de variáveis verdadeiras na cláusula;
    para ( todas as clausulas que possuem a variável x na fase oposta ao valor atribuído)
        decrementa o contador de variáveis falsas na cláusula;
    x.valor = variável livre;
fim
```

Figura 5.8: Função DPLL desfaz atribuição

Se observarmos a árvore D-Sharp apresentada na Figura 5.1, a primeira folha mais à esquerda, está associada com a primeira decisão sobre uma variável. A folha a seguir representa a primeira variável com a fase oposta e uma decisão sobre a segunda variável, o mesmo acontece com as folhas seguintes. Esta é a ordem imposta pelas decisões DSharp e que segue a mesma seqüência imposta pela operação DSharp mostrada no capítulo anterior. Os algoritmos a seguir ilustram quais devem ser as alterações no DPLL original para a inserção desse conceito. O algoritmo base é o mesmo DPLL,

apenas as funções internas se alteram e são aqui apresentadas com o prefixo DSharp. Dessa forma, basta substituir a função DPLL_decide ( ) pela DSharp_decide ( ), DPLL_analisa conflito( ) pela DSharp_analisa conflito ( ) e assim por diante, na rotina principal da Figura 5.2.

Além de alterações no algoritmo, a estrutura de dados é modificada para que:

- uma cláusula possa ser identificada como associada ou não;
- cada variável possua um controle indicando, se estiver associada a uma cláusula, a qual cláusula ela está associada.

A Figura 5.9 ilustra a função DSharp_analisa_conflito (), que é efetivamente onde se dá o início de uma decisão DSharp. Quando um conflito é detectado, em um retrocesso cronológico, o que se segue é a implicação da variável de decisão com a fase oposta. Considerando a árvore de decisão DSharp, isso corresponde exatamente ao comportamento de uma variável na transição de uma folha para a outra, é essa diferença que promove a interseção nula entre estas duas folhas. Nesse ponto, o que nos interessa é identificar a variável de decisão do nível, já que essa decisão gerou, possivelmente, uma série de implicações, mas que culminaram em um conflito. Observando a função, o que fazemos é principalmente identificar qual a variável de decisão e armazená-la como a variável de decisão DSharp. O restante da função é similar à função DPLL_analisa_conflito(), que insere na lista de implicações, essa variável de decisão com a sua fase invertida.

A Figura 5.10 ilustra a função DSharp_decide (). Se novamente retornarmos à árvore de decisão DSharp, observamos que, a segunda folha corresponde à primeira variável com a fase oposta à folha mais a esquerda e uma segunda variável que é originada do cubo a ser subtraído. Esse cubo, na verdade é uma cláusula, e que, se associada à variável de decisão, forçará a escolha dessa segunda variável. Se existir essa variável de decisão DSharp, a próxima variável de decisão é determinada computando-se o próximo cubo disjunto. Subtrai-se a cláusula cubo associada à variável de decisão do atual espaço de procura, que é o resultado do conjunto atual de atribuições das variáveis.

```
inteiro DSharp analisa_conflito ( )
início
    para (i = nível de decisão atual até o nível=0 )
        vetor de atribuições no nível  = vetor de atribuições[i];
        literal de decisão = vetor de atribuições no nível[0];
        Variável de decisão D-Sharp = variáveis[literal de decisão];
        se (variáveis[literal de decisão].invertido = falso)
            variáveis[literal de decisão].invertido = verdadeiro;
            insira literal de decisão com a fase invertida na lista de implicações;
            retorne i;
        fimse
    fimpara
    retorne 0;
fim
```

Figura 5.9: Função DSharp analisa conflito

```
booleano DSharp_decide ( )
início
    nível de decisão=nível de decisão + 1;
        se ( variável de decisão DSharp associada a alguma clausula w)
            se ( cláusula w associada não-SAT)
                escolha uma variável x na cláusula w de acordo com a ordem D-Sharp;
                associe x com a cláusula w ;
                insira x na lista de implicações;
                retorne verdadeiro;
            fimse
        senão
            associe a variável de decisão D-Sharp com alguma cláusula w não-SAT;
            escolha uma variável x na cláusula w de acordo com a ordem D-Sharp;
            associe x com a cláusula w;
            insira x na lista de implicações;
            retorne verdadeiro;
        fimsenão

    para ( i =1 até total de variáveis da instância)
    se (variáveis[i].valor = variável livre)
        variáveis[i].fase= escolha aleatória da fase;
        variáveis[i].invertida = falso;
        insira variáveis[i] na lista de implicações;
        retorne verdadeiro;
    fimse
    retorne falso;
fim
```

Figura 5.10: Função DSharp decide

Essa é a ordem DSharp, e que segue a mesma ordem da operação DSharp descrita no capítulo anterior. A cláusula cubo é escolhida apenas após um retrocesso cronológico se não existir uma cláusula cubo previamente associada à variável de decisão, caso contrário, quando esta associação já existir, iremos insistir nesta cláusula cubo tentando encontrar outro espaço de procura resultante da aplicação do DSharp, se este espaço existir, outra variável livre desta cláusula também será associada com esta cláusula e atribuída de forma a satisfaze-la. Se uma decisão DSharp não for realizada, o algoritmo segue o mesmo critério de decisão do DPLL original conforme pode ser observado.

A Figura 5.11 ilustra as funções DSharp_backtrack ( ) e DSharp_desfaz atribuição ( ). O objetivo dessas funções é o mesmo que o DPLL original, porém, elas agora, também devem desfazer todas as associações de variáveis com as cláusulas, caso o retrocesso aconteça para um nível anterior ao nível onde as associações foram realizadas.

```
D-Sharp backtrack(nível do retrocesso)
início
    para ( i = nível de decisão atual até nível de retrocesso)
        vetor de atribuições no nível  = vetor de atribuições[i];
        para (j=0 até todas as variáveis do vetor de atribuição no nível)
            x = vetor de atribuição no nível [j];
            D-Sharp desfaz atribuição (x);
            variáveis[x].nível de decisão = -1;
        fimpara
    fimpara
    nível de decisão atual = nível de retrocesso;
fim

D-Sharp desfaz atribuição (variável x)
início
    para ( todas as clausulas que possuem a variável x na mesma fase do valor atribuído)
        decrementa o contador de variáveis verdadeiras na cláusula;
        se (variável x associada com alguma cláusula)
            desfaz associação;
    para ( todas as clausulas que possuem a variável x na fase oposta ao valor atribuído)
        decrementa o contador de variáveis falsas na cláusula;
        se (variável x associada com alguma cláusula)
            desfaz associação;
    x.valor = variável livre;
fim
```

Figura 5.11: Função DSharp backtrack e Função DSharp desfaz atribuição

Examplo 1: A forma como o algoritmo trabalha e a intuição do mesmo, podem ser melhor visualizadas na figura 5.12, que ilustra um problema SAT resolvido através de

um solucionador básico DPLL e um solucionador DPLL onde o DSharp foi incorporado.

Vamos assumir por simplicidade que, apenas os retrocessos cronológicos estão sendo utilizados, assim como na resolução clássica do DPLL. Nota-se que quaisquer novas técnicas associadas ao solucionador DPLL clássico também serão absorvidas pelo solucionador DSharp, já que limitamos a sua utilização nas situações onde o retrocesso é não cronológico e onde as mais recentes melhorias não afetam significativamente o desempenho.

Vamos considerar que decisões livres são realizadas e, ao atingirmos o nível 4, a decisão $x_4 = 1$ chega a um conflito devido a x_7. Após inverter a variável de decisão, e executar todas as implicações lógicas devido a essa inversão, se não encontrarmos novos conflitos, iniciamos um novo nível de decisão. Nesse ponto, o solucionador deve escolher uma nova variável de decisão. Se, escolhermos uma nova variável completamente não relacionada com x_4, não estaremos utilizando de forma eficiente todo o conhecimento acumulado até o momento nesse local da procura, isso é o que o solucionador clássico DPLL irá fazer. Contudo, se escolhermos uma variável de alguma cláusula relacionada com x_4, estaremos aumentando a chance de usar o nosso conhecimento local e, ao mesmo tempo, aumentando esse conhecimento se novas cláusulas relacionadas a x_4 forem aprendidas.

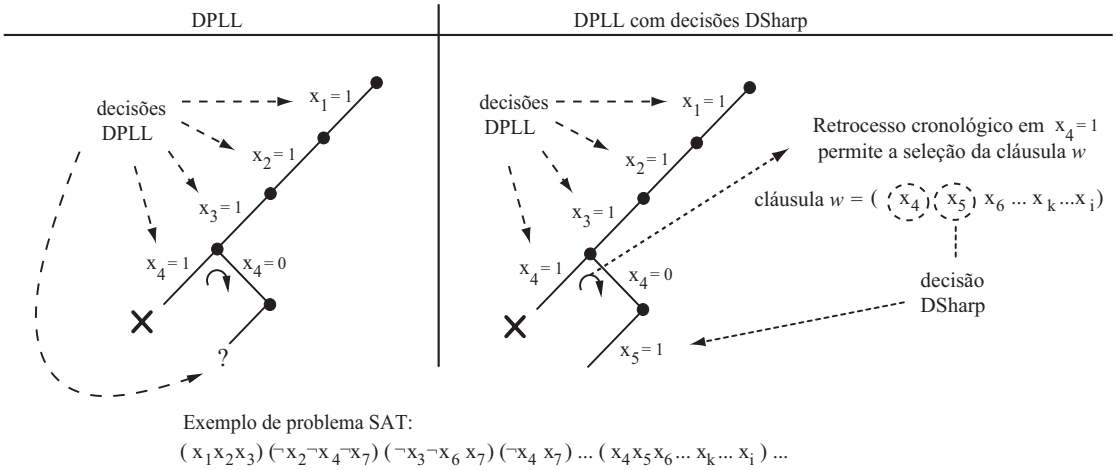

Figura 5.12: Exemplo de resolução de um problema SAT com DPLL e DPLL+DSharp

Nesse Exemplo 1, o algoritmo DSharp inverte a variável x_4 para $x_4 = 0$, seleciona

uma cláusula w para ser a cláusula atual a ser subtraída e a associa com x_4. Observa-se que a cláusula w não é satisfazível com a atribuição $x_4 = 0$. Então, a próxima variável de decisão, deverá vir dessa cláusula associada. Essa decisão forçada, que poderão ser várias se a cláusula associada for longa, torna o algoritmo DSharp mais restrito na decisão de variáveis do que os solucionadores clássicos DPLL. De fato, estamos primeiramente escolhendo cláusulas e não variáveis. Uma vez escolhida uma cláusula, deveremos selecionar variáveis dessa cláusula, uma após a outra, até não mais existirem variáveis a serem selecionadas. É claro que: o BCP é executado após cada atribuição, a análise de conflitos é realizada e, se retrocessos não cronológicos e aprendizado de cláusulas existirem, eles serão realizados. Entretanto, existe uma diferença sutil: quando ocorrer o retrocesso cronológico, iremos retroceder para uma cláusula escolhida e nos fixarmos a ela até resolve-la, para que não ocasione conflitos. Não iremos selecionar e associar cláusulas a variáveis para os retrocessos não cronológicos. A razão é que, esses retrocessos não implicam a variável de decisão com a sua fase invertida, eles simplesmente apagam todas as decisões até um determinado nível e implicam o 1UIP, conforme mencionado no Capítulo 2. Mais detalhes sobre esse aspecto serão dados mais à frente, quando aplicarmos a decisão DSharp sobre um solucionador no estado da arte e que utiliza esse tipo de análise de conflitos.

O algoritmo DSharp integrado ao solucionador baseado em DPLL executa, para a FNC, as mesmas operações definidas anteriormente para os cubos. Não é necessário converter as cláusulas na FNC para cubos e aplicar a operação DSharp como definida anteriormente, o que fazemos é atribuir as variáveis da forma como elas seriam atribuídas ao executarmos uma operação DSharp. Isso é o que o algoritmo da Figura 5.10 faz, e o que esse exemplo está demonstrando.

Continuando com o Exemplo 1, seja $w = (x_4 \vee x_5 \vee x_6 \vee \ldots \vee x_k \vee \ldots \vee x_i)$ a cláusula associada a x_4. O operador DSharp é aplicado negando-se w como demonstrado no Capítulo 4 e executando a subtração desta cláusula-cubo do espaço de solução atual, formado pelas atribuições das variáveis até o momento: $c = (111XXX \ldots X \ldots X)$,

isto é , $x_1 = 1$, $x_2 = 1$ e $x_3 = 1$.

Desde que $\neg w = (\neg x_4 \wedge \neg x_5 \wedge \neg x_6 \wedge \ldots \wedge \neg x_k \wedge \ldots \wedge \neg x_i) = (XXX000...0...0)$, se executarmos a operação DSharp entre o atual espaço de procura e essa cláusula-cubo, iremos obter:

$(111XXX...X...X) \# (XXX\mathbf{000}...\mathbf{0}...\mathbf{0}) =$

$\{(111\mathbf{1}XX...X...X), (111\mathbf{01}X...X...X), (111\mathbf{001}...X...X), ..., (111\mathbf{000}...0...\mathbf{1})\}$

Os cubos acima, representam um conjunto de subespaços disjuntos que podem ser explorados por um solucionador SAT DPLL apenas atribuindo as variáveis de w ilustradas em negrito. O primeiro subespaço de procura disjunto $(111\mathbf{1}XX...X...X)$ corresponde à atribuição $x_4 = 1$. Essa é a atribuição da variável de decisão realizado no nível 4, que gerou um conflito e que nos diz que a solução não está neste subespaço de procura.

Uma atribuição $x_4 = 0$ e $x_5 = 1$ é o mesmo que o segundo subespaço de procura disjunto $(111\mathbf{01}X...X...X)$. Entretanto, nossa procura DSharp implementada no solucionador DPLL não salta imediatamente da tentativa do primeiro espaço disjunto de procura para esse segundo espaço de procura. Para que pudéssemos utilizar o BCP, nós inicialmente invertemos a variável x_4 para $x_4 = 0$ e executamos o BCP. Se novos conflitos forem encontrados resultantes da atribuição de $x_4 = 0$, nós procedemos ao retrocesso e não existe a necessidade da tentativa de procura em um outro subespaço disjunto. Se nenhum conflito for encontrado com a atribuição de $x_4 = 0$ então, uma nova variável de decisão será necessária. Esse é o momento certo de associar a cláusula w com a variável x_4 e selecionar uma nova variável livre dessa cláusula associada.

Ao escolhermos x_5 de w e atribuirmos $x_5 = 1$, obtemos o segundo subespaço disjunto de procura DSharp $(111\mathbf{01}X...X...X)$. Se após o BCP encontrarmos novamente um conflito devido a esta atribuição, tentaremos $x_5 = 0$. O processo será o mesmo para os próximos literais, sempre retornando a essa cláusula toda vez que um conflito for encontrado e, procurando selecionar um literal de w no próximo nível. Se escolhermos x_6 de w e atribuirmos $x_6 = 1$, teremos o terceiro subespaço DSharp $(111\mathbf{001}...X...X)$.

Se após o BCP encontrarmos um conflito devido a essa atribuição, iremos tentar $x_6 = 0$. Tomando x_k de w e atribuindo $x_k = 1$, teremos o $(k-n)$ésimo subespaço DSharp de procura $(111\mathbf{000}...\mathbf{1}...X)$. Se, após o BCP, encontrarmos um conflito após essa atribuição, iremos tentar $x_k = 0$.

Observa-se que o último subespaço $(111\mathbf{000}...0...\mathbf{1})$ será implicitamente avaliado. De fato, a cláusula associada se tornará uma cláusula unitária com a atribuição anterior e a atribuição $x_i = 1$ será forçada pelo BCP, ou simplesmente implicada. Se encontrarmos um conflito, iremos retroceder para a variável x_3 após ter avaliado todos os subespaços DSharp de $c = (111XXX \ldots X \ldots X)$, obtidos através da subtração da cláusula-cubo $\neg w = (XXX000...0...0)$.

Se não encontrarmos conflitos após uma atribuição, iremos executar a próxima decisão de variáveis de acordo com a heurística original de decisão. A rigor, o que deveríamos fazer é escolher a próxima *cláusula de decisão*, não variável. Entretanto, essa decisão sobre uma cláusula é adiada até que um retrocesso cronológico aconteça. Enquanto isso, a heurística de decisão nativa opera livremente, sabendo-se que, quando ela escolher uma variável, estará pré-selecionando as cláusulas dessa variável como possíveis cláusulas de retrocesso no futuro. Essa é uma pequena sutileza que pode induzir o leitor a pensar que abre-se mão da aplicação da subtração de cubos imediatamente após um retrocesso cronológico, não é o que acontece.

Em uma segunda avaliação, aqui denominada DPLL2, o mesmo algoritmo DPLL1 foi modificado de forma a permitir o aprendizado de cláusulas. Para que este procedimento possa ser utilizado, a estrutura de dados foi modificada de forma a identificar o antecedente de uma atribuição e, conseqüentemente, permitir o caminhamento no grafo de implicações durante a análise de conflitos, conforme descrito no Capítulo 3. O retrocesso utilizado nessa segunda versão é ainda o cronológico, conseqüentemente, o algoritmo utilizado, assim como a estrutura de dados adicionada para as decisões DSharp, são as mesmas que no DPLL1.

Os resultados obtidos podem ser visualizados nas Tabelas 5.1 a 5.3. Vale ressal-

tar, que esses resultados representam apenas uma avaliação preliminar dos conceitos teóricos, dessa forma, os tempos obtidos não devem ser comparados com os solucionadores atuais. No momento estamos apenas comparando os algoritmos DPLL1 e DPLL2 com e sem as decisões DSharp. Os algoritmos foram executados em instâncias de problemas da satisfabilidade DIMACs [Url13] e instâncias industriais, originadas da verificação formal de microprocessadores superescalares disponibilizadas por Miroslav N. Velev em [Url17]. O tempo limite para cada instância foi de 600 segundos para os problemas DIMACS e 2000 segundos para os problemas Velev, essas instâncias se mostraram mais complexas e, portanto, um tempo de corte maior foi necessário. As execuções foram realizadas em um processador Pentium 4, 2GHz, uma memória Ram de 1 GB e sistema operacional Linux. Para as decisões DSharp, a escolha da cláusula associada foi completamente aleatória, nenhuma heurística foi aplicada, assim como a escolha da próxima variável de decisão entre os literais desta cláusula associada.

Na Tabela 5.1, cada linha representa um grupo de instâncias, as colunas estão divididas em: *Num*, representando a quantidade de instâncias no grupo, *TL* indicando a quantidade de instâncias que não foram resolvidas dentro do tempo limite, e *tempo* para o somatório dos tempos para a solução de todas as instâncias do grupo. Para uma avaliação justa, o tempo limite é utilizado para as instâncias que o solucionador não conseguiu resolver dentro do tempo limite, assim, podemos avaliar um ganho mínimo de melhoria. Para um efeito comparativo, é apresentado o tempo de solução do solucionador Grasp de João Marques Silva [Sil96], que representou o estado da arte até o ano de 2000. Para o DPLL1, os resultados mostram uma pequena melhora quando as decisões DSharp são inseridas. Entretanto, o ganho de aproximadamente 3% no tempo de solução, é o valor mínimo, já que o tempo de corte foi utilizado como o tempo para aquelas que não foram solucionadas. Já para o solucionador DPLL2, onde o aprendizado de cláusulas está presente, temos uma melhora de 20% sobre o DPLL2 e 12% sobre o Grasp.

A Tabela 5.2 apresenta resultados mais reais. Todas as instâncias que atingiram o

Instância		DPLL1		DPLL1 + Dsharp		DPLL2		DPLL2 + Dsharp		GRASP	
	Inst.	TL	tempo	TL	tempo	TL	tempo	TL	tempo	TL	tempo
Aim-100	24	14	8459,8	12	7426,93	0	0,2	0	0,14	0	0,132
Aim-50	24	19	94,85	18	45,88	0	0,02	0	0,04	0	0,024
Aim-100	24	0	8459,8	0	7426,93	0	0,2	0	0,14	0	0,132
Ais	4	0	20,78	0	65,2	1	0,87	1	15,26	1	607,248
Dubois	6	4	2773,62	4	2664,18	0	0,08	0	128,84	0	0,124
Hanoi	2	1	869,85	1	747,42	0	29,77	0	19,21	2	1200
Logistics	4	4	2400	4	2400	0	15,89	0	63,03	0	37,417
Parity-8-16	20	2	1769,95	0	659,83	2	1866,14	0	466,99	2	2634,4
Pigeon-Hole	5	1	877,38	0	316,45	1	693,88	1	705,19	0	22,656
Pret	8	4	2719,96	4	2722,39	0	0,44	0	0,49	0	1,052
Ssa	8	7	4200,42	4	2402,48	2	1200,71	2	1200,65	0	1,004
Velev-sss.1.0a	9	9	18000	9	18000	6	12019,37	1	5562,92	7	13440,036
Velev-sss.1.0	48	46	92000,33	44	88000,64	26	54706,99	14	32870,06	19	22889,194
Velev-sss-sat-1.0	100	99	198000,65	99	198000,06	80	160692,12	72	152681,36	86	175464,443

Tabela 5.1: DPLL1, DPLL2 com e sem decisões DSharp e Grasp

tempo limite em qualquer dos solucionadores foram retiradas. Também foram retirados os resultados do DPLL1, já que a quantidade de instâncias resolvidas foi pequena dentro do tempo limite fornecido. A coluna *Num* representa a quantidade de instâncias dentro do grupo de instâncias para as quais os solucionadores chegaram a uma resposta, a coluna *Imp.,* a quantidade de implicações, a coluna *Conf.,* a quantidade de conflitos encontrados, a coluna *Dec.,* a quantidade de decisões realizadas, a coluna *tempo,* o somatório dos tempos para solucionar todas as instâncias do grupo e a coluna *retornos*, indica a quantidade de vezes que as decisões DSharp foram executadas. As instâncias onde o DSharp se mostra superior estão em negrito. Comparando o DPLL2 sem e com o DSharp, vemos:

- uma redução no número de implicações de 7.9%;
- uma redução no número de conflitos de 39.8%;
- um aumento no número de decisões de 13.4%;
- uma redução no tempo de solução de 12.8%.

Um aspecto interessante é o aumento do número de decisões. Que se justifica, já que estamos forçando as decisões insistindo sempre em uma cláusula, contudo, esse aumento representa uma redução significativa no número de conflitos e no número de implicações, indicando que: cláusulas mais úteis podem estar sendo aprendidas e

implicadas e que estamos saindo mais rapidamente de regiões do espaço de procura onde possivelmente uma solução não possa ser encontrada. Essas melhorias refletem no tempo necessário para a solução da instância. Observamos ainda que, as maiores melhorias correspondem às instâncias industriais e não às sintéticas ou aleatórias. Isso indica que, a existência de alguma estrutura na instância interfere, assim como nos atuais solucionadores, na atuação das decisões DSharp.

Instâncias	Num	DPLL2				DPLL2 + Dsharp				
		Imp.	Conf.	Dec.	tempo	Imp.	Conf.	Dec.	tempo	retornos
Aim-100	24	17257	825	2879	0,2	16705	811	2753	0,14	306
Aim-200	24	115628	3285	11798	0,79	**71966**	**2064**	**8225**	**0,69**	**822**
Aim-50	24	5860	372	1011	0,02	6383	434	986	0,04	135
Ais	3	23110	727	923	0,87	129057	3767	3782	15,26	1529
Dubois	6	9955	298	540	0,08	581261	44123	44943	128,84	189
Hanoi	0									
Logistics	4	843648	7755	40246	15,89	1751434	13645	39109	63,03	4456
Parity-8-16	17	9214863	68638	69183	666	**7113796**	**53937**	**55859**	**307,08**	**228**
Pigeon-Hole	4	253901	11637	16446	93,88	237356	12165	12508	105,19	7965
Pret	8	50401	2261	13952	0,44	52722	2615	13959	0,49	1004
Ssa	6	31011	153	1714	0,71	**17415**	**62**	**1188**	**0,65**	**13**
Velev-sss.1.0a	1	24127	563238	2418	19,37	**124152**	**6269827**	**31396**	**1078,16**	**6828**
Velev-sss.1.0	18	778241	14736022	116430	2706,18	**482625**	**15955650**	**119623**	**2541,16**	**17263**
Velev-sss-sat-1.0	11	145291	35368765	23072	1793,32	**75604**	**13939879**	**12556**	**451,98**	**3487**

Tabela 5.2: DPLL1 e DPLL2 com e sem decisões DSharp

A Tabela 5.3 é ilustrativa e mostra os resultados do solucionador Grasp para as mesmas instâncias da Tabela 5.2.

Instâncias	Num	GRASP			
		Imp.	Conf.	Dec.	tempo
Aim-100	24	15044	723	1425	0,132
Aim-200	24	74826	2228	4671	2,112
Aim-50	24	4858	350	594	0,024
Ais	3	49808	925	2051	7,248
Dubois	6	10184	935	2418	0,124
Hanoi	0				0
Logistics	4	222706	2004	12039	37,417
Parity-8-16	17	5589500	32081	32374	834,4
Pigeon-Hole	4	45214	3662	3800	14,032
Pret	8	32483	1884	16645	1,052
Ssa	6	9554	68	422	0,344
Velev-sss.1.0a	1	17719	39	232	1,636
Velev-sss.1.0	18	3042236	35544	44347	2880,594
Velev-sss-sat-1.0	11	2776526	7719	103353	1423,169

Tabela 5.3: Solucionador Grasp

Estes resultados foram os responsáveis por encorajar o desenvolvimento de um solucionador onde a decisão DSharp estivesse sobre uma plataforma DPLL e utilizando as principais tecnologias desenvolvidas pelos solucionadores no estado da arte.

5.3 Solucionador DSharp combinado com solucionador DPLL no estado da arte

Como as alterações para a inserção das decisões DSharp foram realizadas na base do algoritmo DPLL, elas podem ser inseridas em qualquer solucionador SAT baseado no algoritmo clássico DPLL. Entre os diversos solucionadores SAT no estado da arte (Minisat1.13, HaifaSat, Jerusat1.31, SatELiteGTI, zChaff e outros) [Url03], a decisão foi de modificar o solucionador zChaff , última versão disponível [Url05], e integrar no seu algoritmo, a estrutura de dados e as funções necessárias ao algoritmo DSharp.

Esse solucionador integra todas as características relevantes dos modernos solucionadores e, em conjunto com o solucionador Grasp [Sil96], é a base de construção de muitos solucionadores atualmente disponíveis, possui o código disponível para utilização pela comunidade e é de fácil entendimento, o que permite a comparação e reprodução das idéias em qualquer outro solucionador. Um exemplo da utilização desses solucionadores pode ser observada no solucionador Minisat, o atual vencedor na competição SAT [Url03]. Conforme descrito pelo seu autor, Niklas Een em [Een03], o solucionador Minisat é apresentado como " *uma implementação minimalista de um solucionador no estilo zChaff baseado na estratégia de observação de dois literais para BCPs rápidos e aprendizado de cláusulas pela análise de conflitos*".

Para a construção do solucionador, comparativamente ao solucionador zChaff, as seguintes alterações foram realizadas:

- A estrutura de dados foi alterada de forma a incluir um vetor de ponteiros para as cláusulas. Essa ação inclui nd ponteiros na estrutura de dados, onde n é o número de variáveis e d é o número médio de cláusulas por variável. A operação sobre esses ponteiros para selecionar a cláusula a associar à variável é a maior sobretaxa imposta ao algoritmo. Os resultados, entretanto, mostram que esse valor agregado é compensado pelo desempenho final.

- Durante a análise de conflitos, foi inserido um algoritmo para identificar e distin-

guir a existência de um retrocesso cronológico e um não cronológico. O algoritmo e o motivo da sua inserção serão descritos mais adiante ainda neste capítulo. Na verdade, o custo adicional dessa ação é pequeno. A principal função do algoritmo é verificar se o primeiro UIP encontrado corresponde à variável de decisão do nível ou não. O que pode ser rapidamente verificado.

- Durante a etapa de limpeza de cláusulas aprendidas, as cláusulas que foram associadas não foram retiradas do conjunto de cláusulas originais evitando-se a sobretaxa de desmarcá-las. Esta ação aumenta o tamanho de memória necessário, mas desde que as cláusulas são associadas apenas nos retrocessos cronológicos, esse número de cláusulas não retiradas é baixo. Fato que pode ser comprovado nos resultados.

- A associação da cláusula com a variável deve ser desfeita se um retrocesso acontece para um nível anterior ao nível da variável de decisão corrente. Esta ação inclui uma sobretaxa pequena, já que é realizada de forma simultânea à limpeza das atribuições às variáveis.

A Figura 5.13 ilustra dois grafos de implicações em um determinado nível conforme descrito no Capítulo 2. A notação utilizada é a mesma descrita anteriormente. Ambos os grafos representam um determinado nível de decisão de dois problemas hipotéticos, onde a variável x_5 é a última variável de decisão e estamos no nível 5. Ambos chegaram a um conflito no nível 5, a variável de conflito é x_{11} para o grafo da esquerda e x_{10} para o da direita.

A Figura 5.13 (esquerda) ilustra a análise do retrocesso considerando-se o primeiro ponto único de implicação (1UIP), que tem provado ser superior a outras soluções para a análise de conflitos [Sil98] [Bea04] [Zha01]. O 1UIP é normalmente o UIP mais próximo do lado do conflito, x_8 na Figura 5.13 (esquerda, representado pelo corte primeiro UIP). O mecanismo de análise de conflito do solucionador zChaff provoca um retrocesso considerando a variável do 1UIP que, eventualmente, pode não ser a variável

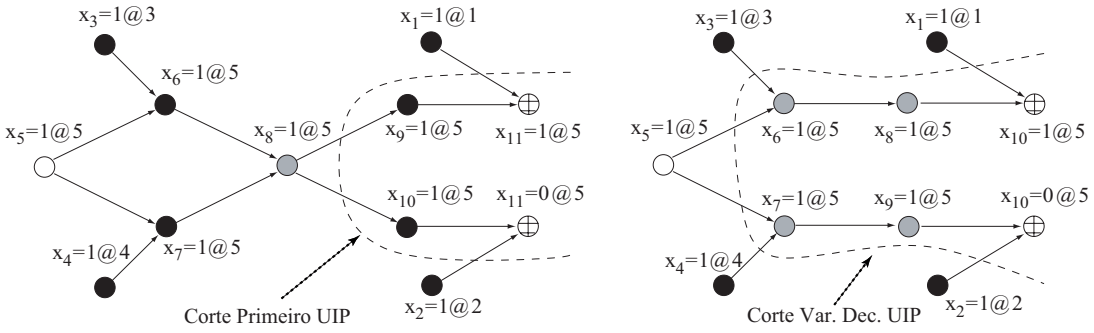

Figura 5.13: Grafo de implicação: 1UIP (esquerda) e var. de decisão (direita)

de decisão. Estes UIPs foram descritos por Silva em [Sil96] como "falsas decisões". Na Figura 5.13 (direita) o UIP é a própria variável de decisão x_5 e é este o momento em que um retrocesso cronológico é executado, assim como no algoritmo DPLL básico. Para a implementação mais eficiente do algoritmo DSharp uma distinção entre estas duas situações deve ser inserida no solucionador.

Quando uma cláusula é selecionada pelo algoritmo DSharp, devemos selecionar uma variável após a outra dentro desta cláusula até rapidamente atingirmos conflitos ou não existirem mais variáveis a ser selecionadas. É claro que: o BCP é executado após as atribuições, a análise de conflitos é realizada e retrocessos não cronológicos são também executados se assim for necessário. Entretanto, existe uma diferença fundamental, entre um retrocesso cronológico, que é o momento de aplicação da decisão Sharp, e um retrocesso não cronológico. Quando retrocedemos de forma cronológica estamos sobre uma decisão real do algoritmo, o que não acontece no retrocesso não cronológico, que não irá tentar uma nova fase de uma variável de decisão mas sim desfazer todas as decisões até um determinado nível e implicar o primeiro UIP.

O algoritmo para executar esta tarefa deve examinar o grafo de implicações. Se o UIP identificado pelo mecanismo de análise de conflitos não possuir antecedentes então ele é uma variável de decisão. Neste caso a variável de decisão deverá ser marcada e utilizada para a associação com a cláusula como anteriormente descrito. Se o UIP encontrado possuir antecedentes então ele não é uma variável de decisão e o algoritmo DSharp não será aplicado.

A Figura 5.14 ilustra como a análise de conflitos utilizando o DSharp é realizada. Se, após percorrer o grafo de implicações, for detectado que o retrocesso será cronológico, a decisão DSharp será acionada. Caso isso não aconteça, a decisão DSharp não será acionada e o algoritmo irá se comportar da forma original.

```
DSharp analise de conflitos ( )
início
    nível de retrocesso = DPLL original analise de conflitos ();
    se ( backtrack (nível de retrocesso) = cronológico)
        DSharp = verdadeiro;
    retorne (nível de retrocesso);
fim
```

Figura 5.14: Função DSharp para analise de conflitos

A Figura 5.15 ilustra o processo de decisão utilizando o DSharp. Se, uma decisão DSharp foi acionada, o algoritmo verifica se a variável de decisão já está associada ou não a alguma cláusula e se essa cláusula ainda é uma cláusula não resolvida, se assim o for, seleciona uma variável dessa cláusula. Isso pode representar ainda um retorno a uma cláusula já associada, indicando que estamos procurando insistir naquela cláusula específica. Caso essa associação ainda não tenha sido realizada, o algoritmo irá procurar por uma cláusula não resolvida e, dentre as variáveis dessa cláusula selecionada, escolher uma. Considerando que as decisões DSharp só serão acionadas nos retrocessos cronológicos, se a decisão DSharp não estiver acionada, o algoritmo executa o processo de decisão original do algoritmo.

Duas questões importantes ainda foram resolvidas e estão presentes no processo de decisão ilustrado na Figura 5.15:

1. Qual a melhor cláusula a subtrair?

2. Uma vez escolhida a cláusula, qual a melhor variável dentro desta cláusula seria a mais indicada a ser escolhida para a próxima variável de decisão?

A Figura 5.16 ilustra a função *DSharp_seleciona_cláusula_de_decisão(v)* que seleciona a cláusula-cubo a ser associada com a variável v. A cláusula cubo é selecionada

```
booleano DSharp decide próximo desvio( )
início
    se (DSharp = verdadeiro)
        se (variável de decisão já associada com uma cláusula w )
            se (w é uma cláusula ≠ satisfazível e com variáveis livres)
                x = DSharp seleciona variável de decisão (w);
                insira x na lista de implicações;
                retorne (verdadeiro);
        senão
            w = DSharp seleciona cláusula de decisão ( variável de decisão );
            x = DSharp seleciona variável de decisão  (w);
            insira x na lista de implicações;
            retorne (verdadeiro);
    senão
        DPLL Original decide próximo desvio ( );
fim
```

Figura 5.15: Função DSharp que decide próximo desvio

a partir das cláusulas de v ainda não resolvidas. Escolhe-se aquela que possui o maior escore de atividade dando preferências ás cláusulas aprendidas. A atividade de uma cláusula é implementada pela simples modificação da heurística VSIDS descrita no Capítulo 3. O VSIDS incrementa um contador de uma constante todas as vezes que uma cláusula é envolvida em um conflito e o objetivo é utilizar este contador durante a eliminação de cláusulas aprendidas, entretanto, ele realmente reflete a atividade da cláusula. Na implementação DSharp, esse contador é incrementado de duas vezes esta mesma constante quando esta cláusula for associada a alguma variável e é também utilizado para a escolha das cláusulas. Observa-se que no solucionador Berkmin, também existe uma pilha de cláusulas aprendidas, outro conceito que foi utilizado na determinação da cláusula a ser associada. Essas duas características, trabalhando juntas, fazem com que o algoritmo seja bem dinâmico.

```
cláusula DSharp_seleciona cláusula de decisão (variável  v)
início
    w = Cláusula de máxima atividade (cláusulas onde v aparece)
    associa  v com w;
    retorne (w);
fim
```

Figura 5.16: Função DSharp que seleciona a cláusula de decisão

A função *DSharp seleciona variável de decisão ( )* ilustrada na Figura 5.17, seleciona

um variável livre da cláusula w, que se torna associada com esta cláusula, e atribui um valor que satisfaça a cláusula. Para selecionar uma variável livre dentro de w usamos o mesmo escore de variáveis encontrados pela heurística VSIDS, já que a mesma possui uma eficiência já comprovada [Mos01].

```
variável DSharp seleciona variável de decisão (cláusula  w)
início
    v = literal de melhor escore VSIDS que satisfaz a cláusula (w)
    associa v com  w;
    retorne ( v );
fim
```

Figura 5.17: Função DSharp que seleciona a variável de decisão

Durante a retirada das cláusulas, como não retiramos as cláusulas que estavam associadas, obtém-se um pequeno aumento no banco de dados original de cláusulas. Contudo, tendo em vista a limitação da utilização da estratégia DSharp apenas nos retrocessos cronológicos, mantivemos o número de cláusulas marcadas baixo. Os experimentos demonstraram que, mesmo em problemas complexos, essa quantidade é realmente baixa, não prejudicando o desempenho e o consumo de memória.

Capítulo 6

Resultados

Esse capítulo apresenta os resultados das decisões DSharp incorporadas a um solucionador DPLL no estado da arte. A inserção das decisões em um solucionador conhecido e de domínio público permite:

- Avaliar o desempenho considerando as decisões DSharp sobrepostas às decisões do algoritmo, comparando para ambos, o tempo de execução;
- Avaliar o comprometimento das alterações nas estruturas de dados desse novo solucionador, através do desempenho final do algoritmo. Como visto no Capítulo 3, além do mecanismo de decisão, as estruturas de dados podem comprometer o desempenho do BCP, principal responsável pelo tempo necessário para a solução da instância.
- Permitir a reprodução do experimento, já que o solucionador é de domínio público.

Para a avaliação, um conjunto de instâncias resultantes de problemas reais foi selecionado e submetido aos dois algoritmos. Grande parte dessas instâncias é a mesma utilizada nas competições mundiais de SAT, porém, nas competições, um número menor de instâncias é utilizado devido ao tempo necessário para a solução. Esse conjunto completo, composto de mais de 1000 instâncias, representou um tempo contínuo de

simulação superior a quatro meses já que, o tempo limite imposto para cada instância, variou entre 10000 e 20000 segundos e, muitas instâncias, não foram resolvidas dentro desse tempo limite. Dessa forma, grande parte dos resultados está apresentada por grupo de instâncias e não individualmente por instâncias.

Resultados

Todos os experimentos foram realizados em um Pentium 4, com a freqüência de clock de 2.0 GHz, com 2Gbytes de memória RAM e sistema operacional Linux. A base do solucionador, que suportou todas as alterações no DPLL necessárias para o DSharp, foi o solucionador zChaff, obtido em [Url05] e identificado como *zChaff-2004.11.15*. Foram utilizados os ajustes padrões no solucionador.

Um total de 1232 instâncias de problemas de uma vasta gama de aplicações foram testadas e os resultados podem ser vistos nas Tabelas 6.1, 6.2 e 6.3. Esses resultados comparam a versão original do zChaff-2004, com o solucionador DSharp construído sobre o mesmo programa, em termos de desempenho (tempo de CPU), número de conflitos e robustez (número de instâncias que não foram resolvidas dentro do tempo limite fornecido). Em cada tabela, as colunas representam as seguintes informações:

- A identificação do conjunto (coluna *conjunto de benchmarks*);
- O número de instâncias do benchmark (coluna *Instâncias*);
- O tempo de CPU para o solucionador zChaff e para o DSharp em segundos (coluna *Tempo*);
- O número de conflitos encontrados (coluna *Conflitos*);
- O número de instâncias abortadas (coluna *TO*) após um tempo limite de 10000 segundos (para as Tabelas 6.1 e 6.3) e 20000 segundos (Tabela 6.2);
- O ganho em tempo proporcionado pela inserção das decisões DSharp (coluna *Speedup*).

Os resultados onde o tempo de CPU ou o número de instâncias não concluídas foi inferior para o solucionador DSharp, estão impressos em negrito. Não estão computados os tempos nos quais o solucionador DSharp ou zChaff não chegaram a um resultado.

Os exemplos ilustrados nas tabelas 6.1 e 6.2 fazem parte de um vasto conjunto de *benchmarks* de *bounded model checking* (BMC) da IBM. Esse conjunto foi escolhido porque todas as instâncias são derivadas de problemas reais de verificação formal de hardware (séries 2004 e 2002). Parte desse conjunto sempre é aceita e utilizada nas competições mundiais de SAT e, o conjunto completo, disponibilizado por E. Zarpas na IBM, pode ser obtido em [Url16] para a reprodução dos experimentos. Para uma discussão mais completa a respeito de geração de *benchmarks* IBM CNF ver [Zar05]. A tabela não contempla alguns conjuntos do pacote completo, o motivo é que para o tempo limite adotado, a quantidade de instâncias não concluídas foi grande para ambos os solucionadores e, a sua apresentação, não traz benefício à análise, apenas acrescenta mais linhas às tabelas. Cada linha das tabelas representa uma família de instâncias.

Conjunto de *benchmarks*		Instâncias	zChaff			Dsharp			Speedup
			Tempo	Conflitos	TO	Tempo	Conflitos	TO	
IBM-FV2004 (k10 até k60)	1	11	10204	1326946	0	**5633**	1050612	0	1.81
	1_02_1	11	249	147735	0	**139**	108583	0	1.79
	1_02_3	11	12.8	7599	0	**0.7**	99	0	18.29
	1_11	11	12625	245158	3	13961	2381039	3	0.9
	1_13	11	47.4	16215	0	**33.6**	10220	0	1.41
	1_14	11	173.2	80794	0	550.1	147353	0	0.31
	1_16_2	11	0.3	35	0	0.3	35	0	1
	1_17_1	11	0.4	0	0	0.4	0	0	1
	1_17_2	11	3.2	511	0	3.3	531	0	0.97
	1_31_1	11	80295	62762	8	80296	62922	8	1
	02_2	11	114.1	65520	0	115.4	69917	0	0.99
	2_14	11	15117	943648	1	**9615**	901178	**0**	1.57
	3	11	2449	306085	0	**2130**	311329	0	1.15
	5	11	2286	257804	0	**2151**	250311	0	1.06
	6	11	9109	535263	0	**3693**	365308	0	2.47
	7	11	209	252280	0	**93**	151749	0	2.25
	9	11	22.2	8230	0	**3.5**	1288	0	6.34
	10	11	10266	693270	0	**2229**	299381	0	4.61
	18	11	10089	2672182	5	**5025**	2464754	**4**	2
	19	11	14802	938943	1	**8467**	828304	0	1.75
	21	11	11127	809517	0	**5576**	722795	0	2
	22	11	17465	1235837	0	**12580**	1007500	0	1.39
	23	11	24393	1810057	3	**21650**	1609201	3	1.12
	26	11	6820	476507	4	**3169**	333762	**1**	2.15
	27	11	1041	263879	0	**927**	266899	0	1.12
	28	11	1328	348921	0	**880**	286381	0	1.51
	29	11	3095	431597	8	**1359**	235112	8	2.27
Total		**297**	**232969**	**13937295**	**33**	**179573**	**13866563**	**27**	**1,30**

Tabela 6.1: Resultados na série 2004 de IBM-CNF BMC.

Conjunto de *benchmarks*		Instâncias	zChaff			Dsharp			Speedup
			Tempo	Conflitos	TO	Tempo	Conflitos	TO	
IBM-FV2002 (k1 até k85)	02_1_rule_1	17	1998	606394	0	**1825**	580045	0	1.09
	02_1_rule_2	17	1401	480230	0	**1286**	498079	0	1.09
	02_1_rule_4	17	128	75848	0	**68**	44923	0	1.88
	02_1_rule_5	17	24.8	18891	0	40.9	30034	0	0.61
	02_3_rule_1	17	703	210528	0	**263**	73472	0	2.67
	02_3_rule_2	17	80.7	61469	0	117.6	68114	0	0.69
	02_3_rule_3	17	406	92693	0	**324**	83474	0	1.25
	02_3_rule_4	17	246	125970	0	**38**	30649	0	6.47
	02_3_rule_5	17	226	65725	0	**217**	65966	0	1.04
	02_3_rule_6	17	306	157382	0	**66**	48254	0	4.64
	03_rule	17	35798	2004650	1	**23539**	1453313	1	1.52
	05_rule	17	10998	988704	0	12205	1077329	0	0.9
	06_rule	17	47497	1870271	4	**44293**	1692961	**3**	1.07
	07_rule	17	1022	601332	0	**154**	285500	0	6.64
	09_rule	17	14.2	3426	0	**5.7**	1807	0	2.49
	15_rule	17	5.4	1948	0	**4.9**	1879	0	1.1
	16_2_1_rule	17	0.7	32	0	**0.6**	32	0	1.17
	17_1_1_rule	17	1.1	0	0	**0.9**	0	0	1.22
	18_rule	17	9673	520633	11	**4278**	306079	**10**	2.26
	19_rule	17	21265	1448723	4	**19445**	1359703	**3**	1.09
	21_rule	17	4032	386775	7	**2441**	321991	**3**	1.65
	27_rule	17	3843	772114	0	**3485**	795780	0	1.1
	28_rule	17	15929	1402626	0	**14085**	1403934	0	1.13
	29_rule	17	13399	9239090	12	**7757**	8524607	12	1.72
Total		**408**	**168870**	**21135454**	**39**	**135769**	**18747925**	**32**	**1,24**

Tabela 6.2: Resultados na série 2002 de IBM-CNF BMC.

As instâncias da Tabela 6.3 foram escolhidas porque também são instâncias industriais. O conjunto Maris2004, obtido da competição de SAT em 2005 [Url03], é derivado de problemas reais de planejamento. As outras instâncias têm origem no teste lógico e na verificação formal de microprocessadores superescalares e VLIW (*Very Large Instruction Word*). Alguns desses *benchmarks* também foram utilizados nas competições de SAT. A série completa, assim como as respectivas descrições, são disponibilizadas por Miroslav Velev e podem ser obtidas em [Url17].

As instâncias da Tabela 6.4 consistem de uma mistura de DIMACS, planejamento e codificação de problemas de *bounded model checking*. São problemas geralmente mais simples, aleatórios e sintéticos. Foram escolhidos, já que são aceitos na comunidade SAT e utilizados em grande parte de publicações, o que nos permite comparar os resultados com outras pesquisas na área. A série completa, assim como a descrição de cada conjunto, pode ser obtida em [Url18].

De acordo com os resultados das tabelas 6.1, 6.2 e 6.3, observa-se que o solucionador DSharp é significativamente mais veloz e mais robusto, considerando robustez, ao fato de se chegar a uma resposta de uma instância dentro do tempo limite fornecido. Na

Conjunto de *benchmarks*	Instâncias	zChaff			Dsharp			
		Tempo	Conflitos	TO	Tempo	Conflitos	TO	Speedup
sss.1.0_cnf	48	8.9	3452	0	**4.4**	1756	0	2.02
sss.1.0a_cnf	9	4.5	5642	0	**4.0**	5263	0	1.13
sss-sat-1.0	100	121	70290	0	**112**	62541	0	1.08
vliw-sat-1.0	100	2033	526189	0	**1807**	486514	0	1.13
fvp-unsat_1.0	4	102.3	56785	0	**97.1**	57603	0	1.05
fvp_unsat_2.0	22	1593	480432	0	1623	492626	0	0.98
liveness_sat_1.0	10	130180	459881	6	**116854**	**321523**	**5**	1.11
pipe_sat_1.1	10	47603	479064	1	**44293**	479699	1	1.07
vliw_unsat-3.0	2	27854	1820390	0	**19438**	1362866	0	1.43
fvp_sat.3.0	20	21904	2173245	0	**17899**	1996919	0	1.22
livness_unsat_2.0	9	298.6	140475	6	**251.1**	117558	6	1.19
Maris2004	67	49.4	115503	0	**22.2**	70724	0	2.23
Total	**401**	**231288**	**6331348**	**13**	**202026**	**5455592**	**12**	**1.14**

Tabela 6.3: Resultados em benchmarks industriais

Tabela 6.4, os resultados de ambos os solucionadores são similares, entretanto, se observarmos apenas as famílias com características industriais Ssa e Bmc, o solucionador DSharp se mostrou superior novamente.

Conjunto de *benchmarks*	Instâncias	zChaff			Dsharp			
		Tempo	Conflitos	TO	Tempo	Conflitos	TO	Speedup
Aim	72	0.15	2833	0	**0.09**	2806	0	1.67
Blocksworld	7	2.14	2390	0	3.38	3405	0	0.63
Parity16	10	7.8	41758	0	9.8	48867	0	0.80
Pigeon-Hole	5	15.6	37626	0	15.7	38404	0	0.99
Hanoi	2	159	66859	0	274	81379	0	0.58
Dubois	13	0.03	3013	0	0.03	2430	0	1.00
Bridge-fault	4	0.06	514	0	0.06	399	0	1.00
Ssa	4	0.17	340	0	**0.15**	326	0	1.13
Beijing	16	506	181795	0	823	277745	0	0.61
Bmc	13	135.5	54322	0	**106.2**	47655	0	1.28
Total	**146**	**665**	**391450**	**0**	**1097**	**503416**	**0**	

Tabela 6.4: Resultados no DIMACS

Observa-se a superioridade do solucionador DSharp apesar do aumento da estrutura de dados. Convém lembrar que, o algoritmo requer que cada variável possua um vetor de ponteiros para as cláusulas onde elas se encontram o qual é utilizado para visitar as cláusulas durante o retrocesso cronológico. Considerando que os retrocessos cronológicos são raros, esse aumento da estrutura de dados com a inserção desse vetor, foi compensado pelo benefício obtido.

As tabelas 6.5 e 6.6 estão relacionadas. A Tabela 6.5 mostra os resultados em instâncias individuais extraídas dos conjuntos das Tabelas 6.1, 6.2 e 6.3. Ela mostra os

tamanhos típicos dos problemas em termos de número de cláusulas e variáveis e que foram submetidos aos solucionadores, no caso, problemas considerados grandes. A última coluna da Tabela 6.4 mostra se o problema é SAT ou não-SAT. Observa-se que, mesmo em grandes problemas, podemos obter melhorias superiores a 2 vezes. Os ganhos de velocidade apresentados, coluna *Speedup*, foram calculados apenas para os solucionadores Dsharp e zChaff e o tempo limite para as instâncias foi fixado em 20000 segundos. As mesmas instâncias foram submetidas ao solucionador Minisat2, versão 061208 disponibilizada em [Een03], esse solucionador foi o vencedor da competição mundial de SAT em 2005. Observa-se que na média, o solucionador Minisat2 é mais veloz que os dois solucionadores e, como um possível trabalho futuro, será interessante verificar se, o aumento da estrutura de dados imposto pelas decisões DSharp, ainda pode ser compensado pelo ganho de velocidade, considerando as estruturas extremamente leves utilizadas pelo solucionador Minisat2. Também observamos que quatro instâncias não conseguiram ser finalizadas para o solucionador Minisat2 devido à exaustão da memória. Isso pode indicar um compromisso entre a robustez e velocidade, entretanto, até o momento da redação desse texto, não me foi possível garantir essa conjectura.

A Tabela 6.6 ilustra mais alguns resultados para as mesmas instâncias, as colunas representam: o número de decisões, o número de cláusulas adicionadas resultantes do aprendizado de cláusulas, o número de cláusulas de conflito retiradas e, duas estatísticas exclusivas do DSharp denominadas "DS-1" e "DS-2". A coluna *DS-1* indica o número de seleções de cláusulas, isto é, o número de vezes que o algoritmo selecionou uma cláusula e retornou a ela para a escolha de uma variável pela segunda vez, após o retrocesso cronológico. A coluna *DS-2* indica a soma do número de retornos superiores ao segundo em alguma cláusula, isto é, se após um novo retrocesso cronológico, o algoritmo retornou até uma determinada cláusula pela terceira, quarta ou mais vezes.

O número de decisões é geralmente inferior para o D-Sharp, o que é consistente com a redução do tempo de CPU. Este efeito também significa que o algoritmo proporciona uma quantidade menor de conflitos e, como conseqüência, um número menor de

cláusulas aprendidas. Como mencionado anteriormente, o número de cláusulas que o solucionador DSharp retira é inferior porque não é permitida a retirada das cláusulas aprendidas que estão associadas às variáveis e, que são poucas, comparadas ao número total de cláusulas aprendidas adicionadas.

Os valores de DS-1 e DS-2, que nos dizem o número de vezes que o algoritmo DSharp é aplicado, isto é, o número de vezes que o processo de decisão DPLL é modificado, conforme se observa, é baixo. Também podemos observar que DS-1, o número de retornos após o retrocesso cronológico é superior a DS-2, o número de retornos a retrocessos posteriores. Esse fato não é surpreendente. Se a cláusula selecionada é uma cláusula com apenas dois literais, apenas um retrocesso, o indicado por DS-1, poderá acontecer. O segundo, e último, subespaço disjunto é obtido através do BCP já que, ao ser invertida a primeira variável para um valor que não satisfaz a cláusula, esta se torna unitária e o valor da segunda variável, que irá tornar a cláusula SAT, é automaticamente imposto pelo BCP. De maneira análoga, uma cláusula com três literais permite no máximo dois retrocessos. Entretanto, desde que damos preferência às cláusulas aprendidas, o algoritmo terá uma tendência a escolher clausulas maiores. Se múltiplos retrocessos não são freqüentes nessas cláusulas longas, isso significa que a subtração de cláusulas induz o BCP a expor regiões onde não encontramos uma solução, sem a necessidade de atribuir valores a muitas variáveis. Atribuir valores às poucas variáveis iniciais de uma mesma cláusula é suficiente para se chegar a uma solução rapidamente, ou provar que essa atribuição não pode ser encontrada, considerando o conjunto de atribuições realizadas pelo algoritmo até esse momento.

Os gráficos ilustrados nas Figuras 6.1 a 6.6 correspondem aos valores individuais das instâncias informadas nas Tabelas 6.1, 6.2 e 6.3. Nesses gráficos, o eixo x corresponde ao tempo de solução do algoritmo DSharp e o eixo y corresponde ao tempo de solução do solucionador zChaff. Os gráficos estão divididos e identificados de acordo com o tempo necessário para a solução de cada uma das instâncias. Foram retiradas todas as instâncias onde não foi possível obter a resposta em qualquer um dos solucionadores.

	Instâncias individuais	Variáveis	Cláusulas	zChaff (seg.)	Dsharp (seg.)	Speed up	Minisat (seg.)	
1	IBM_FV_2004_rule_bath_18_sat dat.k20	34845	143320	71.8	40.4	1.78	13.8	unsat
2	IBM_FV_2004_rule_bath_18_sat dat.k30	52705	217310	2913	1370	2.13	132	sat
3	IBM_FV_2004_rule-bath_18_sat dat.k35	61635	254305	7359	3268	2.25	434	sat
4	IBM_FV_2004_rule-bath_20_sat dat.k35	63136	261850	3717	2443	1.52	225	sat
5	IBM_FV_2004_rule-bath_1_11_sat datk40	128768	511910	4813	2735	1.76	77	sat
6	fvp_unsat2.0-7pipe	23910	751118	612	523	1.17	TO	unsat
7	9dlx_vliw_at_b_iq8_I3_C24	132413	1435600	13742	10251	1.34	TO	unsat
8	9dlx_vliw_at_b_iq8_I3_C24_D	132156	1434887	14112	9186	1.54	TO	unsat
9	pipe_64_4_bug02	35853	1012271	11448	6672	1.72	TO	sat
10	9vliw_bp_mc_bug4	20064	287530	16.2	32.5	-	8.1	sat
11	9vliw_bp_mc_bug20	20079	287978	31.2	39.2	-	10.5	sat
12	fvp_unsat2.0-5pipe	9471	195452	32.8	27.1	1.18	3806	unsat
13	fvp_unsat2.0-4pipe	5237	80213	16	23.3	-	2686	unsat
14	fvp_unsat2.0-4pipe2-000	4941	82207	35.7	25.4	1.4	113.2	unsat

Tabela 6.5: Detalhes em instâncias individuais (tempo de CPU)

	Decisões			Conflitos			Cláusulas retiradas			DS-1	DS-2
	zChaff	DSharp	Minisat2	zChaff	DSharp	Minisat2	zChaff	DSharp	Minisat2		
1	101939	73724	32436	24335	16375	13232	3744	1628	5028	8281	101
2	2009655	1609690	146436	172250	126416	76526	97395	44413	34436	81299	512
3	5369098	2652644	273856	357315	227752	157573	220460	94710	78786	151176	979
4	3010857	1690723	224001	246473	181399	109850	161846	84384	43940	120104	905
5	2645328	739353	164423	212569	236718	47307	120405	105093	13719	151502	1155
6	2151805	1693212	-	81540	71773	-	56180	37260	-	33035	223
7	140170237	87122497	-	908388	707647	-	822915	543945	-	276545	1460
8	145488753	82811428	-	912002	655219	-	826441	500863	-	259016	1501
9	43584502	27394688	-	766523	489895	-	527521	280368	-	215469	2202
10	180319	272781	84265	7347	14158	6699	321	4556	2210	5975	55
11	318455	322899	136849	14262	17816	12344	4524	6766	3703	7532	83
12	242031	194197	4637304	13263	11366	2463505	6847	3386	862226	4240	23
13	99181	105103	4983260	14681	19479	2492596	5852	5550	747778	9724	128
14	132448	99347	406605	25607	17928	147342	7097	9657	58936	11700	190

Tabela 6.6: Detalhes em instâncias individuais (mais estatísticas)

Obviamente, para o solucionador mais robusto, esse tipo de análise é prejudicial, já que não estão inclusas essas instâncias, mas nos permite comparar as heurísticas aplicadas apenas nas instâncias onde ambos os solucionadores efetivamente conseguiram chegar a alguma resposta. Para uma melhor visualização, os gráficos estão divididos de acordo com o tempo necessário para se chegar a uma solução e, as instâncias, agrupadas de acordo com esse tempo. Não foi possível observar nenhuma tendência em uma faixa de tempo específica, aparentemente as decisões DSharp podem ser aplicadas independentemente do tempo necessário para a solução da instância. Os gráficos também ilustram os resultados de acordo com tipo da instância, SAT ou não-SAT e, novamente, não podemos observar nenhuma tendência conclusiva, indicando que se a instância é SAT ou não-SAT também não interfere nas decisões DSharp. Na verdade, podemos concluir

que, eventualmente, se o solucionador DPLL base adotado para a inserção das decisões DSharp possuir alguma tendência, ela será mantida, mas com um tempo de solução menor.

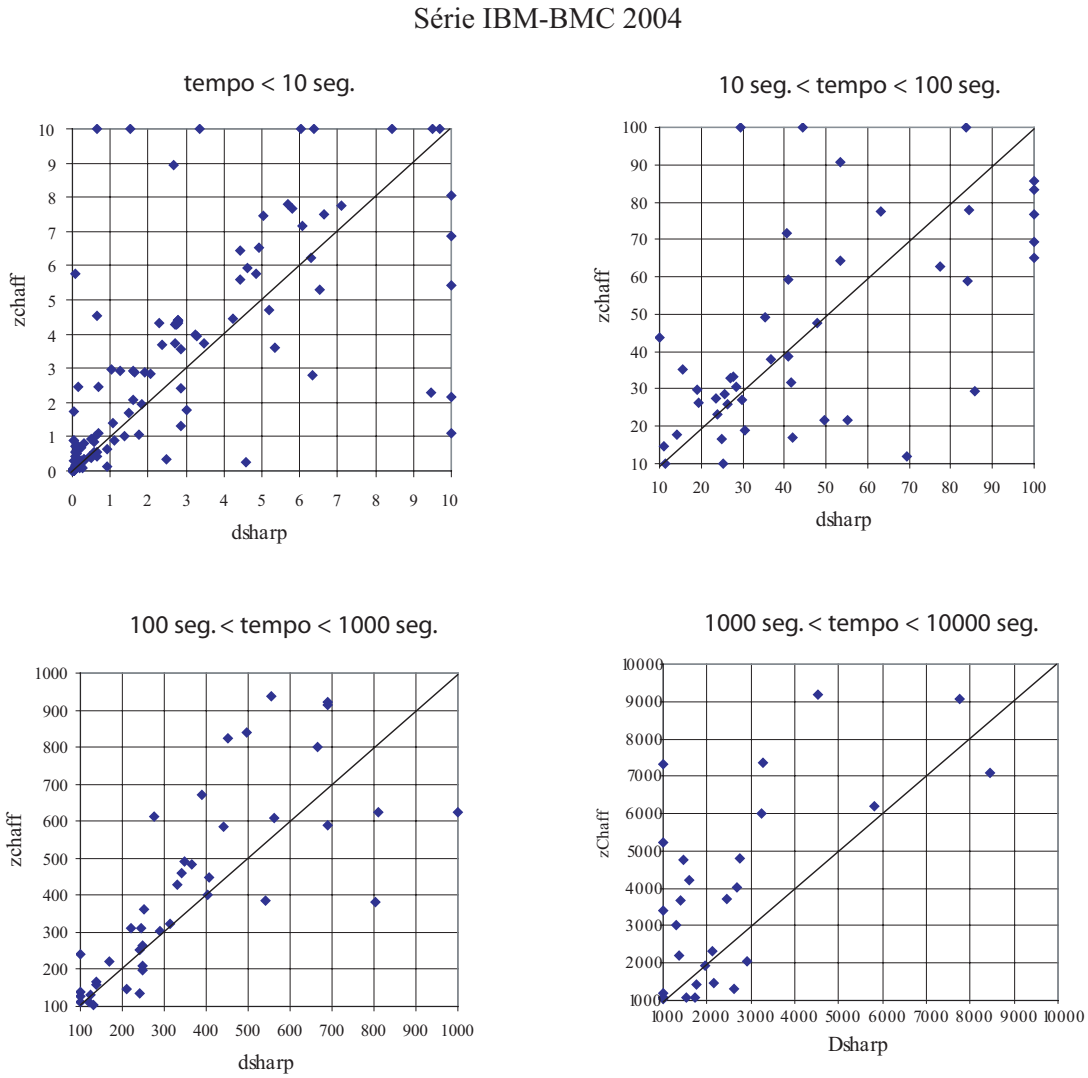

Figura 6.1: Gráfico comparativo com resultados da Tabela 6.1, IBM BMC série 2004

Os gráficos da Figura 6.8 ilustram os resultados das Tabelas 6.1, 6.2, 6.3 e 6.4. A linha pontilhada representa o solucionador DSharp e a linha contínua o solucionador zChaff. O eixo x corresponde às instâncias do conjunto avaliado e o eixo y representa o tempo necessário para solução de cada conjunto de instâncias. O que se observa é que, independentemente do tempo necessário para se chegar a uma solução, existe uma melhoria contínua. Se considerarmos os problemas mais difíceis como aqueles que

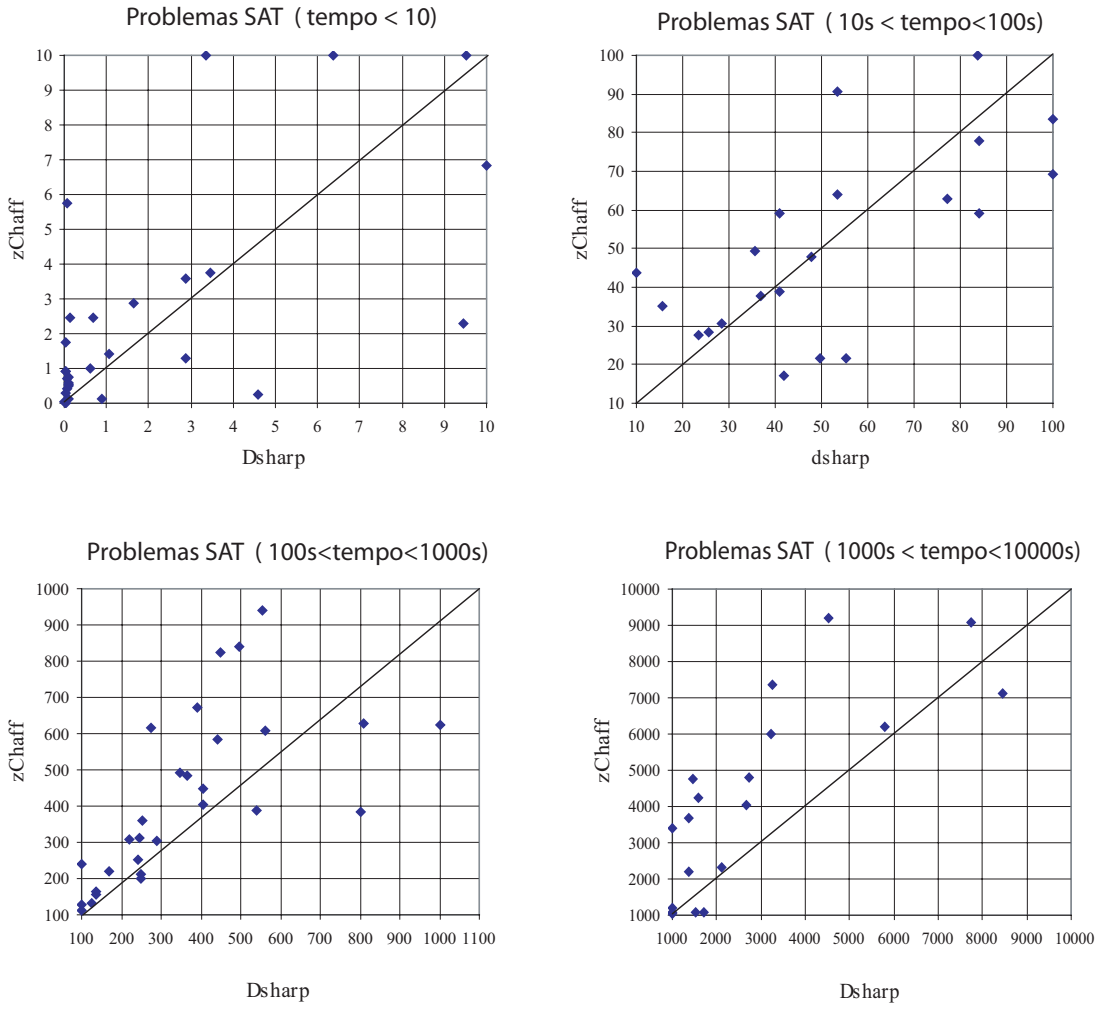

Figura 6.2: Apenas instâncias SAT de IBM BMC série 2004

necessitam de um tempo maior para serem resolvidos, observamos que, novamente, não existe uma distinção para a melhoria. O que se observa claramente é que, para as instâncias não industriais, oriundas do pacote de instâncias do DIMACS, não existe uma melhoria. Isso mostra que, assim como as estratégias modernas, o DSharp também se mostra indicado para os problemas com uma certa estrutura na instância, típico nos problemas industriais.

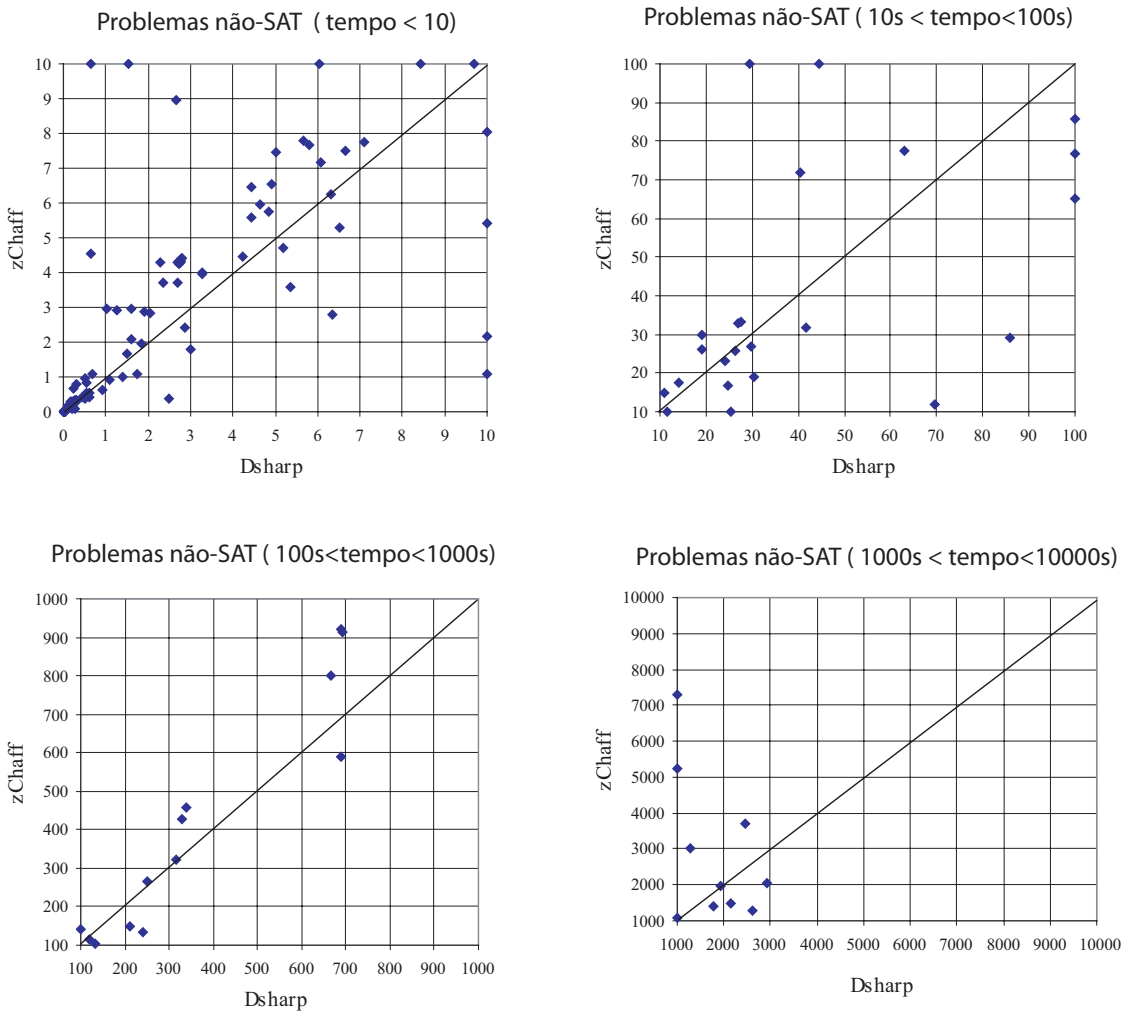

Figura 6.3: Apenas instâncias não-SAT de IBM BMC série 2004

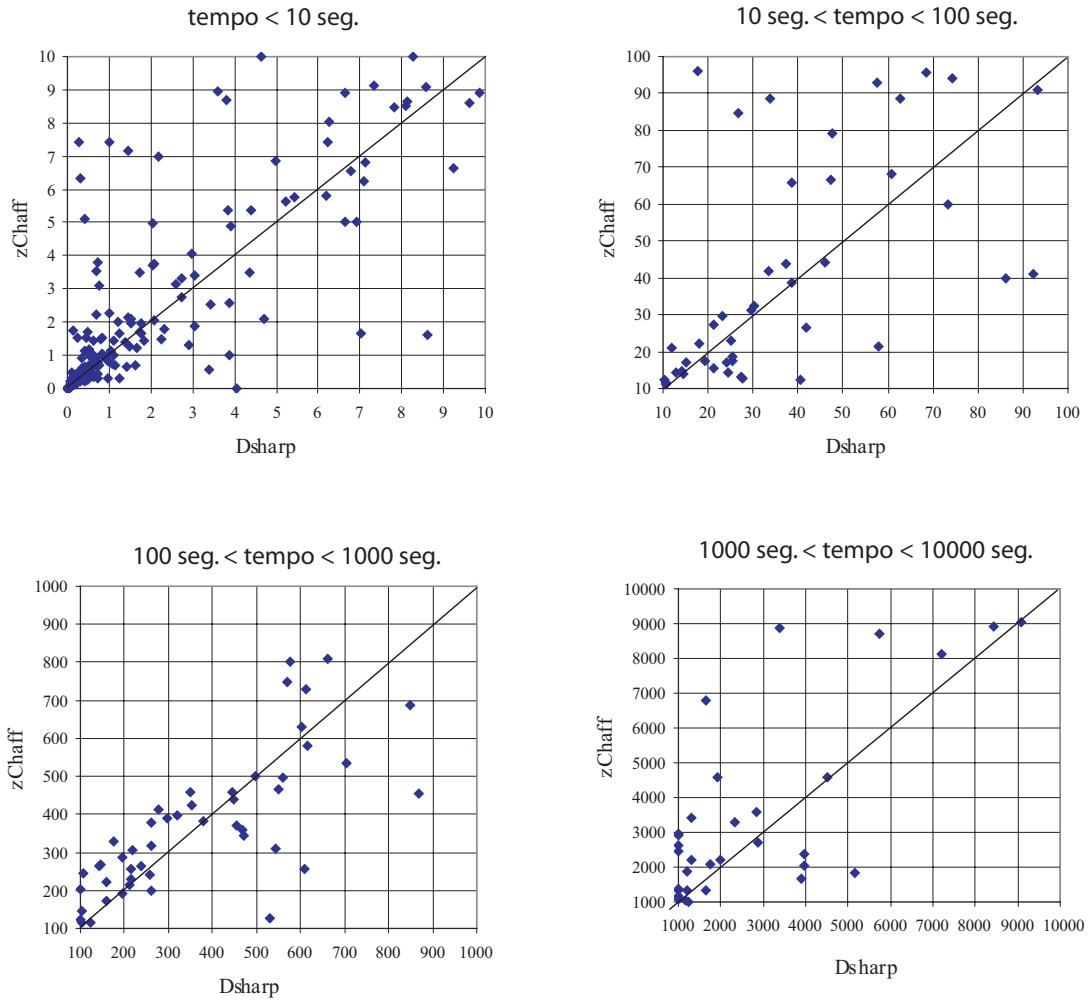

Figura 6.4: Gráfico comparativo com resultados da Tabela 6.2, IBM BMC série 2002

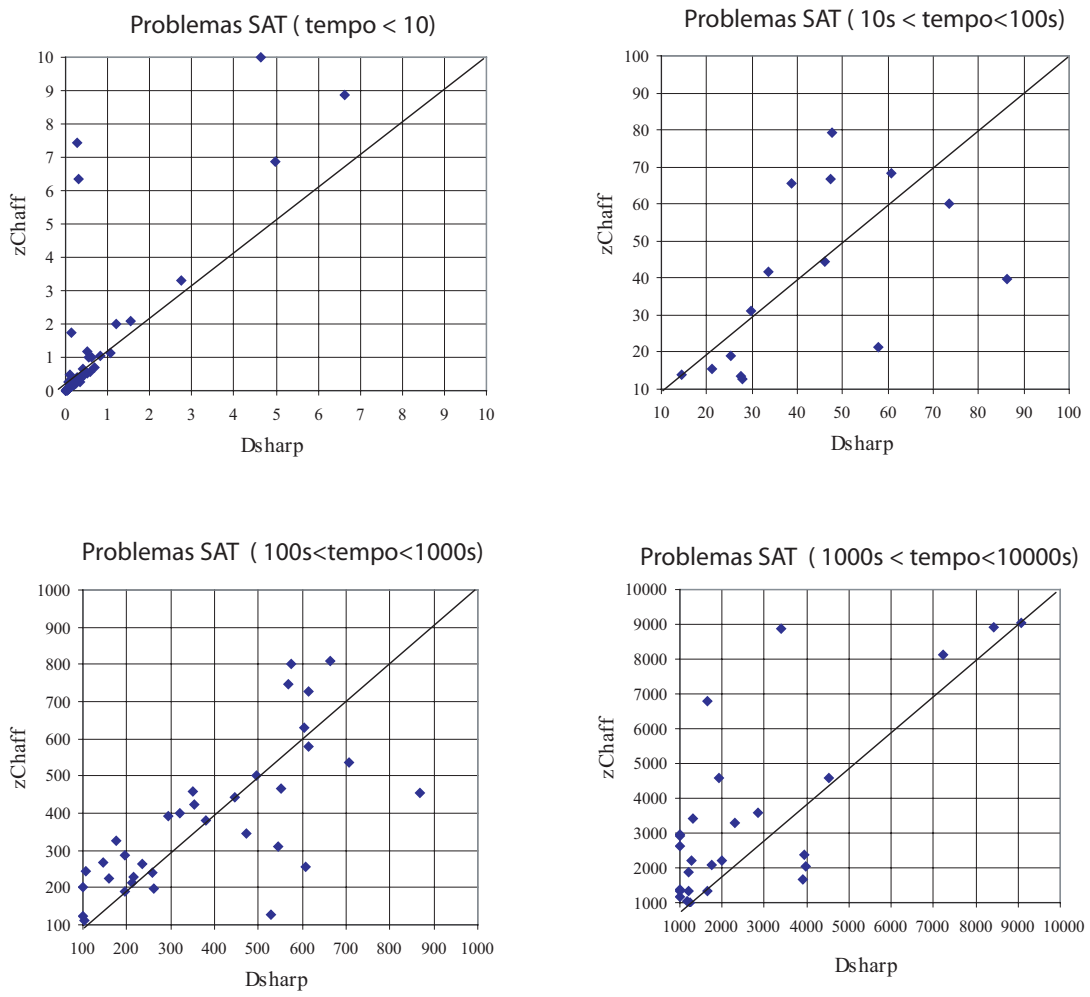

Figura 6.5: Apenas instâncias SAT de IBM BMC série 2002

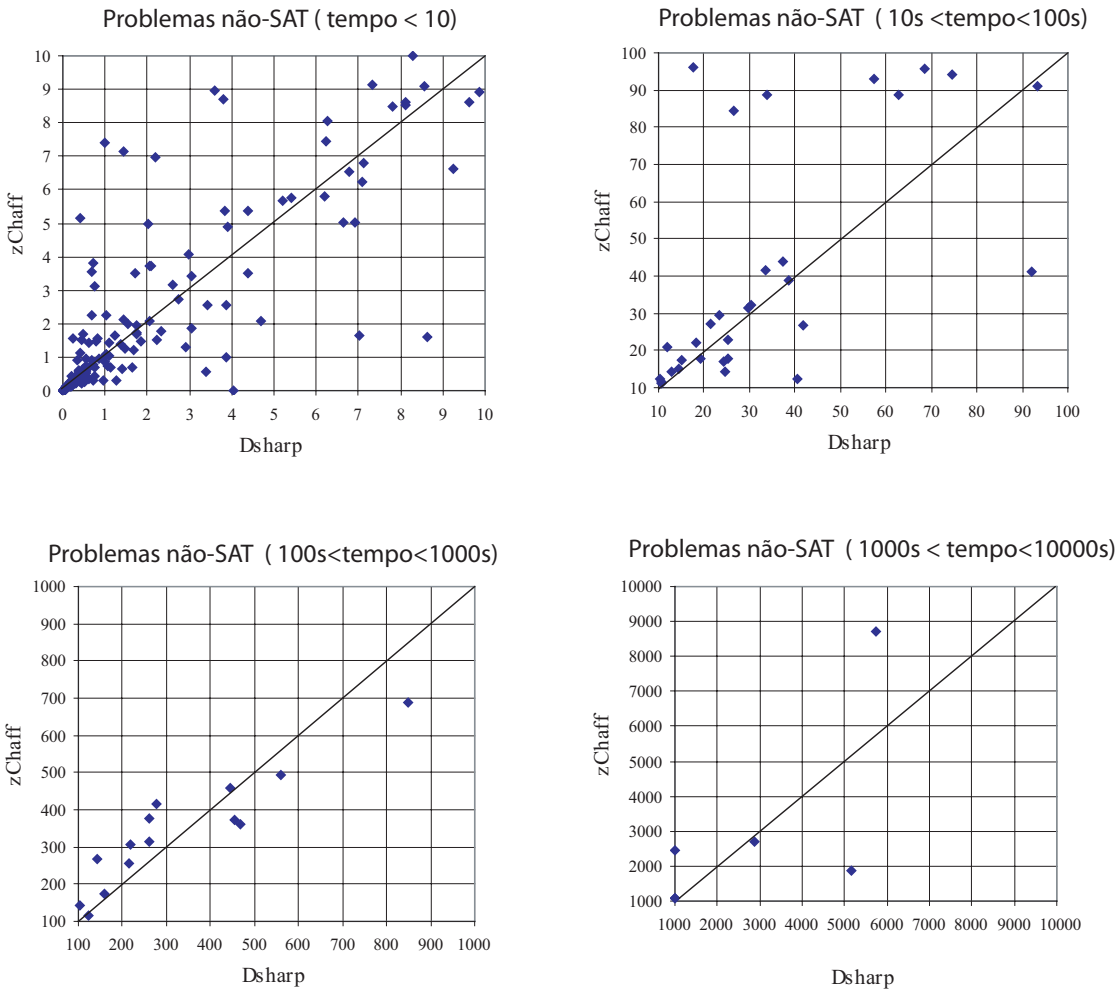

Figura 6.6: Apenas instâncias não-SAT de IBM BMC série 2002

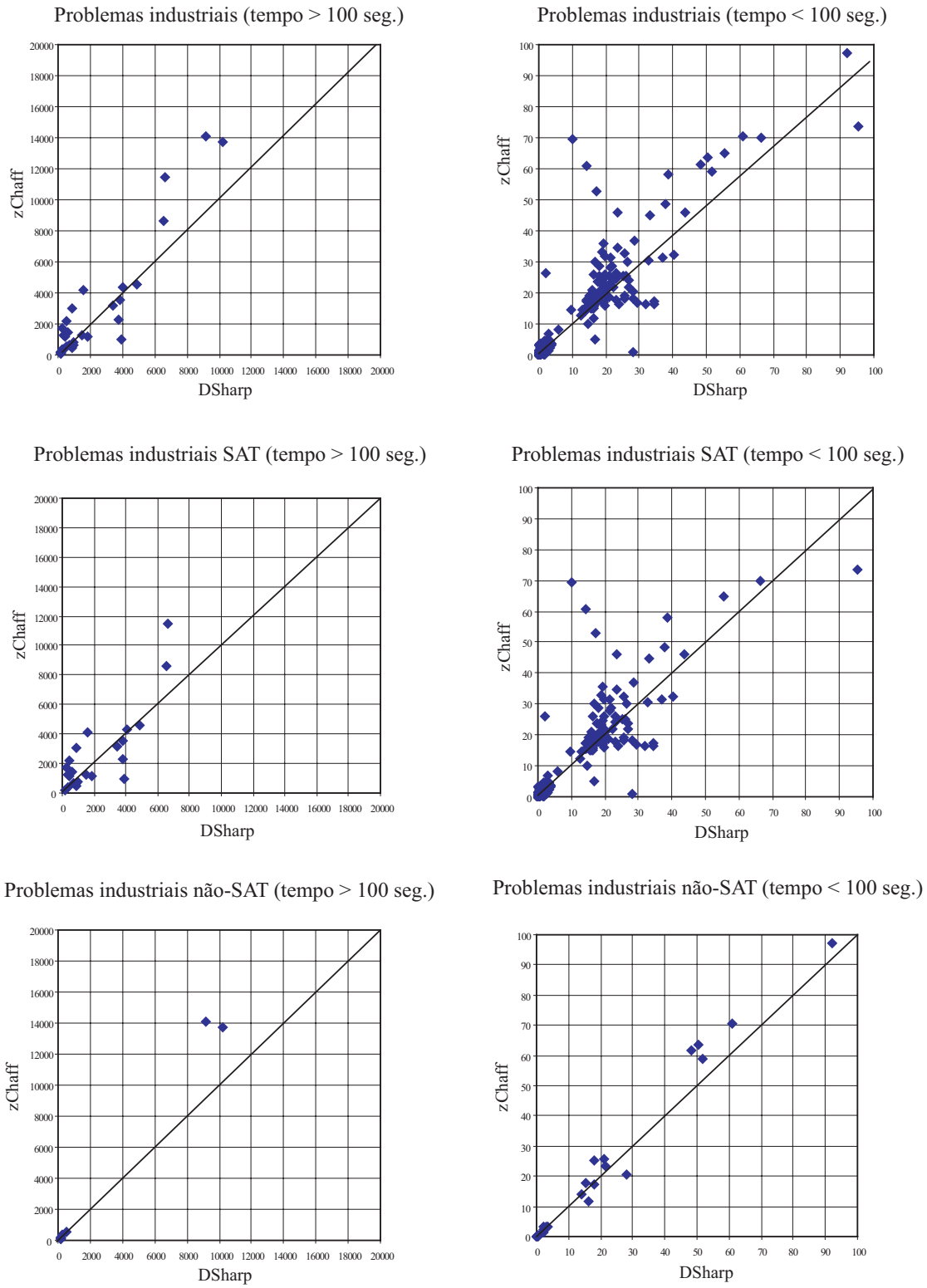

Figura 6.7: Gráfico comparativo com resultados da Tabela 6.3, instâncias industriais (Velev)

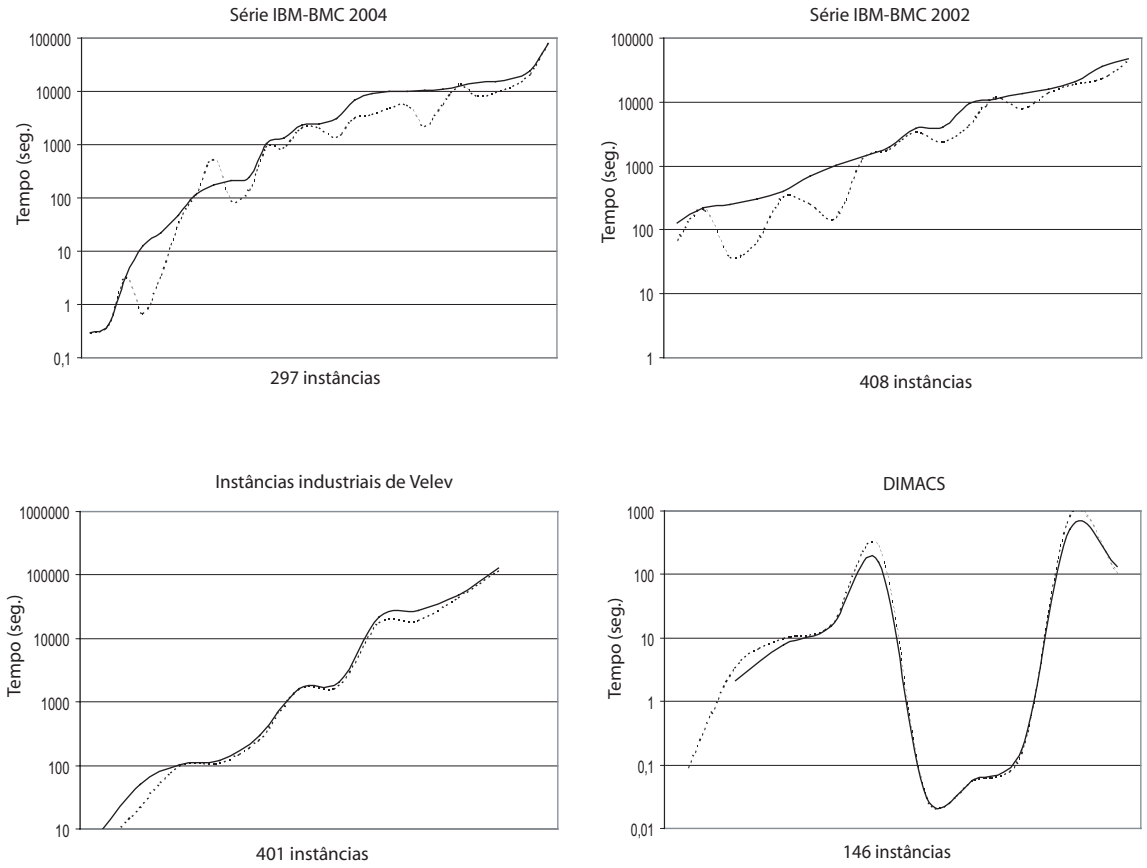

Figura 6.8: Gráfico comparativo com tempos de solução para cada solucionador

Conclusões e trabalhos futuros

Esse trabalho descreve um algoritmo para resolver instâncias do problema da satisfabilidade utilizando a subtração de cubos para escolher novas variáveis de decisão. As cláusulas negadas são vistas como cubos e são subtraídas, uma a uma, do espaço booleano n-dimensional denotado pelas atribuições de valores às variáveis da instância. O algoritmo finaliza com uma solução parcial, após subtrair todas as cláusulas-cubo, indicando que a instância é satisfazível, ou prova que o resultado é um cubo vazio e a instância é não satisfazível. O algoritmo Dsharp é utilizado para subtração, que produz um conjunto de cubos disjuntos como resultado da subtração de um cubo pelo outro.

Essa abordagem pode ser utilizada em solucionadores atuais do problema da satisfabilidade baseados no algoritmo DPLL, modificando-se a estratégia de decisão e utilizando a informação extraída do procedimento de análise de conflitos. Basicamente, o algoritmo influencia e força a escolha da nova variável de decisão após um passo de um retrocesso não cronológico.

O algoritmo de procura baseado na subtração de cubos foi implementado como um solucionador completo, em um solucionador SAT básico DPLL, em um solucionador DPLL incorporando aprendizado de cláusulas e em um solucionador no estado-da-arte, o solucionador SAT zChaff2004. Os resultados mostraram uma melhoria significativa no tempo de execução e uma quantidade menor de instâncias não concluídas dentro de um tempo limite, especialmente nos problemas industriais e apesar da nova estrutura de dados, maior e mais rica em informações, ser incorporada ao solucionador.

A intuição fundamental do algoritmo está na sua capacidade de forçar a escolha de

uma variável, esta escolha determinada pela subtração de cubos. Essa subtração ocorre após um retrocesso cronológico, quando uma cláusula associada à variável de conflito é escolhida e de onde serão selecionadas as próximas variáveis de decisão. Desde que o número de retrocessos cronológicos é pequeno, a aplicação do algoritmo é esporádica e apresenta um ótimo benefício, que se sobrepõe aos custos necessários de infraestrutura para suportar a nova estrutura de decisão.

O algoritmo se mostra indicado aos problemas industriais, uma vez que esses, em geral, possuem uma certa estrutura interna na sua fórmula na FNC. O mesmo tipo de estrutura que tornou o aprendizado de cláusulas fundamental para a eficiência dos atuais solucionadores do problema da satisfabilidade baseados no algoritmo clássico DPLL.

Os benefícios coletados sobre uma vasta e diversificada base de problemas, não deixam dúvidas que esta é uma estratégia eficiente. A intuição, decorrente da subtração de cubos, segue as pesquisas mais recentes na área, que indicam a necessidade de uma atenção maior não apenas às variáveis do problema da satisfabilidade, mas, sobretudo de uma análise das cláusulas do problema, conforme as pesquisas de Dershowitz, Hanna e Nadel em [Der05] e Beame, Kautz e Sabharwal em [Bea04].

Como trabalhos futuros, tencionamos melhorar as heurísticas utilizadas para a seleção das próximas cláusulas-cubo a serem subtraídas e à escolha da variável dentro dessa cláusula: por exemplo a idade da cláusula, calculada durante a análise de conflito no 1UIP foi utilizada, mas pesquisas mais recentes mostraram benefícios na utilização de outras formas de cálculo conforme sugerido por Dershowitz em [Der05]. Acredito ainda que, as cláusulas marcadas e selecionadas pelo algoritmo, podem oferecer uma intuição na pesquisa de núcleos não satisfazíveis e no seu menor tamanho, conforme o trabalho em desenvolvimento por Inês Lynce apresentado em [Lyn04a]. Pesquisas mais recentes têm sido orientadas para novas formas de codificação SAT de circuitos, essa nova formulação, capaz de gerar fórmulas FNC menores e mais fáceis de serem resolvidas pelos solucionadores SAT, utilizam estruturas de dados mais expressivas e

condensadas ou modelos polinomiais para a transformação dos circuitos em fórmulas na FNC. Essas idéias, apresentadas por Audemard [Aud07] e Manolios [Man07], podem indicar uma futura avaliação da estrutura de dados necessária à subtração de cubos e a sua associação com essas novas idéias, uma vez que o solucionador proposto nesse trabalho se mostrou eficiente para a solução de problemas industriais e de circuitos.

Referências bibliográficas

[Abd00] Parosh Aziz Abdulla, Per Bjesse, and Niklas Ellen. Symbolic reachability analysis based on SAT-solvers. In *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2000)*, 2000.

[Abr00] M. Abramovici and J.T. de Sousa. A SAT Solver Using Reconfigurable Hardware and Virtual Logic. In *Journal of Automated Reasoning*, vol. 24, nos. 1-2, pp. 5-36, 2000.

[Abr97] M. Abramovici and D. Saab. Satisfiability on Reconfigurable Hardware. In *Proceedings of Seventh International Workshop Field-Programmable Logic and Applications*, pp. 448-456, 1997.

[Agr04] M. Agrawal, N. Kayal e N. Saxena. PRIMES is in . In *Annals of Mathematics 160*, no.2, pp.781-793, 2004.

[Aim96] Y. Asahiro, K.Iwama, E. Miyano, Random Generation of Test Instances with controled Attributes. In D.S. Johnson and M.A.Trick, editors, *Cliques, Coloring, and Satisfiability: The Second DIMACS Implementation Challenge,* Vol. 26 of DIMACS Series on Discrete Mathematics and Theoretical Computer Science, pp. 377-394. American Math. Soc., pp. 127-154, 1996.

[Ake78] S. B. Akers. Binary Decision Diagram. In *IEEE Transactions on Computers*, vol. c-27, no. 6, June 1978.

[Ant00] Antje Strohmaier **.** A Complete Neural Network Algorithm for Horn-Sat. In *Applied Logic Series Intellectics and Computational Logic*, Vol. 19, pp. 313-325, Kluwer BV, 2000.

[Arei04] S. Areibi and Z. Yang, Effective Memetic Algorithms for VLSI Design = Genetic Algorithms + Local Search + Multi-Level Clustering. *In Journal of Evolutionary Computations, Special Issue on "Memetic Evolutionary Algorithms"*, Vol. 12, pp: 327-353, No. 3, 2004.

[Aud07] Gilles Audemard and Lakhdar Saïs. Circuit Based Encoding of CNF Formula. In *Proceedings of 10th International Conference on Theory and Applications of Satisfiability Testing (SAT 2007), LNCS*, vol 4501/2007, pp. 16-21, Portugal, May, 2007.

[Bap00] L. Baptista, J. Marques Silva. Using Randomization and Learning to Solve Hard Real-World Instances of Satisfiability. In *Proceedings of Principles and Practice of Constraint Programming - CP 2000: 6th International Conference, CP 2000*, p. 489, Singapore, September 2000.

[Bap01] L. Baptista, I. Lynce, J. Marques Silva. Complete Search Restart Strategies for Satisfability. In *Proceedings of IJCAI'01 Workshop on Stochastic Search Algorithms (IJCAI-SSA)*, August 2001.

[Bay97] Bayardo Jr., R. J. and Schrag, R. C. Using CST look-back techniques to solve real world SAT instances. In *Proceedings of 14th National Conference on Artificial Intelligence (AAAI-97),* pp. 203 -208, USA, 1997.

[Bea04] P. Beame, H. Kautz, A. Sabharwal. Towards Understanding and Harnessing the Potential of Clause Learning. *In Journal of Artificial Intelligence Research*, Vol. 22, pp. 319-351, December 2004

[Bie99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99)*, 1999.

[Boy00] M. Boyd and T. Larrabee. ELVIS—a Scalable, Loadable Custom Programmable Logic Device for Solving Boolean Satisfiability Problems. In *Proceedings of Eight IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2000.

[Bra90] K.S. Brace, R.L. Rudell, and R.E. Bryant. Eficient implementation of a BDD package. In *Proceedings of Design Automation Conference (DAC)*, 1990.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. In *IEEE Transactions in Computers*, 8(35):677-691, 1986.

[Bry92] R. E. Bryant, Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. In *ACM Computing Surveys*, vol. 24, no.3, Sep. 1992.

[Bur92] M. Buro and H. Kleine-Büning. Report on a SAT competition. *Technical report, University of Paderborn*, November, 1992.

[Cas03] C. Castellini, E. Giunchiglia, and A. Tachella. SAT-based planning in complex domains: concurrency, constraints and non-determinism. *Artificial Intelligence*, 147:85-117, 2003.

[Chu99] C.K. Chung and P.H.W. Leong. An Architecture for Solving Boolean Satisfiability Using Runtime Configurable Hardware. In *Proceedings of International Workshop Parallel Processing*, pp. 352-357, 1999.

[Coo71] S.A. Cook. The complexity of theorem proving procedures. In *Proceedings of the 3rd annual ACM symposium on the Theory of Computing*, pp. 151-158, 1971.

[Dal01] Dale Schuurmans, Finnegan Southey, and Robert C. Holte. The exponentiated subgradient algorithm for heuristic Boolean programming. In *Proceedings of IJCAI-01*, 2001.

[Dan02] A. Dandalis and V.K. Prasanna. Run-Time Performance Optimization of an FPGA-Based Deduction Engine for SAT Solvers. In *ACM Transactions in Design Automation of Electronic Systems*, vol. 7, no. 4, pp. 547-562, Oct. 2002.

[Dav04] Dave A. D. Tompkins and Holger H. Hoos. Novelty$^+$ and Adaptive Novelty$^+$. In *of Sixth International Conference on Theory and Applications of Satisfiability Testing* SAT2004 - Competition Booklet, (solver description), May 2004.

[Dav05] Dave A. D. Tompkins and Holger H. Hoos. UBCSAT: An Implementation and Experimentation Environment for SLS Algorithms for SAT and MAX-SAT. In *Proceedings of Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, pp. 306-320, Germany, 2004.

[Dav06] Dave A. D. Tompkins and Holger H. Hoos. On the Quality and Quantity of Random Decisions in Stochastic Local Search for SAT. In *Proceedings of the Nineteenth Conference of the Canadian Society for Computational Studies of Intelligence (AI 2006), Lecture Notes in Artificial Intelligence*, Vol. 4013, pp. 146-158, Springer Verlag, Germany, 2006.

[Dav60] M. Davis and H. Putnam. A computing procedure for quantification theory. In *Journal of the ACM*, vol. 7, pp. 201–215, July, 1960.

[Dav62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. In *Communications of the ACM*, 5(7):394-397, July 1962.

[Dav97] David McAllester, Bart Selman, and Henry Kautz. Evidence for invariants in local search. In *Proceedings of the Fourteenth National Conference on ArtificialIntelligence (AAAI'97)*, pp. 321-326, Providence, Rhode Island, 1997.

[Der05] Nachum Dershowitz, Ziyad Hanna and Alexander Nadel. A Clause-Based Heuristic for SAT Solvers. In *Proceedings of 8th International Conference on Theory and Applications of Satisfiability Testing (SAT05)*, June 2005

[Dub01] Olivier Dubois and Gilles Dequen. A backbone-search heuristic for eficient solving of hard 3-SAT formulae. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'01)*, 2001.

[Een03] Eén N., Sörensson N. An Extensible SAT-solver. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT2003*) LNCS 2919, pp 502-518, 2003

[Ele99] Elena Marchiori and Claudio Rossi. A fipping genetic algorithm for hard 3-SAT problems. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pp. 393-400, USA, 1999.

[Ern97] M. Ernst, T.D. Millstein, and D.S. Weld. Automatic SAT-compilation of planning problems. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 1169-1177, 1997.

[Fio03] C. Fiorini, E. Martinelli, and F. Massacci. How to take an RSA signature byencoding modular root finding as a SAT problem. In *Discrete Applied Mathematics*, 130:101-127, 2003.

[Fra97] J. Frank, P. Cheesman and J. Stutz. When gravity fails: Local Search topology. In *Journal of Artificial Intelligence Research*, 7:249-281, 1997.

[Fre06] Frédéric Lardeux, Frédéric Saubion, Jin-Kao Hão. GASAT: A Genetic Local Search Algorithm for the Satisfiability Problem. In *Journal Evolutionary Computation* , Vol. 14, No. 2, pp. 223-253, April 2006.

[Fre95] J.W. Freeman. Improvements to Propositional Satisfiability. *PhD thesis, University of Pennsylvania, Philadelphia(PA),* May 1995

[Gar79] M.R. Garey, D.S. Johnson, *Computers and Intractability: a Guide to the Theory of NP-Completeness*, W.H. Freeman, 1979.

[Gel01] A. V. Gelder. Combining preorder and postorder resolution in a satisfiability solver. In *Electronic Notes in Discrete Mathematics* (H. Kautz and B. Selman, eds.), vol. 9, Elsevier, 2001.

[Gia01] P. Di Giacomo, G. Felici, R. Maceratini, and K. Truemper. Application of a new logic domain method for the diagnosis of hepatocellular carcinoma. In *Proceedings of the Tenth World Congress on Health and Medical Informatics*,2001.

[Gom98] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*, pp. 431-437, Madison, Wisconsin, 1998.

[Gro00] J. F. Groote. The propositional formula checker HeerHugo. In *Journal of Automated Reasoning*, Vol.24, Num.1-2 , Springer Netherlands, February, 2000

[Guc01] S.A. Guccione. Reconfigurable Computing at Xilinx. *Keynote talk, EUROMICRO Symposium on Digital System Design*, Sept. 2001.

[Gup00] Aarti Gupta, Zijiang Yang, Pranav Ashar, and Anubhav Gupta. SAT-based image computation with application in reachability analysis. In *Proceedings of Third International Conference Formal Methods in Computer-Aided Design (FMCAD 2000)*, 2000.

[Hac96] G. Hachtel and F. Somenzi. *Logic Sysntheiss and Verification Algorithms*. Kluwer Academic Publishers, 1996.

[Ham97] Y. Hamadi and D. Merceron. Reconfigurable Architectures: A New Vision for Optimization Problems. In *Proceedings of Third International Conference on Principles and Practice of Constraint Programming*, pp. 209-215, 1997.

[Her05] P.R. Herwig, M.J.H. Heule, P.M. Van Lambalgen, and H. Van Maaren. A new method to construct lower bounds for van der waerden numbers. *The Electronic Journal of Combinatorics*, 12, 2005.

[Hil05] F.S. Hillier and G.J. Lieberman. *Introduction to operations research*. Mc-GrawHill, 8th edition edition, 2005.

[Hir05] Hirsch Edward and Kojevnikov Arist. UnitWalk: A new SAT solver that uses local search guided by unit clause elimination In *Annals of Mathematics and Artificial Intelligence*, Vol. 43, Num. 1, pp. 91-111 (21), January 2005.

[Hol99] Holger H. Hoos. On the run-time behaviour of stochastic local search algorithms for SAT. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI'99)*, 1999.

[Hon79] S.J. Hong and D.L. Ostapko. Fault Analysis and Test Generation for PLAs. In *IEEE Transactions on Computers, vol C-28, n. 9*, September 1979

[Hoo00] J. Hooker. *Logic-based methods for optimizations, combining optimization and constraint satisfaction*. JohnnWiley and Sons, 2000.

[Hoo95] J. N. Hooker and V. Vinay. Branching rules for satisfiability. In *Journal of Automated Reasoning*, vol. 15, pp. 359–383, 1995.

[Ine05] Inês Lynce and João Marques Silva. Efficient data structures for backtrack search SAT solvers. Annals of Mathematics and Artificial Intelligence, 43: 137-152, 2005.

[Ine06a] Inês Lynce and Jöel Ouaknine. Sudoku as a SAT Problem. In *Proceedings of the Ninth International Symposium on Artificial Intelligence and Mathematics (AIMATH 2006)*, Jan. 2006.

[Ine06b] Inês Lynce and João Marques Silva. SAT in Bioinformatics: Making the Case with Haplotype Inference. In *9th International Conference on Theory and Applications of Satisfiability Testing*, pp. 136-141 , Springer, August 2006.

[Jer90] R.G Jeroslaw and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167-187, 1990.

[Jun93] Jun Gu. Local search for satisfiability SAT problem. *IEEE Transactions on Systems and Cybernetics*, 23(3):1108-1129, 1993.

[Kam00] M.B. Do and S. Kambhampati. On the use of LP relaxations in solving SAT encodings of planning problems. In *Working note of the Workshop on the Integration of AI and OR Techniques for Combinatorial Optimization , AAAI-2000*,2000.

[Kau92] H. A. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, pp. 359-363, 1992.

[Kau96] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pp. 1194-1201, Portland, OR, 1996.

[Kau99] Henry A. Kautz and Bart Selman. Unifying SAT-based and graph-based planning. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI'99)*, Stockholm, Sweden, July, 1999.

[Kib06] R. H. Kibria, Y. Li. Optimizing the Initialization of Dynamic Decision Heuristics in DPLL SAT Solvers Using Genetic Programming. In *Proceedings of Genetic Programming: 9th European Conference, EuroGP* 2006, Hungary, LNCS 3905/2006, Springer Verlag, April 2006.

[Kle77] J. de Kleer, An Assumption-Based TMS. In *Artificial Intelligence,* vol. 28, pp. 127-162, 1986.

[Kun94] W. Kunz and D. Pradhan. Recursive Learning: A new implication technique for efficient solutions to CAD problems: Test, verification and optimization. *IEEE Transactions on Computer-Aided Design*, 13(9):1143-1158, September 1994.

[Lar92] Tracy Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(1):6-22, January 1992.

[Lar93] T. Larrabe and Y. Tsuji. Evidence for a satisfiable threshold for random 3cnf formulas. In *Proceedings of AAAI Spring Symposium*, 1993.

[Lee59] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999, 1959.

[Leo04] P.H.W. Leong, C.W. Sham, W.C. Wong, H.Y. Wong, W.S. Yuen, and M.P. Leong. A Bitstream Reconfigurable FPGA Implementation of the WSAT Algorithm. In *IEEE Transactions on VLSI Systems*, vol. 9, no. 1, pp. 197-201, 2001.

[Lia97] C.M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability. In *International Joint Conference on Artificial Intelligence (IJCAI),* pp. 315-326, 1997

[Lyn02] Lynce I. and J. Marques-Silva. Efficient Data Structures for Fast SAT Solvers. *International Symposium on the Theory and Applications of Satisfiability Testing(SAT)*, May 2002

[Lyn04a] Lynce I. and João P. Marques-Silva. On Computing Minimum Unsatisfiable Cores. In *Proceedings of 7th International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, May 2004.

[Lyn04b] Inês Lynce. Propositional Satisfiability: Techniques, Algorithms and Applications . *In PhD dissertation, Instituto Superior Técnico, Univers. Técnica de Lisboa.*, October 2004.

[Mal03] Mali, A.D. Lipen, Y. MFSAT: a SAT solver using multi-flip local search. In *Proceedings of 15th IEEE International Conference on Tools with Artificial Intelligence*, pp. 84- 93, November 2003.

[Man07] Panagiotis Manolios and Daron Vroon. Efficient Circuit to CNF Conversion. In *Proceedings of 10th International Conference on Theory and Applications of Satisfiability Testing (SAT 2007), LNCS, vol 4501/2007, pp. 4-9*, Portugal, May, 2007.

[Man98] V.M. Manquinho, J.P. Marques Silva, A.L. Oliveira, and K.A. Sakallah. Satisfiability-based algorithms for 0-1 integer programming. In *InternationalWorkshop on Logic Synthesis*, 1998.

[Mar99] Elena Marchiori and Claudio Rossi. A flipping genetic algorithm for hard 3-SAT problems. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 393-400, USA,1999.

[Mas00] F. Massacci and L. Marraro. Logical cryptanalysis as a SAT problem:encoding and analysis of the U.S. Data Encryption Standard. *Journal of Automated Reasoning*, 24:165-203, 2000.

[Mei98] C. Meinel and T. Theobald. Algorithms and Data Structures in VLSI Design, OBDD -Foundations and Applications. Springer-Verlag, 1998.

[Men99] O. Mencer and M. Platzner. Dynamic Circuit Generation for Boolean Satisfiability in an Object-Oriented Design Environment. In *Proceedings of 32nd Hawaii International Conference on System Sciences, (HICSS-32)*, 1999.

[Mil02] Ken L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Proceedings of 14th Conference on Computer-Aided Verification (CAV2002)*. Springer Verlag, 2002.

[Min00] Chu-Min Li and Sylvain Grard. On the limit of branching rules for hard random unsatisfiable 3-SAT. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI-2000)*, pp. 98-102, Berlin, 2000.

[Mit92] David Mitchell, Bart Selman, and Hector Levesque. Hard and easy distribution of SAT problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI'92)*, pp. 440-446, 1992.

[Mos01] Matthew Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of 38th Design Automation Conference (DAC01), pp. 530-535*, 2001.

[Nam99] G. Nam, K. A. Sakallah, and R. A. Rutenbar. Satisfiability-based layout revisited: Routing complex FPGAs via search-based Boolean SAT. In *Proceedings of International Symposium on FPGAs*, February 1999.

[Nik05] Niklas Eén, Armin Biere. Effective Preprocessing in SAT through Variable and Clause Elimination. In Proceedings of *International Symposium on the Theory and Applications of Satisfiability Testing (SAT05)*, Jun 2005.

[Nor01] J. Nordström Stalmarck's Method versus Resolution: A Comparative Theoretical Study. In *PhD Thesis, Rice University,* KTH Stockholm, 2001.

[Nov02] Y.Novikov and E.Goldberg. Berkmin: a Fast and Robust Sat-Solver. In *Proceedings of Design, Automation and Test in Europe (DATE02)*, p. 0142, 2002.

[Pap91] C. H. Papadimitriou. On selecting a satisfying truth assignment. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 163-169, 1991.

[Pat04] Patrick Mills, Edward Tsang, Qingfu Zhang, John Ford .A survey of AI-based meta-heuristics for dealing with local optima in local search. In *Technical Report Series, CSM-416, Department of Computer Science, University of Essex*, September 2004

[Pdu06] P. Du, S. Areibi, G. Grewal and D. Banerji. Hierarchical FPGA Placement. In *Accepted for publication in Canadian Journal on Electrical and Computer Engineering*, To Appear in 2007.

[Pla98] M. Platzner and G. De Micheli. Acceleration of Satisfiability Algorithms by Reconfigurable Hardware. In *Field-Programmable Logic: From FPGAs to Computing Paradigm*, R.W. Hartenstein and A. Keevallik, eds., pp. 69-78, Springer, 1998.

[Pot03] N.R. Potlapally. Solving register allocation using boolean satisfiability. Não publicado, 2003, URL: http://www.princeton.edu/~npotlapa/files/papers/regalloc.pdf

[Pra96] M. R. Prasad, P. Chong, and K. Keutzer. Why is ATPG easy? In *Proceedings of the Design Automation Conference (DAC)*, New Orleans, Louisiana, United States, 1996.

[Ras98] Rashid, J. Leonard, and W.H. Mangione-Smith. Dynamic Circuit Generation for Solving Specific Problem Instances of Boolean Satisfiability. In *Proceedings of Sixth IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 196-205, 1998.

[Red00] M. Redekopp and A. Dandalis. A Parallel Pipelined SAT Solver for FPGA's. In *Proceedings of 10th International Conference in Field-Programmable Logic and Applications*, pp. 462-468, 2000.

[Rei02] N.A. Reis and J.T. de Sousa. On Implementing a Configware/ Software SAT Solver. In *Proceedings of 10th IEEE International Symposium on Field-Programmable Custom Computing Machines*, pp. 282-283, 2002.

[Rin04] J. Rintanen, K. Heljanko, and I. Niemela. Parallel encodings of classical planning as satisfiability. In J.J. Alferes and J. Leite, editors, *Logics in Artificial Intelligence: 9th European Conference, JELIA 2004, Lecture Notes in Computer Science, vol.3229*, pp. 307-319. Springer-Verlag, 2004.

[Rip01] R.C. Ripado and J.T. de Sousa. A Simulation Tool for a Pipelined SAT Solver. In *Proceedings of XVI Conference in Design of Circuits and Integrated Systems*, pp. 498-503, Nov. 2001.

[Rob65] J. Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. In *Journal of the* ACM (JACM), Vol. 12, Issue 1, pp. 23–41, 1965.

[Rol98] Rolf Drechsler, Nicole Drechsler, and Wolfgang Gunther. Fast exact minimization of BDDs. In *Proceedings of the Design Automation Conference (DAC)*,1998.

[Rud93] R. Rudell. Dynamic variable ordering for Ordered Binary Decision Diagrams. In *Proceedings of the IEEE/ACM International Conference on Compute- Aided Design (ICCAD)*, 1993.

[Rya04] Ryan, L. Efficient algorithms for clause-learning SAT solvers. In *Master's thesis, Simon Fraser University*, 2004

[Sel92] Bart Selman, Hector Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI'92)*, pp. 459-465, 1992.

[Sel97] Bart Selman, Henry A. Kautz, and David A. McAllester. Ten challenges in propositional reasoning and search. In *Proceedings of the Fifteenth International Joint Conference on ArtificialIntelligence (IJCAI'97)*, pp. 50-54, 1997.

[Sha98] Y. Shang and B.Wah. A discrete lagrangian based global search method for solving satisfability problems. In *Journal of Global Optimization*, 12(1), pp. 61-99, January 1998.

[She00] Mary Sheeran and Gunnar Stalmark. A tutorial on Stalmark's proof procedure for propositional logic. *Formal Methods in System Design*, 16:23-58, January 2000.

[Sil94] João P. Marques-Silva and Karem A. Sakallah. Eficient and robust test generation-based timing analysis. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, May 1994.

[Sil96] J.P. Marques-Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design,* pp. 220-227, November 1996.

[Sil97a] Joonyoung Kim, João P. Marques Silva, Hamid Savoj, and Karem A. Sakallah. RID-GRASP: Redundancy identification and removal using GRASP. In *IEEE/ACM International Workshop on Logic Synthesis*, 1997.

[Sil97b] J.P.M. Silva and A.L. Oliveira. Improving Satisfiability Algorithms with Dominance and Partitioning. In *International Workshop on Logic Systems,* May 1997.

[Sil98] J. P. Marques Silva. An Overview of Backtrack Search Satisfiability Algorithms. In *Fifth International Symposium on Artificial Intelligence and Mathematics*, January 1998.

[Sil99a] João P. Marques-Silva and Thomas Glass. Combinational equivalence checking using satisfiability and recursive learning. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe (DATE)*, 1999.

[Sil99b] J.P. Marques-Silva. The Impact of Branching Heuristics in Propositional Satisfiability Algorithms. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence (EPIA),* September 1999.

[Skl02] Skliarova and A.B. Ferrari. A Hardware/Software Approach to Accelerate Boolean Satisfiability. In *Proceedings of IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop*, pp. 270-277, 2002.

[Skl04a] Y. Skliarova and A.B. Ferrari. A software / Reconfigurable Hardware SAT Solvers. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(4), pp. 408- 419, April 2004

[Skl04b] Y. Skliarova and A.B. Ferrari. Reconfigurable Hardware SAT Solvers: A Survey of Systems. In *IEEE Transactions on Computers*, 53(11), pp. 1449- 1461, November 2004.

[Smi79] J. E. Smith. Detection of Faults in Programmable Logic Arrays. In In *IEEE Transactions on Computers, vol C-28, n.11*, November 1979.

[Soh96] A.Sohn, Parallel Satisfiability Test with Synchronous Simulated Annealing, In *Journal of Parallel and Distributed Computing*, pp.195-204, August 1996.

[Sou01] J. de Sousa, J.P. Marques-Silva, and M. Abramovici. A Configware/ Software Approach to SAT Solving. In *Proceedings of Ninth IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2001.

[Sta77] R.M. Stallman and G.J. Sussman. Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis. *Artificial Intelligence,* vol. 9, pp. 135-196, October 1977.

[Stal89] G. Stalmarck. A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula. In *Technical report*, European Patent N 0403 454 (1995), US Patent N 5 276 897, Swedish Patent N 467 076 (1989), 1989.

[Ste96] P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Transactions on Computer-Aided Design*, 15(9):1167-1176, September 1996.

[Suy01] T. Suyama, M. Yokoo, H. Saw, A. Nagoya. Solving Satisfiability Problems Using Reconfigurable Computing. In *IEEE Transactions on VLSI Systems*, vol. 9, no. 1, pp. 109-116, 2001.

[Url01] Acesso outubro 2007 URL: http://www.claymath.org/millennium/

[Url02] Acesso outubro 2007 URL: http://www.prover.com/

[Url03] Acesso outubro 2007 URL: http://www.satcompetition.org/

[Url04] Acesso outubro 2007 URL: http://www.ecs.soton.ac.uk/~jpms/

[Url05] Acesso outubro 2007 URL: http://www.princeton.edu/~chaff/

[Url06] Acesso outubro 2007 URL: http://cs-svr1.swan.ac.uk/~csoliver/OKsolver.html

[Url07] Acesso outubro 2007 URL: http://www.laria.u-picardie.fr/~dequen/sat/

[Url08] Acesso outubro 2007 URL: http://www.st.ewi.tudelft.nl/sat/download.php

[Url09] Acesso outubro 2007 URL: http://eigold.tripod.com/BerkMin.html

[Url10] Acesso outubro 2007 URL: http://www.cs.chalmers.se/~een/Satzoo/

[Url11] Acesso outubro 2007 URL: http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/

[Url12] Acesso outubro 2007 URL: http://logic.pdmi.ras.ru/~arist/UnitWalk/

[Url13] Acesso outubro 2007 URL: http://dimacs.rutgers.edu/

[Url14] Acesso outubro 2007 URL:http://research.microsoft.com/users/lintaoz/SATSolving/satsolving.htm

[Url15] Acesso outubro 2007 URL: http://www.satcompetition.org/

[Url16] Acesso outubro 2007 URL: http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/bmcbenchmarks.html

[Url17] Acesso outubro 2007 URL: http://www.ece.cmu.edu/~mvelev/

[Url18] Acesso outubro 2007 URL: http://www.satlib.org/ubcsat/

[Vel01] M.N. Velev and R.E. Bryant. Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. In *Proceedings of the Design Automation Conference (DAC)*, June 2001.

[War98] J.P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, 68(2):63-69, 1998.

[Wil93] William M. Spears. Simulated annealing for hard satisfiability problems. In David S. Johnson and Michael A. Trick, editors, Cliques, Coloring and Satisfiability, Second *DIMACS Implementation Challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pp. 533-558. American Mathematical Society, 1993.

[Wil96] William M. Spears. A NN algorithm for Boolean satisfiability problems. In *Proceedings of the 1996 International Conference on Neural Networks*, pp. 1121-1126, 1996.

[Wol01] S.A. Wolfman and D.S. Weld. Combining linear programming and satisfiability solving for resource planning. *The Knowledge Engineering Review*, 16(1):85-99, 2001.

[Xia04] Xiaojun Bao and Shawki Areibi**.** Constructive and Local Search Heuristic Techniques for fpga Placement. In *17th annual IEEE Canadian Conference in Electrical and Computer Engineering*, pp. 13-15, vol 4, May 2004.

[Xyl04] X.Y. Li, M.F. Stallmann and F. Brglez, QingTing: A Local Search SAT Solver Using an Effective Switching Strategy and an Efficient Unit Propagation. In *Proceedings of Sixth International Conference on Theory and Applications of Satisfiability Testing*, pp. 53-68, Italy, May 2004.

[Yap03] R.H.C. Yap, S.Z.Q. Wang, and M.J. Henz. Hardware Implementations of Real-Time Reconfigurable WSAT Variants. In *Proceedings of 13th International Conference on Field-Programmable Logic and Applications*, pp. 488-496, 2003.

[Yok96] M. Yokoo, T. Suyama, and H. Sawada. Solving Satisfiability Problems Using Field Programmable Gate Arrays: First Results. *In Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, pp. 497-509, 1996.

[Yun99] W.H. Yung, Y.W. Seung, K.H. Lee, and P.H.W. Leong. A Runtime Reconfigurable Implementation of the GSAT Algorithm. In *Proceedings of Ninth International Workshop Field Programmable Logic and Applications*, pp. 526-531, 1999.

[Zab88] Zabih, R., and McAllester, D. A. A rearrangement search strategy for determining propositional satisfiability. In Proc. of the National Conference on Artificial Intelligence, pp. 155-160, 1988.

[Zar05] Zarpas E.. Benchmarking SAT Solvers for Bounded Model Checking. In *Proc. of 8th Intern. Conference on Theory and Applications of Satisfiability Testing (SAT05)*, pp. 340-354, June 2005.

[Zha01] L. Zhang, C.F. Madigan, M.H. Moskewicz and S. Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD01),* pp. 279-285, November 2001.

[Zha96] H. Zhang and M. E. Stickel. An eficient algorithm for unit propagation. In *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96)*, Fort Lauderdale (Florida USA), 1996.

[Zha97] Zhang, H. SATO: An Efficient Propositional Prover . In *Proceedings of International Conference on Automated Deduction (CADE-97)*, pp. 272-275, July 1997.

[Zho99] P. Zhong. Using Configurable Computing to Accelerate Boolean Satisfiability. *In PhD dissertation, Dept. of Electrical Eng., Princeton Univ.*, 1999.

[Zui06a] R. Zuim, J.T. Sousa, C. N. Coelho. Cube Subtraction Strategy in Reconfigurable Hardware SAT Solvers. In Proceedings of *Jornadas sobre Sistemas Reconfiguráveis, REC2006*. Porto, Portugal, February 2006.

[Zui06b] Zuim R., Sousa J. T., Coelho C. N.. Cube Subtraction in Sat Solvers. *In Proceedings of the 7*th *IEEE Latin America Test Workshop, LATW06,* pp. 33-40, Buenos Aires, Argentina, ISBN: 85-7727-022-x, March 2006.

[Zui06c] Zuim R., Sousa J. T., Coelho C. N.. A Fast SAT Solver Algorithm Best Suited to Reconfigurable Hardware. *In Proceedings of the* 19th Symposium on Integrated Circuits and System Design – 19th SBCCI, pp. 131-136, Ouro Preto, Brasil, ISBN: 1-59592-479-0, August 2006.

[Zui06d] Zuim R., Sousa J. T., Coelho C. N.. A Fast Sat Solver Strategy Based on Negated Clauses. *In Proceedings of the 14*th *IFIP WG 10.5 Int. Conf. on Very Large Scale Integration and System-on-chip, VLSI-SOC'2006,* pp. 110-115, Nice, France, ISBN:3-901882-19-7, October 2006.

[Zui06e] Zuim R., Sousa J. T., Coelho C. N.. A Decision Heuristic for DPLL SAT Solving based on Cube Subtraction. *In Journal IET Computers & Digital Techniques,* First submission in January.17.2006 and resubmission in December.19.2006., accept for publication in September.03.2007.

Apêndice A

A transformação de um problema real em um problema SAT

Uma gama considerável de problemas pode ser codificada em lógica proposicional. A seguir dois exemplos de como transformar um problema real em uma codificação para um solucionador de satisfabilidade.

A.1 Equivalência combinacional

Umas das formas mais simples são os circuitos combinacionais onde constantes como "verdadeiro" e "falso" podem ser utilizadas para representar as funções de chaves em circuitos eletrônicos. O exemplo a seguir resume as idéias de Marques Silva em [Sil99] e Lynce em [Lyn04b].

Um exemplo simples desta constatação é o circuito da Figura 1.1 que pode ser facilmente codificado através de lógica proposicional como: $(a \vee b) \wedge (\neg a \vee \neg b)$

Problemas SAT obtidos através de circuitos combinacionais normalmente envolvem a verificação de propriedades. Considerando limitações resultantes de uma simulação, é importante verificar a equivalência entre o circuito e o seu modelo comportamental. Normalmente as fórmulas SAT para estes tipos de problemas são descritas através de uma forma normal conjuntiva (CNF). Uma forma normal conjuntiva é uma conjunção

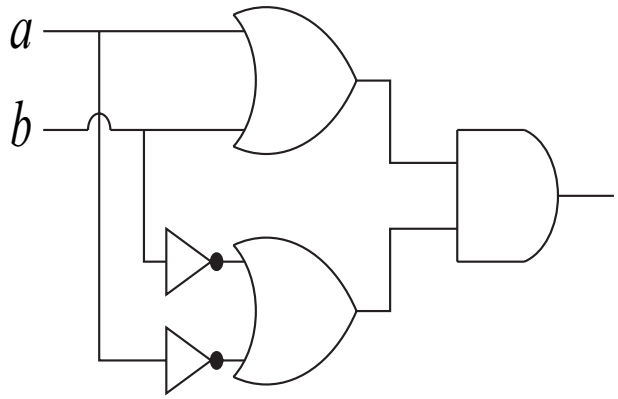

Figura A.1: Figura 1.1: Um simples circuito lógico combinacional

($\wedge$) de cláusulas, ou restrições. Uma cláusula é uma disjunção ($\vee$) de literais. Um literal é uma variável (x) ou o seu complemento ($\neg$x).

O problema de equivalência combinacional consiste em se determinar se dois circuitos digitais implementam a mesma função lógica booleana. É um problema NP-completo e a equivalência pode ser resolvida usando-se SAT para encontramos um contra-exemplo.

A equivalência baseada em SAT pode ser obtida através de um circuito miter. Este tipo de circuito consiste de dois circuitos C_1 e C_2, um conjunto de portas lógicas XOR e OR. Consideraremos as mesmas entradas para os circuitos C_1 e C_2 como $x_1, \ldots, x_n$. Denotaremos as saídas dos circuitos C_1 como $o_{11}, \ldots, o_{1m}$ e para C_2 como $o_{21}, \ldots, o_{2m}$. O circuito terá m portas lógicas XOR tendo como entradas o_{1i} e o_{2i} respectivamente, onde $i = 1, \ldots, m$. Todas as saídas das portas XOR serão as entradas de uma porta OR, cuja saída será O. Este circuito pode ser visto na Figura 1.2. Se a saída $O = 1$ os dois circuitos representam duas funções booleanas diferentes, se a saída $O = 0$, os dois circuitos representam a mesma função booleana.

Assim, ao codificarmos todo o circuito em CNF e adicionarmos esta cláusula O = 1 á fórmula, se C_1 e C_2 forem equivalentes teremos uma resposta não-sat para o problema.

Esta situação pode ser mais claramente ilustrada através de um exemplo. O circuito

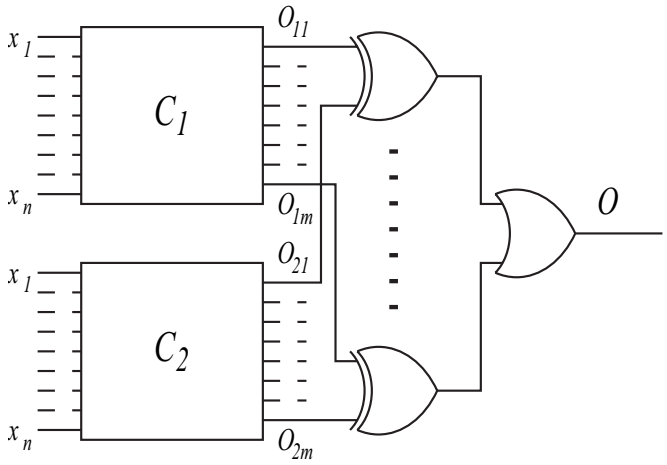

Figura A.2: Figura 1.2: Circuito para equivalência combinacional

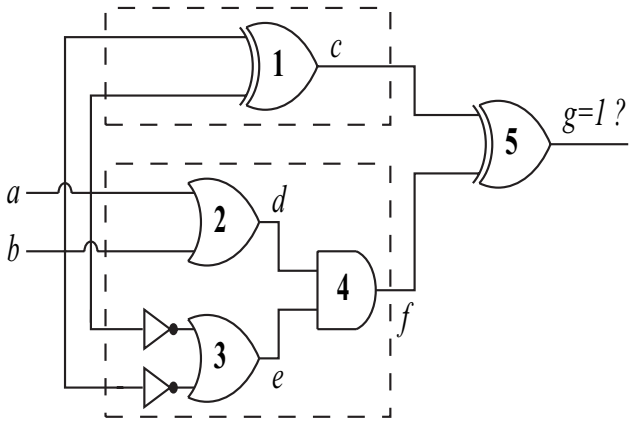

Figura A.3: Figura 1.3: Circuito de equivalência para uma porta lógica XOR

da Figura 1.1 nada mais é do que uma porta lógica XOR. Quando implementarmos o circuito para a equivalência combinacional, obteremos o circuito ilustrado na Figura 1.3.

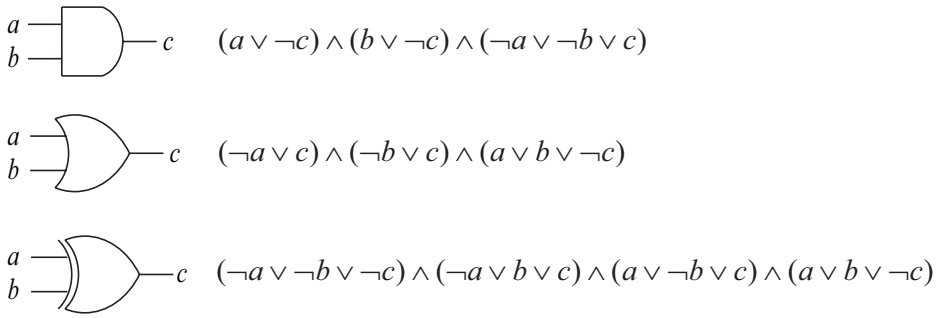

Figura A.4: Figura 1.4: Codificação CNF de portas lógicas

Para codificarmos o circuito da Figura 1.3, deveremos inicialmente considerar as

codificações das portas AND, OR e NOT, componentes do circuito da Figura 1.3, no formato CNF (forma normal conjuntiva). A codificação de cada porta lógica pode ser vista na Figura 1.4.

O problema final de satisfabilidade será construído quando todas as codificações forem agrupadas nas respectivas cláusulas respeitando a identificação de cada nó do circuito.

A numeração da cláusula descrita a seguir, representa a qual porta lógica ela se refere na Figura 1.3:

1. $(\neg a \vee \neg b \vee \neg c) \wedge (\neg a \vee b \vee c) \wedge (a \vee \neg b \vee c) \wedge (a \vee b \vee \neg c)$
2. $(\neg a \vee c) \wedge (\neg b \vee d) \wedge (a \vee b \vee \neg d)$
3. $(a \vee e) \wedge (b \vee e) \wedge (\neg a \vee \neg b \vee \neg e)$
4. $(d \vee \neg f) \wedge (e \vee \neg f) \wedge (\neg d \vee \neg e \vee f)$
5. $(\neg c \vee \neg f \vee \neg g) \wedge (\neg c \vee f \vee g) \wedge (c \vee \neg f \vee g) \wedge (c \vee f \vee \neg g)$
6. (g)

A cláusula 6 corresponde à introdução forçada da contradição, ou seja , para que o problema seja avaliado como sat ela deve ser avaliada como verdade e se assim o for, significa que os circuitos são diferentes. Resolvendo este problema de satisfabilidade podemos concluir que este é um problema não-sat, demonstrando a equivalência entre os dois circuitos.

A.2 Planejamento

Um exemplo simples da utilização de formulação SAT em problemas de planejamento é descrita a seguir:

Vamos considerar a existência de L locais.

Uma ordem de serviço constitui a atividade de levar um pacote de uma origem até um destino. Existe um caminhão com uma capacidade ilimitada que executa ordens de serviço. Ele gasta uma unidade de tempo para ir de um local até outro. O objetivo é encontrar uma rota tal que minimize o tempo de viagem até que todos os pacotes tenham sido transportados da origem até o destino.

As variáveis introduzidas são:

$CM_{xyt} = 1$ se o caminhão vai do local x para o y no tempo t, $= 0$ caso contrário,

$CA_{xt} = 1$ se o caminhão está no local x no tempo t, $= 0$ caso contrário,

$PM_{pxyt} = 1$ se o pacote p vai do local x para o y no tempo t, $= 0$ caso contrário,

$PA_{pxt} = 1$ se o pacote p está no local x no tempo t, $=0$ caso contrário,

Um exemplo de codificação SAT de diferentes cláusulas para este problema seria:

Se o caminhão se move de x para y no tempo t, ele deverá estar em x no tempo t. Este tipo de cláusula indica que para uma ação existe uma pré-condição:

$$\neg CM_{x,y,t} \vee CA_{x,t}$$

Se o caminhão se move de x para y no tempo t, ele estará em y no tempo $t+ 1$. Este tipo de cláusula indica que uma ação implica nos seus efeitos:

$$\neg CM_{x,y,t} \vee CA_{y,t+1}$$

Se o pacote p se move de x para y no tempo t, o caminhão deve se mover de x para y no tempo t, porque o pacote está dentro do caminhão.

$$\neg PM_{x,y,t} \vee CM_{x,y,t}$$

O caminhão não pode estar em dois locais ao mesmo tempo.

$$\neg CA_{x,t} \vee \neg CA_{y,t} \ ,x \neq y$$

O caminhão só poderá estar no local x no tempo t+1 se ele já estivesse lá no tempo t ou se moveu para lá no tempo t.

$$\neg CA_{x,t+1} \vee CA_{x,t} \vee (CM_{y,x,t}) \forall y \neq x$$

Desta forma, diversas outras restrições podem ser adicionadas ao exemplo para completamente formular o problema, mas já se pode perceber a intuição de como a formulação SAT pode ser construída para problemas de planejamento.