BRUNO CÉSAR RIBAS

SATISFATIBILIDADE NÃO-CLAUSAL RESTRITA ÀS VARIÁVEIS DE ENTRADA

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Fabiano Silva

CURITIBA

2011

BRUNO CÉSAR RIBAS

SATISFATIBILIDADE NÃO-CLAUSAL RESTRITA ÀS VARIÁVEIS DE ENTRADA

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Fabiano Silva

CURITIBA

2011

Ribas, Bruno César
Satisfatibilidade não-clausal restrita as variáveis de entrada / Bruno Cesár Ribas. – Curitiba, 2011.
53 f.: il., tab.

Dissertação (mestrado) – Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática.
Orientador: Fabiano Silva

1. Lógica de computador. 2. Logica simbólica e matemática. 3. Algoritmos. I. Silva, Fabiano. II. Universidade Federal do Paraná. III. Título.

CDD: 005.131

Ministério da Educação
Universidade Federal do Paraná
Programa de Pós-Graduação em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Bruno César Ribas, avaliamos o trabalho intitulado, *"SATISFATIBILIDADE NÃO-CLAUSAL RESTRITA ÀS VARIÁVEIS DE ENTRADA"*, cuja defesa foi realizada no dia 13 de abril de 2011, às 10:00 horas, no Auditório do Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação do candidato.

Curitiba, 13 de abril de 2011.

Prof. Dr. Fabiano Silva
DINF/UFPR – Orientador

Prof. Dr. Marcos Alexandre Castilho
DINF/UFPR – Coorientador

Prof. Dr. Marcelo Finger
USP – Membro Externo

Prof. Dr. Renato Carmo
DINF/UFPR – Membro Interno

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
Setor de Ciências Exatas
Coordenação de Pós-Graduação em Informática

AGRADECIMENTOS

Agradeço a minha família, meus amigos, professores e funcionários do Departamento de Informática da UFPR por todo apoio fornecido desde o início da graduação até a conclusão do Mestrado.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Resolvedores de satisfatibilidade Booleana (SAT) são amplamente utilizados em verificação de *software* e *hardware*. A maioria dos resolvedores SAT modernos são baseados no algoritmo Davis-Putnam-Logemann-Loveland(DPLL) e precisam que a fórmula de entrada esteja na forma normal conjuntiva (CNF). No entanto para muitos problemas de verificação a representação em CNF não é a forma mais natural de representação. Tipicamente os problemas são escritos em um modelo proposicional irrestrito e devem ser convertidos para CNF antes de aplicar o resolvedor SAT. A conversão do problema para CNF causa uma perda considerável de informação sobre a estrutura do problema. Apresentamos um novo resolvedor SAT que opera diretamente na forma proposicional irrestrita chamada ISCAS. O procedimento de busca proposto restringe as decisões às variáveis da fórmula e propaga os efeitos da valoração para os operadores lógicos, diferente das abordagens que associam a decisão de valores a qualquer componente da fórmula. A avaliação experimental do resolvedor desenvolvido mostra um desempenho competitivo com o de outros resolvedores atuais para instâncias em representações irrestritas.

ABSTRACT

Boolean satisfiability (SAT) solvers are heavily used in hardware and software verification. Most state-of-art SAT solvers are base on Davis-Putnam-Logemann-Loveland (DPLL) algorithm and require the input formula to be in conjuctive normal form (CNF). However for most problems CNF is not a very natural representation. Typically these problems are more easily expressed using unrestricted propositional formulae and hence must be converted to CNF before SAT solvers can be applied. This conversion entails a considerable loss of information about the problem's structure. We present a new SAT solver that operates directly in an unrestricted propositional formulae named ISCAS. We consider branching from the inputs of the formula instead of choosing any node. We present empirical evidence showing that exploiting the original structure with branching restriction is competitive to current state-of-art SAT solvers of non-clausal benchmarks.

CAPÍTULO 1

INTRODUÇÃO

Sistemas computacionais são cada vez mais usados em aplicações críticas como por exemplo: automóveis, aviões, trens e sistemas de distribuição de energia. A garantia de um funcionamento correto desses sistemas é crucial e quanto maior o sistema maior é a complexidade de afirmar o seu correto funcionamento, por isso, a verificação automática do software e do hardware é crucial para a construção e operação desses sistemas computacionais.

Muitas ferramentas de verificação automática dependem de procedimentos de decisão para verificar a satisfatibilidade de várias fórmulas lógicas proposicionais que são geradas durante o processo de verificação. O objetivo dessa dissertação é o desenvolvimento de um procedimento de decisão para auxiliar a verificação de sistemas.

No contexto desse trabalho um procedimento de decisão é um algoritmo que verifica a existência de alguma interpretação para uma fórmula lógica que a torne verdadeira, ou seja, se uma fórmula lógica é satisfatível ou insatisfatível. Satisfatibilidade Booleana (SAT) é o problema de decidir se uma fórmula é satisfatível, ou seja, verificar se há uma valoração para as variáveis da formula que a torne verdadeira. O problema SAT é de grande importância em várias áreas da Ciência da Computação, incluindo teoria da computação, inteligência artificial e verificação de hardware e software. SAT foi o primeiro problema provado NP-Completo[5] e nenhum algoritmo eficiente é conhecido para resolvê-lo. No entanto muitos resultados relevantes foram obtidos na última década [16, 17, 8] e permitiram a aplicação de resolvedores SAT na solução de problemas reais, em especial, nos problemas de verificação formal.

Os resolvedores SAT desenvolvidos atualmente utilizam duas abordagens principais: procedimentos de busca baseados no algoritmo clássico Davis-Putnam-Logemann-Loveland (DPLL)[7, 6] e procedimentos heurísticos de busca local[20]. Heurísticas de busca local,

em geral, não resultam em algoritmos completos, ou seja, não há garantia que esses algoritmos encontrem uma valoração satisfatível se ela existe, nem que provem que uma fórmula é insatisfatível. Os algoritmos não completos não podem ser utilizados na verificação pois é necessária a garantia da satisfatibilidade da fórmula. Alguns resolvedores não completos são GSAT[20], WALKSAT[19] e POLSAT[21]. Essa dissertação tratará apenas de algoritmos completos para o problema SAT.

A maioria dos resolvedores SAT precisa que a fórmula de entrada, ou instância do problema esteja na Forma Normal Conjuntiva (CNF do inglês *Conjuctive Normal Form*). Esse formato representado por uma conjunção de disjunções facilita o processo de implementação do resolvedor SAT. Dada uma valoração σ para um conjunto de variáveis da fórmula, o algoritmo *Boolean Constraint Propagation* (BCP) determina se σ torna a fórmula falsa, ou provê um conjunto de implicações lógicas.

Apesar do uso intensivo de fórmulas em CNF, as fórmulas típicas de aplicações industriais não estão necessariamente nesse formato. As fórmulas que não estão em CNF chamamos de não-clausais. Para verificar a satisfatibilidade de uma fórmula não-clausal ϕ usando um resolvedor SAT para CNF temos que converter ϕ. A conversão é feita introduzindo novas variáveis[24], resultando em uma nova fórmula ϕ' que é equivalente a ϕ em relação a propriedade de satisfatibilidade, ou seja, equi-satisfatível. Outra maneira de converter uma fórmula qualquer em CNF é aplicando equivalências lógicas como: as leis de De-Morgan, lei da dupla negação e a propriedade distributiva. Neste caso a fórmula equivalente obtida pode possuir um número exponencial de cláusulas em relação ao número de variáveis, enquanto que a conversão com introdução de novas variáveis a fórmula cresce apenas linearmente em relação ao número de variáveis, impactando diretamente na memória para representar a fórmula bem como no desempenho do BCP.

Para evitar as transformações das fórmulas não-clausais em CNF uma família de resolvedores SAT atuam diretamente na representação original do problema, esses resolvedores são chamados de resolvedores SAT não-clausais.

Os resolvedores não-clausais podem atuar de uma forma mais robusta em alguns tipos de problemas, pois agindo diretamente na fórmula original o resolvedor pode utilizar

heurísticas específicas com base na informação estrutural da fórmula, como em caso de circuitos onde operadores podem receber prioridade de valoração em relação aos demais.

O escopo de resolvedores não-clausais ainda é recente e alguns trabalhos se destacam: MINISAT++[22], KW_AIG[1] e NFLSAT[12]. Esses foram os resolvedores inscritos e vencedores na competição de resolvedores SAT de 2010. Alguns dos resolvedores não-clausais não atuam diretamente na fórmula não-clausal, mas durante o pré-processamento da fórmula a mesma é convertida para o formato CNF como é o caso do KW_AIG[1], MINISAT++[22]. Também existem resolvedores que atuam diretamente na fórmula original ou convertem para uma outra representação não-clausal, onde alguns conectivos booleanos ficam hierarquizados. O resolvedor NOCLAUSE[23] atua diretamente na fórmula original enquanto que SATMATE[11] e NFLSAT[12] convertem a fórmula para uma outra representação não-clausal.

Nesse trabalho será apresentada e analisada a implementação de um resolvedor não-clausal que atua diretamente na fórmula original não-clausal num formato de baixo custo de representação em memória, utilizando como base o algoritmo DPLL já adaptado para atuar em fórmulas que não estão em CNF.

No capítulo 2 apresenta-se uma introdução à lógica proposicional, que trata da notação e representação utilizada em todo o texto. O capítulo 3 trata como o processo de satisfatibilidade se desenvolveu desde a introdução do DPLL até os resolvedores atuais que dependem exclusivamente do formato CNF para resolver satisfatibilidade. O capítulo 4 faz uma abordagem não-clausal da satisfatibilidade, identificando os resolvedores atuais e como estes utilizam algumas técnicas para fórmulas em CNF. No capítulo 5 é apresentado um resolvedor para o problema de satisfatibilidade em fórmulas não-clausais representada em ISCAS, o LIAMFSAT. O capítulo 6 apresenta uma avaliação experimental comparativa do resolvedor proposto no capítulo anterior. Finalmente o capítulo 7 apresenta as considerações finais do trabalho e trabalhos futuros.

CAPÍTULO 2

INTRODUÇÃO À LÓGICA PROPOSICIONAL

Este capítulo apresenta definições básicas da lógica proposicional para o uso em satisfatibilidade. Serão tratados as notações, as formas de representação e a conversão entre as formas de representação.

Uma lógica proposicional é um sistema formal no qual fórmulas representam sentenças *declarativas*, ou proposições, que podem ser verdadeiras ou falsas, mas nunca ambas. "O céu é azul." e "A resposta é 42." são exemplos de proposições.

Partindo das proposições, é possível criar sentenças compostas utilizando os conectivos lógicos. Cinco são os conectivos lógicos: $\neg$ (negação), $\wedge$ (e), $\vee$ (ou), $\Rightarrow$ (se ... então) e $\Leftrightarrow$ (se e somente se). A junção de duas proposições por um conectivo forma uma sentença composta. De maneira geral, sentenças são compostas por outras sentenças e operadores. Dessa maneira se torna possível a representação de ideias mais complexas. Para a lógica proposicional, proposições, simples ou compostas, são definidas como fórmulas bem formadas ou, simplesmente, *fórmulas*. As regras que definem recursivamente uma fórmula são as seguintes:

- Um átomo é uma fórmula.
- Se α é uma fórmula, então $(\neg\alpha)$ também é uma fórmula.
- Sendo α e β fórmulas, então $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, $(\alpha \Rightarrow \beta)$ e $(\alpha \Leftrightarrow \beta)$ também serão.
- Todas as fórmulas são geradas pela aplicação das regras anteriores.

2.1 Definições e notações básicas

Ao longo do texto, utilizaremos as definições e notações definidas nessa seção.

Variáveis proposicionais são denotadas como $x_1, ..., x_n$ ou por letras do alfabeto em minusculas como p, q, r e podem receber valores verdade *verdadeiro* (também, V ou 1) ou

falso (também, F ou 0). O valor verdade assinalado em uma variável x é denotado como $v(x)$. Um *literal* l é uma variável x ou a sua negação $\neg x$. Uma *cláusula* c é a disjunção de literais. Uma *fórmula* CNF θ é a conjunção de cláusulas. Uma cláusula é dita *satisfeita* quando pelo menos um de deus literais assumiu o valor V e *não-satisfeita* quando todos os seus literais assumiram o valor F. *Cláusula Unitária* é uma cláusula representada por apenas um literal. *Literal Puro* é o literal que aprece em apenas uma forma em toda a fórmula, ou seja, a variável x aparece apenas na forma x ou $\neg x$ na fórmula.

O problema de satisfatibilidade consiste em decidir se existe uma valoração que torna a fórmula verdadeira, ou seja, uma atribuição de valores verdade para as variáveis da fórmula que tornem a fórmula satisfeita.

2.2 Formas de representação

Neste trabalho utilizamos grafos como estrutura de representação para as fórmulas proposicionais. Os literais são representados por apenas um vértice cada, diferente da forma escrita onde um mesmo literal é listado diversas vezes em uma mesma sentença. A negação também é representada diferente da forma escrita, pois no grafo a representação de negação passa por dois nós, um indicando a negação e outro que é a variável.

2.2.1 Forma Normal Conjuntiva

A Forma Normal Conjuntiva (*Conjuctive Normal Form* - CNF) é a classe de fórmulas da lógica proposicional que utiliza apenas os operadores lógicos de conjunção ($\wedge$), disjunção ($\vee$) e negação ($\neg$), sendo este último aplicável apenas sobre as variáveis proposicionais.

A sua representação é um grafo direcionado acíclico (DAG) de altura 2, onde a raiz é o operador de conjunção ($\wedge$), o primeiro nível são operadores de disjunção ($\vee$) e o segundo nível as folhas, compostas por literais. A figura 2.1 apresenta o diagrama como grafo para a fórmula $(p \vee q) \wedge (q \vee \neg p) \wedge (\neg p \vee \neg q)$.

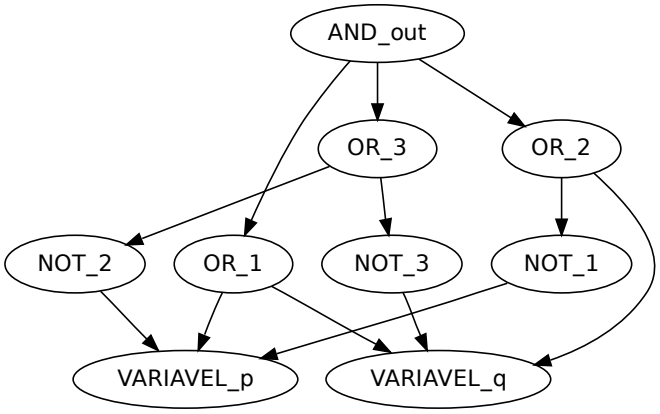

Figura 2.1: Diagrama de uma fórmula CNF. Os operadores são representados pelo nome do operador e uma informação que representa o seu índice, quando nó interno, *out* quando for a raiz do grafo.

2.2.2 Conversão para CNF

As fórmulas podem ser convertidas para CNF aplicando-se as equivalências lógicas que são as leis de De-Morgan, lei da dupla negação e a propriedade distributiva, obtendo uma fórmula equivalente que pode possuir um tamanho exponencial de cláusulas em relação ao número de variáveis, tornando impraticável a sua representação computacional (devido a grande quantidade de memória requerida para a representação). Para evitar essa explosão no tamanho da fórmula a transformação de Tseitin[24] é utilizada para a conversão.

A transformação de Tseitin consiste em adicionar novas variáveis que representam sub-fórmulas, inserindo restrições clausais que ligam o valor dessas variáveis com as sub-fórmulas que representam, e substituem na fórmula principal o seu representante.

Conjunções $x_1 \wedge x_2 \wedge ... \wedge x_n$ podem ser substituídas por uma única variável T_k se adicionarmos as restrições $(T_k \vee \neg x_1 \vee ... \vee \neg x_n) \wedge (\neg T_k \vee x_1) \wedge ... \wedge (\neg T_k \vee x_n)$. Essas restrições codificam a relação $T_k \Leftrightarrow (x_1 \wedge .. \wedge x_n)$.

Disjunções $x_1 \vee ... \vee x_n$ são representadas por T_l adicionando as restrições $(\neg T_l \wedge x_1 \wedge ... \wedge x_n) \vee (T_l \wedge \neg x_1) \vee ... \vee (T_l \wedge \neg x_n)$, codificando $T_l \Leftrightarrow (x_1 \vee ... \vee x_n)$.

A transformação pode ser feita recursivamente das sub-fórmulas de nível mais baixo até a chegada na raiz da fórmula. A nova fórmula codificada cresce apenas linearmente

em relação a quantidade de variáveis da fórmula original.

2.2.3 Forma Normal Negada

Forma Normal Negada (*Negation Normal Form* - NNF) é a classe de fórmulas da lógica proposicional que utiliza apenas os operadores lógicos de conjunção ($\wedge$), disjunção ($\vee$) e negação ($\neg$), sendo este último aplicável apenas sobre as variáveis proposicionais. Por exemplo, a fórmula $(((p \vee q) \wedge \neg r \wedge \neg q) \vee (\neg p \wedge (r \vee \neg s) \wedge q))$ está na NNF pois só apresenta conjunções e disjunções e as negações são aplicadas apenas sobre as variáveis da fórmula.

A sua representação é um grafo direcionado acíclico (DAG) onde cada folha é rotulada por verdadeiro, falso, X ou $\neg X$; cada nó interno é rotulado por $\wedge$ ou $\vee$ e podem ter qualquer quantidade de filhos. A figura 2.2 apresenta o diagrama como grafo para a fórmula $(((p \vee q) \wedge \neg r \wedge \neg q) \vee (\neg p \wedge (r \vee \neg s) \wedge q))$.

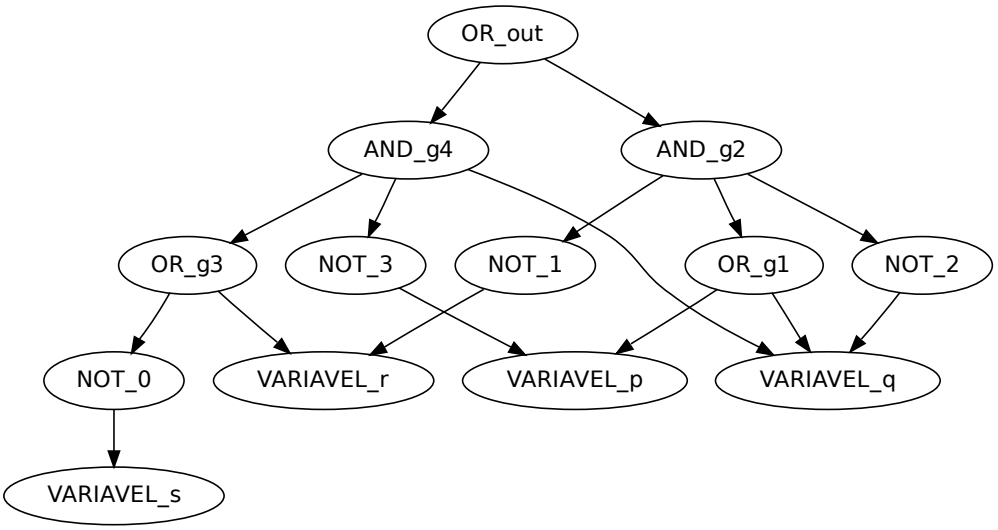

Figura 2.2: Diagrama de uma fórmula NNF

2.2.4 Representação não-clausal ISCAS

A representação ISCAS é um grafo direcionado acíclico (DAG) onde cada folha é rotulada por verdadeiro, falso ou X. cada nó interno é rotulado por $\wedge$, $\vee$, $\neg$, $\oplus$, $\overline{\wedge}$, $\Leftrightarrow$ e vários outros que podem ser utilizados em circuitos, e podem ter qualquer quantidade de filhos e pais, exceto o $\neg$ que pode ter apenas um filho.

Essa representação é comunmente utilizada na representação de circuitos e por isso pode conter várias saídas e diversos conectores, para este trabalho consideraremos fórmulas com apenas uma saída (a raiz do DAG) e os operadores $\wedge$, $\vee$ e $\neg$.

O formato ISCAS não impõe nenhuma restrição nos operadores, permitindo que qualquer fórmula lógica seja representada nesse formato sem nenhuma conversão, incluindo a CNF e a NNF. A figura 2.3 apresenta o diagrama como grafo para a fórmula $((p \vee q) \wedge (r \vee \neg s) \wedge \neg q) \vee \neg(\neg p \wedge (r \vee \neg s) \wedge q)$.

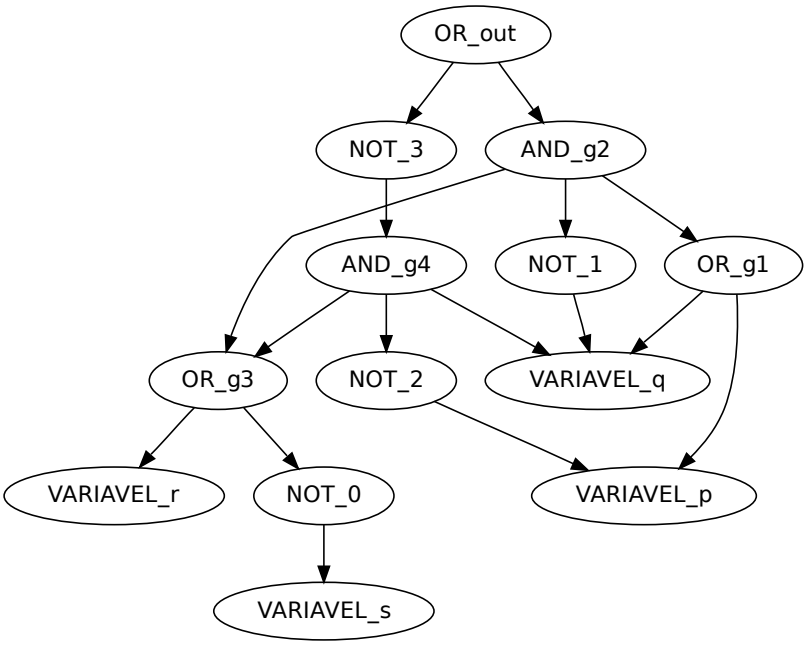

Figura 2.3: Diagrama de uma fórmula ISCAS

CAPÍTULO 3

SATISFATIBILIDADE

O algoritmo Davis-Putnam-Logemann-Loveland (DPLL)[6] foi desenvolvido por Martin Davis, George Logemann e Donald W. Loveland sendo um refinamento do algoritmo desenvolvido por Martin Davis e Hilary Putnam em 1960[7]. O algoritmo DPLL é usado como base dos principais resolvedores da atualidade, alguns deles são: GRASP[16], ZCHAFF[17] e MINISAT[8].

O DPLL é um algoritmo simples que escolhe uma variável da fórmula e então define um valor verdade para ela. A fórmula é simplificada e um processo recursivo verifica se a fórmula simplificada é satisfatível. Se a fórmula simplificada não for satisfatível a simplificação é desfeita e o valor verdade da variável é trocado e o processo se repete.

Durante a década de 90 o interesse pelo problema aumentou quando alguns grupos de pesquisa apresentaram estratégias que otimizaram o processo de verificação da fórmula em uma dada valoração, além de estimar qual seria a melhor escolha de variável para assinalar um valor verdade.

Os saltos de desempenho foram marcados por passos importantes na otimização de pedaços do algoritmo DPLL. O primeiro resolvedor a contribuir com o alto desempenho foi o SATO[29] que conseguiu otimizar o processo de avaliar as consequências de uma valoração na fórmula. Essa otimização foi atingida pelo modo em que as cláusulas foram representadas. A representação das cláusulas é feita com um vetor, onde dois marcadores são colocados: um no início do vetor, e o outro no fim do vetor. Sempre que o literal marcado pelo marcador for valorado como falso o marcador se desloca em uma posição, de forma que quando os marcadores se encontrem o literal marcado por ambos será ainda não valorado e todos os outros já estão valorados como falso, é feita a inferência do literal para verdadeiro. Essa representação de marcadores envolve um aumento de desempenho porque não é mais necessário avaliar todas as cláusulas a cada valoração.

Depois do SATO uma grande contribuição veio com o GRASP[16] (*Generic seaRch Algorithm for the Satisfiability Problem*), que introduziu o conceito de retrocesso não cronólogico sobre o algoritmo DPLL, e será tratado com detalhes na seção 3.4. Esse conceito do GRASP fez com que a revisão de valoração deixasse de ser feita pela inversão da última valoração feita e passou a ser analisada de forma mais inteligente. A análise do erro acontece com a identificação de qual decisão gerou a inferência da valoração que causou a inconsistência na fórmula. Vale notar que quando identificada essa causa o DPLL desfaz todas as valorações até o ponto determinado como o causador do erro.

A proposta do GRASP realmente era boa, mas com sua implementação ainda clássica do BCP, fez com que os resolvedores da época, como o SATO, ainda tivessem resultados melhores. Foi então que uma nova implementação feita em 2001, denominada ZCHAFF, conseguiu unir o que existia de melhor nas pesquisas em BCP, escolha de variável e retrocesso não-cronológico. A maior contribuição do ZCHAFF foi a otimização feita no BCP, onde as cláusulas passam a ser verificadas pelo BCP apenas quando entram em um momento crítico de valoração, ou seja, quando apenas um literal da cláusula ainda não foi valorado e todos os outros possuem valor verdade falso, se a cláusula não está nesse estado então o BCP, grosso modo, não irá gastar processamento verificando se a cláusula ficou vazia.

Nesse capítulo apresentaremos a evolução dos resolvedores que utilizam o algoritmo DPLL como base.

3.1 O algoritmo DPLL

Dada uma fórmula ϕ e seu conjunto de cláusulas ψ. O Algoritmo DPLL consiste das seguintes regras:

Tautologia - Remove de ψ todas as cláusulas que são tautologias. O conjunto de cláusulas resultante é Insatisfatível apenas se ψ também for;

Cláusula Unitária - Se existe alguma cláusula unitária em ψ dada pelo literal ϕ, remova de ψ todas as cláusulas contendo θ. Se ψ for vazia, então ϕ é Satisfatível;

Literal Puro - Um literal L no conjunto de cláusulas ψ é dito puro se e somente se o literal $\neg L$ não aparece em nenhuma cláusula do conjunto ψ. Se L for puro, remova toda cláusula onde L aparece;

Ramificação - Quando nenhuma das regras acima for aplicável, ψ é reescrita na forma: $(A_1 \vee L) \wedge \ldots \wedge (A_m \vee L) \wedge (B_1 \vee \neg L) \wedge \ldots \wedge (B_n \vee \neg L) \wedge R$

Onde A_i, B_i, R são livres de L e $\neg L$, então obtemos os conjuntos $\psi_1 = A_1 \wedge \ldots \wedge A_m \wedge R$ e $\psi_2 = B_1 \wedge \ldots \wedge B_n \wedge R$. ψ é insatisfatível se e somente se ψ_1 e ψ_2 foram insatisfatíveis.

Tendo as 4 regras descritas, percebemos que o ponto crucial está justamente na regra de Ramificação. A regra de Ramificação é o momento em que o algoritmo escolhe uma variável e define um valor. Com essa valoração o algoritmo deverá propagar o efeito resultando nos dois conjuntos de cláusulas, uma definindo a variável escolhida como verdadeira e outro conjunto com ela definida como Falsa.

A implementação computacional do DPLL não possui a regra do Ramificação estritamente como na definição, pois é fácil notar que a cada escolha de variável o conjunto de cláusulas dobra de tamanho e não haveria memória suficiente para guardar a grande quantidade de cláusulas em fórmulas muito grandes. Por isso, as implementações são baseadas em um conceito de nível e retrocesso, ou seja, no momento da regra do ramificação é escolhida uma variável e dado um valor (verdadeiro ou falso) e esse momento é guardado como nível de decisão i sendo o nível de decisão de uma variável x denominado por $\delta(x)$. Se a variável recebeu o valor verdadeiro é guardado que no nível de decisão i a variável L foi decidida como verdadeiro e com isso todas as cláusulas que possuem L são marcadas como removidas, e pode ser escrito como $x = v@d$ representando que a variável x recebeu o valor verdade v(verdadeiro) no nível de decisão d.

Quando alguma cláusula se torna Falsa a fórmula em questão também fica falsa, mas ainda não é possível afirmar a sua insatisfatibilidade, pois o espaço de busca ainda não foi completamente esgotado. Nesse momento o algoritmo desfaz o último nível de decisão e faz a troca de valor da variável, se a variável L foi decidida como verdadeiro ela é marcada

como Falsa e o algoritmo continua propagando o valor. Se os dois valores de uma variável já foram testados, o nível anterior sofre a mudança de valor e assim sucessivamente.

```
1  //Procedimento DPLL
2  //Entrada: Fórmula em CNF
3  //Saída: Informação SAT ou INSTATISFATIVEL
4  enquanto 1==1 faça
5  |  se Decide() == OK então
6  |  |  BCP()
7  |  |  se BCPRetornouConflito() == SIM então
8  |  |  |  se Diagnostico() == NaoPodeSerResolvido então
9  |  |  |  |  retorna INSTATISFATIVEL
10 |  |  |  fim
11 |  |  |  senão
12 |  |  |  |  RETROCESSO()
13 |  |  |  fim
14 |  |  fim
15 |  fim
16 |  senão
17 |  |  retorna SAT
18 |  fim
19 fim
```

Algoritmo 1: DPLL

A organização genérica do algoritmo de busca DPLL está representada no algoritmo 1. Esse algoritmo captura a essência dos resolvedores mais competitivos. O algoritmo conduz uma busca pelo espaço das possíveis valorações para as variáveis do problema. Em cada passo da busca um valor verdade é escolhido com a função *Decide*(). E observamos que o nível de decisão d é associado com a variável valorada, representando em qual nível a variável foi valorada.

As implicações de valorações são identificadas pela função $BCP()$ (Boolean Constraint Propagation), que é o momento em que o algoritmo procura alguma cláusula que ficou falsa, ou retorna as implicações causadas pela última decisão. Quando alguma cláusula se torna falsa o $BCP()$ retorna o conflito que é analisado pela função *Diagnostico*(). A função de diagnóstico analisa o conflito e identifica em qual nível de decisão foi cometido um erro, e então um retrocesso é feito, desfazendo todas valorações do nível de decisão atual até o nível marcado como fonte do erro. A maneira que um conflito é analisado será explicada mais adiante no texto. A seguir veremos como as funções *Decide*() e

$BCP()$ foram trabalhadas ao longo dos anos até o modelo utilizado hoje nos principais resolvedores.

A função *Decide*() escolhe uma variável, ainda não valorada, e atribui um valor verdade, enquanto que a função $BCP()$ percorre a fórmula aplicando a decisão feita em *Decide*(). Na descrição das 4 regras do DPLL clássico podemos dizer que o $BCP()$ é a aplicação das regras 1-3 enquanto que a *Decide*() é a regra da ramificação.

O BCP propaga o efeito da valoração feita até o momento, marcando os literais valorados como falso fora da cláusula e marca as cláusulas com pelo menos um literal verdadeiro como verdadeiro, excluindo essas cláusulas do conjunto de cláusulas. Sempre que o BCP é chamado essa verificação é feita em torno da valoração parcial, sendo um dos procedimentos mais importantes e mais custosos do resolvedor, o seu consumo varia entre $80-90\%$ do tempo gasto pelo resolvedor[17]. Vários estudos foram feitos para melhorar o desempenho do BCP além de estudos para a escolha da variável mais adequada para aplicar a regra da ramificação.

Cada função possui a sua equivalência nas regras do DPLL, exceto pela função *Diagnostico()* que nos resolvedores modernos tem o objetivo de analisar o estado da valoração e identificar o motivo de a fórmula estar em um estado que a torne insatisfatível. O maior avanço dessa função foi atingido com o GRASP[16] que conseguiu identificar o motivo, retroceder para a decisão que foi a base para o estado atual de valoração e ainda adicionar ao conjunto de cláusulas uma nova cláusula que representa o conjunto de literais que devem sempre ficar verdadeiro. O processo de analisar, diagnosticar e adicionar uma nova cláusula será apresentado na seção 3.4.

A ordem de escolha de variáveis para aplicar a regra da ramificação pode melhorar significativamente o tempo de execução do resolvedor SAT, pois se uma fórmula for satisfatível e a cada passo do algoritmo a variável escolhida para a ramificação tiver seu valor verdade marcado tal como deve ser na linha da tabela verdade que torne a fórmula verdadeira, o número de decisões e retrocessos feitos durante a execução do resolvedor pode ser muito menor que escolher uma variável qualquer.

Vários estudos foram feitos no sentido de se encontrar a melhor heurísticas de escolha de variável, mas essa escolha depende muito de como a fórmula foi gerada e mais ainda da instância do problema. Por isso é muito difícil de afirmar o que faz com que uma heurística seja um boa estratégia em relação a outras. Em 1999, João Marques-Silva publicou um estudo com algumas heurísticas e seus resultados empíricos com diversas fórmulas e resolvedores SAT. A seguir veremos algumas heurísticas.

3.2 Heurísticas para ramificação

A estratégia mais simples de escolha de variável está na escolha aleatória de uma variável ainda não valorada, essa estratégia é denominada RAND. Testes empíricos mostram que a utilização da estratégia RAND pode ser tão eficiente quanto a mais sofisticada técnica de escolha.

As heurísticas que conseguiram ter resultado melhor que o RAND na maioria dos testes levam em consideração informação dinâmica fornecida pelo BCP. O BCP repassa para a máquina de decisão essas informações. Dentre as heurísticas temos a BOHM[4, 15], MOM[9, 28, 15] e VSIDS[17].

A heurística BOHM (o nome vem do autor) era bastante competitiva para instâncias aleatórias em 1992 como foi descrito em [4] e a ideia é a de escolher a variável com maior vetor $(H_1(x), H_2(x), ..., H_n(x))$ em ordem lexicográfica onde $H_i(x)$ é computado da seguinte maneira:

$$H_i(x) = \alpha \times max(h_i(x), h_i(\neg x)) + \beta min(h_i(x), h_i(\neg x))$$

onde $h_i(x)$ representa o número de cláusulas não resolvidas com i literais que possuam o literal x. Com isso cada literal selecionado dá preferência em satisfazer cláusulas pequenas (quando valoradas como verdadeiras) ou reduzir o tamanho das cláusulas pequenas (quando valoradas como falsas). Os valores de α e β são dependentes da implementação da heurística, no caso de [4] os valores sugeridos são $\alpha = 1$ e $\beta = 2$.

Maximum Occurrences on clauses of Minimum size (MOM) é a heurística que, pelo

nome, busca pelo maior número de ocorrências de um literal nas menores cláusulas presentes na fórmula. Para isso temos $f^*(l)$ o número de ocorrências de um literal l na menor cláusula ainda não satisfeita. Uma boa escolha de variável a ser selecionada é uma que maximiza a função:

$$(f^*(x) + f^*(\neg x)) \times 2^k + f^*(x) \times f^*(\neg x)$$

Intuitivamente a preferência é dada para variáveis x que tenham a maior quantidade de ocorrência dos literais x ou $\neg x$ (considerando que k possua um valor suficientemente alto), e também para variáveis com várias ocorrências de seus literais x e $\neg x$.

Jeroslow e Wang em [14] propuseram duas heurísticas, chamadas *one-sided Jeroslow-Wang* (JW-OS) e *two-sided Jeroslow-Wang* (JW-TS). Para um dado literal l compute:

$$J(l) = \sum_{l \in w \wedge w \in \phi} 2^{-|w|}$$

A JW-OS valora o literal l que possui o maior $J(l)$, enquanto que a JW-TS identifica a variável x com a maior soma $J(x) + J(\neg x)$ e valora x como verdadeira se $J(x) >= J(\neg x)$, e falso caso contrário.

Mesmo tendo um grande número de heurísticas para tentar escolher a melhor variável, é importante saber como avaliar e escolher a melhor. Alguns podem considerar a melhor escolha aquela que influencia no número de decisões que o algoritmo de busca irá fazer, pois menos decisões indicam que as tomadas foram mais inteligentes. O problema é que cada decisão influencia o trabalho do BCP de forma diferente, ou seja, uma sequência de decisões menor pode causar mais operações no BCP do que uma sequência maior. E ainda o custo computacional de cada heurística de escolha de variável é diferente, a melhor decisão a ser feita pode custar muito computacionalmente para ser calculada, ao passo que uma heurística simples como a RAND não consome quase processamento e pode fazer algumas escolhas boas, e no tempo final o resolvedor SAT pode ser menor. O fato é que não existe uma resposta clara na literatura que indique uma estratégia que seja boa em qualquer caso.

O grupo que desenvolveu o resolvedor ZCHAFF apresentou uma nova estratégia denominada VSIDS (*Variable State Independent Decaying Sum*) que conseguiu atuar de forma melhor que várias outras estratégias pensadas anteriormente, e funciona da seguinte maneira:

1. todo literal tem um contador iniciado como 0;
2. quando uma cláusula nova é adicionada, o contador associado com cada literal presente na cláusula é incrementado;
3. a variável (não valorada) e seu literal com o contador mais alto é escolhida para a decisão;
4. periodicamente todos os contadores são divididos por uma constante.

A implementação dessa heurística conta com uma lista das variáveis não valoradas ordenadas pelo valor do contador que é atualizado durante o BCP e na análise de conflitos. Dessa forma a escolha de variável é feita de modo muito rápido no momento da decisão.

O MINISAT utiliza uma abordagem semelhante, porém não distingue os literais de uma variável. A contagem é feita apenas pela variável (que é a soma dos literais). Após gravar o conflito, o contador de todas as variáveis é multiplicado por um valor menor que 1, diminuindo a atividade das variáveis com o tempo.

O ponto chave dessa estratégia é que ela exige pouco processamento pois é atualizada apenas quando uma nova cláusula é adicionada decorrente de um conflito. Os autores do ZCHAFF afirmam que essa estratégia melhorou o resolvedor em uma ordem de magnitude com problemas difíceis e não prejudicou em problemas mais simples, o que pode ser visto como uma métrica para afirmar o seu sucesso.

3.3 BCP rápido

Motivados pela grande parcela de processamento do BCP os autores do ZCHAFF decidiram otimizá-lo e perceberam que uma cláusula entra em estado crítico de valoração quando todos os literais foram valorados como falso e resta apenas um literal não valorado.

Quando a cláusula entra nesse estado crítico o literal não valorado pode ser inferido facilmente, ou seja, deve ser valorado como verdadeiro para que a fórmula não fique com uma valoração insatisfatível.

Para determinar se uma cláusula está no estado crítico, o ZCHAFF então implementa a seguinte ideia: cada cláusula possuirá uma lista de observação de dois literais, e quando uma variável for valorada apenas as cláusulas que possuem o literal da variável na lista de observação serão verificados. Se o literal for falso, a lista de observação mudará fazendo com que o literal valorado como falso deixe de ser observado e será trocado por um outro literal qualquer que ainda não tenha sido valorado na cláusula. Quando não existe um literal a ser escolhido o literal restante é inferido como verdadeiro.

Quando um literal é inferido a máquina de busca não muda de nível de decisão, pois é uma inferência feita pela decisão tomada no nível, e propaga o efeito até que não existam mais inferências.

3.4 Aprendizado de cláusulas e retrocesso não-cronológico

A contribuição mais relevante para a resolução de satisfatibilidade foi com a introdução do algoritmo GRASP[16] , onde o retrocesso deixou de ser apenas com a troca do valor verdade da última decisão e passou a ser inteligente, analisando o motivo da cláusula se tornar falsa e voltar todos os níveis de decisão até a causa do erro. Esse processo é chamado de retrocesso não-cronológico pois o resolvedor deixa de voltar apenas um nível de decisão e passa a voltar vários. O GRASP ainda permite que o sistema aprenda o motivo do erro evitando, assim, a recorrência. Desse modo dizemos que o GRASP contribuiu com a análise de conflito e com o aprendizado de cláusulas.

Todos os resolvedores da atualidade fazem uso das técnicas que o GRASP introduziu, apenas variando o modo que guardar as cláusulas aprendidas ou outras pequenas modificações. Veremos agora como o GRASP permite esse retrocesso não-cronológico e como é feito o aprendizado.

A principal modificação do GRASP no algoritmo clássico do DPLL, mostrado no Algoritmo 1, está na função *Diagnostico*(). Quando o BCP retorna conflito é feita uma

análise da causa. Essa análise é feita com a criação de um grafo dirigido de implicação I definido da seguinte forma:

1. cada vértice de I corresponde à valoração de uma variável $x : v(x)$;

2. os predecessores de um vértice $x : v(x)$ em I são valorações antecedentes $A^w(x)$ correspondente a cláusula unitária w que causou a implicação de x. As arestas dirigidas partindo do vértice $A^w(x)$ para o vértice $x : v(x)$ são nomeados com w. Vértices sem antecedentes correspondem à decisões;

3. vértices especiais de conflitos são adicionados em I para indicar a ocorrência de conflito. Os predecessores de um vértice de conflito K correspondem a valoração de variáveis que forçaram a cláusula w a ficar Falsa e são vistos como a valoração antecedente de $A^w(x)$. As arestas partindo dos vértices $A^w(x)$ para K são nomeados como w.

O nível de decisão de uma variável implicada x é relacionado com os seus antecedentes de acordo com:

$$\delta(x) = max((\delta y|(y, v(y)) \in A^w(x))$$

Com o grafo sendo atualizado a cada decisão ou implicação, temos que identificar uma valoração de conflito e efetuar alguma ação. Quando o BCP retorna um conflito a sequência de implicações aparece convergindo para um vértice K e esse caminho é analisado para determinar o responsável pelo conflito. A conjunção das valorações conflitantes é um implicante suficiente para o conflito aparecer. A negação desse implicante gera uma função booleana f (cuja satisfatibilidade é procurada) que não existe no conjunto de cláusulas. Esse implicante é denominado cláusula de conflito induzido e provê o mecanismo para a implementação de um retrocesso não cronológico.

Quando o conflito aparece na fórmula é verificado se a decisão feita no nível de decisão corrente está presente, se estiver o seu valor é trocado e nesse momento a variável deixa de ser uma decisão e a implicação dela está nas outras decisões presentes na cláusula de conflito induzido. E o processo continua.

Quando as variáveis presentes na cláusula de conflito induzido são todas de um nível mais baixo que o nível corrente então o algoritmo escolhe a que foi decidida no nível mais alto dentro da cláusula de conflito induzido. Quando o maior nível é $d-1$ (sendo d o nível corrente) o algoritmo faz um retrocesso cronológico, quando for menor que $d-1$ o algoritmo então está fazendo um retrocesso não-cronológico.

Para evitar que o mesmo erro seja feito várias vezes durante o processo de busca, sempre que o retrocesso for feito a cláusula de conflito induzido é adicionada ao conjunto de clásulas. Esse processo é denominado aprendizado de cláusula.

Exemplificando o processo, tomemos a fórmula:

$$(\neg x_1 \vee x_2 \vee x_4) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_5 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_6 \vee x_5)$$

E tomemos a seguinte valoração conflitante:

x_6 verdadeira no nível 1 (v@1);

x_5 verdadeira no nível 1 (v@1);

x_4 falsa no nível 2 (f@2);

x_1 verdadeira no nível 3 (v@3);

x_2 verdadeira no nível 3 (v@3);

x_3 verdadeira e falsa no nível 3 (*conflito*);

O resolvedor fará uma busca no grafo a partir dos nós que indicam x_3 e retrocederão até que seja encontrada um nó que não foi implicação do nível de decisão corrente. Na figura 3.1 podemos identificar um corte no grafo que representa o local onde podemos extrair os componentes da cláusula de conflito induzido que envolvem as variáveis x_2 e x_5. Portanto a cláusula aprendida é negação das valorações das variáveis aprendidas, ficando: $(\neg x_2 \vee \neg x_5)$. Outra cláusula aprendida poderia ser $(x_4 \vee \neg x_1 \vee \neg x_6)$, porém essa última é menos sucinta que a primeira pois x_2 gera implicação em x_4 e x_1.

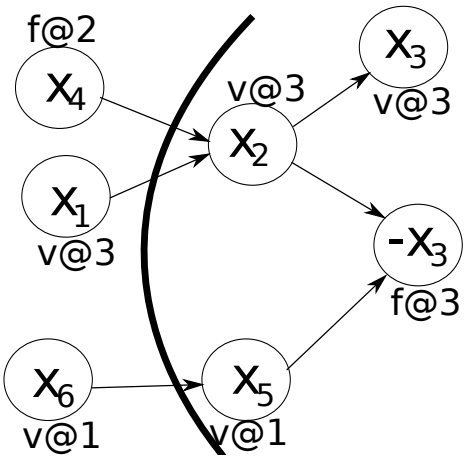

Figura 3.1: Grafo de decisão com conflito na variável x_3

3.5 Outras otimizações

Além das técnicas apresentadas existem ainda mais dois procedimentos que podem ser usados para evitar que o resolvedor fique em uma busca sem progresso substancial e ainda um limpador de cláusulas aprendidas que não possuem utilidade e apenas acrescentam uma perda de desempenho para o BCP e para o consumo de memória.

Os resolvedores que implementam o aprendizado de conflitos passaram a efetuar reinícios automáticos do processo de busca: periodicamente após uma quantidade definida de decisões ou quantidade de cláusulas aprendidas o algoritmo reinicia toda a busca guardando apenas o conjunto de cláusulas aprendidas.

O reinício é feito para evitar que a busca fique presa em um sub-espaço onde nenhum progresso substancial possa ser feito. Experimentos com várias heurísticas para se determinar o momento de fazer o reinício pode ser visto em[10].

Muitos resolvedores colocam um limite em relação a quantidade de cláusulas que são mantidas, para evitar uma explosão no uso de memória. As técnicas variam entre as implementações; alguns resolvedores removem cláusulas apenas durante os reinícios, outros mesclam algumas cláusulas ou tentam identificar quais cláusulas são subconjuntos de outras e guardam apenas uma delas.

O ZCHAFF, ao adicionar uma cláusula, faz uma análise para determinar em que ponto no futuro ela pode ser removida. A métrica usada é a relevância, ou seja, quando mais que

N (onde N varia entre 100 e 200) literais na cláusula ficam sem valoração pela primeira vez, a cláusula será marcada para remoção.

3.6 Considerações finais

Ao longo do capítulo vimos as principais estratégias para melhorar o desempenho na execução de fórmulas grandes. Hoje é impossível identificar um resolvedor moderno com bons resultados sem a implementação de estruturas otimizadas para efetuar um BCP rápido, principalmente sem um sistema de aprendizado de conflitos que permita um retrocesso não-cronológico.

Todas essas estratégias implementadas nos resolvedores, causaram um enorme impacto no tempo e tamanho das fórmulas resolvidas. Cada estratégia implementada separadamente agrega alguma melhoria no tempo do relógio, mas a implementação de todas as estratégias - BCP rápido; aprendizado de cláusulas; retrocesso não cronológico e; heurística VSIDS na decisão - unidas apresentam o grande potencial de cada uma delas.

A implementação de todas as regras reflete em um resolvedor mais rápido e com capacidade de resolver problemas complexos em tempos aceitáveis, como exemplo de desempenho, um simples DPLL consegue resolver fórmulas, aleatórias, de até 100 variáveis e 430 cláusulas ao passo que o mais moderno resolvedor consegue resolver fórmulas industriais com mais de 1 milhão de variáveis e mais de 5 milhões de cláusulas.

CAPÍTULO 4

SATISFATIBILIDADE EM FÓRMULAS NÃO-CLAUSAIS

A maior parte dos resolvedores SAT atuais precisa que a fórmula esteja em formato CNF, porém os problemas do mundo real mapeados em lógica dificilmente estarão em CNF. Em geral podem ser representados em um formato não-clausal, ou seja, em um formato onde operadores podem estar em qualquer posição na fórmula tendo, inclusive, um operador como filho de outro operador e negações em operadores. Para permitir que os resolvedores SAT resolvam essas fórmulas é necessário que uma conversão seja feita afim de transformar essas fórmulas em CNF. O processo de conversão introduz novas variáveis e pode "perder" a noção estrutural da fórmula original, podendo impactar de forma negativa na eficiência da verificação de satisfatibilidade.

Alguns resolvedores podem ser considerados o estado da arte em resolvedores não-clausais, o NOCLAUSE[23] e o NFLSAT[12]. Veremos nas seções seguintes a estrutura desses dois resolvedores.

Neste capítulo iremos considerar que fórmula Booleanas possuem apenas os operadores $\wedge$ (E), $\vee$ (OU) e $\neg$ (NÃO).

4.1 Satisfatibilidade em NNF

Para resolver satisfatibilidade em uma fórmula em NNF (seção 2.2.3) o método utilizado é por um diagrama bi-dimensional que foi chamado de *General Matings*[2] definido inicialmente para resolver fórmulas em CNF. Anos mais tarde o modelo foi apresentado para fórmulas em NNF[3]. Seguindo a linha de utilizar *General Matings* para resolver satisfatibilidade em NNF temos dois resolvedores: o SATMATE[11] e um aperfeiçoamento desse, o NFLSAT[12], desenvolvidos pelo mesmo autor, sendo o NFLSAT uma melhoria em cima do SATMATE.

A representação interna utiliza um formato bi-dimensional da fórmula NNF, chamada

caminho vertical-horizontal (vhpform, *vertical-horizontal path form*) como está descrito em [11]. Nessa forma as disjunções ($\vee$) são escritas horizontalmente e as conjunções ($\wedge$) verticalmente. Uma valoração para a fórmula em NNF é satisfatível se e somente se existe um caminho vertical na *vhpform* tal que a valoração satisfaz cada literal no caminho. Uma valoração para a fórmula em NNF é insatisfatível se existe um caminho horizontal na *vhpform* tal que a valoração não satisfaz cada literal no caminho.

A figura 4.1 mostra a representação da fórmula $(((p \vee q) \wedge \neg r \wedge \neg q) \vee (\neg p \wedge (r \vee \neg s) \wedge q))$.

$$\left[\begin{bmatrix} p \vee q \\ \neg r \\ \neg q \end{bmatrix} \vee \begin{bmatrix} \neg p \\ r \vee \neg s \\ q \end{bmatrix} \right]$$

Figura 4.1: *vhpform* para a fórmula $(((p \vee q) \wedge \neg r \wedge \neg q) \vee (\neg p \wedge (r \vee \neg s) \wedge q))$

Caminho Vertical : Um caminho vertical na *vhpform* é a sequência de literais na *vhpform* que resulta pela escolha do filho esquerdo ou direito de cada disjunção ($\vee$). Para a *vhpform* na figura 4.1 os caminhos verticais são $\{\langle p, \neg r, \neg q\rangle$, $\langle q, \neg r, \neg q\rangle, \langle \neg p, r, q\rangle, \langle \neg p, \neg s, q\rangle\}$;

Caminho Horizontal : Um caminho horizontal na *vhpform* é a sequência de literais na *vhpform* que resulta pela escolha do filho esquerdo ou direito de cada conjunção ($\wedge$). Para a *vhpform* na figura 4.1 os caminhos horizontais são $\{\langle p, q, \neg p\rangle$, $\langle p, q, r, \neg s\rangle, \langle p, q, q\rangle, \langle \neg r, \neg p\rangle, \langle \neg r, r, \neg s\rangle, \langle \neg r, q\rangle, \langle \neg q, \neg p\rangle, \langle \neg q, r, \neg s\rangle, \langle \neg q, q\rangle\}$

Os caminhos verticais e horizontais podem ser dispostos mais explicitamente como um grafo direcionado acíclico, onde os nós são os literais e os vértices representam as conexões da fórmula na representação bi-dimensional. O processo de construção do grafo vertical consiste em construir grafos para as sub-fórmulas, partindo dos literais mais internos e criando os vértices de forma apropriada para cada operador:

Para disjunções: uma união dos grafos, representando as duas sub-fórmulas. Visualmente, corresponde a colocar os dois grafos um ao lado do outro.

Para conjunções: uma concatenação dos grafos é feita. Visualmente é o mesmo que colocar um grafo em cima do outro.

O processo é não determinístico no sentido de que os operadores $\wedge$ e $\vee$ podem ser pegos em uma ordem qualquer, resultando em grafos diferentes, ficando dependente da implementação. A figura 4.2 exemplifica um grafo vertical gerado da fórmula descrita na figura 4.1.

Para criar o grafo horizontal o processo é bem semelhante ao do grafo vertical, mas os grafos são concatenados em disjunções e unidos em conjunções. A figura 4.3 exemplifica um grafo vertical gerado da fórmula descrita na figura 4.1.

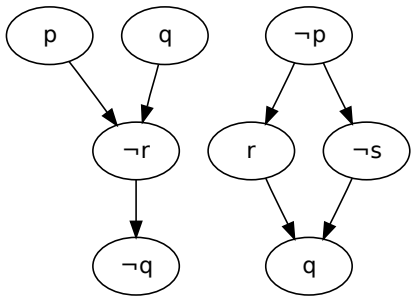

Figura 4.2: Representação do Grafo Vertical da fórmula na figura 4.1

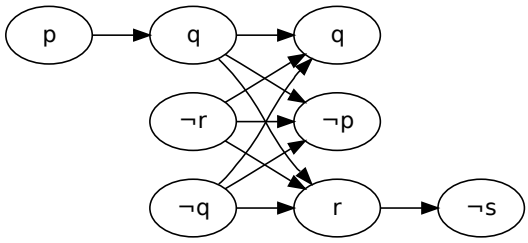

Figura 4.3: Representação do Grafo Horizontal da fórmula na figura 4.1

Andrews [3] mostrou que, para qualquer fórmula booleana, escrever todos os caminhos verticais em forma de conjunção resulta em uma fórmula na forma normal disjuntiva. Ao

passo que podemos escrever os caminhos horizontais e obter a fórmula em sua forma normal conjuntiva.

Observando a figura 4.2 é fácil perceber que basta encontrar um caminho no grafo vertical onde todos os nós possuem o valor verdade Verdadeiro para encontrar a valoração satisfatível para a fórmula. O problema desse método está na possibilidade da existência de um número exponencial na quantidade de caminhos verticais em relação ao tamanho da fórmula.

Para evitar que o algoritmo fique andando pelo grafo para buscar um caminho satisfatível, técnicas foram trazidas de algoritmos modernos para CNF que são o BCP e a observação de literais.

4.1.1 Aplicação de estratégia de resolvedores baseados em DPLL em fórmulas NNF

Definimos como *corte* um conjunto de nós do grafo vertical tal que se removidos do grafo horizontal, este ficará com todos os caminhos desconectados. Um corte da figura 4.1 é o caminho vertical $\langle \neg p, r, q \rangle$, pois ao se remover esses três nós do grafo horizontal, todos os caminhos ficarão desconectados. Dada uma valoração parcial da variáveis, um corte é *aceitável* quando nenhum dos literais possui valor verdade falso. Cortes podem possuir um ou mais nós em comum com outro corte.

Uma valoração parcial das variáveis é conflitante quando nenhum corte aceitável pode ser encontrado. Quando isso ocorre o estado da fórmula se encontra em uma situação onde a valoração corrente falsifica todos os caminhos do grafo vertical.

Quando um conflito acontece, o grafo horizontal é examinado. Os caminhos horizontais codificam uma representação de cláusulas em CNF da fórmula. Para uma valoração conflitante, pelo menos um caminho horizontal deve ser completamente falso e ele deve ser encontrado. O caminho horizontal totalmente falso atua diretamente na causa do conflito, e o retrocesso é feito da mesma maneira que os algoritmos baseados em DPLL do capítulo 3.

Durante a execução do resolvedor dois cortes disjuntos são mantidos, e são cortes

disjuntos, ou seja, não possuem nós em comum. Ao longo da busca o algoritmo tenta manter pelo menos um corte aceitável e ainda tenta manter os cortes o mais disjuntos possível. Esses cortes se equiparam ao sitema de observação de literais em cláusulas no algoritmos baseados em DPLL, pois repensando a fórmula CNF no grafo vertical a lista de observação de todas as cláusulas são dois cortes no grafo, da mesma forma em que são mantidos os cortes nessa abordagem não-clausal.

4.2 NOCLAUSE

Ao contrário do NFLSAT e SATMANT, que transformam a fórmula em uma forma vhp, o resolvedor NOCLAUSE [23] atua diretamente na fórmula representada em ISCAS, que como vimos, é representada por um DAG. Para o NOCLAUSE, cada nó do DAG é associado com a seguinte estrutura:

1. identificador único;
2. uma lista de pais;
3. uma lista de filhos;
4. o tipo do nó (variável, $\wedge$, $\vee$, $\neg$);
5. valoração (verdadeiro, falso, irrelevante, desconhecido);
6. nível de decisão no qual o nó foi valorado;
7. razão da valoração do nó.

Com a fórmula já representada na memória, o objetivo do resolvedor agora é valorar os nós do DAG de uma forma consistente até que o nó raiz seja valorado como verdadeiro. Uma valoração é consistente quando a lógica dos operadores é respeitada, por exemplo, se um nó operador $\wedge$ for valorado como verdadeiro todos os seus filhos serão valorados como verdadeiro.

O procedimento de busca do NOCLAUSE escolhe um nó qualquer do DAG e o valora como verdadeiro ou falso, propaga a consequência dessa valoração, e, recursivamente tenta

valorar os nós ainda não valorados. A escolha de um nó interno se reflete na valoração de uma sub-fórmula.

Quando um nó é valorado, as consequências dessa valoração são propagadas pelo DAG, por exemplo, se um nó é valorado falso então falso é propagado para todos os pais que são nós operadores E; se for valorado como verdadeiro e esse nó for um operador E, então verdadeiro é propagado para todos os seus filhos e assim por diante. As propagações sobem e descem no DAG guiados por regras simples de propagação, como veremos mais adiante.

Uma contradição é detectada quando um nó tenta ser valorado como verdadeiro e falso. Quando a contradição é identificada, o processo deve fazer um retrocesso para tentar uma outra valoração, da mesma forma como ocorre com os procedimentos baseados em DPLL para fórmulas em CNF.

O NOCLAUSE possui uma propagação eficiente pois adota a ideia da lista de observação como é feita nos resolvedores em CNF. As observações são usadas onde elas podem auxiliar na eficiência da regra de propagação. São definidas quatro regras para propagar operadores $\wedge$:

1. se um operador $\wedge$ se torna verdadeiro, propaga verdadeiro para todos filhos;
2. se um filho torna falso, propaga falso para todos os pais $\wedge$;
3. se todos os filhos se tornam verdadeiro, propaga verdadeiro para o $\wedge$ pai;
4. se um operador $\wedge$ se torna falso e todos menos um filho é verdadeiro, propaga falso para o filho sem valoração.

A lista de observações não ajuda nas duas primeiras regras. A primeira porque é dependente apenas do operador. A segunda pois é implementada com a divisão da lista de pais de um nó baseadas no tipo do pai. Então existe uma lista separada para os pais $\wedge$, outra para os $\vee$, e assim por diante, dessa forma é eficiente propagar falso para cada um dos pais $\wedge$ de um nó qualquer.

As listas de observação melhoram a eficiência justamente das regras 3 e 4. Para cada operador $\wedge$ dois filhos são marcados como observação de verdadeiro, e para nó é mantido

uma lista de pais que os observa. Como a lista é para um operador E, o filhos que causam alguma diferença são justamente os que são valorados como falso. Sempre que um filho for valorado como verdadeiro o filho observado é trocado por outro não valorado, mas só existe essa troca se a outra observação já não for Falsa. Quando a outra observação for verdadeira significa que todos os outros filhos desse operador já são verdadeiros e a terceira regra de propagação já pode ser ativada. Se um outro nó não valorado não existir para o operador, então nada é feito e o nó recém valorado continua sendo observado, ficando uma observação sem valoração e outra valorada, também é verificado se o operador $\wedge$ já não foi valorado como falso. Se a regra quatro for ativada e propaga falso para o único filho não valorado. Por fim, sempre que um nó $\wedge$ for valorado falso, olhamos para as suas duas observações. Se uma delas for verdadeira, sabemos que a outra observação é o único filho desse operador ainda não valorado, e a regra quatro é ativada propagando falso para esse filho.

4.3 Considerações finais

Ao longo deste capítulo identificamos duas abordagens para se resolver fórmulas codificadas em uma representação não-clausal sem que a transformação de Tseitin seja utilizada, ou seja, atuando na fórmula em uma representação não-clausal.

Primeiramente vimos a abordagem em NNF que utiliza uma representação bi-dimensional denominada *General Matings*, essa representação permite codificar fórmulas tanto em CNF como NNF e um procedimento levemente diferente do DPLL pode ser aplicado. Como a representação pode ser feita em ambas representações, as estratégias dos resolvedores em CNF podem ser avaliadas e trazidas para fórmulas em NNF sem modificações, isto é, desde que as estratégias em CNF caibam dentro do escopo de *General Matings*.

A segunda abordagem foi a implementação de um outro resolvedor baseado na ideia do DPLL para CNF. Esse método utiliza um DAG bem menos restritivo que a NNF, no sentido de poder existir a negação em qualquer nó da árvore, contrapondo a NNF onde a negação é aplicada apenas nas variáveis.

A abordagem com ISCAS desenha o grafo da mesma forma que está descrito no ar-

quivo, sem modificação, e aplica regras de propagação da mesma forma que os resolvedores em CNF, a diferença está nas regras que cada nó pode possuir para propagar seu valor, e ainda a técnica de observação é transportada para esse modelo de uma forma elegante e adaptável para os operadores existentes na fórmula.

CAPÍTULO 5

LIAMFSAT

Este capítulo apresenta um resolvedor para satisfatibilidade que além de atuar diretamente na fórmula original, implementa todas as técnicas dos resolvedores modernos, tais como: retrocesso não-cronológico; aprendizado de cláusula; BCP rápido; e escolha de variável bem definida. O formato escolhido foi o ISCAS, pela sua grande facilidade em representar qualquer fórmula lógica. O formato ISCAS é apresentado na seção 2.2.4.

Esse novo resolvedor foi carinhosamente batizado de LIAMFSAT. O nome é a união das siglas do Laboratório de Inteligêngia Artifical e Métodos Formais (LIAMF) da Universidade Federal do Paraná (UFPR) com Satisfatibilidade (SAT).

5.1 Estrutura de dados

O LIAMFSAT faz a leitura do arquivo contendo a fórmula em ISCAS e gera um DAG no que cada nó do grafo que representa a fórmula é dado pelos seguintes atributos:

- identificador numérico único;
- o nome original do nó na fórmula;
- o tipo do nó (Variável, $\wedge, \vee, \neg$);
- o valor verdade (que pode ser: verdadeiro, falso, não-valorado);
- a lista de nós filhos;
- a lista de nós pais;
- o nível de decisão em que foi valorado;
- a causa da valoração (decisão ou implicação).

A partir da estrutura da fórmula representada na memória o resolvedor procura por uma valoração para as variáveis da fórmula que torne o nó raiz da estrutura verdadeiro.

5.2 Processo de busca

O processo de busca é semelhante ao utilizado pelos resolvedores baseados no DPLL. Nos algoritmos baseados em CNF já é definido que todos os operadores devem possuir uma valoração verdadeira e para garantir essa valoração as variáveis são valoradas para manter essa propriedade das cláusulas, pois queremos determinar se a fórmula é satisfatível, e isso é definido na estrutura do grafo, que é uma conjunção de várias cláusulas.

A busca por satisfatibilidade em fórmulas não-clausais não parte da premissa de que todos os operadores devem ser valorados como verdadeiro , pois pode ser necessário que algum operador seja valorado como falso. O LIAMFSAT utiliza os operadores apenas como guia de valorações parciais das variáveis, ou seja, a decisão acontece apenas nas variáveis.

Antes de começar o processo de busca na fórmula, o nó raiz é valorado como verdadeiro. Depois de definir a raiz da fórmula como verdadeira, uma variável qualquer é valorada como verdadeira. É feita a análise do efeito dessa valoração e o processo continua até que seja encontrado uma valoração que torne a fórmula verdadeira ou até que um conflito seja alcançado.

A análise do efeito da valoração de uma variável ocorre de forma semelhante a outros resolvedores, porém como temos a restrição de decidir a valoração apenas das variáveis, podemos garantir que a propagação ocorra em apenas uma direção, ou seja, a propagação vai de filho para pai e nunca de pai para filho. Isto permite que cada nó acumule todas as modificações indicadas por seus nós filhos. A propagação das modificações segue um modelo de sinais que será detalhado mais adiante.

A idéia de garantir que cada nó será visitado apenas uma vez para executar os sinais pendentes. Para garantir que uma única verificação seja suficiente, o DAG tem que ser reordenado a partir de uma ordem de verificação. Todas as variáveis já são marcadas como nível 0 e os nós internos possuem um nível dado pela fórmula:

$$Nivel(G) = max_{i \in filho(G)}(Nivel(i)) + 1$$

Como exemplo diagramado do nivelamento, aplicamos o nivelamento para a fórmula $((p \vee q) \wedge (r \vee \neg s) \wedge \neg q) \vee \neg(\neg p \wedge (r \vee \neg s) \wedge q)$, que foi diagramada na figura 2.3. O diagrama da fórmula nivela pode ser visto abaixo, na figura 5.1.

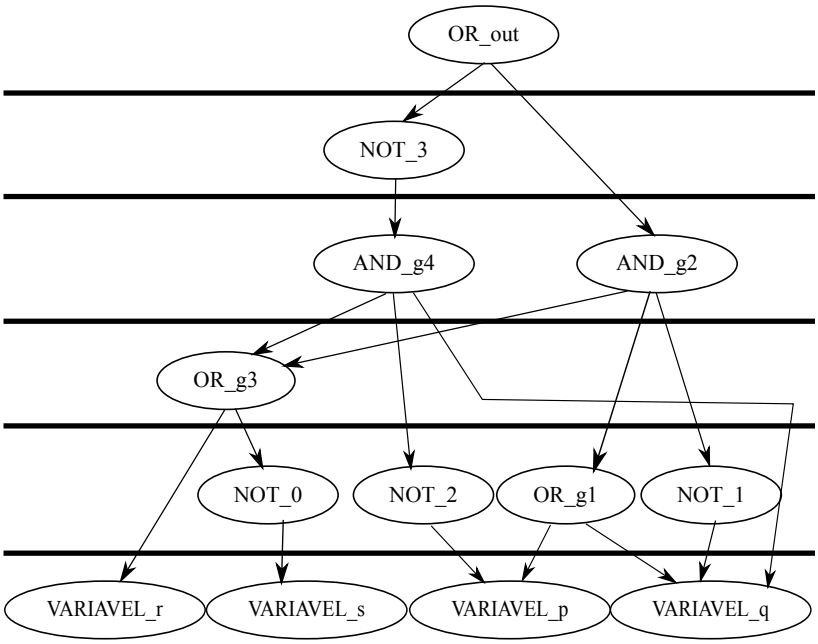

Figura 5.1: Diagrama da fórmula $((p \vee q) \wedge (r \vee \neg s) \wedge \neg q) \vee \neg(\neg p \wedge (r \vee \neg s) \wedge q)$ com o nivelamento aplicado

A propagação dos sinais no DAG é feita de forma simples, tal que cada nó cuida dos seus próprios sinais da seguinte forma:

- se todos os filhos de um operador $\wedge$ são marcados como verdadeiro, o nó se torna verdadeiro e envia um sinal de verdadeiro para os pais;
- se qualquer filho de um operador $\wedge$ for falso, o nó se torna falso e é enviado um sinal falso para todos os pais;

Quando o DAG está nivelado pelo processo de reordenação podemos garantir que é possível verificar os sinais pendentes apenas uma vez por nó, então o algoritmo verifica os sinais por nível em ordem crescente até chegar na raiz do DAG. Para garantir que não

seja feita uma verificação para cada nó em busca de sinais pendentes, existe uma fila de sinais por nível, onde cada nó que possui sinais pendentes é colocado.

Sempre que um nó mudar a valoração de desconhecido para verdadeiro ou falso, um sinal é enviado para todos os pais do nó. Cada operador possui uma estrutura de acumulação de sinais pendentes e é nessa estrutura que os sinais enviados são guardados. O BCP então busca em cada nível os operadores com sinais pendentes e verifica se os sinais forçam a propagação de valoração. Quando o operador fica valorado novos sinais são enviados para os pais, e o BCP continua o processo até que não existam mais sinais pendentes.

Esse processo de não valorar os nós internos do DAG está relacionado com a dificuldade de se saber qual seria o valor verdade ideal para os nós internos, pois diferentemente do CNF existem operadores que devem ser falsos para que a a raiz se torne verdadeira.

O modelo de efetuar decisão apenas em variáveis significa que o motivo de valoração das variáveis será sempre marcado como decisão. Com isso o momento crítico no estado de valoração será apenas quando a raiz se tornar falsa, e então um retrocesso será feito.

A causa de valoração de um operador será sempre a inferência, e com isso o nível de decisão será dado pelo operador e pelo valor associado:

(∧, verdadeiro) : nível de decisão de todos os filhos;

(∧ falso) : nível de decisão do filho falso de menor nível de decisão;

(∨ verdadeiro) : nível de decisão do filho verdadeiro de menor nível de decisão;

(∨ falso) : nível de decisão de todos os filhos.

Sempre que os filhos da raiz da fórmula forçarem uma propagação de valor falsa o processo de busca é interrompido e a fórmula entra em uma valoração conflitante, pois a raiz já foi valorada como verdadeira. Mas os filhos estão forçando uma valoração falsa para o nó. Neste é iniciada a "análise de conflito".

Conforme explicado, a análise de conflito tem como objetivo identificar o erro cometido durante o processo de busca ao valorar alguma variável. Esse processo primeiramente

verifica se a última decisão já foi testada tanto para verdadeiro como para falso. Quando algum dos valores não foi testado, o algoritmo troca o valor da última decisão e continua o processo de busca. Quando ambos valores foram testados então é procurado o maior nível de decisão entre os filhos que ajudaram com a valoração falsa da raiz, ou seja, quando a raiz for um operador $\wedge$ é identificado o maior nível de decisão entre os filhos falsos da raiz.

Identificando a causa da valoração inconsistente um retrocesso é feito até o nível problemático, com a mudança do valor verdade. Quando o retrocesso chega até o primeiro nível de decisão e esta decisão já foi testada com os dois valores verdade, então o algoritmo para e a fórmula é insatisfatível.

Além de analisar e identificar o nível de decisão onde se deve efetuar o retrocesso, também é feito o aprendizado de uma cláusula para evitar que o mesmo erro seja cometido diversas vezes. A cláusula é gerada a partir de uma busca no DAG pelos filhos da raiz que tornaram a valoração conflitante. A busca percorre o DAG até que o nível de decisão dos nós seja diferente do nível atual, as variáveis decididas nesses níveis definem a cláusula que representa o conflito.

Para guardar as cláusulas aprendidas, todas são penduradas na fórmula. Quando a raiz da fórmula é um operador $\vee$ então um novo operador $\wedge$ é incluído como nova raiz da fórmula, nele são colocados a raiz anterior e todas as cláusulas aprendidas.

A ordem de decisão das variáveis é feita aleatoriamente iniciando pelo valor verdade verdadeiro e depois com o falso. O resolvedor avalia sistematicamente as valorações para as variáveis da fórmula até que uma satisfatível seja encontrada ou que a fórmula seja verificada como insatisfatível.

Para tomar como exemplo o processo de busca, retrocesso não-cronológico e aprendizado de cláusula, tomamos a fórmula $((p \vee q) \wedge (r \vee \neg s) \wedge \neg q) \vee \neg(\neg p \wedge (r \vee \neg s) \wedge q)$ já nivelada e representada na figura 5.1. Agora imaginemos que a seguinte valoração já foi feita:

r - verdadeiro no nível de decisão 1 ($r_{v@1}$);

q - verdadeiro no nível de decisão 2 ($q_{v@2}$);

p - falso no nível de decisão 3 ($p_{f@3}$).

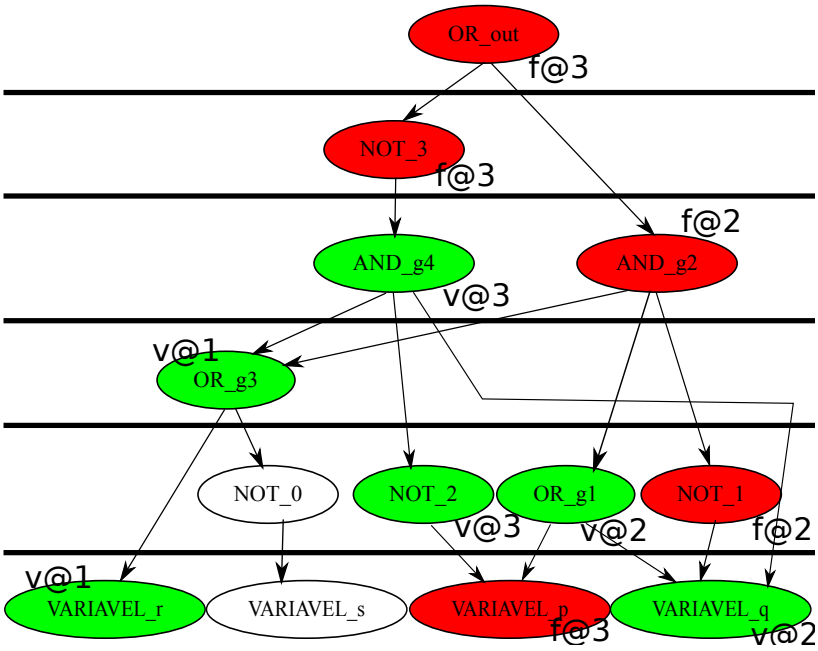

Figura 5.2: Diagrama nivelado com as valorações $r_{v@1}$, $q_{v@2}$, $p_{f@3}$

Após a valoração $p_{f@3}$ podemos identificar na figura 5.2 que a raiz da fórmula se torna falsa e por isso temos um conflito, pois estamos procurando uma valoração onde a raiz se torne verdadeira. Nesse momento o resolvedor tenta identificar o motivo desse conflito e começa a descer a fórmula, a partir da raiz, a procura dos níveis de decisão associados a essa valoração na raiz. O processo inicia identificando as causas da raiz. Como a raiz é um operador $\vee$ ele só se torna falso quando todos os seus filhos são falsos, então empilha os filhos como causa. Avaliando o filho mais a esquerda da raiz temos o NOT_3 que foi valorado como falso no nível 3 (mesmo da raiz) e por isso vai avaliar seu filho que é o AND_{g4} que foi valorado como verdadeiro no nível 3 (ainda o mesmo nível da raiz) e continua o processo. Para um operador $\wedge$ ser valorado como verdadeiro todos os seus filhos também devem ser valorados como verdadeiro, por isso os filhos de AND_{g4} são empilhados, o filho mais a esquerda é o OR_{g3} que foi valorado no nível 1, como é um operador do tipo $\vee$ basta um filho valorado como verdadeiro para se tornar verdadeiro, então a variável decidida nesse nível é adicionada à cláusula aprendida e no caso é r. Continuando a busca o segundo filho de AND_{g4} é o NOT_2 que ficou verdadeiro no nível 3

e o seu motivo foi a valoração de p como falso no nível 3, como p é uma causa de conflito ela será incluída na cláusula aprendida. Agora avaliando o último filho de AND_{g4} temos a variável q valorada como verdadeira no nível 2, q será incluída na cláusula aprendida. O filho mais a esquerda da raiz já foi todo explorado e suas causas aprendidas, o segundo filho da raiz será avaliado que é o AND_{g2} valorado como falso no nível 2. Para um operador $\wedge$ se tornar falso basta que um de seus filhos seja valorado como falso então a variável decidida nesse nível é adicionada à cláusula aprendida, que no caso é a variável q que já foi adicionada à cláusula aprendida. Ao fim do processo adicionamos todas as variáveis identificadas como participantes do conflito a adicionamos à uma cláusula, no exemplo temos a seguinte cláusula aprendida: $(\neg r \vee \neg q \vee p)$.

O processo de busca continua e inverte a polaridade da variável decidida no nível corrente, caso já não tenha sido testada, e repete todo o processo. Em nosso exemplo a variável p será re-valorada como verdadeira e a raiz será verdadeira, portanto a fórmula é satisfatível, como podemos ver na figura 5.3.

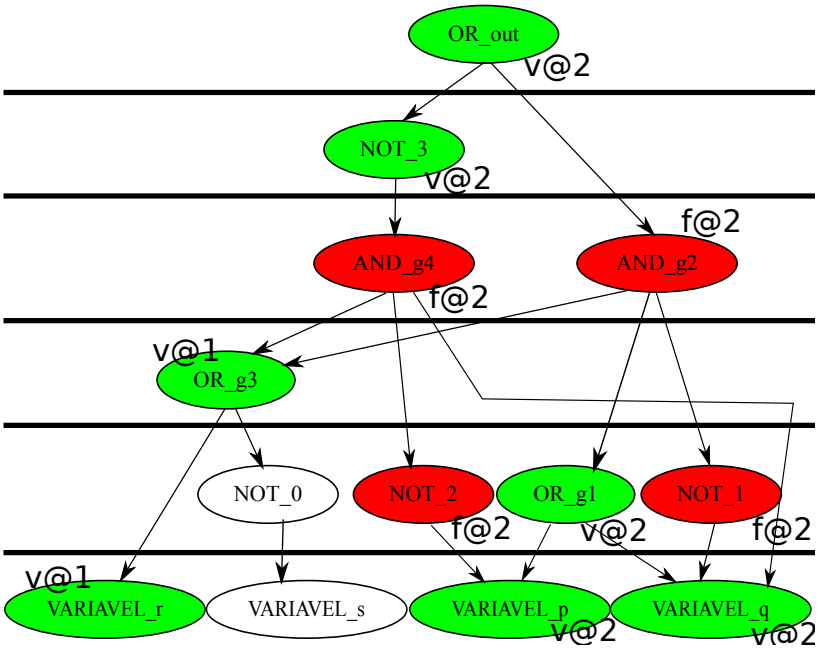

Figura 5.3: Diagrama nivelado com as valorações $r_{v@1}$, $q_{v@2}$, $p_{v@3}$

5.3 Considerações Finais

Neste capítulo apresentamos uma abordagem para resolver satisfatibilidade sem a necessidade de converter a fórmula original para o formato CNF. Essa abordagem gerou um resolvedor não-clausal chamado LIAMFSAT, no mesmo espírito do NOCLAUSE.

O LIAMFSAT aproveita a estrutura definida na descrição ISCAS, que permite operadores com vários nós pais. O uso dessas características permitiu a definição de níveis para os operadores a propagação de valorações de baixo para cima.

A propagação das valorações tem uma característica diferenciada: todas as decisões executadas são feitas sobre as variáveis da fórmula e não sobre qualquer nó do grafo como acontece com o NOCLAUSE[23]. Assim, os operadores da fórmula tem apenas a função de guiar a valoração parcial e atrasam alguns conflitos para a saída da fórmula, enriquecendo o aprendizado de cláusulas e o retrocesso não cronológico.

O aprendizado de cláusulas implementado ainda não verifica se aprendeu cláusulas repetidas nem se algumas cláusulas podem ser subjugadas por outras já aprendidas. Esse problema não se limita ao resolvedor LIAMFSAT mas também faz parte da discussão dos resolvedores modernos, tanto para fórmulas em CNF quanto para fórmulas não clausais.

Trazemos nesse resolvedor um apanhado de técnicas utilizadas nos principais resolvedores existentes, clausais ou não. Incorporamos o retrocesso não-cronológico, que traça as decisões do algoritmo e descobre qual delas foi a que causou o conflito; também implementamos o aprendizado de cláusulas, onde a sumarização do que é descoberto para executar o retrocesso é adicionada à fórmula; um BCP rápido guiado por sinais e orientado das folhas à raiz.

No próximo capítulo avaliamos experimentalmente nossa implementação.

CAPÍTULO 6

AVALIAÇÃO EXPERIMENTAL

O resolvedor, aqui apresentado, executa a ramificação apenas nas variáveis da fórmula ISCAS deixando os operadores da fórmula apenas como guias para saber se a valoração ainda possui chance de se tornar consistente ou se já deixa a fórmula em um estado inconsistente.

Na execução da avaliação experimental do LIAMFSAT observamos um comportamento diferente daquele indicado por Järvisalo e Junttila[13], que afirma que um resolvedor restrito à variáveis de entradas não consegue ser suficientemente bom em comparação com os resolvedores que fazem ramificações em qualquer nó da fórmula. Apesar da busca possuir um espaço $O(2^v)$, onde v representa o número de variáveis, menor que o espaço $O(2^o)$, onde o representa o número de operadores, a quantidade de retrocessos executados no resolvedor restrito à entradas passa a ser muito maior em comparação ao resolvedor não restrito.

Nos capítulos anteriores defendemos que aplicar um resolvedor diretamente na representação original da fórmula ajuda a aproveitar a informação estrutural, permitindo que, o resolvedor possa atuar em áreas críticas e até evitar um mínimo local por muito tempo. Em [13] a fórmula é convertida e com isso várias variáveis são injetadas na conversão de Tseitin, além da inserção de várias cláusulas para deixar a fórmula equivalente em satisfatibilidade. O nosso resolvedor traz uma abordagem diferente, pois atua diretamente na fórmula original.

A avaliação experimental foi dividida em duas partes: a primeira mostra a evolução do resolvedor aqui desenvolvido em um conjunto de fórmulas geradas a partir de instâncias de problemas de planejamento; a segunda parte é a execução do resolvedor com um conjunto de fórmulas utilizado em competições de satisfatibilidade não-clausal.

A divisão da avaliação experimental foi feita para identificar os pontos chave das

técnicas avaliadas e implementadas no resolvedor, principalmente quando se diz respeito às técnicas trazidas dos resolvedores em CNF.

Os experimentos foram realizados em um computador AMD Opteron 2.8GHz com 128GB de memória RAM. Cada instância teve um tempo limite de 3600 segundos. Quando esse limite é extrapolado as tabelas de comparação mostram como TLE (tempo limite excedido).

6.1 Avaliação das técnicas empregadas

Para avaliar a influência do aprendizado de conflitos e retrocesso não cronológico no desempenho do resolvedor foram escolhidas algumas fórmulas mais simples, geradas a partir de planejamento de jogos. O conjunto é formado por 49 fórmulas insatisfatíveis e 1 fórmula satisfatível.

Separamos o resolvedor em três configurações, cada uma contendo as seguintes características:

LIAMFSAT - possui aprendizado e retrocesso não-cronológico;

LIAMFSATr - possui apenas retrocesso não-cronológico;

LIAMFSAT0 - não possui aprendizado nem retrocesso não cronológico.

Todas as três variações conseguiram executar todas as fórmulas, salvo o LIAMFSAT0 que só não resolveu uma fórmula. Na tabela 6.1 tem-se o tempo total de execução dos problemas resolvidos. O problema mais difícil nesse conjunto é justamente a fórmula satisfatível, principalmente pelo seu tamanho, pois temos 184071 operadores e 2460 variáveis de entrada, sendo 411963 literais da fórmula.

O LIAMFSAT conseguiu resolver a fórmula satisfatível em apenas 92 segundos, ao passo que LIAMFSATr resolveu em $100,2$. Apesar do LIAMFSATr possuir retrocesso não cronológico ele ainda não consegue evitar que os mesmos erros sejam cometidos repetidas vezes e por isso o seu tempo de execução é mais elevado que o LIAMFSAT. O LIAMFSAT0 não conseguiu resolver a fórmula em seu tempo limite de 3600 segundos, por-

que o resolvedor não possui retrocesso não-cronológico e com isso não consegue identificar os erros cometidos e o algoritmos faz uma busca exaustiva no espaço de busca.

Tabela 6.1: Tempo gasto em segundos para resolver os problemas

LIAMFSAT	LIAMFSATr	LIAMFSAT0
92,28(49)	100,60(49)	0,58(48)

Dentre as fórmulas do experimento, a fórmula *buttons.iscas_6_UNSAT* foi resolvida em $0,35$ segundos pelo LIAMFSAT0 e em $0,004$ segundos pelo LIAMFSAT e $0,000$ pelo LIAMFSATr. O LIAMFSAT possuiu um desempenho um pouco inferior que o LIAMFSATr justo pelo processo de aprendizagem pois a fórmula é muito pequena, apenas 96 variáveis, 398 operadores e 419 instâncias de variáveis, e todo o maquinário de aprender cláusulas fez com que o LIAMFSAT tivesse uma pequena diferença em relação ao LIAMFSATr. O LIAMFSAT0 demorou mais que as outras variações justo pelo fato de não conseguir efetuar o retrocesso não-cronológico. É interessante perceber também que o LIAMFSAT aprendeu 17 cláusulas nessa fórmula.

Entrando em um cenário com fórmulas maiores (que serão discutidas na seção seguinte com outros resolvedores), pegamos a menor fórmula, *2pipe*, que possui 96 variáveis, 812 operadores e 1326 instâncias de variáveis. O LIAMFSAT0 não conseguiu resolver essa fórmula com um tempo limite de 3600 segundos, já o LIAMFSATr resolveu esse problema em $25,72$ segundos, enquanto que o LIAMFSAT resolveu em $0,04$ e aprendeu 223 cláusulas. Nesse ponto fica evidente a vantagem que o maquinário para efetuar o retrocesso não-cronológico é importante, e ainda tão importante quanto o retrocesso é o aprendizado de cláusulas, onde o resolvedor consegue evitar que os mesmos erros sejam cometidos diversas vezes.

6.2 Comparação com outros resolvedores

Para comparar o LIAMFSAT com outros resolvedores no estado da arte em satisfatibilidade não-clausal obtivemos fórmulas que são utilizadas nas competições de resolvedores[1]. Essas fórmulas foram geradas para verificação formal de microprocessadores superescalares e *Very Long Instruction Word* (VLIW), a descrição de como foram geradas as fórmulas estão descritas em [18, 26, 27, 25].

Processadores VLIW aproveitam o paralelismo das instruções de uma forma diferente de processadores superescalares, que geralmente possuem pipeline. A abordagem VLIW executa as operações baseadas em uma agenda determinada quando os programas são compilados. A ordem de execução das instruções, incluindo as instruções que executarão em paralelo, são definidas pelo compilador.

Dentre as fórmulas disponíveis escolhemos quatro conjuntos, que são:

fvp-unsat.1.0 - 4 fórmulas insatisfatíveis para verificação formal de processadores superescalares e VLIW corretos;

fvp-unsat.2.0 - 21 fórmulas insatisfatíveis e 1 satisfatível de processadores superescalares corretos;

sss-sat1.0 - 100 fórmulas satisfatíveis geradas para verificação formal de processadores superescalares com defeito;

vliw-sat.1.0 - 100 fórmulas satisfatíveis para verificação formal de processadores VLIW com defeito.

As fórmulas de verificação são geradas a partir da união de duas fórmulas: uma a especificação correta do circuito; e outra a implementação do circuito. As saídas dos circuitos são ligadas por um $\oplus$ que, por sua vez, são ligados a um $\vee$. Quando a especificação e a implementação possuem o mesmo funcionamento a fórmula será insatisfatível, quando não forem equivalentes a fórmula será satisfatível.

[1] A competição SAT-Race começou em 2006 e tem periodicidade bienal a última edição 2010 está no endereço http://baldur.iti.uka.de/sat-race-2010

As fórmulas utilizadas possuem apenas $\wedge, \vee, \neg$, como foi dito nos capítulos anteriores. A transformação do $\oplus$ foi feita pelo fornecedor das fórmulas, portanto nenhuma modificação foi feita nas fórmulas testadas.

Dentre os resolvedores escolhidos para executar os testes temos:

SATMATE - resolvedor não-clausal baseado em NNF, utilizando as técnicas de *General Matings*;

NFLSAT - resolvedor não-clausal baseado em NNF, evolução do SATMATE, utiliza as técnicas de *General Matings* mas também aplica algumas técnicas de algoritmos baseados em DPLL para ajudar na busca nos grafos, foi discutido no capítulo 4;

BCMINISAT - resolvedor MINISAT baseado em fórmulas em CNF, mas com um sistema que converte a fórmula não-clausal em CNF pela transformação de Tseitin. Esse resolvedor marca as variáveis originais da fórmulas para efetuar a ramificação apenas nelas;

LIAMFSAT - resolvedor não-clausal com ramificação restrita as variáveis de entrada, foi discutido no capítulo 5.

O resolvedor NOCLAUSE não foi utilizado nos testes pois o seu código é muito dependente de condições específicas e ainda falha na leitura de algumas fórmulas. Em trabalhos futuros temos o objetivo de criar um ambiente mais controlado para pode executar todos os resolvedores, mesmo que isso implique na alteração do código de terceiros.

A seguir são apresentadas as tabelas com tempos de processamento para os problemas testados. Na tabela 6.2 tem-se os tempos para todos os problemas do conjunto fvp-unsat.2.0. Na tabela 6.3 tem-se os tempos para todos os problemas do conjunto fvp-unsat.1.0. Na tabela 6.4 tem-se a sumarização de problemas resolvidos em cada uma dos 4 conjuntos de problemas e por fim, tem-se a tabela 6.5 que mostra a soma do tempo de processamento utilizado para resolver as instâncias, entre parenteses está a quantidade de problemas resolvidos para aquela soma de tempo.

Com os tempos obtidos em cada um dos testes, algumas conclusões podem ser obtidas. Primeiramente, apesar do LIAMFSAT obter tempos consideravelmente maiores que o

Tabela 6.2: Tempo de execução em segundos para cada fórmula no conjunto fvp-unsat.2.0

Problema	SATMATE	NFLSAT	LIAMFSAT	BCMINISAT
2pipe.iscas	29,47	0,21	0,02	47,57
2pipe_1_ooo.iscas	21,60	0,17	0,06	4,46
2pipe_2_ooo.iscas	311,42	0,19	0,07	111,49
3pipe.iscas	TLE	3,21	0,30	TLE
3pipe_1_ooo.iscas	TLE	2,79	312,95	TLE
3pipe_2_ooo.iscas	TLE	3,66	114,17	TLE
3pipe_3_ooo.iscas	TLE	3,56	10,91	TLE
4pipe.iscas	TLE	56,82	10,59	TLE
4pipe_3_ooo.iscas	TLE	13,49	397,73	TLE
4pipe_1_ooo.iscas	TLE	10,19	1310,39	TLE
4pipe_2_ooo.iscas	TLE	10,33	TLE	TLE
4pipe_4_ooo.iscas	TLE	14,30	TLE	TLE
5pipe.iscas	TLE	15,42	131,26	TLE
5pipe_1_ooo.iscas	TLE	19,28	TLE	TLE
5pipe_2_ooo.iscas	TLE	14,78	TLE	TLE
5pipe_3_ooo.iscas	TLE	20,93	TLE	TLE
5pipe_4_ooo.iscas	TLE	45,21	TLE	TLE
5pipe_5_ooo.iscas	TLE	19,02	TLE	TLE
6pipe.iscas	TLE	TLE	1265,19	TLE
6pipe_6_ooo.iscas	TLE	64,85	TLE	TLE
7pipe.iscas	TLE	TLE	1260,44	TLE
7pipe_bug.iscas	TLE	70,04	TLE	1123,02

Tabela 6.3: Tempo de execução em segundos para cada fórmula no conjunto fvp-unsat.1.0

Problema	SATMATE	NFLSAT	LIAMFSAT	BCMINISAT
1dlx_c_mc_ex_bp_f.iscas	TLE	0,11	22,04	76,44
2dlx_ca_mc_ex_bp_f_new.iscas	TLE	0,66	TLE	TLE
2dlx_cc_mc_ex_bp_f_new.iscas	TLE	1,42	424,34	TLE
9vliw_bp_mc.iscas	TLE	26,12	1243,71	TLE

Tabela 6.4: Quantidade de problemas resolvidos nos conjuntos de fórmulas

Conjunto	Total	SATMATE	NFLSAT	LIAMFSAT	BCMINISAT
fvp-unsat.1.0	4	0	4	3	1
fvp-unsat.2.0	22	3	20	13	4
sss-sat-1.0	100	39	100	11	100
vliw-sat-1.1	100	15	100	33	16

Tabela 6.5: Tempo gasto em segundos para resolver os problemas para cada conjunto de problemas

Conjunto	SATMATE	NFLSAT	LIAMFSAT	BCMINISAT
fvp-unsat.1.0(4)	0(0)	28,31(4)	1690,09(3)	76,44(1)
fvp-unsat.2.0(22)	362,49(3)	388,45(20)	4814,12(13)	1286,54(4)
sss-sat-1.0(100)	26463,12(39)	33,72(100)	245,90(11)	6501,59(100)
vliw-sat-1.1(100)	1889,07(15)	293,76(100)	3428,80(33)	1300,04(16)

NFLSAT, ele apresentou um desempenho compatível em problemas insatisfatíveis. Dentro dos dois conjuntos de problemas satisfatíveis o LIAMFSAT foi pior no conjunto sss-sat-1.0 se comparado com os outros 3 resolvedores, mas nos problemas vliw-sat-1.1 ele foi superior em relação ao SATMATE e o BCMINISAT.

As causas do desempenho inferior, principalmente, em problemas satisfatíveis pode ser atribuído à dificuldade da fórmula, deixando a máquina de busca enumerar as possibilidades em um ramo do DAG onde o algoritmo não consegue obter informação suficiente para executar um retrocesso não cronológico mais distante, ficando assim, em um mínimo local até esgotar o espaço de busca desse ramo.

O BCMINISAT tem um bom desempenho no conjunto sss-sat-1.0, pois ele consegue resolver os 100 problemas enquanto que o LIAMFSAT resolve apenas 11, um dos motivos desse resultado está na estrutura do MINISAT, um dos melhores resolvedores em CNF, que executa toda a busca na fórmula sendo guiado pela ramificação restrita às variáveis de entrada. O MINISAT já possui mecanismos de reinícios sofisticados que ajudam a evitar o gasto desnecessário de tempo em um mínimo local, mas também notamos que no conjunto vliw-sat-1.1 o MINISAT já não consegue o mesmo desempenho, com 16 problemas resolvidos contra 33 do LIAMFSAT.

Finalmente, mesmo com os problemas complexos utilizados em competições de satisfatibilidade o resolvedor LIAMFSAT mostra que possui um bom desempenho e mais importante que isso, é um contra exemplo ao discutido em [13] quanto à restrição a variáveis de entrada, conforme é percebido nos testes executados.

6.3 Avaliação do tamanho das fórmulas

Para comparar o espaço de busca da fórmula em ISCAS e a sua versão em CNF, obtida pela transformação de Tseitin[24], contamos a quantidade de operadores, variáveis e literais. Também fizemos a contagem na fórmula ISCAS sem considerar que cada operador poderia possuir mais de um pai. Dessa forma reinstanciamos todos os operadores da fórmula e a chamamos de ISCAS-r. A comparação está na tabela 6.6.

Tabela 6.6: Tabela comparativa em relação ao número de Variáveis, Cláusulas e Literais para os problemas em CNF, ISCAS e ISCAS-r

Problema	Representação	Variáveis	Cláusulas/Operadores	Literais
2pipe	CNF	892	6695	18637
	ISCAS	94	812	1326
	ISCAS-r	94	11146143	23666560
2pipe_1_ooo	CNF	834	7026	19768
	ISCAS	136	2105	5208
	ISCAS-r	136	13204376	24910751
2pipe_2_ooo	CNF	925	8213	23161
	ISCAS	139	2194	5400
	ISCAS-r	139	27282559	50776426
2dlx_cc_	CNF	917	4320	11802
mc_ex_	ISCAS	193	1132	1517
bp_f2_bug099	ISCAS-r	193	54809999	147700361
3pipe	CNF	2468	27533	78509
	ISCAS	198	2385	3923
	ISCAS-r	198	5052426967	11432712830
3pipe_1_ooo	CNF	2223	26561	76153
	ISCAS	302	7227	18585
	ISCAS-r	302	8675515688	15919231162
3pipe_2_ooo	CNF	2400	29981	86081
	ISCAS	305	7413	18851
	ISCAS-r	305	18129245059	33285561031
3pipe_3_ooo	CNF	2577	33270	95618
	ISCAS	308	7592	19230
	ISCAS-r	308	26788900451	52079743741

Como podemos identificar na tabela 6.6, a quantidade de operadores aumenta de forma descontrolada, inibindo assim qualquer tentativa de explorar esse espaço de busca altamente redundante. Ainda podemos identificar que a quantidade de cláusulas e variáveis aumenta consideravelmente quando a fórmula é convertida para CNF. Um resolvedor

que atua diretamente em ISCAS pode se beneficiar da representação do DAG tratando instâncias maiores e com um menor número de variáveis que um resolvedor que utilize CNF.

O resolvedor LIAMFSAT aproveita a estrutura original da fórmula escrita em ISCAS, onde cada operador pode possuir múltiplos pais. Essa característica é fundamental para que o resolvedor consiga resolver as fórmulas.

CAPÍTULO 7

CONCLUSÃO

Nessa dissertação fizemos uma análise de técnicas para se resolver satisfatibilidade em fórmulas em CNF, em formatos mais gerais não-clausais, terminando com a implementação de um resolvedor que abriga uma grande quantidade de técnicas do estado da arte.

O grande avanço nos resolvedores em CNF aconteceu com o surgimento do retrocesso não-cronológico, onde o resolvedor passou a identificar o motivo da valoração parcial ter deixado a fórmula inconsistente e retroceder a busca exatamente no ponto do erro, ao invés de executar um retrocesso cronológico onde apenas o último passo é desfeito.

Junto com o retrocesso não-cronológico ganhamos a possibilidade de identificar o conjunto de decisões que causou a inconsistência na fórmula, permitindo assim incluir uma nova cláusula que ajuda a evitar que o mesmo erro seja cometido, ou seja, a nova cláusula evita que a máquina de busca continue quando um conjunto de literais torna esta cláusula falsa. Esse mecanismo evita que o resolvedor avance vários níveis de decisão para encontrar o mesmo erro.

Além do retrocesso não-cronológico outra grande contribuição dos resolvedores em CNF foi a criação da lista de observação de literais por cláusula que deixou mais eficiente a identificação do momento crítico de uma cláusula, que é quando a cláusula está com apenas um literal não valorado, identificando, assim, uma cláusula unitária. Esse processo permite uma inferência do literal não valorado sem uma busca pelo conjunto de cláusulas.

As heurísticas de ramificação também ganharam um braço de pesquisa, mas é impossível criar uma heurística que consiga escolher sempre a melhor a variável para ramificar, pois cada tipo de problema gera uma fórmula com alguma característica específica. Com essa dificuldade o consenso foi o de criar uma heurística que consuma pouco tempo de processamento, ou seja, deve escolher a próxima variável a ser decidida sem efetuar

muitos cálculos. A partir daí uma heurística se destacou entre os resolvedores, a *Variable State Independent Decaying Sum* (VSIDS), que aproveita a informação dinâmica gerada pelo aprendizado de cláusulas, e mesmo assim uma porcentagem das decisões ainda são escolhas aleatórias e a quantidade varia de resolvedor para resolvedor.

O aprendizado de conflitos apresenta alguns problemas nos resolvedores. O principal deles está relacionado com a cláusula que deve ser aprendida a cada conflito, esta cláusula pode ser enorme, com centenas de literais, como também pode ser tão sucinta quanto uma clausula de três literais. Quanto maior a cláusula aprendida, menor é a chance daquela combinação acontecer novamente e mais memória o resolvedor consumirá. Outra questão relevante é a quantidade de cláusulas que deve ser aprendida, pois quanto mais cláusulas, mais tempo o BCP levará para processar. Vale lembrar que o BCP representa 80% do tempo gasto pelo resolvedor. Essas questões ainda são objeto de estudo e algumas limitações são feitas, como guardar uma cláusula de no máximo 5 literais, e ainda de vez em quando, executar uma rotina que sub-julgue as cláusulas aprendidas, reduzindo assim, a quantidade de cláusulas.

Também encontramos algumas características que tendem a evitar um mínimo local, pois mesmo possuindo retrocesso não-cronológico, com as escolhas "erradas" na ramificação ainda podemos cair na enumeração da tabela verdade, e por isso os resolvedores modernos reiniciam todo o processo de busca e partem de uma decisão diferente. Apesar do reinício, os resolvedores continuam completos pois as cláusulas aprendidas ainda são guardadas e possuem informação dos caminhos que o resolvedor já explorou e sabe que são inconsistentes.

Caminhando para o cenário não-clausal também caímos nos mesmos problemas do CNF e tentamos trazer várias soluções como o retrocesso não-cronológico, a lista de observação de literais e também as heurísticas de ramificação.

Nos resolvedores não-clausais que foram estudados identificamos todas essas características trazidas de CNF e algumas abordagens criativas para tratar os operadores de forma que se aproveite o máximo a estrutura da fórmula. O NFLSAT é o melhor exemplo pois ele reescreve a fórmula em dois grafos e depois aplica as técnicas de observação de

literais, que passa a ser a observação de dois caminhos no grafo vertical. O retrocesso não-cronológico está na busca no grafo horizontal.

O resolvedor criado aqui, o LIAMFSAT, aproveita a estrutura da fórmula para efetuar uma propagação baseada estritamente na ramificação das variáveis de entrada da fórmula, deixando os operadores apenas como propagadores da valoração parcial. Já o NOCLAUSE trata tanto as variáveis quanto os operadores como variáveis da fórmula e efetua ramificações em qualquer nó do grafo.

Mesmo com a abordagem restrita que o LIAMFSAT faz em relação as variáveis de entrada, conseguimos obter um resultado próximo ao apresentado pelos resolvedores atuais. Alguns problemas de desempelho podem estar ligados as estruturas implementadas, e um dos trabalhos futuros é a revisão da estrutura interna e de alguns algoritmos, como por exemplo o BCP, afim de otimizar a implementação.

O destaque neste trabalho é o uso de fórmulas em um formato não-clausal para resolver satisfatibilidade, evitando, assim a conversão da fórmula para a CNF, tentando atingir um desempenho próximo aos melhores resolvedores atuais.

Outro trabalho futuro diz respeito ao processo de aprendizagem, onde ainda aprendemos cláusulas do mesmo modo que os resolvedores em CNF e baseados no DPLL fazem. Agora que as fórmulas estão em um formato diferente da CNF, parece ser possível aprender sub-fórmulas em qualquer operador, evitando que várias cláusulas sejam aprendidas junto a raiz da fórmula, deixando os espaços de busca disjuntos com aprendizados diferentes, deixando mais evidente quais sub-fórmulas são críticas para a busca de consistência da fórmula, podendo até antecipar o momento em que o algoritmo descobre que a fórmula como um todo é insatisfatível.

BIBLIOGRAFIA

[1] J. Alfredsson e O. Consulting. The SAT Solver kw. *The SAT race 2008: Solver descriptions, 2008*.

[2] Peter B. Andrews. Theorem proving via general matings. *J. ACM*, 28:193–214, April de 1981.

[3] Peter B. Andrews. *An introduction to mathematical logic and type theory: to truth through proof*. Academic Press Professional, Inc., San Diego, CA, USA, 1986.

[4] H. Bohm. Report on a SAT competition. Relatório técnico, Technical report.

[5] Stephen A. Cook. The complexity of theorem-proving procedures. *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, páginas 151–158, New York, NY, USA, 1971. ACM.

[6] M. Davis, G. Logemann, e D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):397, 1962.

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

[8] Niklas Eén e Niklas Sörensson. An extensible sat-solver. Enrico Giunchiglia e Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, páginas 333–336. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-24605-3_37.

[9] Jon William Freeman. *Improvements to propositional satisfiability search algorithms*. Tese de Doutorado, Philadelphia, PA, USA, 1995. UMI Order No. GAX95-32175.

[10] J. Huang. The effect of restarts on the efficiency of clause learning. *Proceedings of the 20th international joint conference on Artifical intelligence*, páginas 2318–2323. Morgan Kaufmann Publishers Inc., 2007.

[11] Himanshu Jain, Constantinos Bartzis, e Edmund M. Clarke. Satisfiability checking of non-clausal formulas using general matings. *Theory and Applications of Satisfiability Testing (SAT)*, páginas 75–89, 2006.

[12] Himanshu Jain e Edmund M. Clarke. Efficient sat solving for non-clausal formulas using dpll, graphs, and watched cuts. *46th Design Automation Conference (DAC)*, 2009. To appear.

[13] Matti Järvisalo e Tommi Junttila. Limitations of restricted branching in clause learning. Christian Bessiere, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP 2007)*, volume 4741 of *Lecture Notes in Computer Science*, páginas 348–363. Springer, 2007.

[14] Robert G. Jeroslow e Jinchang Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990. 10.1007/BF01531077.

[15] J. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. *Progress in Artificial Intelligence*, páginas 850–850, 1999.

[16] J.P. Marques-Silva e K.A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *Computers, IEEE Transactions on*, 48(5):506–521, 2002.

[17] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, e S. Malik. Chaff: Engineering an efficient SAT solver. *Design Automation Conference, 2001. Proceedings*, páginas 530–535. IEEE, 2005.

[18] Laurence Pierre e Thomas Kropf, editors. *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*, volume 1703 of *Lecture Notes in Computer Science*. Springer, 1999.

[19] B. Selman, H. Kautz, e B. Cohen. Local search strategies for satisfiability testing. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 26:521–532, 1996.

[20] B. Selman, H. Levesque, e D. Mitchell. A new method for solving hard satisfiability problems. páginas 440–446, 1992.

[21] Zbigniew Stachniak e Anton Belov. Speeding-up non-clausal local search for propositional satisfiability with clause learning. *Proceedings of the 11th international conference on Theory and applications of satisfiability testing*, SAT'08, páginas 257–270, Berlin, Heidelberg, 2008. Springer-Verlag.

[22] N. Sörensson e N. Eén. MiniSat 2.1 and MiniSat++ 1.0—SAT Race 2008 Editions. *SAT 2009 competitive events booklet: preliminary version*, páginas 31, 2009.

[23] C. Thiffault, F. Bacchus, e T. Walsh. Solving non-clausal formulas with DPLL search. *Principles and Practice of Constraint Programming–CP 2004*, páginas 663–678, 2004.

[24] G.S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in constructive mathematics and mathematical logic*, 2(115-125):10–13, 1968.

[25] Miroslav N. Velev. Using positive equality to prove liveness for pipelined microprocessors. *ASP-DAC '04: Proceedings of the 2004 conference on Asia South Pacific design automation*, páginas 316–321, Piscataway, NJ, USA, 2004. IEEE Press.

[26] Miroslav N. Velev e Randal E. Bryant. Formal verification of superscalar microprocessors with multicycle functional units. páginas 112–117, 2000.

[27] Miroslav N. Velev e Randal E. Bryant. Effective use of boolean satisfiability procedures in the formal ve rification of superscalar and vliw microprocessors. *Journal of Symbolic Computation*, páginas 226–231, 2001.

[28] Ramin Zabih. A rearrangement search strategy for determining propositional satisfiability. *in Proceedings of the National Conference on Artificial Intelligence*, páginas 155–160, 1988.

[29] Hantao Zhang. Sato: An efficient propositional prover. William McCune, editor, *Automated Deduction—CADE-14*, volume 1249 of *Lecture Notes in Computer Science*, páginas 272–275. Springer Berlin / Heidelberg, 1997. 10.1007/3-540-63104-6_28.